清华大学电子工程系
**媒体与认知** 课堂 2
2021-2022 学年春季学期

**<u>作业 1</u>**

孙一 2019010976 2022 年 3 月 6 日

# 理论部分

## 1 单选题（15 分）

### 1.1 <u>B</u>

### 1.2 <u>C</u>

### 1.3 <u>A</u>

### 1.4 <u>B</u>

### 1.5 <u>B</u>

## 2 计算题（15 分）

| $x_1$ | $x_2$ | $y$ |
|---|---|---|
| 0 | 0 | **0** |
| 0 | 1 | **0** |
| 1 | 0 | **0** |
| 1 | 1 | 1 |

AND

| $x_1$ | $x_2$ | $y$ |
|---|---|---|
| 0 | 0 | **0** |
| 0 | 1 | **1** |
| 1 | 0 | **1** |
| 1 | 1 | 1 |

OR

| $x_1$ | $x_2$ | $y$ |
|---|---|---|
| 0 | 0 | **0** |
| 0 | 1 | **1** |
| 1 | 0 | **1** |
| 1 | 1 | **0** |

异或

图 1: AND,OR, 异或三种逻辑运算

### 2.1 基于如下单个人工神经元，设计实现两种逻辑门 AND、OR 运算。

$$z = w_1x_1 + w_2x_2 + b \tag{1}$$

$$y = f(z) = \begin{cases} 1, z > 0 \\ 0, z \leq 0 \end{cases} \tag{2}$$

解:

- AND: 令 $w_1 = 1$, $w_2 = 1$, $b = -1.1$，则只有 $x_1$, $x_2$ 都取 1 时，$z$ 才大于 0，$y = f(z) = 1$，其余情况，$z$ 均小于 0，$y = 0$
- OR: 令 $w_1 = 1$, $w_2 = 1$, $b = -0.1$，则只有 $x_1$, $x_2$ 都取 0 时，$z$ 才小于等于 0，$y = f(z) = 0$，其余情况，$z$ 均大于 0，$y = 1$

### 2.2 上述形式的单个神经元是否可以实现逻辑门异或运算？如果是，请给出具体设计；若否，请解释理由。

解:

不可以用上述形式的单个神经元实现异或运算。

因为 $z = w_1x_1 + w_2x_2 + b$ 为线性分类模型，其本质是在以 $x_1$，$x_2$ 为横纵坐标的二维平面上确定分界面，从而把样本分为两类（$z > 0$ 和 $z \leq 0$）。如果把异或运算对应的 $(x_1, x_2)$ 关系表示在二维平面上，如图 2所示，二维平面中不存在将所有样本分为两类的一条直线，即为线性不可分情形，因此不可以用单个神经元实现逻辑异或。

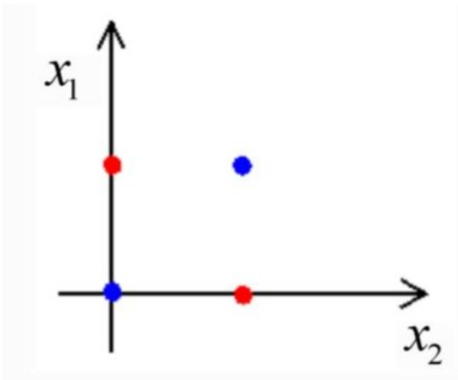



图 2: 异或运算对应的 $(x_1, x_2)$ 关系

# 编程部分

# 3 编程作业报告

## 3.1 导入模型依赖库

这部分没有修改

```
# ==== Part 0: import libs
from importlib_metadata import requires
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import matplotlib.pyplot as plt
from pylab import *
import sys
```

## 3.2 定义数据类模块

自定义的 MyDataset 需要继承 torch.utils.data.Dataset 类，并重载函数 ___init___(), ___len___() 和 ___getitem___()。
注意两点:

1. $data$ 的形状是 $shape(N, 3)$，共有 $N$ 行，每一行前两个值是特征值，第三个值是标签

2. ___*getitem*___() 每次返回 $index$ 指示的一条数据及其 $label$，$tensor$ 索引只传一个参数 $index$ 时指的是其第 $index + 1$ 行的数据

```
# ==== Part 1: definition of dataset
class MyDataset(Dataset):
    def __init__(self, file_path):
        # TODO: load all items of the dataset stored in file_path
        # you may need np.load() function
        self.data = np.load(file_path)
        self.feats = self.data[:,:2].astype(np.float64)
        self.labels = self.data[:,2].astype(np.float64)

    def __len__(self):
        # TODO: get the number of items in the dataset
```

```
        return self.feats.shape[0]

    def __getitem__(self, index):
        # TODO: get the feature and label of the current item
        # pay attention to the type of the outputs
        # index 表示哪一行的数据
        assert index <= len(self), 'index range error'
        feat = self.feats[index]
        label = self.labels[index]
        return feat, label
```

### 3.3 定义模型结构

自定义线性层，需要注意三点：

1. 参数必须用 $nn.Parameter()$ 来定义，否则无法像 $nn.Linear$ 等层参数一样被 $Model.parameters()$ 返回，也无法通过 torch.save() 保存；
2. 注意权重矩阵的形状，这里采用 $z = \mathbf{x}w + b$ 的格式，输入 $\mathbf{x}$ 的形状是 $(batch\_size, 2)$，因此 $w$ 的形状是 $(2, 1)$；
3. 输出值只有一维，因此需要转变形状。

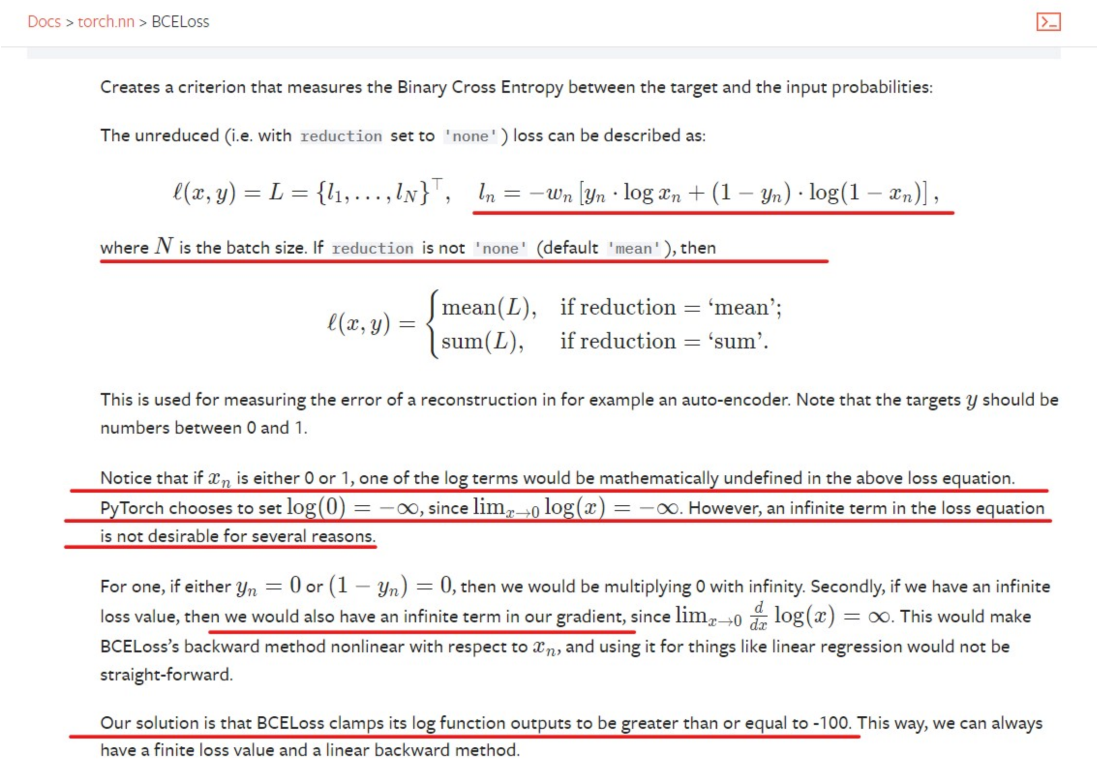
```
# ==== Part 2: network structure
# -- linear classifier
class Model(nn.Module):
    def __init__(self, input_size, output_size):
        '''
        :param input_size: dimension of input features: 2
        :param output_size: 1
        '''
        # TODO: initialization of the linear classifier
        # the model should include a linear layer and a Sigmoid layer
        # remember to initialize the father class and pay attention to the dimension
        super(Model,self).__init__()
        # 用 nn.Parameter(data=None, requires_grad=True) 来定义模型参数
        # 初始化为权重和偏置为 (0,1) 的随机数，并且开启梯度跟踪
        self.weights = nn.Parameter(torch.rand((2, 1), dtype = torch.double))
        self.bias = nn.Parameter(torch.rand((1), dtype=torch.double))
```

```python
        self.act = nn.Sigmoid()

    def forward(self, x):
        '''
        :param x: input features of shape (batch_size, 2)
        :return pred: output of model of shape (batch_size, )
        '''
        # TODO: forward the model
        '''pred should be shape of (batch_size, ) rather than (batch_size, 1)'''
        out = self.act(torch.mm(x, self.weights) + self.bias)
        return out.view(-1) # 输出只有一维
```

## 3.4 定义损失函数

*bce_loss*() 函数的书写参考了 *pytorch nn.BCELoss*() 文档，如图 3，可见 *BCELoss*() 输出默认是 *mean*(一个数)，即批次内样本 *loss* 的均值；此外，为了解决 *log*0 导致的数值问题，需要把对数值钳位到 $(-100, \infty)$


Docs > torch.nn > BCELoss

Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n\left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)\right],$$

where $N$ is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{`mean'}; \\ \operatorname{sum}(L), & \text{if reduction} = \text{`sum'}. \end{cases}$$

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets $y$ should be numbers between 0 and 1.

Notice that if $x_n$ is either 0 or 1, one of the log terms would be mathematically undefined in the above loss equation. PyTorch chooses to set $\log(0) = -\infty$, since $\lim_{x \to 0} \log(x) = -\infty$. However, an infinite term in the loss equation is not desirable for several reasons.

For one, if either $y_n = 0$ or $(1 - y_n) = 0$, then we would be multiplying 0 with infinity. Secondly, if we have an infinite loss value, then we would also have an infinite term in our gradient, since $\lim_{x \to 0} \frac{d}{dx} \log(x) = \infty$. This would make BCELoss's backward method nonlinear with respect to $x_n$, and using it for things like linear regression would not be straight-forward.

Our solution is that BCELoss clamps its log function outputs to be greater than or equal to -100. This way, we can always have a finite loss value and a linear backward method.

图 3: nn.BCELoss() 文档说明

```
# ==== Part 3: define binary cross entropy loss for classification task
def bce_loss(pred, label):
    '''
    Binary cross entropy loss function
    ------------------------------
    :param pred: predictions with size [batch_size, *], * is the dimension of data
    :param label: labels with size [batch_size, *]
    :return: loss value, divided by the number of elements in the output
    '''
    # TODO: calculate the mean of losses for the samples in the batch
    # you should not use the nn.BCELoss class to implement the loss function
    # 钳位到 (-100,+∞)
    log_q = torch.clamp(torch.log(pred),-100)
    log_1_q = torch.clamp(torch.log(1-pred),-100)
    los = torch.mean(-(label* log_q + (1-label)*log_1_q))
    return los
    # 注意变为平均数，这样才和 nn.BCELoss() 的返回值对应
    # pytorch 自带的优化器默认不除以 batch_size，只负责迭代参数。
```

在运行阶段，发现自定义的交叉熵损失函数容易出现 $loss = nan$ 的情况，通过调试，发现减小学习率和增大数据精度都可以解决这一问题，因此本实验中数据的精度使用 $torch.double$。

## 3.5 训练-验证代码

这一部分需要注意的是每次反向传播之前先要清空优化器中的梯度，同时注意验证时准确率的计算方法。

```
# ==== Part 4: training and validation
def train_val(train_file_path='data/character_classification/train_feat.npy',
              val_file_path='data/character_classification/val_feat.npy',
    n_epochs=20, batch_size=8, lr=1e-3, momentum=0.9, valInterval=5, device='cpu'):
    '''
    The main training procedure
    ----------------------------
    :param train_file_path: file path of the training data
    :param val_file_path: file path of the validation data
    :param n_epochs: number of training epochs
```

```
    :param batch_size: batch size of training and validation
    :param lr: learning rate
    :param momentum: momentum of the stochastic gradient descent optimizer
    :param valInterval: the frequency of validation, valInterval = 5
    do validation after each 5 training epochs
    :param device: 'cpu' or 'cuda', we can use 'cpu' for our homework
    if GPU with cuda support is not available
    '''

    # TODO: instantiate training and validation data loaders
    trainset = MyDataset(train_file_path)
    valset = MyDataset(val_file_path)
    trainloader = DataLoader(trainset, batch_size=batch_size, shuffle=True)
    valloader = DataLoader(valset, batch_size=batch_size, shuffle=True)

    # TODO: instantiate the linear classifier
    # you can change the device if you have a gpu
    model = Model(input_size=2, output_size=1)  # 形状其实没啥用
    model = model.to(device)

    # TODO: instantiate the SGD optimizer
    optimizer = optim.SGD(model.parameters(), lr, momentum)
    # to save loss of each training epoch in a python "list" data structure
    losses = []

    # now everything is prepared for training the model!
    for epoch in range(n_epochs):
        total_loss = 0.0
        # ===== training stage =====
        # TODO: set the model in training mode
        model.train()
        # TODO: train the model for one epoch, which may include data loading, model
        # pay attention to clear the gradient before the back propagation
        '''step 不一定有用，所以不用时可以直接迭代 loader，不需要枚举'''
        for step, (feats, labels) in enumerate(trainloader):
            # set data type and device
```

```
            feats, labels = feats.to(device), labels.to(device)
            # call a function to clear gradients in the optimizer
            optimizer.zero_grad()
            #run the model which is the forward process
            out = model(feats)
            # compute the binary cross entropy loss,
            # and call backward propgation function
            loss = bce_loss(out, labels)
            #print(loss.requires_grad)
            loss.backward()
            # sum up of total loss, loss.item() return the value of the tensor
            # as a standard python number,this operation is not differentiable
            total_loss += loss.item()
            # call a function to update
            optimizer.step()
        # TODO: calculate average of the total loss for iterations
        # and store it in losses
        average_loss = total_loss / len(trainloader)
        losses.append(average_loss)
        print('epoch {:02d}: loss = {:.3f}'.format(epoch + 1, average_loss))

        # ===== validation stage =====
        # validate the model every valInterval epochs
        if (epoch + 1) % valInterval == 0:
            # TODO: set the model in evaluation mode
            model.eval()    '''必须进入评估模式! '''
            n_correct = 0
            n_ims = 0
            # TODO: evaluate the model on the validation set, which may
            # include data loading, model forwarding and accuracy calculating
            # remember to use torch.no_grad() because we do not need to
            # compute gradients during validation
            with torch.no_grad():
                for feats, labels in valloader:
                    feats = feats.to(device)
                    labels = labels.to(device)
```

```
                out = model(feats)
                predictions = torch.round(out)  # 四舍五入到 0 或 1
                n_correct += torch.sum((predictions == labels).float())
                # 找出和标签匹配的预测值的数量
                n_ims += feats.size(0)  # 增加一个批次的大小
        print('Epoch {:02d}: validation accurancy = {:.1f}
        \%'.format(epoch + 1, 100*n_correct/n_ims))

        # TODO: save model parameters in model_save_path
        # 在某些 epoch 时保存模型参数
        model_save_path = 'saved_models/model_epoch{:02d}.pth'.format(epoch + 1)
        torch.save({'state_dict': model.state_dict()}, model_save_path)
        print('model saved in {}\n'.format(model_save_path))
    # draw the loss curve
    plot_loss(losses)   # 绘制所有 epoch 的误差构成的曲线
```

### 3.6 测试代码

这一部分没有修改

### 3.7 结果可视化

纠正了第二幅子图的图例名称，同时为了和之前定义的数据精度匹配，把下面的数据类型改成了 *torch.double*()

```
    # get weights and bias of the linear layer
    input_a = torch.tensor([1.0, 0.0]).double().view(1, 2)
    input_b = torch.tensor([0.0, 1.0]).double().view(1, 2)
    input_zeros = torch.tensor([0.0, 0.0]).double().view(1, 2)
```

### 3.8 运行入口

这一部分没有修改

### 3.9 运行结果

执行 python classification.py train 的结果如下：

```
epoch 01: loss = 0.462
epoch 02: loss = 0.313
```

```
epoch 03: loss = 0.281
epoch 04: loss = 0.268
epoch 05: loss = 0.261
Epoch 05: validation accurancy = 92.8%
model saved in saved_models/model_epoch05.pth

epoch 06: loss = 0.257
epoch 07: loss = 0.255
epoch 08: loss = 0.253
epoch 09: loss = 0.252
epoch 10: loss = 0.252
Epoch 10: validation accurancy = 92.4%
model saved in saved_models/model_epoch10.pth

epoch 11: loss = 0.251
epoch 12: loss = 0.250
epoch 13: loss = 0.250
epoch 14: loss = 0.250
epoch 15: loss = 0.250
Epoch 15: validation accurancy = 92.8%
model saved in saved_models/model_epoch15.pth

epoch 16: loss = 0.250
epoch 17: loss = 0.249
epoch 18: loss = 0.249
epoch 19: loss = 0.249
epoch 20: loss = 0.249
Epoch 20: validation accurancy = 92.6%
model saved in saved_models/model_epoch20.pth
```

loss 随训练 epoch 的变化如图 4

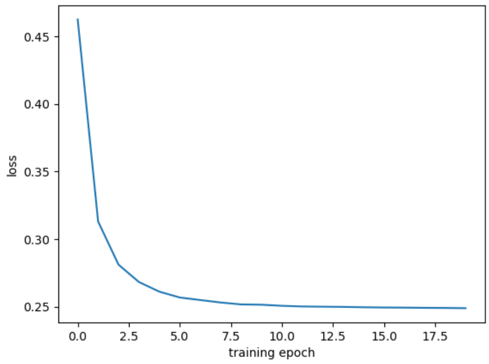



图 4: loss-epoch 图

执行 python classification.py test 的结果如下:

```
[Info] Load model from saved_models/model_epoch20.pth
[Info] Test accuracy = 90.3%
```

执行 python classification.py visual 的结果如下:

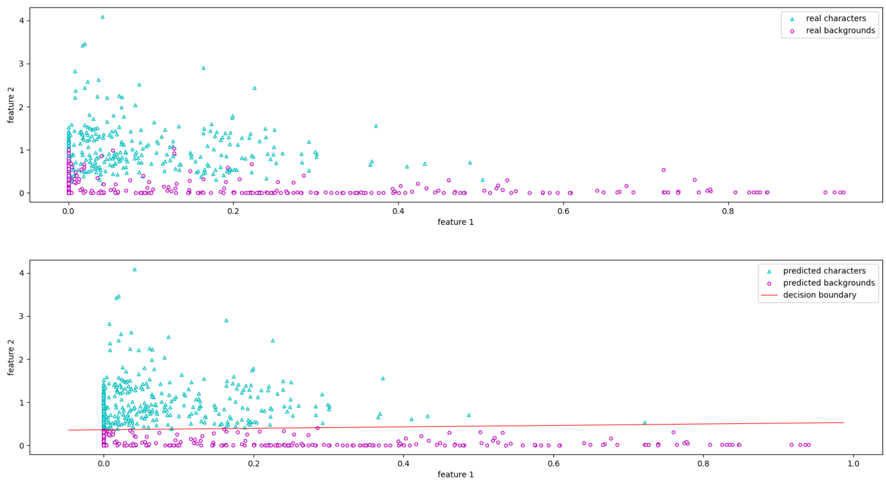



图 5: 模型线性分类决策面

# 4 学习总结

为了完成此次作业，我学习了《动手学深度学习》pytorch 版第三章的部分内容，要点总结如下：

1. 注意线性回归前向计算的形式，课件上是 $z = \mathbf{w}\mathbf{x} + b$，权重矩阵在左；而 torch.nn.Linear() 官网上的形式是 $z = \mathbf{x}\mathbf{A}^{\mathbf{T}} + b$，权重矩阵在右，且有一个转置！本次实验中采用的是 $z = \mathbf{x}w + b$ 形式。

2. 继承 torch.utils.data.Dataset 后需要重载其 ___init___(), ___len___() 和 ___getitem___()。torch.utils.data.Dataloader() 返回的是一个迭代器，可以用 for i in ⋯⋯来访问迭代器返回的每一批次 data。

3. 注意 tensor 的形状！在用 +-*/或者 torch 提供的函数进行 tensor 计算时，如果形状不匹配很容易报错，尤其是前向传播和误差的计算，必要时通过 tensor.view() 修改。

4. 执行反向传播的 tensor 必须是一个标量！我们一般对 loss 做 backward，所以得到的 loss 必须是标量 tensor。pytorch 自带的损失函数默认返回的是 $\hat{y}$ 和 $y$ 对应 loss 的 'mean'，这是一个标量，所以如果我们不用 pytorch 自带的损失函数的话，必须在返回 loss 时将其调整为标量（可以求和或者求平均）。注意用 SGD 更新参数 $\theta$ 时按照小批量随机梯度下降的公式，会有一个 $\frac{1}{batch\_size}$，所以如果 loss 函数中没有 $\frac{1}{batch\_size}$ 的话，需要在参数更新中实现。而 torch.optim 中默认是不除以 batch_size 的，所以需要提前除。

5. 注意在每次更新完参数后要将参数的梯度清零。

6. 鉴于每一个批次的数据的形状是 (batch_size,⋯⋯)，我们可以自定义一层 FlattenLayer(nn.Module)，它的 forward(self, x) 中只有一句：return x.view(x.shape[0],-1)，这样确保数据是二维 tensor，共有 batch_size 行，每一行对应一条数据。