同濟大學
TONGJI UNIVERSITY

王睿智 ruizhiwang@tongji.edu.cn

人工智能技术与应用

第五章 深度学习

PyTorch

- PyTorch是一个开源的机器学习库。基于Torch，由Facebook人工智能研究小组开发。

是目前最受欢迎的深度学习框架之一。

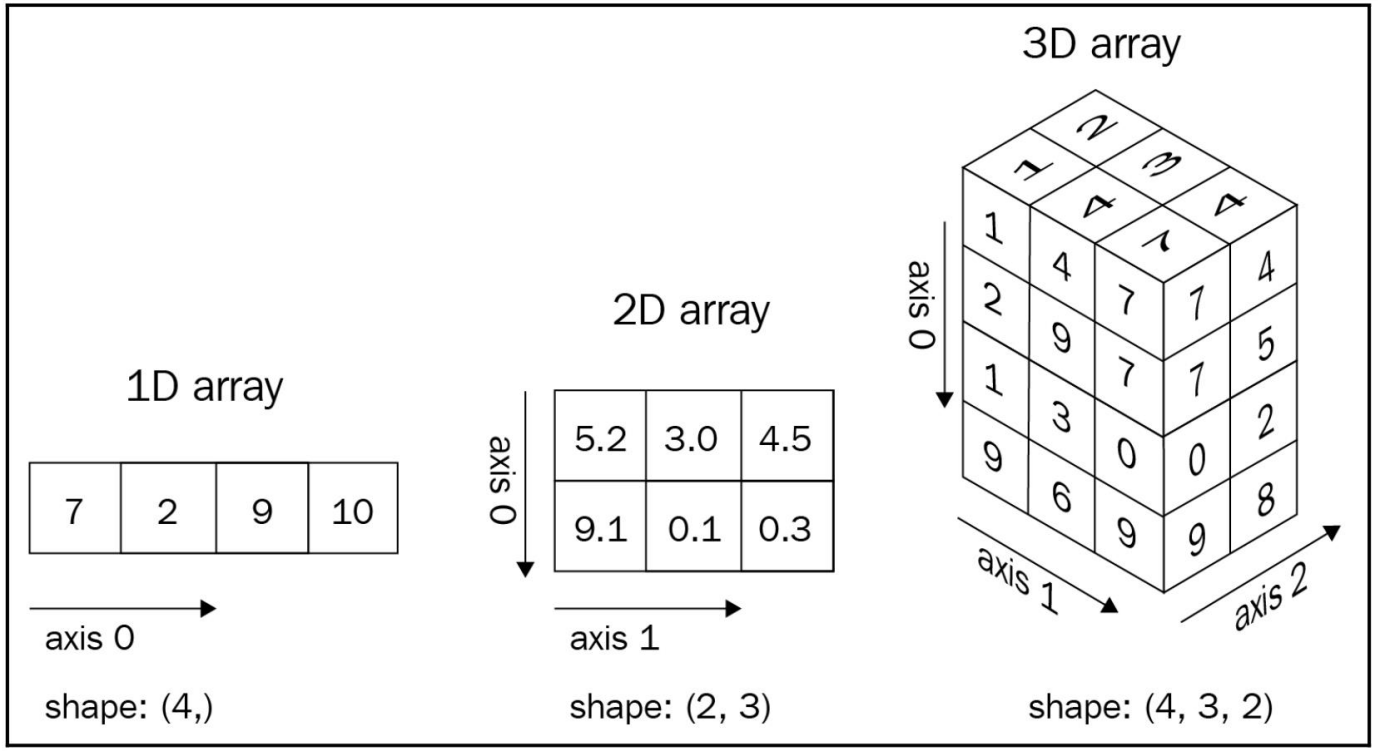
  - PyTorch在CPU和GPU上都可运行。还支持移动端部署。
  - PyTorch使用动态计算图。动态计算图比静态图更灵活、易于调试。
  - PyTorch提供了包括管理文本、图像和音频（torchtext, torchvision和torchaudio）模块，以及诸如ResNet等流行架构的内置实现。

- PyTorch文档： https://pytorch.org/docs/stable/index.html
- 下载安装：https://pytorch.org/get-started/locally/
- **国内镜像安装(CPU)**：pip install torch –i https://pypi.tuna.tsinghua.edu.cn/simple

  pip install torchvision –i https://pypi.tuna.tsinghua.edu.cn/simple

5.2 PyTorch深度学习基础 | 5.2.1 张量

- **张量**是PyTorch的基本数据类型。
- **张量是一个多维矩阵**，类似与NumPy的ndarrays：
  - **标量**可表示为零维张量；
  - **向量**可表示为一维张量；
  - **二维矩阵**可表示为二维张量；
  - **多维矩阵**可表示为多维张量。



5.2.1 张量

一幅彩色图像可看作像素值的三维张量，因为它由height × width × 3像素组成，其中3个通道对应于RGB通道。

一幅灰度图像可看成二维张量，因为它由height × width 像素组成。

除可作为图像等的基本数据结构外，张量还可用来初始化NN不同层的权重。

5.2.1 张量

1. 创建张量

- 由列表、元组、NumPy数组等创建张量：torch.tensor()、torch.from_numpy()函数。
- 内置函数初始化张量对象：torch.zeros()、torch.randn()等函数。与NumPy数组类似
- 查看张量形状和元素类型：张量名.shape，张量名.dtype 属性。与NumPy数组类似

2. 张量操作

- 张量转置：张量名.transpose()
- 交换维度：张量名.permute()
- 张量变形：张量名.reshape()重塑、张量名.squeeze()压缩、使用None索引添维度
- 更改元素/设备类型：张量名.to()
- 拆分、连接张量：torch.split()拆分、torch.cat()连接

3. 张量运算

- 数学运算，特别是元素乘法、矩阵乘法、简单统计、张量的范数等。
- 张量的自动梯度，张量名. backward()

1. 创建张量 | torch.tensor()

例 调torch.tensor()由列表创建张量，并查看其形状和数据类型。

```
import torch
import numpy as np
x = torch.tensor([1,2])
y = torch.tensor(np.array([[1,2],[3,4]]))
```

```
print(x.shape)
print(y.shape)
print(y.dtype)
```

```
torch.Size([2])
torch.Size([2, 2])
torch.int32
```

```
z = torch.tensor([False,1,2.0])
print(z)
```

```
tensor([0., 1., 2.])
```

说明：

① torch.tensor(data): data可以是列表、元组、NumPy数组等。

② x.shape：获取张量x的形状。

③张量内所有元素的数据类型是相同的。若一个张量包含不同数据类型的数据(如 z)，整个张量被强制转换为一种最为通用的数据类型。

1. 创建张量 | 利用内置函数


```
# 生成一个张量对象，有2行3列，填充0
x = torch.zeros((2,3))
print(x)
```

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

```
# 生成输入值服从正态分布的2行3列
x = torch.randn((2,3))
print(x)
```

```
tensor([[-0.5255, -1.2915, -0.4492],
        [ 0.5951,  0.2402,  1.2151]])
```

```
# 生成输入值服从[0,1)均匀分布的2行3列
x = torch.rand(2,3)
print(x)
```

```
tensor([[0.9437, 0.8465, 0.3539],
        [0.5110, 0.4352, 0.0089]])
```

2. 张量操作 | 张量转置 与 交换维度

张量转置： torch.transpose(tensor, dim0, dim1) 或 张量名.transpose(dim0,dim1)

dim0 和 dim1交换，返回转置后的结果。

```
x = torch.rand(1,8,5) # 形状为(1,8,5)
x_tr = torch.transpose(x,1,2)
print(x.shape,'-->',x_tr.shape)
```

```
torch.Size([1, 8, 5]) --> torch.Size([1, 5, 8])
```

交换维度： torch.permute(tensor, dims) 或 张量名.permute(dims)

返回原始张量的视图，其维度已按dims排列,dims是int型元组。

```
x_pm = torch.permute(x,(1,0,2))
print(x.shape,'-->',x_pm.shape)
```

```
torch.Size([1, 8, 5]) --> torch.Size([8, 1, 5])
```

2. 张量操作 | 张量变形

重塑：张量名.reshape(*shape) 返回一个张量，其元素个数和类型与原张量相同，但形状是由参数shape指定的形状，shape是int型元组

```
y = torch.zeros(30)    # 形状(30)
y = y.reshape(5,6,1)   # 形状(5,6,1)
print(y.shape)
```

```
torch.Size([5, 6, 1])
```

压缩：张量名.squeeze(dim)：删除由dim指定的维度，只适用于要删除的维度中只有一项的情况，即该维大小维1

```
z = y.squeeze(2)  # 第3个维度被删除
print(z.shape)
```

```
torch.Size([5, 6])
```

添新维：利用None索引，添加新的维度。

```
z1,z2,z3 =  z[:,:,None],z[:,None,:],z[None]
print(z1.shape, z2.shape,z3.shape)
```

```
torch.Size([5, 6, 1]) torch.Size([5, 1, 6]) torch.Size([1, 5, 6])
```

2. 张量操作 | 改变元素/设备类型

tensor.to()方法作用一：数据类型转换

```
x = torch.zeros(30)
x1 = x.to(torch.int64)
print(x.dtype,'-->',x1.dtype)
```

```
torch.float32 --> torch.int64
```

tensor.to()方法作用二：注册设备，以便利用GPU执行张量操作

```
y = torch.rand(6400,6400)
device = 'cuda' if torch.cuda.is_available() else 'cpu'
y = y.to(device) # 将张量对象注册到设备中。
print(y.device)
```

定义存储张量对象的设备。
若没有GPU, 设备将是CPU。

```
cpu
```

说明：注册张量对象意味着在设备中存储信息。

2. 张量操作 | 连接、拆分张量

连接张量：torch.cat(tensors, dim=0) 将给定的张量序列在指定的维度上连接起来。

参数：tensors - 相同类型的张量序列。提供的非空张量必须具有相同的形状，但 cat 维度除外。

dim - int，可选参数，张量被连接的维度。

```
A = torch.ones(3)
B = torch.zeros(3)
C = torch.cat([A,B],axis = 0)
print('沿axis=0连接:', C.shape)
print(C)

D = torch.cat([A[:,None],B[:,None]],axis=1)
print(A[:,None])
print('沿axis=1连接:', D.shape)
print(D)
```

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

```
沿axis=0连接: torch.Size([6])
tensor([1., 1., 1., 0., 0., 0.])
```

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

```
沿axis=1连接: torch.Size([3, 2])
tensor([[1., 0.],
        [1., 0.],
        [1., 0.]])
```

2. 张量操作 | 连接、拆分张量

拆分张量：torch.split(tensor,split_size_or_sections,dim=0) 将张量拆分成块，返回张量的元组。

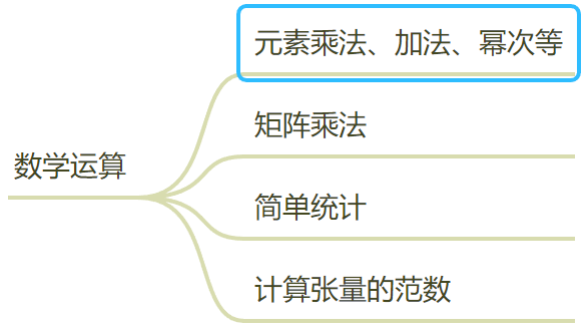
参数：split_size_or_sections - int 或 list（int), 单个块的大小或 每个数据块的大小列表。

若是int，则拆分成指定块大小的块，最后一个块可能小于指定大小。

dim - int,分割张量的维度。

```
t = torch.tensor([1,1,2,2,3,3])
print(t)
t_split1 = torch.split(t,split_size_or_sections = 3) # 指定块大小
print(t_split1)
t_split2 = torch.split(t,split_size_or_sections = [2,3,1])
print(t_split2)
```

3. 张量运算 | 数学运算

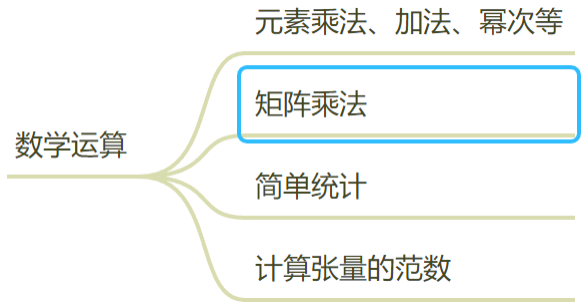


元素乘法、加法、幂次等，是对应元素的运算。

```
x1 = torch.tensor([[1,2,3],[4,5,6]])
x2 = torch.tensor([[3,1,0],[3,1,0]])
y = torch.multiply(x1,x2)    # 或 x1 * x2
print("x1和x2的元素乘法:\n",y)
print("x1和x2的元素加法:\n", x1 + x2)
print("将x1的所有元素乘以10:\n", x1 * 10)
print("将x1的所有元素加10:\n", x1 + 10)
print('对x1的每个元素平方:\n',x1.pow(2))
```

```
x1和x2的元素乘法:
 tensor([[ 3,  2,  0],
        [12,  5,  0]])
x1和x2的元素加法:
 tensor([[4, 3, 3],
        [7, 6, 6]])
将x1的所有元素乘以10:
 tensor([[10, 20, 30],
        [40, 50, 60]])
将x1的所有元素加10:
 tensor([[11, 12, 13],
        [14, 15, 16]])
对x1的所有元素平方:
 tensor([[ 1,  4,  9],
        [16, 25, 36]])
```

3. 张量运算 | 数学运算

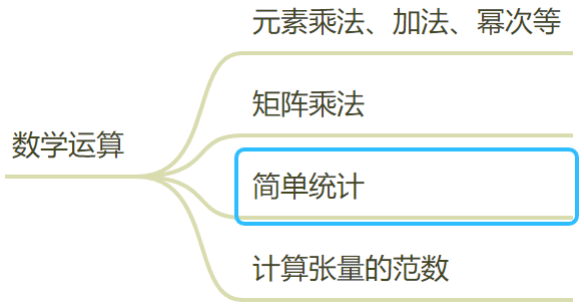


矩阵乘法。如，神经网络常用的输入数据与权重的矩阵乘法

```
x = torch.tensor([[1,2,3],[4,5,6]])
w = torch.tensor([3,1,0])
print("x和w的矩阵乘法:",torch.matmul(x,w))
# 或用@
print("x和w的矩阵乘法:", x@w)
```

```
x和w的矩阵乘法: tensor([ 5, 17])
x和w的矩阵乘法: tensor([ 5, 17])
```

3. 张量运算 | 数学运算

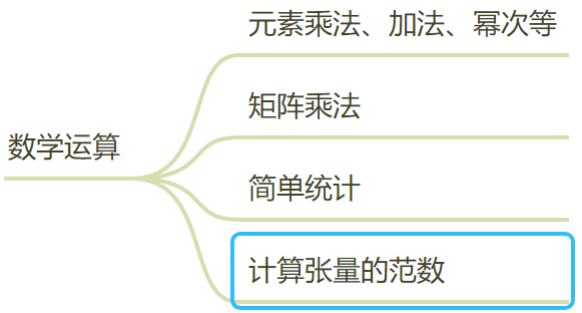


简单统计：沿某轴的均值、和，标准差等

```
x = torch.tensor([[4.,1.],[-4.,1.]])
m = x.mean(axis = 0)
print('沿0轴的均:',m )
s = x.pow(2).sum()
print('x的所有元素平方和:',s)
std = torch.std(x,axis=0)
print('沿0轴的标准差:',std)
```

```
沿0轴的均: tensor([0., 1.])
x的元素平方和: tensor(34.)
沿1轴的标准差: tensor([2.1213, 3.5355])
```

3. 张量运算 | 数学运算



计算张量的范数：利用torch.linalg.norm()函数

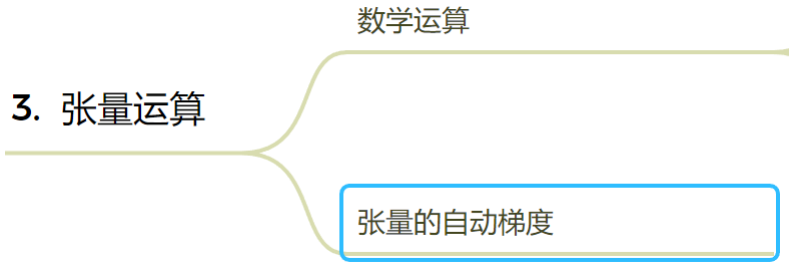
```
t1 = 2 * torch.rand(5,2) - 1  # 取值[-1,1)，形状为(5,2)
norm_t1 = torch.linalg.norm(t1,ord=2,dim=1) # 计算L2范数
print(norm_t1)
```

```
tensor([0.9543, 1.0209, 0.6177, 1.1800, 0.7683])
```

```
# 与下列计算结果比较
_t1 = np.sqrt(t1.pow(2).sum(axis=1))
print(_t1)
```

```
tensor([0.9543, 1.0209, 0.6177, 1.1800, 0.7683])
```

3. 张量运算 | 张量对象的自动梯度



梯度计算在更新NN权重中起关键作用。

PyTorch的张量自带计算梯度的功能。调用<张量名>.backward()方法来计算梯度。

```
# 创建一个张量，指定要为该张量计算梯度
x = torch.tensor([[2.,-1.],[1.,1.]],requires_grad = True)
```

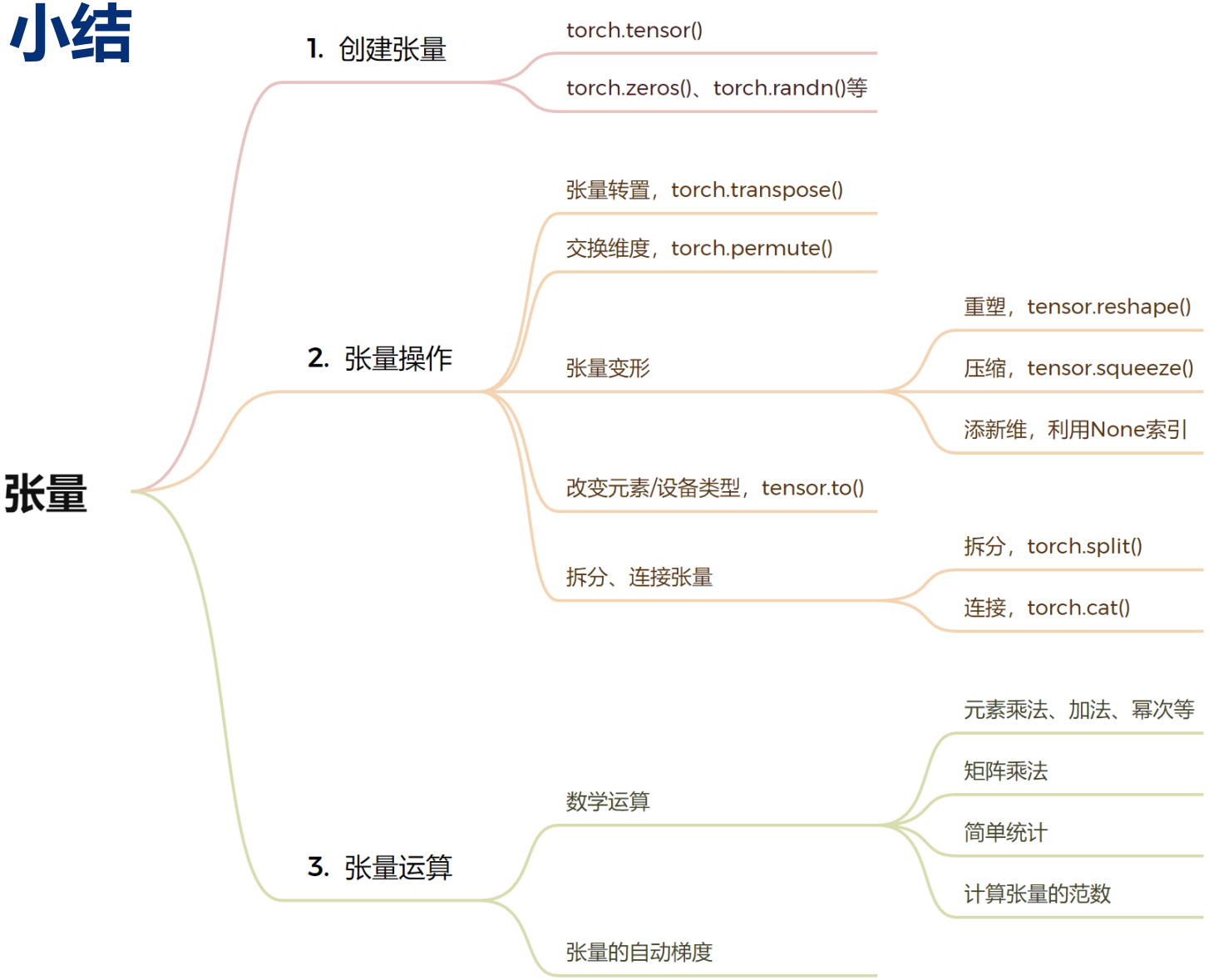
```
out = x.pow(2).sum() # 输出out是所有输入的平方和 out = ∑_{i=1}^{4} x_i^2
print(out)
```

```
tensor(7., grad_fn=<SumBackward0>)
```

```
out.backward()   # 对out调用backward()
print(x.grad)    # 得到out关于x的梯度
```

```
tensor([[ 4., -2.],
        [ 2.,  2.]])
```

小结



深度学习为什么用张量?

1. 同一次迭代中，各个权重更新互不影响，因此，每个权重更新可分别由不同的内核并行完成。
2. CPU一般≤64个内核，而GPU由数千个内核组成，并行计算效率高。Torch张量对象被优化为与GPU一起工作，其运算效率得到极大提高。

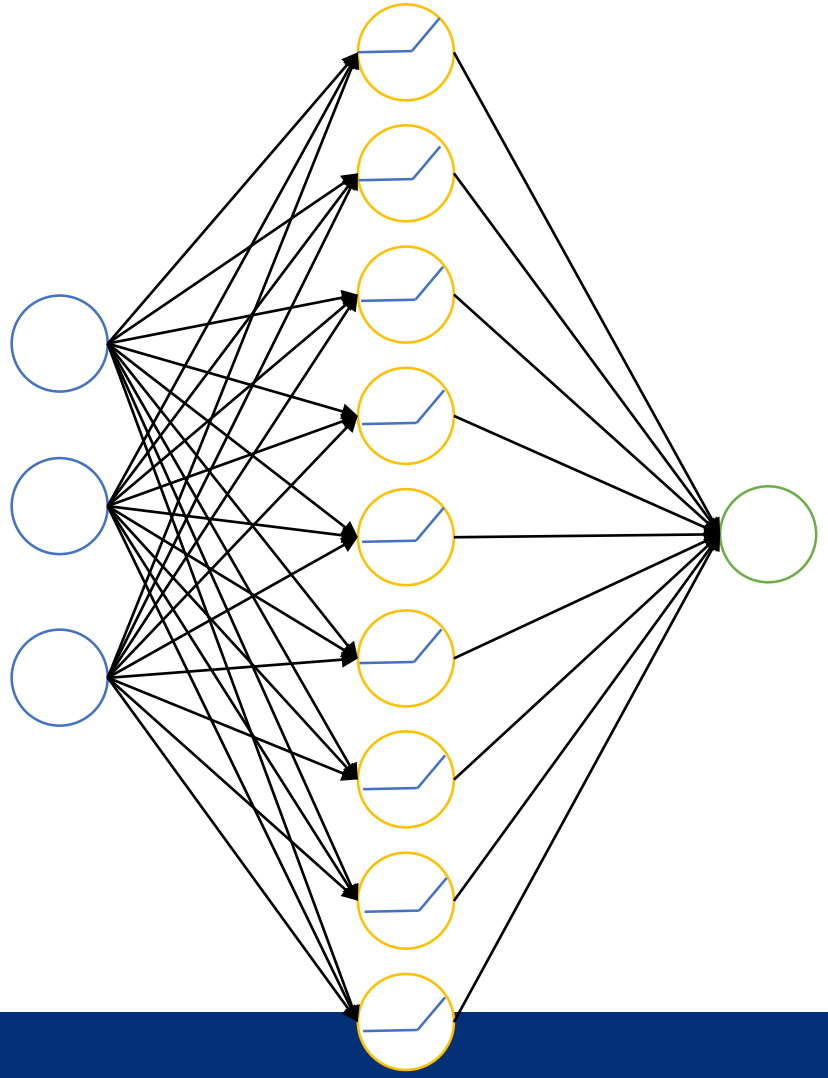
5.2.2 使用PyTorch构建神经网络

神经网络的组件:

- 隐藏层的数量
- 隐藏层中的单元数量
- 各个层中的激活函数
- 实现最优化的损失函数
- 与神经网络有关的学习率
- 用于构建神经网络的数据的批大小
- 前向传播和反向传播的轮次

例5.3 一个简单神经网络

构建一个含1个隐藏层的神经网络，模拟三个数字相加。



网络架构：

1. 定义一个线性层(函数)，计算由输入 (3个) 到隐藏层 (8个) 的线性和。
2. 定义隐藏层的激活函数：ReLU()
3. 定义一个线性层(函数)，计算由隐藏层 (8个) 到输出 (1个) 的线性和。

代码分步解析

1．给定输入值(x)和输出值(y)，y中各个元素值是x中各个元素值之和。

```
import torch
x = [[1,2,3],[3,4,5],[5,6,7],[7,8,9]]
y = [[6],[12],[18],[24]]
```

2．将输入列表转换为张量对象，再将元素对象转换为float

```
X = torch.tensor(x).float()
Y = torch.tensor(y).float()
```

```
# 若有GPU,将输入和输出注册到GPU；否则用CPU
device = 'cuda' if torch.cuda.is_available() else 'cpu'
X = X.to(device)
Y = Y.to(device)
```

为什么要在训练中将整数输入转为浮点数?

因为梯度下降法等优化时要求数据是连续的。此外，浮点数具有更高的精度，可以减少在反向传播过程中由于数值舍入误差导致的累积误差。

代码分步解析

3.定义神经网络架构

```
import torch.nn as nn

class MyNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Linear(3,8)
        self.hidden_layer_activation = nn.ReLU()
        self.hidden_to_output_layer = nn.Linear(8,1)
    def forward(self, x):
        x = self.input_to_hidden_layer(x)
        x = self.hidden_layer_activation(x)
        x = self.hidden_to_output_layer(x)
        return x
```

创建一个类(MyNN), 用它构建神经网络架构。

① torch.nn，包含构建神经网络模型的常用函数

② nn.Module 是所有神经网络的基类。自定义网络需继承该类，填写__init__()和forward()方法。

③ __init__()，初始化神经网络的所有组件。

④ forward()，定义计算步骤，将__init__()中已定义的组件连接起来。

代码分步解析

4.定义一个MyNN实例

```
mynet = MyNN().to(device)  # 定义一个MyNN实例
```

```
# 查看该网络的初始权重
print(mynet.input_to_hidden_layer.weight)
print(mynet.hidden_to_output_layer.weight)
```

```
Parameter containing:
tensor([[ 0.2328, -0.4363, -0.2729],
        [ 0.1076,  0.3382,  0.1827],
        [ 0.5264, -0.1423, -0.2411],
        [-0.5068,  0.4302,  0.5141],
        [-0.3370,  0.0530, -0.4750],
        [ 0.5617, -0.3761, -0.5344],
        [ 0.3548, -0.3608, -0.2831],
        [ 0.4582, -0.3443, -0.2039]], requires_grad=True)
```

```
Parameter containing:
tensor([[ 0.0010,  0.3098, -0.0779,  0.1561, -0.0024, -0.0406, -0.2317,  0.2370]],
       requires_grad=True)
```

5.定义损失函数

```
loss_func = nn.MSELoss() # 因预测值是连续值，这里用均方误差
```

```
_Y = mynet(X)                  # 给定输入X，计算预测值_Y
loss_value = loss_func(_Y,Y) # 计算损失值
print(loss_value)
```

```
tensor(216.8368, grad_fn=<MseLossBackward0>)
```

其他重要的损失函数如：

- CrossEntropyLoss (多分类)
- BCELoss (二分类的二元交叉熵损失)

代码分步解析

6. 设置优化器

```
from torch.optim import SGD
opt = SGD(mynet.parameters(), lr = 0.001) # 这里采用随机梯度下降
```

7. 多个轮次训练

```
# 这里轮次设为50
loss_history = []
for _ in range(50):
    opt.zero_grad()  # ①
    loss_value = loss_func(mynet(X),Y) # ②
    loss_value.backward() # ③
    opt.step() # ④
    loss_history.append(loss_value.item())
```

每轮要执行的步骤：

① 下一轮之前，**刷新**上一轮的**梯度**

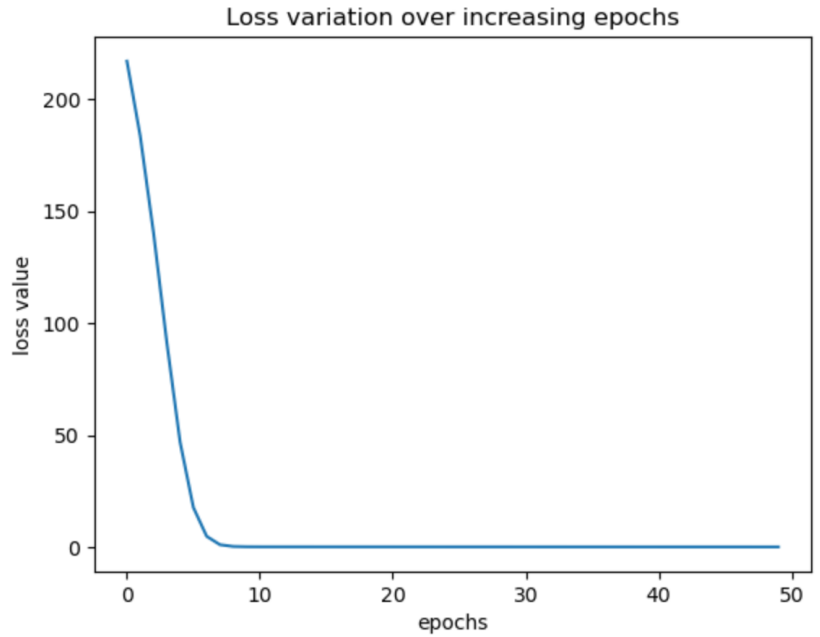
② **计算**给定输入和输出对应的**损失**

③ 执行反向传播**计算**每个参数对应的**梯度**

④ 根据梯度和学习率**更新权重**

代码分步解析

```
# 绘制损失值随轮次增加而变化的曲线
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(loss_history)
plt.title('Loss variation over increasing epochs')
plt.xlabel('epochs')
plt.ylabel('loss value')
```



损失值随着轮数的增加而减少。

这里，根据数据集中所有数据点计算损失，更新神经网络的权重。
神经网络更常用**小批量**计算损失、更新权重！

关键步骤

1. 输入和目标张量转换为float
2. 定义神经网络架构类，生成一个NN实例
3. 定义损失函数，设置优化器
4. 多轮次训练
   - ① 刷新梯度
   - ② 计算损失
   - ③ 反向传播（计算梯度）
   - ④ 更新权重

完整代码

```
# 1. 给定输入值(x)和输出值(y)
import torch
x = [[1,2,3],[3,4,5],[5,6,7],[7,8,9]]
y = [[6],[12],[18],[24]]

# 2. 将输入列表转为张量对象，再将元素类型转换为float
X = torch.tensor(x).float()
Y = torch.tensor(y).float()
# 若有GPU,将输入和输出注册到GPU；否则用CPU
device = 'cuda' if torch.cuda.is_available() else 'cpu'
X = X.to(device)
Y = Y.to(device)

# 3.定义神经网络架构
import torch.nn as nn
class MyNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Linear(3,8)
        self.hidden_layer_activation = nn.ReLU()
        self.hidden_to_output_layer = nn.Linear(8,1)
    def forward(self, x):
        x = self.input_to_hidden_layer(x)
        x = self.hidden_layer_activation(x)
        x = self.hidden_to_output_layer(x)
        return x

# 4.定义MyNN的一个实例
mynet = MyNN().to(device) # 定义一个MyNN实例

# 5.定义损失函数
loss_func = nn.MSELoss() # 这里用均方误差
_Y = mynet(X) # 给定输入X，计算预测值_Y
loss_value = loss_func(_Y,Y) # 计算损失值
print(loss_value)

# 6. 设置优化器
from torch.optim import SGD
opt = SGD(mynet.parameters(), lr = 0.001)

# 7. 多个轮次训练, 这里轮次设为50
loss_history = []
for _ in range(50):
    opt.zero_grad() # 刷新梯度
    loss_value = loss_func(mynet(X),Y) # 计算损失
    loss_value.backward()  # 反向传播
    opt.step()  # 更新权重
    loss_history.append(loss_value.item())

# 绘制损失值随轮次增加而变化的曲线
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(loss_history)
plt.title('Loss variation over increasing epochs')
plt.xlabel('epochs')
plt.ylabel('loss value')
```

PyTorch 的神经网络模块

- torch.nn，包含构建神经网络模型的常用函数。
  - **nn.Linear(in_features, out_features)** ： **全连接层**，对输入数据进行线性变换。

    参数 in_features、out_features 分别是输入特征数目、输出特征数目。
  - **nn.ReLU()**： **激活函数**，对输入进行逐元素relu函数变换，$ReLU(x) = max(0, x)$
  - **nn.MSELoss()**：**损失函数**，回归任务常用的均方误差（MSE）损失。
- torch.nn.Module 是所有神经网络的基类。自定义网络需继承该类，填写__init__和forward方法。
  - **__init__(self)**：初始化神经网络的所有组件。必须调用super().__init__()来确保类继承nn.Module，以便利用预制功能。
  - **forward(self, * input)**：定义每次训练和预测时执行的计算步骤，它将__init__中已定义的组件连接起来。必须使用forward作为函数名。
- torch.nn.optim，提供多种优化器。如SGD, Adam

5.2.2 使用PyTorch构建神经网络 | 分批加载数据

背景：

- 在训练深度神经网络模型时，通常使用**迭代优化**算法（如梯度下降算法）多轮训练，需**反复加载**数据集。
- 若训练集不大且可加载到内存中，就直接用来训练模型。但通常数据集都太大，无法加载到计算机内存。需要从硬盘中分块加载数据，即**分批加载数据**。**批大小**是指用于计算损失值、更新权重的数据点数量。
- Pytorch提供了**数据集类**和**数据加载器类**，来加载数据并转换成可进行分批训练的格式。两个类在torch.utils.data下。

数据集、数据加载器

torch.utils.data下提供的**数据集类**和**数据加载器类**，组织数据成可进行**分批迭代**的格式。

- **Dataset** **数据集**抽象类

  要求自定义数据集类包含__init__()、__getitem__()和__len__()方法，以供数据加载器使用。

- **TensorDataset**(*data_tensor, target_tensor*)

  将样本**数据张量**和**目标张量**合成一组，包装成一个数据集。

- **DataLoader**(*dataset, batch_size*=1, ...) **数据加载器类**

  返回一个对象，可遍历输入**数据集**，依次向神经网络加载**批量**数据。

例5.4　可批量训练的神经网络

改进例5.3，要点如下：

（1）用一个数据集，存储数据样本和目标，提供键到数据样本映射。

（2）用一个数据加载器，依次向神经网络加载批量数据。

1.导入用于加载数据和处理数据集的方法

```
import torch
import torch.nn as nn
from torch.optim import SGD
from torch.utils.data import DataLoader,TensorDataset,Dataset
```

2. 导入数据，将数据转为浮点数，并注册到设备

同例5.3中步骤1和2，这里略

代码分步解析

3. 实例化一个数据集类MyDataset

```
class MyDataset(Dataset):
    def __init__(self,x,y):
        self.x = torch.tensor(x).float()
        self.y = torch.tensor(y).float()
    def __len__(self):
        return len(self.x)
    def __getitem__(self, ix):
        return self.x[ix], self.y[ix]
```

4. 创建已定义类MyDataset的实例

```
ds = MyDataset(X, Y)
```

可用TensorDataset简化：

```
ds = TensorDataset(X,Y)
```

- __init__()：初始化方法，如读取现有数组、加载文件，过滤数据等。
- __getitem__()：返回给定索引对应的样本。
- __len__()：返回数据集的大小。

5. 定义数据加载器对象,用于依次加载批量数据

```
dl = DataLoader(ds, batch_size=2,
                shuffle=True)
```

代码分步解析

6. 定义神经网络类

```
class MyNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Linear(3,8)
        self.hidden_layer_activation = nn.ReLU()
        self.hidden_to_output_layer = nn.Linear(8,1)
    def forward(self, x):
        x = self.input_to_hidden_layer(x)
        x = self.hidden_layer_activation(x)
        x = self.hidden_to_output_layer(x)
        return x
```

代码分步解析

7.定义模型对象、损失函数和优化器

```
mynet = MyNN().to(device) # MyNN的一个实例
loss_func = nn.MSELoss()
opt = SGD(mynet.parameters(), lr = 0.001)
```

8. 循环遍历各批数据点以最小化损失值

```
loss_history = []
for _ in range(50):
    for data in dl:
        x, y = data # 每次2个数据点
        opt.zero_grad()
        loss_value = loss_func(mynet(x),y)
        loss_value.backward()
        opt.step()
        loss_history.append(loss_value)
```

9. 预测新数据

```
new_x = [[10,11,12]] # 新数据点
# 转换为浮点张量并注册到设备
val_x = torch.tensor(new_x).float().to(device)

mynet(val_x) # 用训练好的模型预测新数据
```

5.2.2 使用PyTorch构建神经网络 | 自定义损失函数

例5.5 自定义损失函数

修改例5.4, 自定义均方差函数，并将该函数作为损失函数。

1.导入数据，构建数据集和数据加载器

```
x = [[1,2,3],[3,4,5],[5,6,7],[7,8,9]]
y = [[6],[12],[18],[24]]
X = torch.tensor(x).float()
Y = torch.tensor(y).float()

device = 'cuda' if torch.cuda.is_available() else 'cpu'
X = X.to(device)
Y = Y.to(device)

ds = TensorDataset(X,Y)
dl = DataLoader(ds, batch_size=2, shuffle=True)
```

例5.5

2. 定义一个神经网络，创建模型对象

```
class MyNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Linear(3,8)
        self.hidden_layer_activation = nn.ReLU()
        self.hidden_to_output_layer = nn.Linear(8,1)
    def forward(self, x):
        x = self.input_to_hidden_layer(x)
        x = self.hidden_layer_activation(x)
        x = self.hidden_to_output_layer(x)
        return x
mynet = MyNN().to(device)
```

例5.5

3. 自定义损失函数,并调用该函数

```
def my_mean_squared_error(_y, y):
    loss = (_y-y)**2
    loss = loss.mean()
    return loss

print(my_mean_squared_error(mynet(X),Y))
```

```
loss_func = nn.MSELoss() # PyTorch提供的均方误差损失函数
loss_value = loss_func(mynet(X),Y)
print(loss_value)
```

5.2.2 使用PyTorch构建神经网络 | 获取中间层的值

方法：直接调用训练好的网络中间层，将输入张量X作为参数。

例5.5续 查看中间层的值

```
input_to_hidden = mynet.input_to_hidden_layer(X)
print(input_to_hidden)
hidden_activation = mynet.hidden_layer_activation(input_to_hidden)
print(hidden_activation)
```

```
tensor([[-7.2171e-01,  1.8732e+00, -1.7521e+00, -1.5268e+00, -2.1947e+00,
          3.3219e+00, -4.8157e-01,  4.8190e-03],
        [-1.2766e+00,  2.3441e+00, -4.1325e+00, -2.2054e+00, -4.1436e+00,
          6.0953e+00, -4.7029e-01, -8.1965e-02],
        [-1.8315e+00,  2.8150e+00, -6.5130e+00, -2.8841e+00, -6.0925e+00,
          8.8687e+00, -4.5901e-01, -1.6875e-01],
        [-2.3863e+00,  3.2859e+00, -8.8934e+00, -3.5628e+00, -8.0414e+00,
          1.1642e+01, -4.4773e-01, -2.5553e-01]], grad_fn=<AddmmBackward0>)
tensor([[0.0000e+00, 1.8732e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 3.3219e+00,
         0.0000e+00, 4.8190e-03],
        [0.0000e+00, 2.3441e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 6.0953e+00,
         0.0000e+00, 0.0000e+00],
        [0.0000e+00, 2.8150e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 8.8687e+00,
         0.0000e+00, 0.0000e+00],
        [0.0000e+00, 3.2859e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 1.1642e+01,
         0.0000e+00, 0.0000e+00]], grad_fn=<ReluBackward0>)
```

注意：

必须在调用hidden_layer_activation前调用input_to_hidden_layer，因为input_to_hidden_layer的输出是hidden_layer_activation层的输入。

5.2.3 使用序贯法构建神经网络

- **torch.nn.Squential**，可简化定义一个神经网络架构。
- **Squential**是一个顺序容器类，可以将组件依次添加到其中。

```
model = nn.Sequential(
    nn.Linear(3, 8),
    nn.ReLU(),
    nn.Linear(8, 1)
    )
```

例5.6 利用序贯方法构建一个神经网络

本例中，用Squential类取代例5.4中手工定义的神经网络架构类。

```
import torch
import torch.nn as nn
import numpy as np
from torch.utils.data import TensorDataset, DataLoader
from torch.optim import SGD
```

1.导入数据，构建数据集和数据加载器

```
torch.manual_seed(2) # 设置随机数种子
device = 'cuda' if torch.cuda.is_available() else 'cpu'

x = [[1,2,3],[3,4,5],[5,6,7],[7,8,9]]
y = [[6],[12],[18],[24]]
X = torch.tensor(x).float().to(device)
Y = torch.tensor(y).float().to(device)

ds = TensorDataset(X,Y)
dl = DataLoader(ds, batch_size=2, shuffle=True)
```

例5.6

2. 用torch.nn.Sequential方法定义模型架构

```
model = nn.Sequential(
    nn.Linear(3, 8),
    nn.ReLU(),
    nn.Linear(8, 1)
    ).to(device)
```

3.定义损失函数和优化器

```
loss_func = nn.MSELoss()
opt = SGD(model.parameters(), lr = 0.001)
```

4. 训练模型

```
loss_history = []
for _ in range(50):
    for ix, iy in dl:
        opt.zero_grad()
        loss_value = loss_func(model(ix),iy)
        loss_value.backward()
        opt.step()
        loss_history.append(loss_value)
```

5. 预测新数据

```
val = [[8,9,23],[5,10,11],[1.5,2.5,8.5]]
val = torch.tensor(val).float()
_y = model(val.to(device))
print(_y)
```

```
tensor([[41.5605],
        [25.6688],
        [13.0353]], grad_fn=<AddmmBackward0>)
```

```
print(val.sum(-1))  # 目标的真实值
```

```
tensor([40.0000, 26.0000, 12.5000])
```

保存并加载PyTorch模型

- **保存模型：**保存训练好的模型参数名和值。
  - **<模型对象名>.state_dict()**

    返回一个字典，对应于模型相应的参数名(键) 和值（权重和偏置）。
  - **torch.save(<模型对象名>.state_dict(), 文件名)**

    将训练好的模型（参数名和值）保存到指定名称的磁盘文件中。

- **加载模型**

① 首先初始化一个网络结构与训练时相同的新模型

② 然后加载模型参数文件，返回参数键值字典

③ 加载参数键值字典到新模型。

例5.6续 将模型保存到当前文件夹的mymodel.pth文件中。然后重新加载模型。

```
# 保存模型
save_path = 'mymodel.pth'
torch.save(model.state_dict(), save_path)
```

```
# 加载模型
model_new = nn.Sequential(
    nn.Linear(3, 8),
    nn.ReLU(),
    nn.Linear(8, 1)
    ).to(device)  # 初始化一个同类新模型

dict = torch.load('mymodel.pth') #加载模型文件
model.load_state_dict(dict) #加载模型参数字典
```

```
model.state_dict()
```

```
OrderedDict([('0.weight',
              tensor([[ 0.2697,  0.0196,  0.3351],
                      [ 0.3529,  0.6842,  0.6301],
                      [-0.0664, -0.4668,  0.1318],
                      [-0.5112,  0.0759,  0.0384],
                      [ 0.1567,  0.7961,  0.4029],
                      [ 0.2176,  0.2194, -0.3687],
                      [-0.2220,  0.5552, -0.5655],
                      [-0.1207, -0.1430,  0.2965]])),
             ('0.bias',
              tensor([-0.0351,  0.2071, -0.0277, -0.3233, -0.2869, -0.2836, -0.5245, -0.3904])),
             ('2.weight',
              tensor([[ 0.3595,  0.9908,  0.0174, -0.1621,  0.7667, -0.1292, -0.0783, -0.0297]])),
             ('2.bias', tensor([0.0291]))])
```

小结

使用PyTorch建神经网络的**关键步骤**：

1. 创建数据集和数据加载器
2. 构建神经网络模型
3. 设置损失函数和优化方法
4. 训练模型
5. 预测新数据
6. 保存模型和重新加载