第一周

1. 安装

1. Anaconda
    - 勾选添加环境变量
    - 更换下载源
2. CUDA与CuDNN
    - https://developer.nvidia.com/cuda-downloads
    - https://developer.nvidia.com/cudnn
    - 将CuDNN压缩包中的文件复制至CUDA安装文件夹中
    - 执行demo_suite文件夹下`bandwidthTest.exe`与`deviceQuery.exe`两个文件
3. PyTorch与torchvision
    - https://download.pytorch.org/whl/torch_stable.html
    - `conda create -n 环境名`
    - `pip install 文件名.whl`
    - `torch.__version__`
    - `torch.cuda.is_available()`

2. 张量Tensor

2.1 张量的概念

- 张量是一个多维数组，它是标量（0维）/向量（1维）/矩阵（2维）的高维拓展

`torch.tensor`

- `data`：被包装的Tensor
- `grad`：data的梯度
- `grad_fn`：创建Tensor的Function，是自动求导的关键
- `requires_grad`：指示是否需要梯度
- `is_leaf`：指示是否是叶子结点（张量）
- `dtype`：张量的数据类型，如`torch.FloatTensor`，`torch.cuda.FloatTensor`
- `shape`：张量的形状
- `device`：张量的设备

2.2 张量的构建

直接创建

```
# 1.
torch.tensor( data,    # 数据，可以是list、numpy
              dtype=None,    # 数据类型，默认与data一致
              device=None,    # 所在设备，gpu/cpu
              requires_grad=False,    # 是否需要梯度
              pin_memory=Flase )    # 是否存于锁页内存
# 2.
torch.from_numpy(ndarray)    # tensor与ndarray共享内存，修改一个数据，另一个也会被改动
```

依据数值创建

```
# 1. 全零张量
torch.zeros( *size,    # 张量的形状
             out=None,    # 输出的张量
             dtype=None,
             layout=torch.strided,    # 内存中布局形式，可以是strided、sparse_coo
             device=None,
             requires_grad=False )

torch.zeros_like( input,    # 创建与input同形状的全0张量
                  dtype=None,
                  layout=None,
                  device=None,
                  requires_grad=False )
# 2. 全一张量
torch.ones( *size,    # 张量的形状
            out=None,    # 输出的张量
            dtype=None,
            layout=torch.strided,    # 内存中布局形式，可以是strided、sparse_coo
            device=None,
            requires_grad=False )

torch.ones_like( input,    # 创建与input同形状的全0张量
                 dtype=None,
                 layout=None,
                 device=None,
                 requires_grad=False )
# 3. 自定义张量
torch.full( size,    # 张量的形状
            fill_value,    # 张量的值
            out=None,    # 输出的张量
            dtype=None,
            layout=torch.strided,    # 内存中布局形式，可以是strided、sparse_coo
            device=None,
            requires_grad=False )

torch.full_like()
# 4. 等差数列
# 数值区间为[start,end)
torch.arange( start=0,    # 数列起始值
              end,    # 数列结束值
              step=1,    # 数列公差
```

```
            out=None,
            layout=torch.strided,
            device=None,
            requires_grad=False )
# 5. 均分数列
# 数值区间为[start,end]
torch.linspace( start,    # 数列起始值
            end,    # 数列结束值
            steps=100,    # 数列长度
            out=None,
            dtype=None,
            layout=torch.strided,
            device=None,
            requires_grad=False )
# 6. 对数均分数列
torch.logspace( start,    # 数列起始值
            end,    # 数列结束值
            steps=100,    # 数列长度
            base=10.0,    # 对数函数的底
            out=None,
            dtype=None,
            layout=torch.strided,
            device=None,
            requires_grad=False )
# 7. 单位对角张量
torch.eye( n,    # 矩阵行数
        m=None,    # 矩阵列数
        out=None,
        dtype=None,
        layout=torch.strided,
        device=None,
        requires_grad=False )
```

依据概率分布创建

```
# 1. 正态分布
# mean与std均可为标量或张量；当二者均为标量时，需要设置size
torch.normal( mean,    # 均值
            std,    # 标准差
            size,
            out=None)
# 2. 标准正态分布（mean=0,srd=1）
torch.randn( *size,    # 张量的形状
            out=None,
            dtype=None,
            layout=torch.strided,
            device=None,
            requires_grad=False)

torch.randn_like()
# 3. 均匀分布
# 数值区间为[0,1)
torch.rand( *size,    # 张量的形状
            out=None,
            dtype=None,
            layout=torch.strided,
            device=None,
```

```
            requires_grad=False)

torch.rand_like()
# 数值区间为[low,high)
torch.randint( low=0,
            high,
            size,
            out=None,
            dtype=None,
            layout=torch.strided,
            device=None,
            requires_grad=False)

torch.randint_like()
# 4. 0-1分布
torch.bernoulli( input,    # 概率值
            *,
            generator=None,
            out=None )

# 生成从0到n-1的随机排列，常用于生成索引
torch.randperm( n,    # 张量的长度
            out=None,
            dtype=torch.int64,
            layout=torch.strided,
            device=None,
            requires_grad=False)
```

2.3 作业

- 安装anaconda,pycharm, CUDA+CuDNN（可选），虚拟环境，pytorch，并实现hello pytorch查看pytorch的版本

```
import torch

print(torch.__version__)    # 1.3.0
```

- 张量与矩阵、向量、标量的关系是怎么样的?
  - 张量是其三者的高维拓展
- Variable“赋予”张量什么功能?
  - 自动求导
- 采用 `torch.from_numpy` 创建张量，并打印查看ndarray和张量数据的地址

```
import numpy as np

a = np.arange(1.,5.)
b = torch.from_numpy(a)
print("a:",id(a))    # a: 2610267966480
print("b:",id(b))    # b: 2610270812776
```

- 实现 `torch.normal()` 创建张量的四种模式。

```
# mean为标量，std为标量
c = torch.normal(0.,1.,size=(4,))
print("c:",c)    # c: tensor([-0.6274, 0.5585, -0.3253, -0.6051])
# mean为标量，std为张量
d = torch.normal(0.,b)
print("d:",d)    # d: tensor([ 1.0161, -1.0010, -1.0991, -1.6921],
dtype=torch.float64)
# mean为张量，std为标量
e = torch.normal(b,1.)
print("e:",e)    # e: tensor([2.0389, 3.1449, 3.2940, 4.2294],
dtype=torch.float64)
# mean为张量，std为张量
f = torch.normal(b,b)
print("f:",f)    # f: tensor([ 2.2841, -1.1381, 6.2357, 8.5782],
dtype=torch.float64)
```

3. 张量操作与线性回归

3.1 张量的操作

拼接与切分

```
# 1. 拼接
# 将张量按维度dim进行拼接
torch.cat(tensors,    # 张量序列
        dim=0,    # 要拼接的维度
        out=None)
# 在新创建得维度dim上进行拼接
torch.stack(tensors,
        dim=0,
        out=None)
# 2. 切分
# 将张量按维度dim进行平均切分
# 注：若不能整除，最后一个张量小于其他张量
torch.chunk(input,    # 要切分的张量
        chunks,    # 要切分的份数
        dim=0)
torch.split(tensor,
        split_size_or_sections,    # 为int时，表示每一份长度；为list时，按list元
素切分
        dim=0)
```

索引

```
# 1.
# 在维度dim上，按index索引数据
torch.index_select(input,    # 要索引的张量
        dim,    # 要索引的维度
        index,    # 要索引数据的序号
        out=None)
# 2.
# 按mask中True进行索引，返回一维列表
torch.masked_select(input,
        mask,    # 与input同形状的布尔类型张量
        out=None)
```

```
# 生成mask
# ge/gt/le/lt
mask = t.ge(5)    # greater than or equal >=
```

变换

```
# 1. 形状变换
# 注：当张量在内存中是连续时，新张量与input共享数据内存
# shape中-1表示不关注的维度
torch.reshape(input,    # 要变换的张量
        shape    # 新张量的形状)
# 2. 维度变换
torch.transpose(input,
        dim0,
        dim1)
# 二维张量转置，等价于torch.transpose(input,0,1)
torch.t(input)
# 3. 压缩与扩展变换
torch.squeeze(input,
        dim=None,    # 若为None，移除所有长度为1的维度；若指定维度，当且仅当该维度长度为1时，可以被移除
        out=None)
torch.usqueeze(input,
        dim,    # 扩展的维度
        out=None)
```

3.2 张量的数学运算

加减乘除

```
# input + alpha * other
torch.add(input    # 第一个张量
        alpha=1,    # 乘项因子
        other,    # 第二个张量
        out=None)
# input + value * tensor1 / tensor2
torch.addcdiv()
# input + value * tensor1 * tensor2
torch.addcmul(input,
        value=1,
        tensor1,
        tensor2
        out=None)
torch.sub()
torch.div()
torch.mul()
```

对指幂函数

```
torch.log(input,out=None)
torch.log10(input,out=None)
torch.log2(input,out=None)
torch.exp(input,out=None)
torch.pow()
```

三角函数

```
torch.abs(input,out=None)
torch.acos(input,out=None)
torch.cosh(input,out=None)
torch.cos(input,out=None)
torch.asin(input,out=None)
torch.atan(input,out=None)
torch.atan2(input,other,out=None)
```

3.3 线性回归

- 线性回归是分析一个变量与另外一个或多个变量之间线性关系的方法

求解步骤：

- 确定模型：$y = w \times x + b$
- 选择损失函数：`MSE`
- 求解梯度并更新 w，b

```
import torch
import matplotlib.pyplot as plt

torch.manual_seed(10)
lr = 0.1
# 创建训练数据
x = torch.rand(20, 1) * 10
y = 2 * x + (5 + torch.randn(20, 1))
# 构建线性回归参数
w = torch.randn((1), requires_grad=True)
b = torch.zeros((1), requires_grad=True)

for iteration in range(1000):
    # 前向传播
    wx = torch.mul(w,x)
    y_pred = torch.add(wx, b)
    loss = (0.5 * (y - y_pred) ** 2).mean()
    loss.backward()    # 反向传播
    # 更新参数
    b.data.sub_(lr * b.grad)
    w.data.sub_(lr * w.grad)
    # 绘图
    if iteration % 20 == 0:
        plt.scatter(x.data.numpy(), y.data.numpy())
        plt.plot(x.data.numpy(), y_pred.data.numpy(), 'r-', lw=5)
        plt.text(2, 20, 'Loss=%.4f' % loss.data.numpy(), fontdict={'size' : 20, 'color' : 'red'})
        plt.xlim(1.5, 10)
        plt.ylim(8, 28)
        plt.title("Iteration: {}\nw:{} n:{}" . format(iteration, w.data.numpy(), b.data.numpy()))
        plt.pause(0.5)
        if loss.data.numpy() < 1:
            break
```

4. 计算图与动态图机制

4.1 计算图

- 计算图是用来描述运算的有向无环图
- 计算图两个主要元素：**结点node**，**边edge**
  - 结点表示**数据**，如向量/矩阵/张量
  - 边表示**运算**，如加/减/乘/除/卷积

```
import torch

w = torch.tensor([1.,], requires_grad=True)
x = torch.tensor([2.,], requires_grad=True)
a = torch.add(w,x)
# a.retain_grad() 保存非叶子结点a的梯度
b = torch.add(w,1)
y = torch.mul(a,b)
y.backward()
print(w.grad)    # w=5
```

- 叶子节点
  - 用户创建的结点，如 x/w
  - `is_leaf`：指示张量是否为叶子结点
  - 非叶子结点梯度会被释放

```
# 查看叶子节点
print("is_leaf:\n", w.is_leaf, x.is_leaf, a.is_leaf, b.is_leaf, y.is_leaf)
 # TTFFF
# 查看梯度
print("gradient:\n", w.grad, x.grad, a.grad, b.grad, y.grad)    # 52NNN
```

- `grad_fn`
  - 用于记录创建张量时使用的方法/函数
  - 叶子结点 `grad_fn=None`

```
# y.grad_fn = <MulBackward0>
# a.grad_fn = <AddBackward0>
# b.grad_fn = <AddBackward0>
print("grad_fn:\n", w.grad_fn, x.grad_fn, a.grad_fn, b.grad_fn, y.grad_fn)
 # NNAAM
```

4.2 动态图机制

- 动态图：搭建与运算同时进行
- 静态图：先搭建图，后运算

4.3 作业

- 调整线性回归模型停止条件以及 $y = 2 \times x + (5 + torch.randn(20, 1))$ 中的斜率，训练一个线性回归模型
  - 修改上面线性回归示例代码即可

- 计算图的两个主要概念是什么?
  - 结点与边
- 动态图与静态图的区别是什么?
  - 动态图搭建与运算同时进行，灵活易调节
  - 静态图先搭建后运算，不够灵活但高效

5. autograd与逻辑回归

5.1 autograd

```
# 1.
torch.autograd.backward(tensors,     # 用于求导的张量，如loss
             grad_tensors=None,     # 多梯度权值
             retain_graph=None,     # 保存计算图
             create_graph=Flase     # 创建导数计算图，用于高阶求导)
# 2.
torch.autograd.grad(outputs,     # 用于求导的张量，如loss
             inputs,     # 需要梯度的张量
             grad_tensors=None,     # 多梯度权值
             retain_graph=None,     # 保存计算图
             create_graph=Flase     # 创建导数计算图，用于高阶求导)
```

注:

- 梯度不自动清零，使用 `w.grad.zero_()` 操作清零
- 依赖于叶子结点的结点， `requires_grad` 默认为 `True`
- 叶子结点不可执行 `in-place`

```
# in-place操作示例
import torch

a = torch.ones((1, ))
print(id(a), a)     # 原始内存地址
a = a + torch.ones((1, ))
print(id(a), a)     # 开辟了新内存地址
a += torch.ones((1, ))
print(id(a), a)     # 依然是原始内存地址
```

5.2 逻辑回归

- 逻辑回归是**线性**的**二分类**模型，在线性回归的基础上增加了sigmoid激活函数
- 模型表达式： $y = f(wx + b)$ ， $f(x) = \frac{1}{1+e^{-x}}$ （sigmoid函数）

$$\text{class} = \begin{cases} 0, & 0.5 > y \\ 1, & 0.5 \leq y \end{cases}$$

- 机器学习模型训练步骤
  - 数据
  - 模型
  - 损失函数
  - 优化器
  - 迭代训练
- 逻辑回归示例

```python
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
torch.manual_seed(10)

# 1. 生成数据
sample_nums = 100
mean_value = 1.7
bias = 1
n_data = torch.ones(sample_nums, 2)
x0 = torch.normal(mean_value * n_data, 1) + bias
y0 = torch.zeros(sample_nums)
x1 = torch.normal(-mean_value * n_data, 1) + bias
y1 = torch.ones(sample_nums)
train_x = torch.cat((x0,x1), 0)
train_y = torch.cat((y0,y1), 0)

# 2. 选择模型
class LR(nn.Module):
    def __init__(self):
        super(LR, self).__init__()
        self.features = nn.Linear(2,1)
        self.sigmoid = nn.Sigmoid()

    def forward(self,x):
        x = self.features(x)
        x = self.sigmoid(x)
        return x

lr_net = LR() # 实例化

# 3. 选择损失函数
loss_fn = nn.BCELoss()

# 4. 选择优化器
lr = 0.01
optimizer = torch.optim.SGD(lr_net.parameters(), lr=lr, momentum=0.9)

# 5. 训连模型
for iteration in range(1000):

    # 前向传播
    y_pred = lr_net(train_x)
    loss = loss_fn(y_pred.squeeze(), train_y)

    # 反向传播
    loss.backward()
    optimizer.step()

    # 绘图
    if iteration % 20 == 0:
        mask = y_pred.ge(0.5).float().squeeze() # 以0.5为阈值进行分类
        correct = (mask == train_y).sum() # 计算正确预测的样本个数
        acc = correct.item() / train_y.size(0) # 计算分类准确率

```

```
        plt.scatter(x0.data.numpy()[:,0], x0.data.numpy()[:,1], c='r', label='class 0')
        plt.scatter(x1.data.numpy()[:,0], x1.data.numpy()[:,1], c='b', label='class 1')

        w0, w1 = lr_net.features.weight[0]
        w0, w1 = float(w0.item()), float(w1.item())
        plot_b = float(lr_net.features.bias[0].item())
        plot_x = np.arange(-6, 6, 0.1)
        plot_y = (-w0 * plot_x - plot_b) / w1

        plt.xlim(-5, 7)
        plt.ylim(-7, 7)
        plt.plot(plot_x, plot_y)

        plt.text(-5, 5, 'Loss=%.4f' % loss.data.numpy(), fontdict={'size':20, 'color':'red'})
        plt.title("Iteration:{}\nw0:{:.2f} w1:{:.2f} b:{:.2f} accuracy:{:.2%}".format(iteration, w0, w1, plot_b, acc))
        plt.legend()

        plt.show()
        plt.pause(0.5)

        if acc > 0.99:
            break
```

5.3 作业

- 逻辑回归模型为什么可以进行二分类?
  - sigmoid函数将预测值y映射至 $(0,1)$ 之间，当预测值 $y>0.5$ 时，视为一类； $y<0.5$ 时，视为另一类
- 采用代码实现逻辑回归模型的训练，并尝试调整数据生成中的 `mean_value` ，将 `mean_value` 设置为更小的值，例如1，或者更大的值，例如5，会出现什么情况？再尝试仅调整bias，将bias调为更大或者负数，模型训练过程是怎么样的?
  - 当 `mean_value=1` 时样本点更密集，难以分类。
  - 当 `mean_value=5` 时样本点过于稀疏，极易分类。
  - 当 `bias=2` 时样本点向右上角偏移，loss值难以收敛。
  - 当 `bias=-2` 时样本点向左下角偏移，loss值难以收敛。