《人工智能与Pytorch程序设计》——PyTorch简介

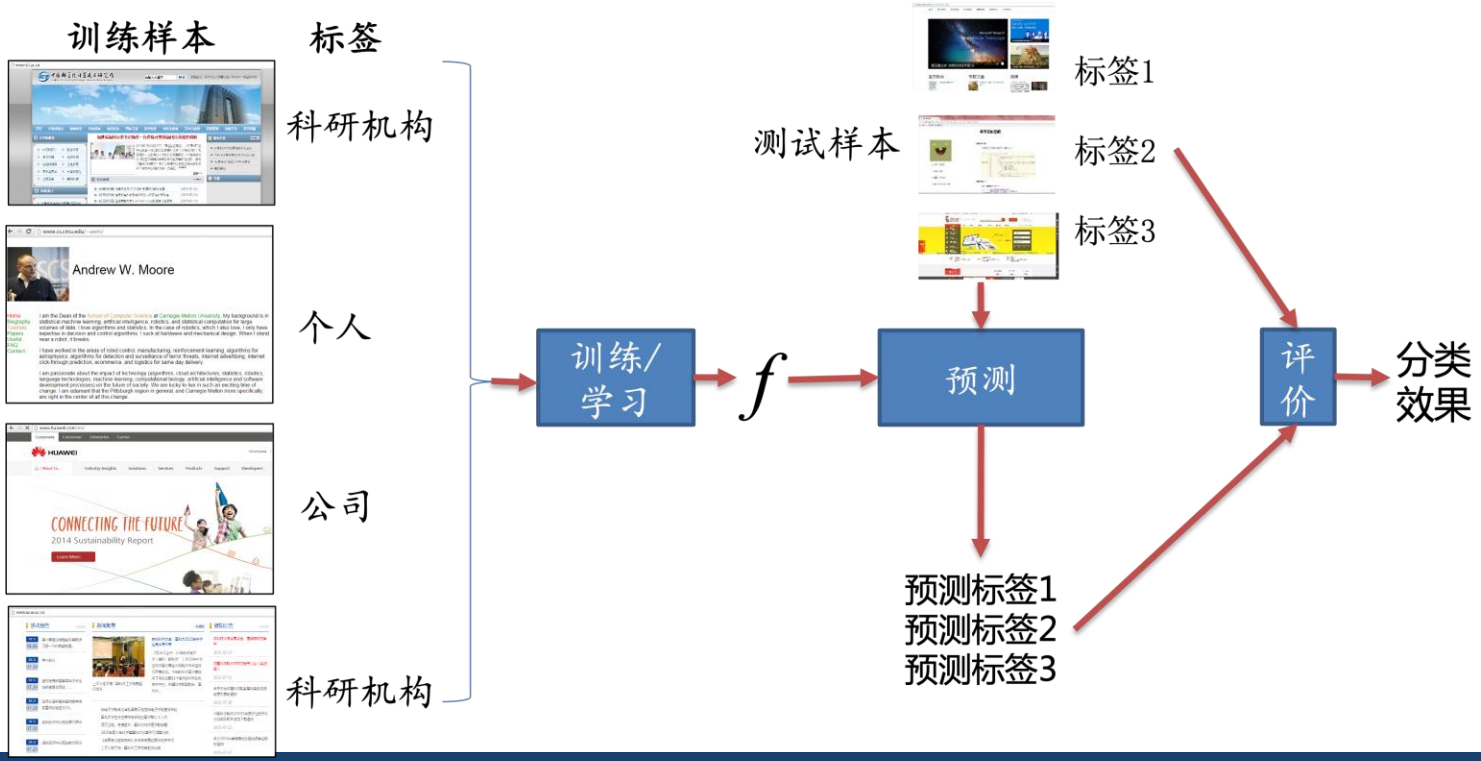
人工智能与Python程序设计 教研组

复习：什么是机器学习

- 研究一类算法，使之
  - 在某些**任务**上(task)
  - 通过已有的观测**经验**(数据)(experience)
  - 提升算法**效果**(performance)
- 机器学习应用的例子：
  - 网页分类
  - 手写数字识别
  - 人脸识别
  - 垃圾信息分类
  - 图像分类
- 机器学习任务分类：
  - 有监督学习
    - 回归、分类
  - 无监督学习
  - 强化学习

复习：机器学习流程

- 以网页分类为例



复习：机器学习流程

- 构建分类器的流程

数据准备

- 数据标注
- 训练集/验证集/测试集分割
- 特征提取

模型训练

- 分类损失函数
- 损失函数优化和参数调优

模型测试

- 性能评价指标

复习：二分类的性能评价指标

- 评测一个模型在实际应用环境的性能
  - 评估不同机器学习模型、不同参数设置的优劣
  - 在线应用模型前对预测精度进行估计
  - 注意区分模型测试时使用的性能评价指标和训练/学习时使用的损失函数
- 二分类问题的性能评价指标
  - 混淆矩阵
  - 正确率（Accuracy): $\frac{\mathrm{TP}+TN}{TP+FN+FP+TN}$
    - 仅仅使用正确率评价的问题
  - 精确率（Precision）：$\frac{TP}{TP+FP}$
  - 召回率（Recall）：$\frac{TP}{TP+FN}$
  - F1值（F1 score）：$\mathrm{F1}=2\times\frac{\mathrm{Precision}\times\mathrm{Recall}}{\mathrm{Precision}+\mathrm{Recall}}$
    - 精确率和召回率的调和平均数
    - 更接近精确率和召回率中较小的一个

	预测为正样本	预测为负样本
标注为正样本	TP (true positive)	FN (false negative)
标注为负样本	FP (false positive)	TN (true negative)

PyTorch简介

提纲

- 人工智能平台之PyTorch
- PyTorch入门
  - Tensor
  - AutoGrad

近年来涌现的人工智能/深度学习平台

- 几大阵营
  - Facebook：PyTorch，2017年1月
  - Google：TensorFlow，2015年11月
  - Amazon：MXNet，2015年9月
  - Microsoft: CNTK，2016年1月
  - 学术界：Caffe，2013年9月
  - 百度：PaddlePaddle，2016年8月

阿里：X-Deep Learning (XDL)

为何需要上述平台?

- 人工智能学术研究和教学
  - 简单易用，降低编程门槛
  - 融合多种计算硬件和资源
  - 聚焦于模型和算法逻辑，减少辅助代码

- 人工智能生产应用
  - 快速、灵活并适合产品级大规模应用
  - 应对多样化的挑战：适应五花八门、瞬息万变的应用环境
  - 便捷开发、快速迭代与部署

为何选择PyTorch?

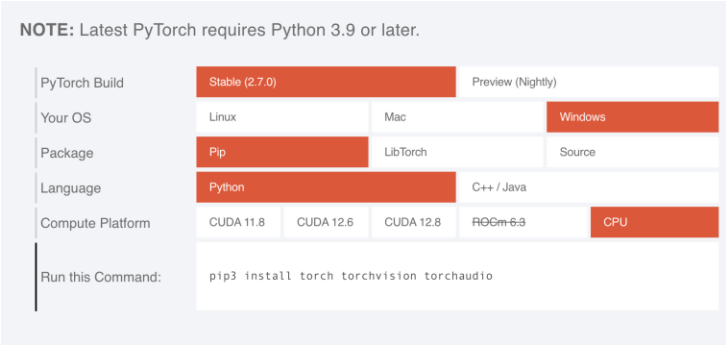
- 每一种平台都有自己的优缺点
  - 大部分平台仍然在不断更新完善
  - 大多都足以支持人工智能模型研究和产品研发

- PyTorch（在科研教学方面）具有独特的优势
  - 充分利用普通Python代码的灵活性和能力来构建、训练神经网络 →能解决更广泛的问题
  - PyTorch 让自定义的实现更加容易，将更多时间专注于算法中
  - 能让初学者更深入地了解每个算法中发生了什么（相比于TensorFlow而言）

用TensorFlow我能找到很多别人的代码
用PyTorch我能轻松实现自己的想法

安装PyTorch

- https://pytorch.org/



- 推荐使用Anaconda安装
- 选择版本、操作系统、Package、语言、是否有GPU等信息
- 复制command命令行，输入命令行窗口执行
  - 需要安装numpy

- 中文文档：https://pytorch.apachecn.org/
- 查看PyTorch是否安装成功

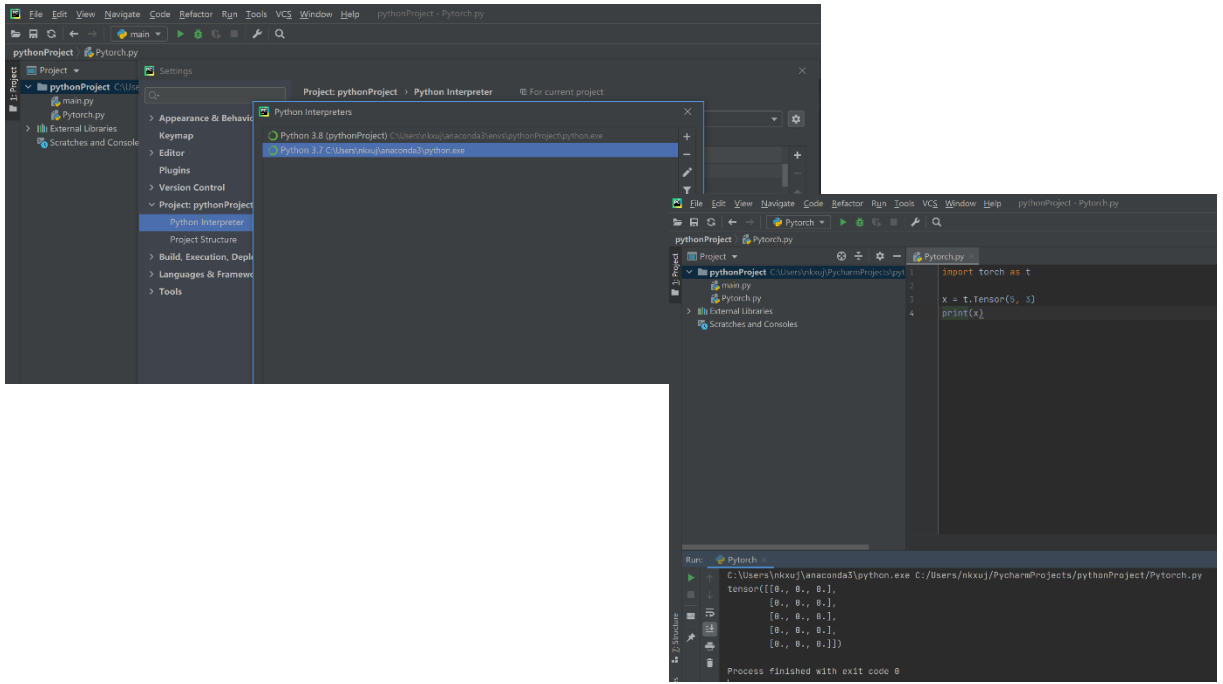
```
(base) C:\Users\nkxuj>python
Python 3.7.6 (default, Jan  8 2020, 20:23:39) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> print(torch.__version__)
1.6.0
```

在Jupyter Notebook和PyCharm中使用PyTorch

- Jupyter Notebook

```
In [1]: import torch as t
        t.Tensor(3, 4)

Out[1]: tensor([[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.]])
```

- PyCharm如果出错，尝试改变一下Python解析器的路径



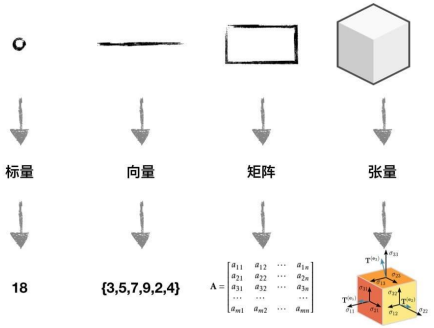
PyTorch简介

提纲

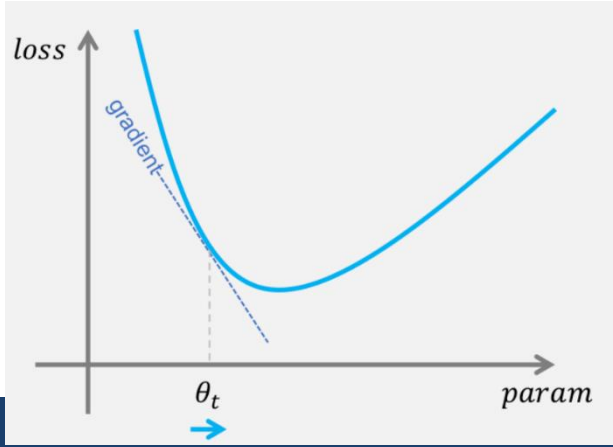
- 人工智能平台之PyTorch
- PyTorch入门
  - Tensor
  - AutoGrad

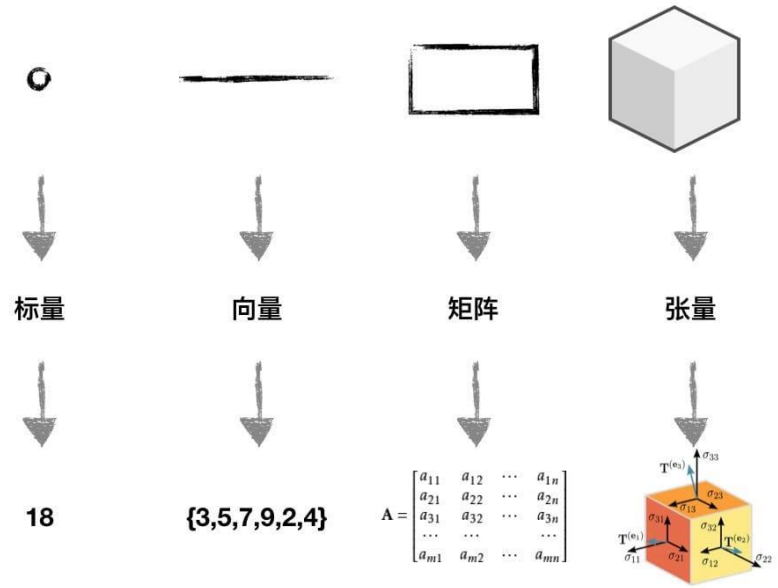
PyTorch入门

- 可以认为PyTorch是一个基于Python的科学计算包
  - 替代numpy发挥GPU潜能（增强版的numpy）
  - 提供了高度灵活性和效率的深度学习实验性平台

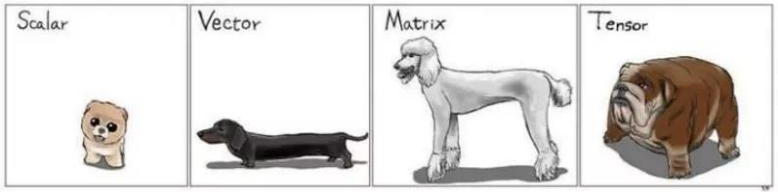


- PyTorch中最主要的两个概念
  - Tensor（张量）
  - Autograd（自动求导）





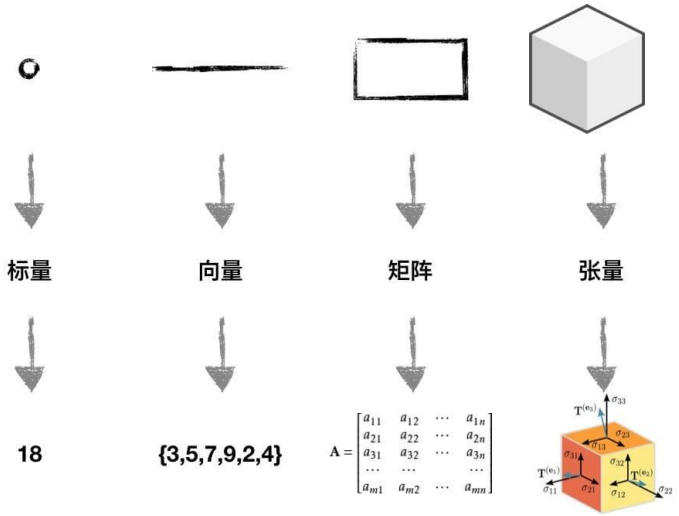
张量



PyTorch入门——Tensor

- Tensor可以认为是一个高维数组
  - 与numpy中的ndarray相似
  - 可以使用GPU加速

- Tensor的操作与numpy中的定义相似
  - 构建、初始化一个新的Tensor
  - 数学运算、线性代数运算
  - 选择、切片



创建一个Tensor

- 0-1初始化
  - torch.empty(size)：返回形状为size的空tensor
  - torch.zeros(size)：全部是0的tensor
  - torch.ones(size)：全部是1的tensor

- 随机初始化
  - torch.rand(size)：构建形状为size的Tensor，取值为[0,1)内的均匀分布随机数
  - torch.randn(size)：取值为标准正态分布N(0,1)的随机数
  - torch.normal(mean, std, out=None)：取值为标准正态分布N(mean,std^2)的随机数

```
import torch as t
t.Tensor(3, 4)
```

```
1]: tensor([[0., 0., 0., 0.],
            [0., 0., 0., 0.],
            [0., 0., 0., 0.]])
```

```
# 生成一个随机Tensor
x = t.rand(2, 3, 4)
x
```

```
8]: tensor([[[0.2171, 0.6213, 0.7121, 0.3788],
             [0.3043, 0.3532, 0.7203, 0.4110],
             [0.4478, 0.1721, 0.1924, 0.4209]],

            [[0.5084, 0.0502, 0.1158, 0.5984],
             [0.2916, 0.1080, 0.3094, 0.3991],
             [0.2528, 0.2338, 0.2153, 0.6788]]])
```

```
# 查看x的形状
print(x.size())

print(x.size()[0], x.size()[1], x.size()[2])
```

```
torch.Size([2, 3, 4])
2 3 4
```

Tensor操作：加

- 操作符：+ 、add和add_
  - 注意：大小要匹配（pytorch同样有广播机制）

```
#两个Tensor相加
y = t.rand(2, 3, 4)
z = x + y
z
```

```
tensor([[[0.3955, 1.5931, 0.9961, 1.1492],
         [0.9788, 0.6793, 0.8304, 0.5727],
         [1.2582, 0.2156, 0.2669, 0.4646]],

        [[0.6923, 0.6903, 0.5903, 0.6059],
         [0.8083, 0.9475, 0.7094, 0.8159],
         [1.1215, 0.8542, 1.0859, 0.7217]]])
```

```
# 注意两个张量对应的维度要匹配
y1 = t.rand(3, 4, 2)
z1 = x + y1
z1
```

```
RuntimeError                              Traceback (most recent call last)
<ipython-input-21-82390817fe76> in <module>
      1 # 注意两个张量对应的维度要匹配
      2 y1 = t.rand(3, 4, 2)
----> 3 z1 = x + y1
      4 z1

RuntimeError: The size of tensor a (4) must match the size of tensor b (2) at non-singleton dimension 2
```

```
#通过函数add实现加法
print(t.add(x, y))
```

```
tensor([[[0.3955, 1.5931, 0.9961, 1.1492],
         [0.9788, 0.6793, 0.8304, 0.5727],
         [1.2582, 0.2156, 0.2669, 0.4646]],

        [[0.6923, 0.6903, 0.5903, 0.6059],
         [0.8083, 0.9475, 0.7094, 0.8159],
         [1.1215, 0.8542, 1.0859, 0.7217]]])
```

```
#另外一个写法，结果输出到提前定义好的Tensor中
result = t.Tensor(2, 3, 4)
t.add(x, y, out = result)
result
```

```
tensor([[[0.3955, 1.5931, 0.9961, 1.1492],
         [0.9788, 0.6793, 0.8304, 0.5727],
         [1.2582, 0.2156, 0.2669, 0.4646]],

        [[0.6923, 0.6903, 0.5903, 0.6059],
         [0.8083, 0.9475, 0.7094, 0.8159],
         [1.1215, 0.8542, 1.0859, 0.7217]]])
```

```
# 函数名带下划线 x.add_(y)会改变x，不带下划线x.add(y) 返回一个新的Tensor，x不改变
print(x)
x.add(y)
print(x)
x.add_(y)
print(x)
```

```
tensor([[[0.5739, 2.5648, 1.2802, 1.9196],
         [1.6533, 1.0054, 0.9404, 0.7345],
         [2.0685, 0.2591, 0.3413, 0.5084]],

        [[0.8761, 1.3303, 1.0649, 0.6135],
         [1.3251, 1.7870, 1.1094, 1.2327],
         [1.9903, 1.4747, 1.9564, 0.7646]]])
tensor([[[0.5739, 2.5648, 1.2802, 1.9196],
         [1.6533, 1.0054, 0.9404, 0.7345],
         [2.0685, 0.2591, 0.3413, 0.5084]],

        [[0.8761, 1.3303, 1.0649, 0.6135],
         [1.3251, 1.7870, 1.1094, 1.2327],
         [1.9903, 1.4747, 1.9564, 0.7646]]])
tensor([[[0.7523, 3.5366, 1.5642, 2.6900],
         [2.3279, 1.3315, 1.0505, 0.8962],
         [2.8789, 0.3026, 0.4157, 0.5521]],

        [[1.0600, 1.9704, 1.5394, 0.6211],
         [1.8418, 2.6264, 1.5093, 1.6494],
         [2.8590, 2.0951, 2.8270, 0.8075]]])
```

Tensor操作：减

- 操作符：-
  - 与加法类似，两个矩阵大小必须匹配

```
a =t.ones(2, 3)
b = 3.0 * t.ones(2, 3)
c = a - b
print(c)
```

```
tensor([[-2., -2., -2.],
        [-2., -2., -2.]])
```

```
d = t.ones(2,2)
e = a - d
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-208-d8132db88a45> in <module>
      1 d = t.ones(2,2)
----> 2 e = a - d

RuntimeError: The size of tensor a (3) must match the size of tensor b (2) at non-singleton dimension 1
```

Tensor操作：乘

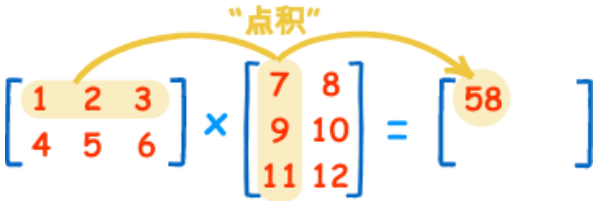
- 矩阵(Tensor)的乘法在人工智能算法极为普遍
  - 数乘 $a * \mathbf{X}$：把X中每一个元素都乘a

$$2 \times \begin{pmatrix} 2 & 1 \\ 4 & 3 \end{pmatrix} = \begin{pmatrix} 4 & 2 \\ 8 & 6 \end{pmatrix}$$

  - 对应点相乘 x.mul(y)或 x*y（点乘再求和为卷积）

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ h & i \end{bmatrix} = \begin{bmatrix} ae & bf \\ ch & di \end{bmatrix}$$

  - 线性代数中的矩阵相乘，x.mm(y)




```
a = t.Tensor([[1,2], [3,4], [5, 6]])
print(a)

#数乘矩阵
b = 2.0 * a
print(b)

#对应点相乘，sum后即为卷积
c = a * b
print(c)
print(c.sum())

# a: 3 * 2, b.t():2 * 3
# d: 3 * 3
d = a.mm(b.t())
print(d)


# a.t(): 2 * 3, b.t():3 * 2
# e: 2 * 2
e = a.t().mm(b)
print(e)
```

```
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
tensor([[ 2.,  4.],
        [ 6.,  8.],
        [10., 12.]])
tensor([[ 2.,  8.],
        [18., 32.],
        [50., 72.]])
tensor(182.)
tensor([[ 10.,  22.,  34.],
        [ 22.,  50.,  78.],
        [ 34.,  78., 122.]])
tensor([[ 70.,  88.],
        [ 88., 112.]])
```

矩阵乘操作A.mm(B)

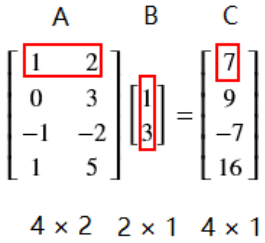
- A.mm(B)，矩阵A乘以矩阵B
  - A：$m \times n$，B：$n \times k$ （中间维度相同）
  - 返回结果矩阵大小为$m \times k$
    - 第i行j列的值：A的第i行内积B的第j列
- 如果大小不匹配，则出错

- A.mm(B)，矩阵A乘以矩阵B
  - A：$m \times n$，B：$n \times k$ （中间维度相同）
  - 返回结果矩阵大小为$m \times k$
    - 第i行j列的值：A的第i行内积B的第j列
- 如果大小不匹配，则出错


```
a = t.Tensor([[1,2], [3,4], [5, 6]]) # 3 * 2
b = t.Tensor([[1, 2],[3,4],[5,6],[7,8]]) # 4 * 2

c = a.mm(b.t()) # 3 * 4
print(c)


#如果矩阵大小不满足 x: i * n, y:n * j的方式，则出错
c = a.mm(b)
```

```
tensor([[ 5., 11., 17., 23.],
        [11., 25., 39., 53.],
        [17., 39., 61., 83.]])

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-204-71d00edcc7f0> in <module>
      7
      8 #如果矩阵大小不满足 x: i * n, y:n * j的方式，则出错
----> 9 c = a.mm(b)

RuntimeError: size mismatch, m1: [3 x 2], m2: [4 x 2] at ..\aten\src\TH/generic/THTensorMath.cpp:41
```

矩阵与向量（向量与矩阵）的乘



- 把向量看成1*n的矩阵，然后利用矩阵乘完成

- 在右图的例子中
  - x的定义为1*2的矩阵[[1, 2]]，而非向量[1, 2]
  - Tensor([[1, 2]])返回1*2的矩阵
  - A.t()：矩阵转置

  - A：3*2；x：1*2
  - A.mm(x.t())，A*xT：返回3*1的矩阵
  - x.mm(A.t())，x*AT：返回1*3的矩阵

```python
#A: 3 * 2
#x: 1 * 2
A = t.Tensor([[1,2], [3,4], [5, 6]])
x = t.Tensor([[1, 2]])
print(A)
print(x)

# A: 3*2; b.t(): 2 * 1
# 结果：3 * 1
c = A.mm(x.t())
print(c)

#x: 1 * 2, A.t(): 2 * 3
# 结果: 1 * 3
d = x.mm(A.t())
print(d)
```

```
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
tensor([[1., 2.]])
tensor([[ 5.],
        [11.],
        [17.]])
tensor([[ 5., 11., 17.]])
```

Tensor的其它数学操作

- 基本数学操作函数
  - 除：torch.div(input, other, out=None)
  - 指数：torch.pow(input, exponent, out=None)
  - 开方：torch.sqrt(input, out=None)
  - 四舍五入到整数：torch.round(input, out=None)

- 神经网络中常用到的数值变换
  - sigmoid函数：torch.sigmoid(input, out=None)
  - tanh函数：torch.tanh(input, out=None)
  - 绝对值：torch.abs(input, out=None)
  - 向上取整：torch.ceil(input, out=None)
  - 限制范围：torch.clamp(input, min, max, out=None) 把输入数据规范在min-max区间，超过范围的用min、max代替

Tensor视图(viev

- 相当于numpy中的reshape功能
  - 把原先tensor中的数据按照行优先的顺序排成一个一维的数据
  - 然后按照参数组合成其他维度的tensor
  - 返回的数据和传入的tensor一样，**只是形状不同**

```
import torch as t
a=t.Tensor([[[1,2,3],[4,5,6]]])
b=t.Tensor([1,2,3,4,5,6])

print(a.view(1,6))
print(b.view(1,6))
```

```
tensor([[1., 2., 3., 4., 5., 6.]])
tensor([[1., 2., 3., 4., 5., 6.]])
```

```
a=t.Tensor([[[1,2,3, 4],[5,6,7,8]]])
print(a)
print('--------------------------------')
print(a.view(4,2))
print('--------------------------------')
print(a.view(2, 2 ,2))
print('--------------------------------')
```

```
tensor([[[1., 2., 3., 4.],
         [5., 6., 7., 8.]]])
--------------------------------
tensor([[1., 2.],
        [3., 4.],
        [5., 6.],
        [7., 8.]])
--------------------------------
tensor([[[1., 2.],
         [3., 4.]],

        [[5., 6.],
         [7., 8.]]])
--------------------------------
```

Tensor的选取

- 与Numpy中类似，[:, :, s]

- 其它操作
  - 拼叠：torch.cat(seq, dim=0, out=None)
  - 切块：torch.chunk(tensor, chunks, dim=0)
  - 把size为1的维度删除：torch.squeeze(input)
  - 变换形状：torch.reshape(input, shape)

```
# Tensor的选取操作与numpy类似
print(x)
x[0, 0, :]
```

```
tensor([[[0.7523, 3.5366, 1.5642, 2.6900],
         [2.3279, 1.3315, 1.0505, 0.8962],
         [2.8789, 0.3026, 0.4157, 0.5521]],

        [[1.0600, 1.9704, 1.5394, 0.6211],
         [1.8418, 2.6264, 1.5093, 1.6494],
         [2.8590, 2.0951, 2.8270, 0.8075]]])
```

```
]: tensor([0.7523, 3.5366, 1.5642, 2.6900])
```

```
x[:, 0, 0]
```

```
]: tensor([0.7523, 1.0600])
```

```
x[:, :, 0]
```

```
]: tensor([[0.7523, 2.3279, 2.8789],
        [1.0600, 1.8418, 2.8590]])
```

```
x[:,1:3, 2:4]
```

```
]: tensor([[[1.0505, 0.8962],
         [0.4157, 0.5521]],

        [[1.5093, 1.6494],
         [2.8270, 0.8075]]])
```

Numpy与PyTorch Tensor转换

- 虽然结构和功能非常类似，Python的numpy中的 (array) 与PyTorch Tensor属于不同类型数据结构
- 可以互相进行转换
  - Tensor-> numpy: numpy()
  - numpy -> tensor: from_numpy(a)

- 注意：转换后**numpy的变量和原来的tensor会共用底层内存地址**，如果原来的tensor改变了，numpy变量也会随之改变

```
#PyTorch Tensor与Numpy之间的转换
#tensor->numpy
a = t.ones(2, 3, 4)
b = a.numpy()
b
```

```
1]: array([[[1., 1., 1., 1.],
            [1., 1., 1., 1.],
            [1., 1., 1., 1.]],

           [[1., 1., 1., 1.],
            [1., 1., 1., 1.],
            [1., 1., 1., 1.]]], dtype=float32)
```

```
# numpy --> Tensor
import numpy as np
a = np.ones(5)
b = t.from_numpy(a)
print(a)
print(b)
```

```
[1. 1. 1. 1. 1.]
tensor([1., 1., 1., 1., 1.], dtype=torch.float64)
```

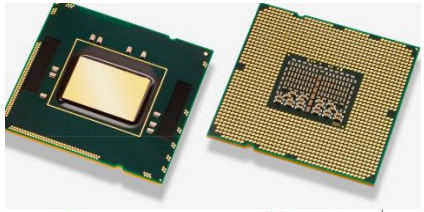
```
# 上述转换中，Tensor和Numpy共享内容，所以修改一个另一个也随之改变

print(a)
b.add_(1) #注意是下划线add_
print(a)
```

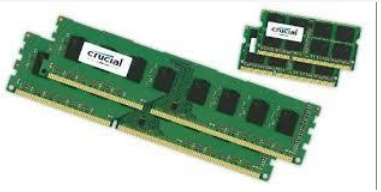
```
[1. 1. 1. 1. 1.]
[2. 2. 2. 2. 2.]
```

GPU与内存

- GPU (graphics processing unit)
  - 原为计算机图像处理设计的设备（显卡），包含**图像处理器**和**显存**
  - 相对CPU，GPU有更多的运算单元，非常适合深度学习中大量的并行计算（如：Tensor上的运算），近年来已经广泛被用于加速大规模神经网络训练（歪打正着）

- CPU (central processing unit) + 内存
  - 计算机的核心，执行指令+存储数据
  - CPU有更多的控制和缓存机制
  - 即使装备了GPU，CPU和内存也不可或缺
    - GPU的功能需要CPU进行调用
    - GPU中显存的数据从内存中读入，最终结果放入内存等待下一步处理





PyTorch支持GPU计算

- PyTorch通过CUDA框架对GPU进行调用
- 什么是CUDA?
  - Compute Unified Device Architecture
  - NVIDIA推出的用于GPU的**并行计算**框架
  - PyTorch通过CUDA与GPU进行交互

- PyTorch中使用GPU计算
  - t.cuda.is_available()：检查本机是否支持CUDA
  - x.cuda()，y.cuda()：把Tensor x和y的内容从内存移入显存
  - 操作既可在GPU上进行

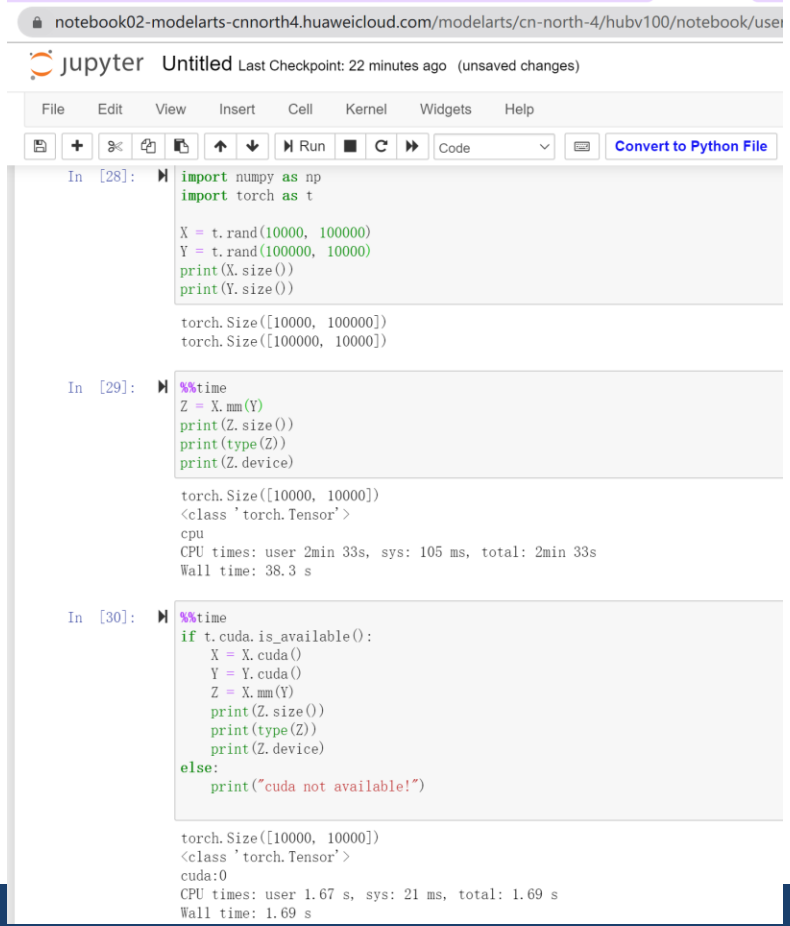
```
#支持GPU，将x和y都转移到GPU中进行运算
# 在大规模数据复杂Tensor运算时，具有优势；
# 小规模数据时由于具有数据转移开销，CPU会更快一点

#本机器不支持GPU，不会运行以下代码
if t.cuda.is_available():
    x = x.cuda() #将张量x转移到GPU中
    y = y.cuda() #将张量y转移到GPU中
    z = x + y #在GPU中运算x + y
else:
    z = x + y
print(z)
```

```
tensor([[[0.9307, 4.5083, 1.8482, 3.4604],
         [3.0024, 1.6576, 1.1606, 1.0579],
         [3.6893, 0.3461, 0.4902, 0.5958]],

        [[1.2438, 2.6105, 2.0140, 0.6287],
         [2.3585, 3.4659, 1.9093, 2.0662],
         [3.7277, 2.7156, 3.6976, 0.8504]]])
```

```
#如果不进行判断，x.cuda()语句在没有GPU环境的情况下出错
x = x.cuda() #将张量x转移到GPU中
y = y.cuda() #将张量y转移到GPU中
x + y #在GPU中运算x + y
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-65-4a8d1ea12d04> in <module>
      1 #如果不进行判断，x.cuda()语句在没有GPU环境的情况下出错
----> 2 x = x.cuda() #将张量x转移到GPU中
      3 y = y.cuda() #将张量y转移到GPU中
      4 x + y #在GPU中运算x + y

~\anaconda3\lib\site-packages\torch\cuda\__init__.py in _lazy_init()
    184                 raise RuntimeError(
    185                     "Cannot re-initialize CUDA in forked subprocess. " + msg)
--> 186         _check_driver()
    187         if _cudart is None:
    188             raise AssertionError(

~\anaconda3\lib\site-packages\torch\cuda\__init__.py in _check_driver()
     59 def _check_driver():
     60     if not hasattr(torch._C, '_cuda_isDriverSufficient'):
---> 61         raise AssertionError("Torch not compiled with CUDA enabled")
     62     if not torch._C._cuda_isDriverSufficient():
     63         if torch._C._cuda_getDriverVersion() == 0:

AssertionError: Torch not compiled with CUDA enabled
```

GPU vs CPU



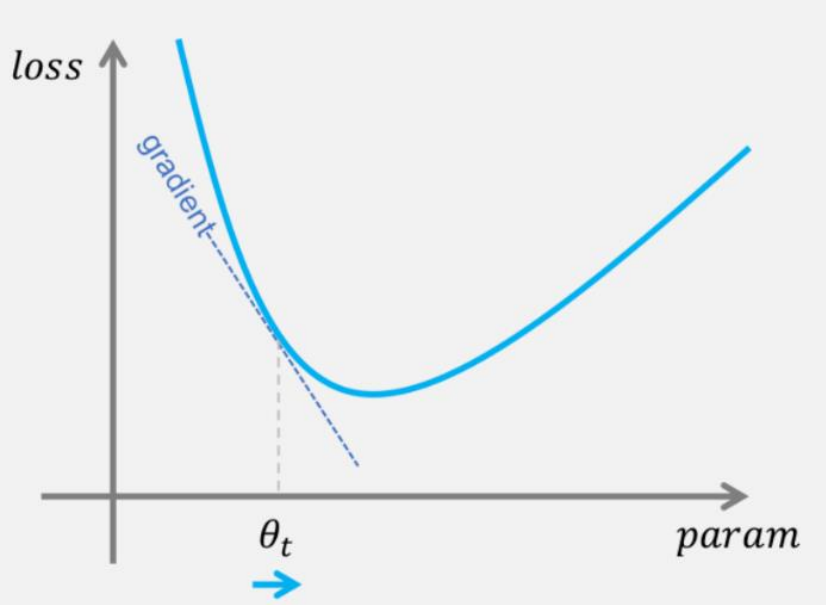
- 测试环境：华为云，1 V100 GPU
- 任务：矩阵乘法 Z = X.mm(Y)
  - X: 10000 * 100000
  - Y: 100000 * 10000
  - Z: 10000 * 10000

- CPU: 总耗时38.3秒
  - 结算结果Z在内存中(Z.device: cpu)
- GPU：耗时1.69秒
  - GPU加载时间
    - X = X.cuda()：把X移入显存
    - Y = Y.cuda()：把Y移入显存
  - 计算时间
  - 结果矩阵Z在第一块显卡中 (cuda:0)

Tensor部分小结

- Pytorch的基本数据结构，在Pytorch中几乎所有的数据都以Tensor的方式存储和操作
- 定义了大量的与线性代数相关的运算和操作（包括求行列式的值、特征根和特征向量），为实现人工智能的算法和模型奠定了基础
- 统一了GPU与CPU接口
- Tensor中文文档： https://pytorch-cn.readthedocs.io/zh/latest/package_references/Tensor/

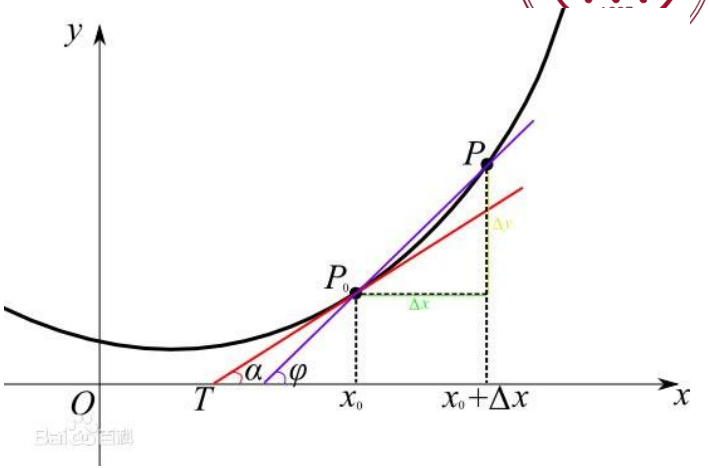
- 接下来的内容：基于Tensor的自动求导AutoGrad



自动求导

函数的导数

- 函数f 在某一点的导数描述了这个函数在这一点附近的变化率，记为f'
- 几何意义
  - 函数$y=f(x)$在x_0点的导数$f'(x_0)$ 表示函数曲线在点$P_0\ (x_0, f(x_0))$处的切线的斜率
- 常用求导数方法
  - 1. 通过原函数求得导函数的表达式
  - 2. 把x_0代入导函数
- 在人工智能（计算机）中求导数
  - 往往只关心一个函数（如：损失函数）在某些指定位置的导数的数值，而不是导函数的表达形式
  - 问题：是否可以跳过导函数的表达式直接求某些位置的导数值?



原函数	导函数
$y=C$ （C为常数）	$y'=0$
$y=a^x$	$y'=a^x\ln a$
$y=e^x$	$y'=e^x$
$y=x^n$	$y'=nx^{n-1}$
$y=\log_a x$	$y'=\frac{1}{x\ln a}$
$y=\ln x$	$y'=\frac{1}{x}$
$y=\sin x$	$y'=\cos x$
$y=\cos x$	$y'=-\sin x$

AutoGrad

- 如：函数 $y = x^2$，求$x = 3$处的导数
  - $\frac{dy}{dx} = 2x$
  - $x = 3 \Rightarrow \frac{dy}{dx} = 2 \times 3 = \mathbf{6}$
- AutoGrad是PyTorch中自动求导机制，能够求得一个给定的函数在某一给定点的导数值
  - 定义输入Tensor x，设置requires_grad=True
  - 定义原函数$y = x^2$
    - 用torch中定义的tensor运算符
  - 自动求导：y.backward()
    - 在y处求导，即dy
  - 在x.grad处获得对x的导数值dy/dx
    - 为x=3处的导数值

```
# 求函数 y= x^2 在x=3时的导数：y'(3) = dy/dx|x=3

x=t.tensor(3.0,requires_grad=True)
y=x.mul(x)

#判断x,y是否是可以求导的
print(x.requires_grad)
print(y.requires_grad)

#求导，通过backward函数来实现
y.backward()

#查看导数，也即所谓的梯度
print(x.grad)
```

```
True
True
tensor(6.)
```

更多的例子

```
# 求y = 1/x
x = t.tensor(0.5, requires_grad=True)
y = t.reciprocal(x)
y.backward()
print(x.grad)

x = t.tensor(1.0, requires_grad=True)
y = t.reciprocal(x)
y.backward()
print(x.grad)

x = t.tensor(2, requires_grad=True)
y = t.reciprocal(x)
y.backward()
print(x.grad)
```

```
tensor(-4.)
tensor(-1.)
tensor(-0.2500)
```

$$y = \frac{1}{x}$$

```
# 求y = 1/\sqrt(x)
x = t.tensor(0.5, requires_grad=True)
y = t.reciprocal( t.sqrt(x))
y.backward()
print(x.grad)

x = t.tensor(1.0, requires_grad=True)
y = t.reciprocal( t.sqrt(x))
y.backward()
print(x.grad)

x = t.tensor(2.0, requires_grad=True)
y = t.reciprocal( t.sqrt(x))
y.backward()
print(x.grad)
```

```
tensor(-1.4142)
tensor(-0.5000)
tensor(-0.1768)
```

$$y = \frac{1}{\sqrt{x}}$$

填空题 3分

设置

函数y=max(0, x)在x=-1处的导数为 [填空1]，在x=1处的导数为 [填空2]， 在x=2处的导数为 [填空3]

正常使用填空题需3.0以上版本雨课堂

作答

练习

- 写代码求函数$y = \max(0, x)$ 在$x = 1$和$x = -1$处的导数
  - $x > 0: y = x, y' = 1$
  - $x < 0: y = 0, y' = 0$

- 提示：最大值函数为tensor.max(,)

```
# 求y = max(0, x)在x=1和x=-1处的导数

#定义x=1.0 和y=max(0, x)
x = t.tensor(1.0, requires_grad=True)
y = t.max(x, t.zeros(1,1))

#调用backward并打印导数
y.backward()
print(x.grad)

# 改变x后需要重新执行y和backward
x = t.tensor(-1.0, requires_grad=True)
y = t.max(x, t.zeros(1,1))
y.backward()
print(x.grad)
```

```
tensor(1.)
tensor(0.)
```

多元函数自动求导

- 类似的调用方式
- 每一个输入参数的.grad变量保存了其对应的导数值

- $f = x^2 + 2y^2 + xy$
  - $\frac{df}{dx} = 2x + y$
  - $\frac{df}{dy} = 4y + x$
- 在(x=1, y=1)处的导数
  - 2 + 1 = 3
  - 4 * 1 + 1 = 5

```
# f(x,y) = x^2 + 2 * y^2 + xy
# 在 (1.0, 1.0)处的导数
x = t.tensor(1.0, requires_grad=True)
y = t.tensor(1.0, requires_grad=True)

f = x.pow(2) + t.tensor(2.0).mul(y.pow(2)) + x.mul(y)
f.backward()
print(x.grad)
print(y.grad)
```

```
tensor(3.)
tensor(5.)
```

多元函数自动求导(续)

- 求$f(x,y,z)=\ln(e^x+e^y+e^z)$在(0, 2, 5)处的导数

- 思考：为什么z的导数最大?

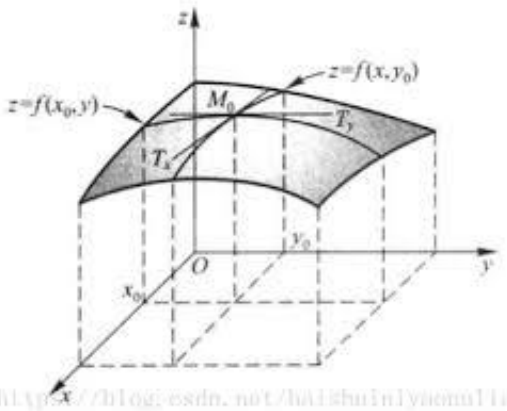
- 课后练习
  - 计算在其它点的导数，找出规律

```
# f(x,y, z) = ln(e^x + e^y + e^z)
# 在 (0.0, 2.0, 5.0)处的导数
x = t.tensor(0.0, requires_grad=True)
y = t.tensor(2.0, requires_grad=True)
z = t.tensor(5.0, requires_grad=True)

f = t.log(t.exp(x) + t.exp(y) + t.exp(z))
f.backward()
print(x.grad)
print(y.grad)
print(z.grad)
```

```
tensor(0.0064)
tensor(0.0471)
tensor(0.9465)
```

以向量为输入的函数的导数

- 函数$f(x)$的输入x可以是向量、矩阵甚至张量(tensor)
  - 如果f的输入是一个标量，$\frac{df}{dx}$也是一个标量
  - 如果f的输入是一个N维向量，$\frac{df}{dx}$也是一个N维的向量
  - 如果f的输入是一个张量，$\frac{df}{dx}$也是一个同样大小的张量
  - 注意：在AutoGrad中我们要求f的返回值为标量



- 举例：$f(\mathbf{x}=[x_1,x_2])=x_1^2+2x_2^2+2x_1x_2+x_1=\mathbf{x}^{\mathrm{T}}\begin{bmatrix}1 & 1\\ 1 & 2\end{bmatrix}\mathbf{x}+\mathbf{x}^{\mathrm{T}}\begin{bmatrix}1\\ 0\end{bmatrix}$
  - $\frac{df}{d\mathbf{x}}=\left[\frac{df}{dx_1},\frac{df}{dx_2}\right]^T=[2x_1+2x_2+1,4x_2+2x_1]^T$
  - $\frac{df}{d\mathbf{x}}=2\begin{bmatrix}1 & 1\\ 1 & 2\end{bmatrix}\mathbf{x}+\begin{bmatrix}1\\ 0\end{bmatrix}$
  - 如果$\mathbf{x}=[1,2]$，导数为[7, 10]

以向量为输入函数的求导

- $f(\mathbf{x}) = \mathbf{x}^{\mathrm{T}} \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \mathbf{x} + \mathbf{x}^{\mathrm{T}} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ 在(1, 2)位置的导数（向量）

- 矩阵乘: mm()
- 矩阵转置: t()

```
# 注意：把x和b定义为一个1*2的矩阵，x.mm()是矩阵乘法
x = t.tensor([[1.0, 2.0]], requires_grad=True)

A = t.tensor([[1.0, 1.0],
              [1.0, 2.0]])
b = t.tensor([[1.0, 0.0]])
print(x, A, b)
print()

f = x.mm(A).mm(x.t()) + x.mm(b.t())
print(f)
print()
f.backward()
print(x.grad)
```

```
tensor([[1., 2.]], requires_grad=True) tensor([[1., 1.],
        [1., 2.]]) tensor([[1., 0.]])

tensor([[14.]], grad_fn=<AddBackward0>)

tensor([[ 7., 10.]])
```

以向量为输入的多元函数求导

- $f(\mathbf{x},\mathbf{y}) = \mathbf{x}^{\mathrm{T}}\begin{bmatrix}1 & 1\\ 1 & 2\end{bmatrix}\mathbf{y} + \mathbf{x}^{\mathrm{T}}\begin{bmatrix}1\\ 0\end{bmatrix}$ 在位置((1,2),(3,4))的导数（向量）

```
# 注意：把x和b定义为一个1*2的矩阵，x.mm()是矩阵乘法
x = t.tensor([[1.0, 2.0]], requires_grad=True)
y = t.tensor([[3.0, 4.0]], requires_grad=True)

A = t.tensor([[1.0, 1.0],
              [1.0, 2.0]])
b = t.tensor([[1.0, 0.0]])
print(x, y, A, b)
print()

f = x.mm(A).mm(y.t()) + x.mm(b.t())
print(f)
print()
f.backward()
print(x.grad)
print(y.grad)
```

```
tensor([[1., 2.]], requires_grad=True) tensor([[3., 4.]], requires_grad=True) tensor([[1., 1.],
        [1., 2.]]) tensor([[1., 0.]])

tensor([[30.]], grad_fn=<AddBackward0>)

tensor([[ 8., 11.]])
tensor([[3., 5.]])
```

基于PyTorch的线性回归梯度计算

```
import numpy as np
import torch as t
import matplotlib.pyplot as plt
```

```
def __calc_gradient(self, x, y):
    '''
    类的方法：计算对w的梯度
    '''
#numpy version
    N = x.shape[0]
     diff = (x.dot(self.w) - y)
     grad = x.T.dot(diff)
     d_w = (2 * grad) / N

#torch version
    x_tensor = t.tensor(x)
    y_tensor = t.tensor(y)
    w_tensor = t.tensor(self.w, requires_grad=True)
    loss = t.sum(t.pow(x_tensor.mm(w_tensor) - y_tensor, 2)) / N
    loss.backward()
    d_w = w_tensor.grad.numpy()
    return d_w
```

- 用PyTorch实现的要点
  - 引入torch包
  - 把numpy变量转为Tensor变量，注意只对w求导
  - 用pyTorch中的方法实现损失函数
  - 调用backward()
  - 返回对应的梯度值
- 优点
  - 不用显式求解和实现梯度公式
  - 在一般情况下，梯度函数比损失函数更加难以实现

总结

- Pytorch是简单易学的人工智能平台
- 两个主要的组成部分
  - Tensor
  - AutoGrad
- 本次课
  - 以数学运算为背景，介绍了如何利用Pytorch进行简单计算和求导
- 下次课
  - Pytorch在深度学习中的应用介绍

谢谢！