Neural Network and Applications
大作业一

陈轶洲 MF20330010

November 17, 2020

1 设计方案

本章将从两个方面介绍感知器神经网络的设计方案：

1) 网络结构设计：定义网络层数，各层激活函数，输入输出及各隐藏层规模，损失函数类型；

2) 参数设计：定义训练数据集，以及训练轮次、学习率等超参数。

1.1 网络结构设计

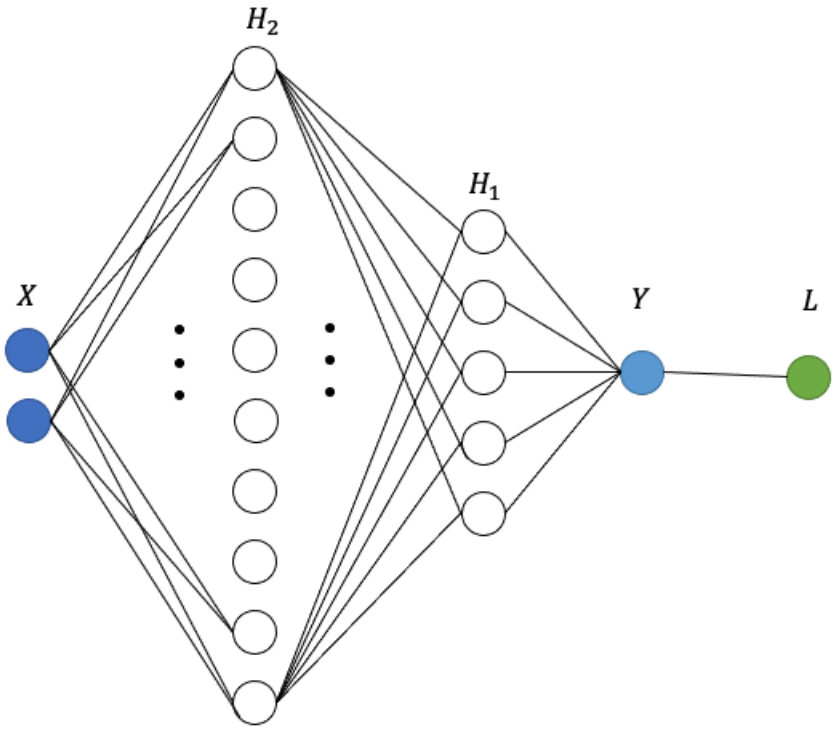

Figure 1: 神经网络结构

1) 网络层数：本网络除输入输出层之外，包含两个隐藏层 H_1, H_2；

2) 激活函数：通过实验对比 Sigmoid、Relu、Tanh 函数，Tanh 激活函数取得了更好的效果，故模型选择 Tanh 作为激活函数；

3) 神经元规模：由于待拟合函数为二元非线性函数，故输入层神经元个数为 2，隐藏层 H_1 包含 10 个神经元，隐藏层 H_2 包含 5 个神经元，输出层为 1 个神经元；

4) 损失函数：本模型选择均方误差作为损失函数，其计算方法如下：

$$LOSS = \sum_{i=1}^{n} (y_i - t_i)^2 \tag{1.1}$$

1.2 参数设计

1) 训练数据集：在 [-5,5] 的区间中以 0.1 的步长取点，将取得的 100 个点作为二维矩阵的横纵坐标，从而生成 10000 个数据点，以这 10000 个点作为训练数据集；

2) 迭代次数 iters_num：5000000

3) 批处理 batch_size：100

4) 学习率 learning_rate：0.0001

5) 轮次 epoch：50000

2 编程实现方案

随报告提交的代码文件中包含 functions.py、layers.py、neuralnet.py、main.py，分别对应本章将介绍的神经网络层、神经网络结构、模型训练三个层面的设计，下面将自底向上地介绍编程实现方案。

2.1 神经网络层设计

本节将介绍神经网络中各隐藏层的设计，包括其前向传播与反向传播的实现方法。（具体实现见 layers.py）

2.1.1 全连接层

图 2给出了全连接层前向传播与反向传播的推导公式：
前向传播：$Y = W^T X + B$
反向传播：$\frac{\partial L}{\partial X} = W\frac{\partial L}{\partial Y}$，$\frac{\partial L}{\partial W} = X(\frac{\partial L}{\partial Y})^T$，$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial Y}$

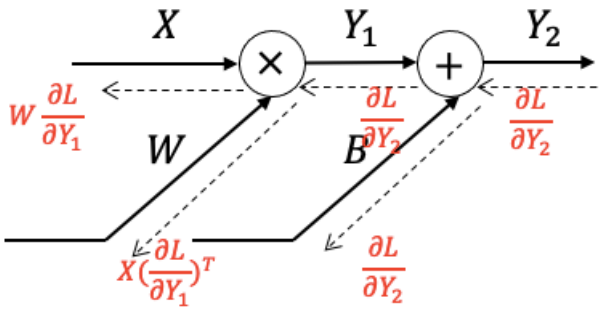

Figure 2: Fully Connected

编程实现类 $FC()$，其包含 $forward(), backward()$ 两种方法，分别实现对前向传播与反向传播的计算：

```
32  class FC:
33      def __init__(self, W, B):
34          self.W = W
35          self.B = B
36
37          self.x = None
38          self.dW = None
39          self.dB = None
40
41      def forward(self, x):
42          self.x = x
43          out = self.W.T.dot(self.x)+self.B
44          return out
45
46      def backward(self, dout):
47          dx = self.W.dot(dout)
48          self.dW = self.x.dot(dout.T)
49          self.dB = np.sum(dout, axis=1, keepdims=True)
50          return dx
```

Figure 3: fully connection layer

2.1.2 Relu 层

relu 层的作用是将输入矩阵中的非正元素置为零，所以在前向传播中使用矩阵 mask 记录下所有非正元素位置，并将相应位置元素置为 0。在反向传播中，再次利用 mask 矩阵，将传播来的偏导矩阵中的对应元素置为零。
下图是通过编程实现的 $Relu()$ 类，其包含 $forward(), backward()$ 两种方法：

```python
class Relu:
    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = (x<=0)
        out = x.copy()
        out[self.mask] = 0
        return out

    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout
        return dx
```

Figure 4: relu layer

2.1.3 Tanh 层

tanh 激活函数的计算与求导如下所示:

$$\begin{aligned} tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\ tanh'(x) &= 1 - tanh^2(x) \end{aligned} \tag{2.1}$$

通过上式实现类 $Tanh()$:

```
19  class Tanh:
20      def __init__(self):
21          self.out = None
22
23      def forward(self, x):
24          out = tanh(x)
25          self.out = out
26          return out
27
28      def backward(self, dout):
29          dx = dout*(1-self.out**2)
30          return dx
```

Figure 5: tanh layer

2.1.4 Loss 层

本网络采用均方误差作为损失函数，故损失函数层的计算与求导如下所示：

$$\begin{aligned} loss(y) &= (y-t)^2 \\ loss'(y) &= 2(y-t) \end{aligned} \tag{2.2}$$

通过上式实现 $Loss()$ 类：

```python
class Loss:
    def __init__(self):
        self.y = None
        self.t = None

    def forward(self, y, t):
        self.y = y
        self.t = t
        loss = (self.y-self.t)**2
        return loss

    def backward(self, dout):
        dy = dout*2*(self.y - self.t)
        return dy
```

Figure 6: loss layer

2.2 神经网络结构

通过上节实现了各隐藏层的功能，本节将介绍如何通过已有的隐藏层完成神经网络类 $ThreeLayerNet()$ 的实现。（具体实现见 neuralnet.py）

2.2.1 初始化

功能：对神经网络模型必要的参数，如隐藏层、激活函数等进行初始化。
实现方法：首先定义字典 params 用来保存全连接层中的权重矩阵与偏置，接着定义字典 layers 保存神经网络需要的各层。为了方便后续反向传播的计算，将 layers 定义为 OrderdDict()：

```python
def __init__(self, input_size, hidden_size1, hidden_size2, output_size, weight_init_std = 0.01):
    # 初始化权重
    self.params = {}

    self.params['W1'] = weight_init_std*np.random.randn(input_size, hidden_size1)
    self.params['B1'] = np.zeros((hidden_size1, 1))
    self.params['W2'] = weight_init_std*np.random.randn(hidden_size1, hidden_size2)
    self.params['B2'] = np.zeros((hidden_size2, 1))
    self.params['W3'] = weight_init_std*np.random.randn(hidden_size2, output_size)
    self.params['B3'] = np.zeros((output_size, 1))

    # 初始化隐藏层
    self.layers = OrderedDict()

    self.layers['fc1'] = FC(self.params['W1'], self.params['B1'])
    self.layers['relu1'] = Relu()
    self.layers['fc2'] = FC(self.params['W2'], self.params['B2'])
    self.layers['relu2'] = Relu()
    self.layers['fc3'] = FC(self.params['W3'], self.params['B3'])

    self.lastlayer = Loss()
```

Figure 7: inital

2.2.2 predict 方法

功能：对输入数据进行预测并返回预测结果。
实现方法：得到输入数据后，依次调用各层的 forward() 方法，同时将输出作为下一层的输入，依次传递：

```python
def predict(self, x):
    for layer in self.layers.values():
        x = layer.forward(x)
    return x
```

Figure 8: predict

2.2.3 loss 方法

功能：计算模型当前的误差。
实现方法将中间输出输入到 Loss() 层中即可计算出模型的误差：

```python
def loss(self, x, t):
    y = self.predict(x)
    return self.lastlayer.forward(y, t)
```

Figure 9: loss

2.2.4 gradient 方法

功能：计算反向传播时损失函数对各隐藏层参数的偏导并返回。
实现方法：首先调用 loss() 方法求出模型误差，接着利用字典 layers 的有序性，使用 reverse() 方法将字典中各层的顺序反转，然后利用各层的 backward() 方法将 dout 传递下去。最后利用字典 grads 将各层的偏导保存并返回：

```
39      def gradient(self, x, t):
40          # 前向传播
41          self.loss(x, t)
42
43          # 反向传播
44          dout = 1
45          dout = self.lastlayer.backward(dout)
46
47          layers = list(self.layers.values())
48          layers.reverse()
49          for layer in layers:
50              dout = layer.backward(dout)
51
52          # 保存偏导
53          grads = {}
54          grads['W1'], grads['B1'] = self.layers['fc1'].dW, self.layers['fc1'].dB
55          grads['W2'], grads['B2'] = self.layers['fc2'].dW, self.layers['fc2'].dB
56          grads['W3'], grads['B3'] = self.layers['fc3'].dW, self.layers['fc3'].dB
57
58          return grads
```

Figure 10: gradient

2.3 模型训练

本节将介绍如何训练模型。(具体实现见 main.py)
利用 batch 处理方法，每次从数据集中随机选择 100 个样本作为神经网络的输入，接着计算其输出与标签的误差，并将误差进行反向传播。在每个 epoch 完成后后记录模型的损失加入到列表 train_loss_list 中，直至训练结束。

```python
    # 模型训练
    for i in range(iters_num):
        batch_mask = np.random.choice(train_size, batch_size)
        x_batch = x_train[:, batch_mask]
        y_batch = y_label[batch_mask]

        grad = network.gradient(x_batch, y_batch)

        for key in (network.params.keys()):
            network.params[key] -= learning_rate*grad[key]

        loss = network.loss(x_train, y_label)
        # print(loss)
        mean_loss = np.sum(loss, axis=1) / loss.shape[1]
        print("The {} iteration'loss:{}".format(i, mean_loss))
        train_loss_list.append(mean_loss)
```

Figure 11: train

3 实验结果展示

利用训练数据完成 50000 个 epoch 的训练后，模型的误差如下：

```
No.49980 epoches' loss:[0.00027916]
No.49981 epoches' loss:[7.35387338e-05]
No.49982 epoches' loss:[7.03023774e-05]
No.49983 epoches' loss:[6.75871167e-05]
No.49984 epoches' loss:[6.45597286e-05]
No.49985 epoches' loss:[7.05153945e-05]
No.49986 epoches' loss:[6.79537965e-05]
No.49987 epoches' loss:[6.643333e-05]
No.49988 epoches' loss:[6.69514185e-05]
No.49989 epoches' loss:[8.93737563e-05]
No.49990 epoches' loss:[6.50609537e-05]
No.49991 epoches' loss:[6.63429531e-05]
No.49992 epoches' loss:[6.89290751e-05]
No.49993 epoches' loss:[6.623437e-05]
No.49994 epoches' loss:[7.89120252e-05]
No.49995 epoches' loss:[6.81798675e-05]
No.49996 epoches' loss:[6.50050452e-05]
No.49997 epoches' loss:[6.73683397e-05]
No.49998 epoches' loss:[6.49899293e-05]
No.49999 epoches' loss:[6.70696084e-05]
No.50000 epoches' loss:[7.27270229e-05]
(base) ➜  code git:(master) ✗
```

Figure 12: result

训练过程中的损失函数记录如下，模型已训练至收敛：

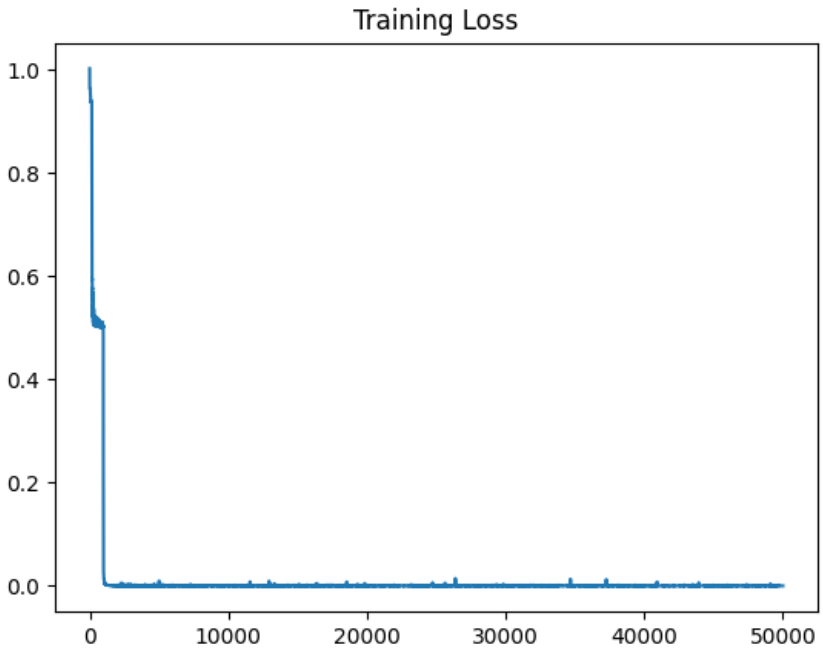

Figure 13: train loss list

利用三维图像显示模型训练前与训练后的拟合差别：

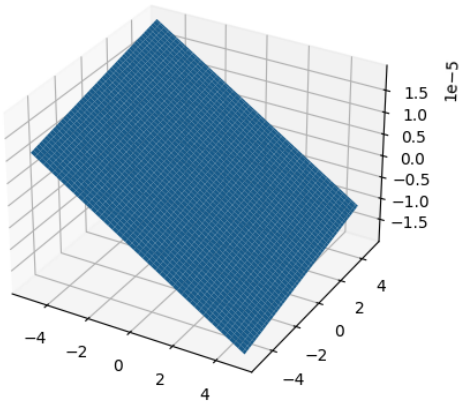

Figure 14: before training

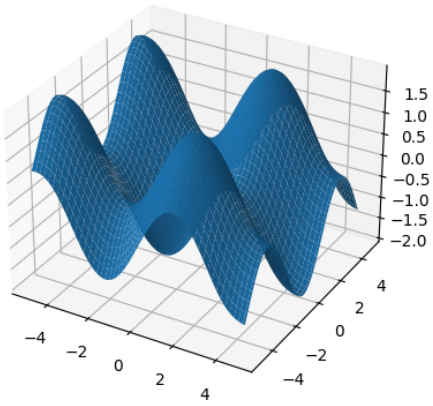

Figure 15: after training

4 分析与总结

通过上一章的实验结果展示，该神经网络已经具备了拟合二元非线性函数 $y = sin(x_1) - cos(x_2)$ 的能力。其之所以能够拟合非线性曲线，原因在于：

1) 单神经元只具有拟合线性函数的功能，单层中的多个神经元拥有拟合不同线性函数的功能。延伸至多隐藏层的神经网络，将各线性函数的加权和进行反复迭代可以达到拟合非线性函数的效果；

2) 仅利用权重矩阵与偏置，拟合效果往往并不理想，所以需要利用激活函数。本神经网络模型采用的激活函数为 tanh，它具有整流的作用，能够更好地提取函数非线性部分的特征，弱化其它不重要的特征。并且 tanh 的值域与待拟合函数的值域相同，均为 $[-2, 2]$，进一步提升了拟合效果，这一优势是 relu、sigmoid 激活函数所不具备的。

5 参考资料

[1]https://zhuanlan.zhihu.com/p/47519999