

LAB1-实验报告

SA22011035 李钊佚

目录:

- 实验过程陈述
 - 关键代码展示
 - 测试结果和超参数分析
-

正文:

➤ 实验过程陈述:

本次实验研究使用简单前馈神经网络近似一个初等函数： $y = \sin x + e^{-x}$, $x \in [0, 4\pi]$ ，按照实验指导书的指引，我在 `pytorch` 环境下进行了本次实验。我的代码主要分为：数据集的产生与划分、模型的构建、模型训练部分代码、模型性能测试部分代码。经过调试，我的代码可以顺利运行，其执行逻辑流程如下：

【数据集产生和划分：随机且独立地从 $[0, 4\pi]$ 中采样出若干个点，并将其随机划分为 `training \ validation \ testing` 集合】 ->

【模型的构建：使用 `pytorch` 构建简单的前馈神经网络，其中利用 `nn.ModuleList()` 来动态定义前馈神经网络中的隐藏层数】 ->

【模型训练：把数据按照 `batch` 送入定义的前馈神经网络中，进行 `forward` 和 `backward` 过程，更新参数】 ->

【模型性能测试：每个 `epoch` 结束时，记录 `epoch` 中平均的 `batch_loss`，和当前模型在 `validation set` 上面的 `loss`，当模型训练

结束时，根据 validation loss 计算模型在 testing set 上的 loss】

➤ 关键代码展示：

【数据集产生和划分】

training set:

```
train_size = 4000
x=np.linspace(0,4*np.pi,train_size)
print(x[1])
y=np.sin(x)+np.exp(-x)

X=np.expand_dims(x,axis=1)
Y=y.reshape(train_size,-1)
```

validation set:

```
val_size = 500
x_val=np.linspace(0,4*np.pi,val_size)
print(x_val[1])
y_val=np.sin(x_val)+np.exp(-x_val)

X_val=np.expand_dims(x_val,axis=1)
Y_val=y_val.reshape(val_size,-1)
```

testing set:

```
test_size = 128
x_test=np.linspace(0,4*np.pi,test_size)
print(x_test[1])
y_test=np.sin(x_test)+np.exp(-x_test)
X_test=np.expand_dims(x_test,axis=1)
Y_test=y_test.reshape(test_size,-1)
```

【模型的构建】

```
class Net(nn.Module):
```

```

def __init__(self):
    super(Net, self).__init__()
    self.inp_layer = nn.Linear(1, args.width)
    self.hiddens = nn.ModuleList()
    '''
    to make parameter in the net.parameters()
    '''
    for i in range(args.depth-1):
        self.hiddens.append(nn.Linear(args.width,
            args.width).cuda())
    self.out_layer = nn.Linear(args.width, 1)
    if args.actfunc == 'relu':
        self.activate = F.relu
    elif args.actfunc == 'tanh':
        self.activate = torch.tanh
    elif args.actfunc == 'sigmoid':
        self.activate = torch.sigmoid
    else:
        raise Exception("ERROR: parameter `actfunc` is invalid!")

    def forward(self, inp):
        x = self.activate(self.inp_layer(inp))
        for hid_layer in self.hiddens:
            x = self.activate(hid_layer(x))
        x = self.out_layer(x)
        return x

```

【模型训练】

```

epoch_num = 500
train_losses = list()
val_losses = list()
steps = list()

for epoch in range(epoch_num):
    loss=None

```

```
loss_epoch = 0
cnt=0
for batch_x,batch_y in dataloader:
    net.train()
    #print('flag_')
    y_predict=net(batch_x)
    loss=Loss(y_predict,batch_y)
    #print('flag__')
    optim.zero_grad()
    loss.backward()
    optim.step()
    loss_epoch += loss.item()
    cnt += 1
loss_epoch /= cnt
```

【模型性能测试】

validation loss :

```
if (epoch+1)%10==0:
    print("step: {0} , loss: {1}".format(epoch+1,loss.item()))
    steps.append(epoch+1)
    train_losses.append(loss_epoch)

    net.eval()
    with torch.no_grad():
        y_val_pred =
        net(torch.tensor(X_val,dtype=torch.float).cuda())
        val_loss =
        Loss(torch.tensor(Y_val,dtype=torch.float).cuda(),
        y_val_pred)
        print(val_loss.item())
        val_losses.append(val_loss.item())
```

```
# testing loss
```

```
y_test_pred=net(torch.tensor(X_test,dtype=torch.float).cuda())
test_loss =
Loss(torch.tensor(Y_test,dtype=torch.float).cuda(),
y_test_pred).item()
```

➤ 测试结果和超参数分析:

我们在如下空间搜索超参数:

【learning rate = 1e-2, 5e-3, 1e-3, 5e-4】

【activation function = relu, tanh, sigmoid】

关于网络宽度和网络深度的影响, 为了公平比较, 我们设置隐藏层神经元的数量一致: 即隐藏层深度与隐藏层宽度的乘积约为 40

【width = 20, depth = 2;

width = 13, depth = 3;

width = 10, depth = 4】

测试结果:

@1: width = 20, depth = 2:

testing loss	1e-2	5e-3	1e-3	5e-4
relu	0.0003	0.001	0.0009	0.0004
tanh	0.0003	4e-5	7e-5	7e-5
sigmoid	0.0001	7e-5	2e-5	0.0507

@2: width = 13, depth = 3:

testing loss	1e-2	5e-3	1e-3	5e-4
relu	0.0006	0.0002	0.001	0.0017
tanh	0.0002	2e-5	2e-5	1e-5
sigmoid	0.0176	0.0175	8e-6	0.0012

@3: width = 10, depth = 4:

testing loss	1e-2	5e-3	1e-3	5e-4
relu	0.0003	0.0002	0.0005	0.0006
tanh	4e-5	0.0004	6e-6	0.0001
sigmoid	6e-5	0.0173	0.2498	0.02

【超参数分析】

诚实地说，实验结果并没有表现出对某组超参数有明显的倾向规律。

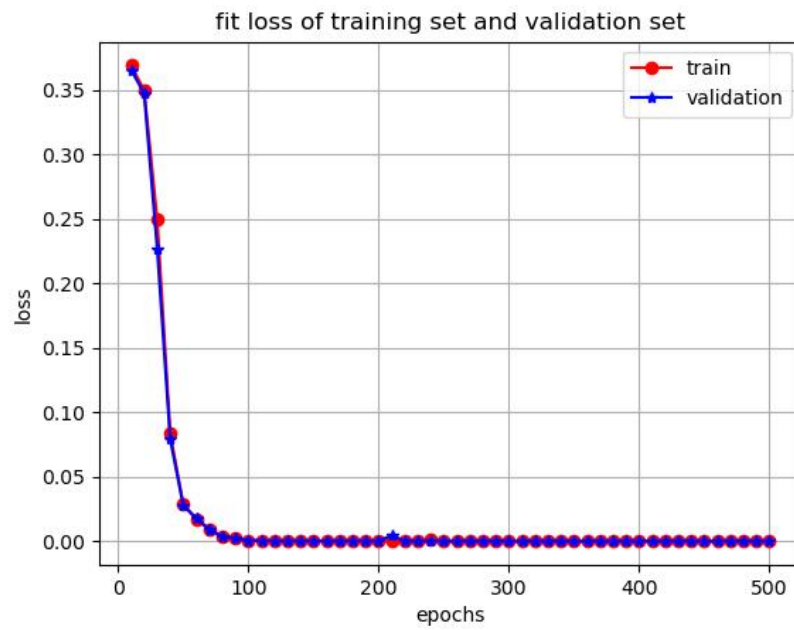
但是观察实验结果我们可以得到一些显然的结论：

@1:在网络结构固定时：学习率和激活函数都有可能对模型的训练产生巨大的影响。

@2:综合来看，对于该回归任务，选用 **tanh** 激活函数得到的在测试集合上的拟合性能（MSE loss）几乎是最优的（相比较于 **relu** 和 **sigmoid**），**relu** 激活函数的性能次之，但是相对来讲也比较稳定，但是 **sigmoid** 函数的性能不是很稳定，经常熟练到局部最优处，具体可以观察下面给出的【width=10, depth=4; lr=1e-3; sigmoid】和

【width=10, depth=4; lr=1e-3; tanh】的训练曲线对比:

@1:tanh



@2:sigmoid

