Lab1

实验目的

使用前馈神经网络拟合函数

$$y = \log_2(x) + \cos(\frac{\pi x}{2}), x \in [1, 16]$$

实验原理

1.代码框架

1. `utils.py` : 工具函数包
2. `model.py` : 前馈神经网络模型
3. `train.py` : 训练脚本,并在验证集上评估模型性能
4. `eval.py` : 在测试集上评估模型性能
5. `accelerate_config.ymal` : 配置文件

2.网络结构

根据训练配置文件搭建网络结构,隐藏层的features分别为:

$$[hidden\_size, hidden\_size * 2, hidden\_size * 4, \cdots, hidden\_size * 4, hidden\_size * 2, hidden\_size]$$

可选择的激活函数分别为:Sigmoid, ReLu, Tanh, SiLU
`model` 定义如下:

```
class MLP(nn.Module):
    def __init__(self, cfg):
        super(MLP, self).__init__()

        # Activation function
        if cfg.act_fn == "sigmoid":
            self.act_fn = F.sigmoid
        elif cfg.act_fn == "relu":
            self.act_fn = F.relu
        elif cfg.act_fn == "tanh":
            self.act_fn = F.tanh
        elif cfg.act_fn == "silu":
            self.act_fn = F.silu
```

```
            else:
                raise ValueError("Not supported activation function")

            # Layers
            self.layer_in = nn.Linear(1, cfg.hidden_size)
            self.feature_size = [cfg.hidden_size * (2 ** i) for i in range(cf
    g.n_muti_layers + 1)]

            self.layers = nn.ModuleList([])

            for i in range(cfg.n_muti_layers):
                self.layers.add_module(f"diff_layer_{i}", nn.Linear(self.featu
    re_size[i], self.feature_size[i+1]))

            self.layers.add_module("mid_layer", nn.Linear(self.feature_size[-
    1], self.feature_size[-1]))

            self.feature_size = list(reversed(self.feature_size))
            for i in range(cfg.n_muti_layers):
                self.layers.add_module(f"coll_layer_{i}", nn.Linear(self.featu
    re_size[i], self.feature_size[i+1]))

            self.layer_out = nn.Linear(cfg.hidden_size, 1)


        def forward(self, x):
            x = self.act_fn(self.layer_in(x))
            for i, layer in enumerate(self.layers):
                x = self.act_fn(layer(x))
            x = self.layer_out(x)
            return x
```

以N=200时为例,我使用的网络结构为:

```
==========================================================================================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param #
==========================================================================================
MLP                                      [200, 1, 1]               [200, 1, 1]               --
├─Linear: 1-1                            [200, 1, 1]               [200, 1, 128]             256
├─ModuleList: 1-2                        --                        --                        --
│    └─Linear: 2-1                       [200, 1, 128]             [200, 1, 256]             33,024
│    └─Linear: 2-2                       [200, 1, 256]             [200, 1, 512]             131,584
│    └─Linear: 2-3                       [200, 1, 512]             [200, 1, 512]             262,656
│    └─Linear: 2-4                       [200, 1, 512]             [200, 1, 256]             131,328
│    └─Linear: 2-5                       [200, 1, 256]             [200, 1, 128]             32,896
├─Linear: 1-3                            [200, 1, 128]             [200, 1, 1]               129
==========================================================================================
Total params: 591,873
Trainable params: 591,873
Non-trainable params: 0
Total mult-adds (M): 118.37
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 2.87
Params size (MB): 2.37
Estimated Total Size (MB): 5.24
==========================================================================================
```

3.训练过程

1. 首先读取配置文件,并设置随机种子,根据配置文件初始化模型
2. 使用 `print_model_summary` 函数打印模型结构
3. 读取数据集,并将数据集分为训练集,验证集,测试集
4. 把所需参数和配置文件导入 `Trainer` 类,损失函数为 `MSE` ,优化器为 `Adam`
5. 训练完成后加载 `Evaluator` 类,评估模型在验证集上的性能
6. 选择一组合适的超参数

`Trainer` 类定义如下:

```
class Trainer:
    """
    Trainer of the model

    Args:
        - model: the model to train
        - train_loader: the training data loader
        - accelerator: the accelerator
        - make_opt: a function that takes the model parameters and returns
an optimizer
        - config: the configuration
        - results_path: the path to save the results
        - var_loader: optional, the validation data loader
    """
    def __init__(self, model, train_loader, accelerator, make_opt, config,
results_path, var_loader: Optional[torch.utils.data.DataLoader] = None):
        super().__init__()
        self.model = accelerator.prepare(model)
        self.train_loader = train_loader
        self.var_loader = var_loader
        self.accelerator = accelerator
        self.opt = accelerator.prepare(make_opt(self.model.parameters()))
        self.critierion = torch.nn.MSELoss()
        self.cfg = config
        self.results_path = results_path
        self.device = self.accelerator.device
        print('Train on', self.device)
        self.model.to(self.device)
        self.checkpoint_path = self.results_path / f"model.pt"

        self.results_path.mkdir(parents=True, exist_ok=True)
        with open(self.results_path / "config.yaml", "w") as f:
            yaml.dump(dataclasses.asdict(self.cfg), f)

    def train(self):
        loss_list = []
        for epoch in tqdm(range(self.cfg.epochs)):
            for x, y in self.train_loader:
                x, y = x.float(), y.float()
```

```
                x, y = x.to(self.device), y.to(self.device)
                self.opt.zero_grad()
                y_pred = self.model(x)
                loss = self.critierion(y_pred, y)
                self.accelerator.backward(loss)
                self.opt.step()
                loss_list.append(loss.item())
            if self.var_loader is not None:
                for x, y in self.var_loader:
                    x, y = x.float(), y.float()
                    x, y = x.to(self.device), y.to(self.device)
                    self.opt.zero_grad()
                    y_pred = self.model(x)
                    loss = self.critierion(y_pred, y)
                    self.accelerator.backward(loss)
                    self.opt.step()
                    loss_list.append(loss.item())

        with open(self.results_path / 'loss_list.txt', 'w') as f:
            print(loss_list, file=f)
        plt.plot(loss_list)
        plt.savefig(self.results_path / 'loss_list.png')

        self.save()

    def save(self):
        """
        Save model to checkpoint_path
        """
        self.model.eval()
        checkpoint_path = Path(self.checkpoint_path)
        checkpoint_dir = checkpoint_path.parent

        if not os.path.exists(checkpoint_dir):
            os.makedirs(checkpoint_dir)

        checkpoint = {
            "model": self.accelerator.get_state_dict(self.model),
            "opt": self.opt.state_dict(),
        }
        torch.save(checkpoint, checkpoint_path)
        log(f"Saved model to {checkpoint_path}")
        if self.var_loader is not None:
            return checkpoint_path
```

4.评估过程

1. 读取配置文件,并设置随机种子,根据配置文件初始化模型

2. 读取数据集,并将数据集分为训练集,验证集,测试集
3. 把所需参数和配置文件导入 `Trainer` 类,这次把训练集和验证集都用于训练
4. 训练完成后再测试集上评估模型的性能,并画出拟合的曲线与真实曲线的比较

`Evaluator` 定义如下:

```
class Evaluator:
    """
    Evaluate the model on the validation set

    Args:
        - model: the model to evaluate
        - val_loader: the validation data loader
        - accelerator: the accelerator
        - config: the configuration
        - results_path: the path to save the results
    """
    def __init__(self, model, val_loader, accelerator, config, results_pat
h):
        super().__init__()
        self.model = accelerator.prepare(model)
        self.val_loader = val_loader
        self.accelerator = accelerator
        self.cfg = config
        self.device = self.accelerator.device
        self.model.to(self.device)
        self.results_path = results_path
        self.checkpoint_path = results_path / "model.pt"

    def evaluate(self):
        checkpoint = torch.load(self.checkpoint_path, map_location=self.de
vice)
        self.model.load_state_dict(checkpoint["model"])
        self.model.eval()
        loss_list = []
        y_pred_list = []
        y_true_list = []
        x_list = []
        for x, y in self.val_loader:
            x, y = x.float(), y.float()
            x, y = x.to(self.device), y.to(self.device)
            with torch.no_grad():
                y_pred = self.model(x)
            loss = torch.nn.MSELoss()(y_pred, y)
            loss_list.append(loss.item())
            y_pred = y_pred.cpu().numpy()
            y_true = y.cpu().numpy()
            x = x.cpu().numpy()
            y_pred_list.append(y_pred)
            y_true_list.append(y_true)
            x_list.append(x)
```

```
            del x, y, y_pred, y_true
            torch.cuda.empty_cache()

        print(f'Loss_mean: {np.mean(loss_list)}')
        y_pred_list = np.concatenate(y_pred_list)
        y_true_list = np.concatenate(y_true_list)
        x_list = np.concatenate(x_list)

        plt.figure()
        plt.scatter(x_list, y_true_list, label='True')
        plt.scatter(x_list, y_pred_list, label='Pred')
        plt.legend()
        plt.savefig(self.results_path / 'eval.png')
```

5.Utils工具包介绍

我在 `utils.py` 中预先写好了一些工具函数,在这里列出它们的作用:

1. `class TrainConfig`:用于加载训练参数配置
2. `func()`:要拟合的函数
3. `class FunctionDataset(Dataset)`:加载数据集
4. `print_model_summary()`:打印网络结构并估计需要的显存
5. `make_dataloader()`:从数据集中创建数据加载器
6. `get_date_str()`:用于记录评估时间
7. `handle_results_path()`:处理结果路径,如果不存在则创建
8. `zero_init()`:零初始化
9. `init_config_from_args()`:从命令行初始化配置文件
10. `init_logger()`:初始化记录器
11. `log()`:记录器

实验步骤

1.环境配置

安装依赖

```
conda create -n pytorch python=3.9
pip install -r requirements.txt
```

2.训练模型

配置文件:

```
compute_environment: LOCAL_MACHINE
distributed_type: NO
fp16: False
mixed_precision: no
num_processes: 1
gpu_ids: all
use_cpu: false
```

运行以默认参数配置开始训练:

```
accelerate launch --config_file accelerate_config.yaml train.py
```

或在Linux服务器上:

```
bash train.sh
```

3.评估模型

运行评估脚本

```
python eval.py
```

实验结果

在 `train.py` 中,我们能控制的超参数如下:

```
# Data
parser.add_argument("--N", type=int, default=200)

# Architecture
parser.add_argument("--input-size", type=int, default=1)
parser.add_argument("--hidden-size", type=int, default=256)
parser.add_argument("--n-muti-layers", type=int, default=2)
parser.add_argument("--act-fn", type=str, default="relu")

# Training
parser.add_argument("--batch-size", type=int, default=64)
parser.add_argument("--lr", type=float, default=2e-4)
parser.add_argument("--seed", type=int, default=123)

parser.add_argument("--results-path", type=str, default=None)
```

```
parser.add_argument("--epochs", type=int, default=1_000)
```

除去 `input-size` 和 `results-path` 这两个参数,下面我将逐一分析各个超参数的影响,最终给出每个任务合适的超参数配置:

1. `N` : 数据集大小,
2. `hidden-size` : 第一层隐藏层的features数量,决定模型的宽度,宽度不够时容易拟合出一条直线,当复杂度太大时会出现过拟合
3. `n-muti-layers` : 隐藏层features增长层的数量,决定模型的深度,深度不够时容易拟合出一条直线,当复杂度太大时会出现过拟合
4. `act-fn` : 激活函数类型,当层数较深时使用Sigmoid这种函数会出现梯度爆炸/消失等情况,目前来看ReLU更好
5. `batch-size` : 批量大小,当N=200时我们把整个数据集作为一个batch,这样使得训练比较稳定,当数据集较大时我们选择batch-size=64,节省显存
6. `lr` : 由于我们选用了较大的batch,我们同时使用较大的学习率2e-4,这样模型收敛较快,并且采用Adam算法自适应调整学习率大小
7. `seed` : 用于划分数据集和复现结果,影响不是很大
8. `epoch` : 数据集较小时需要更多的epoch,数据集较大时可以用更少的epoch达到比较不错的效果

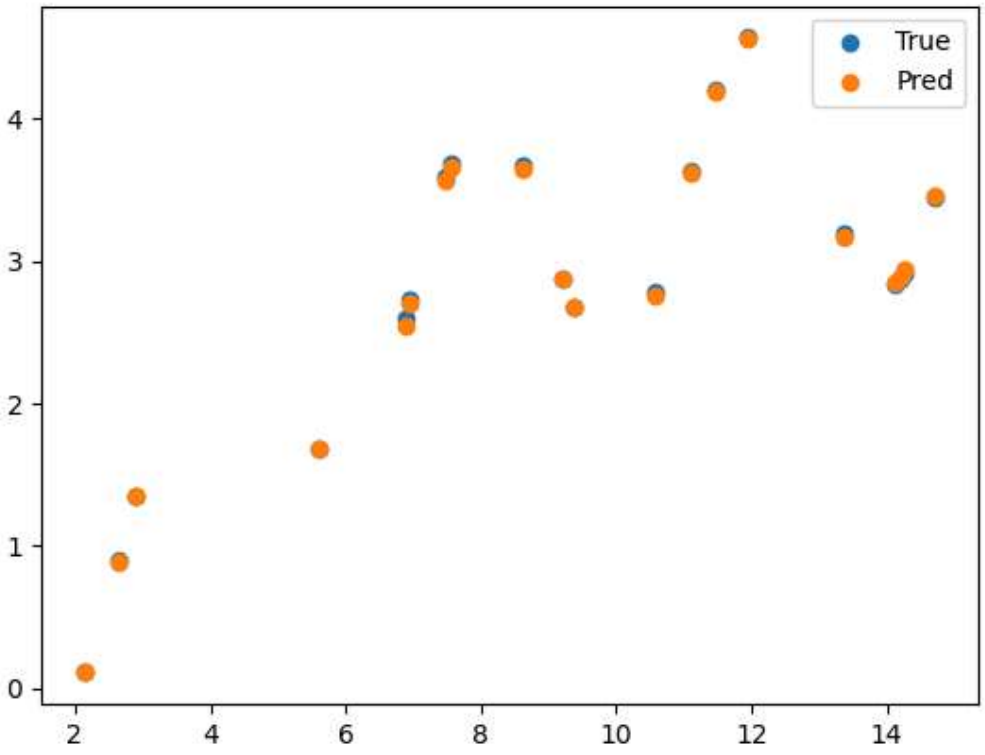
下面给出我实验后给每个N值合适的超参数配置:

N=200

```
N: 200
act_fn: relu
batch_size: 200
epochs: 5000
hidden_size: 128
input_size: 1
lr: 0.0002
n_muti_layers: 2
results_path: results/test6/
seed: 123
```

$MSE = 0.00047427104436792433$

效果图:

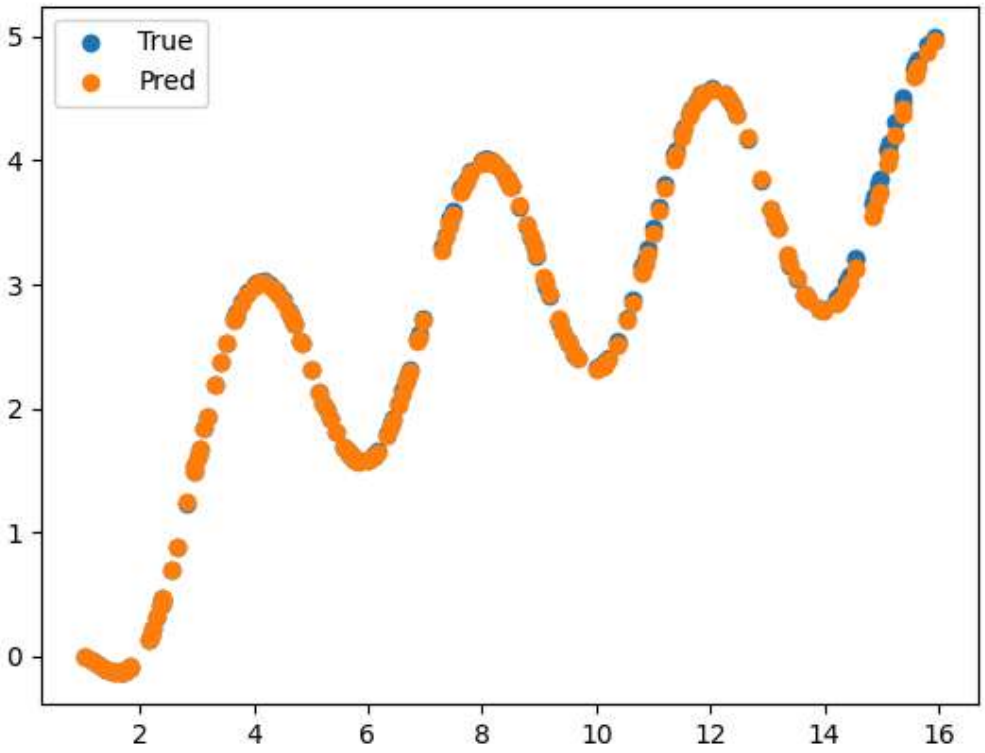


N=2000

```
N: 2000
act_fn: relu
batch_size: 128
epochs: 5000
hidden_size: 128
input_size: 1
lr: 0.0002
n_muti_layers: 3
results_path: results/test5/
seed: 123
```

$MSE = 0.0009722792601678 52$

效果图:

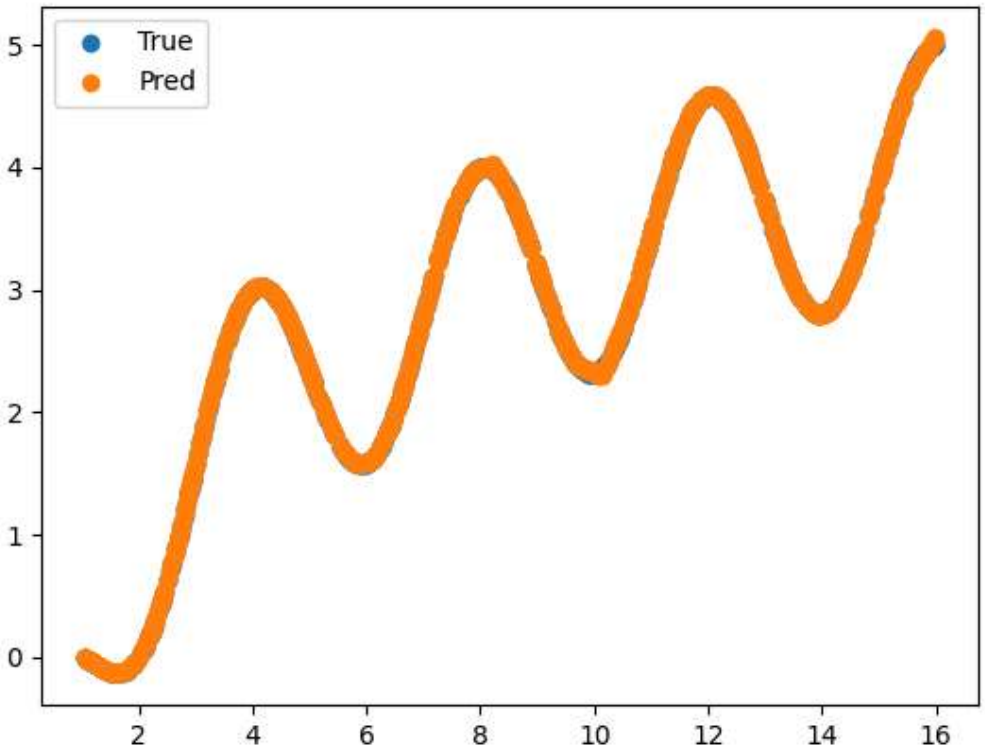


N=10000

```
N: 10000
act_fn: relu
batch_size: 64
epochs: 1000
hidden_size: 256
input_size: 1
lr: 2.0e-06
n_muti_layers: 2
results_path: results/test4/
seed: 123
```

$MSE = 7.7236501169863 91e - 05$

效果图:


True
Pred
5
4
3
2
1
0
2
4
6
8
10
12
14
16