

快速上手 Pytorch

这篇文章需要大家对深度学习里的神经网络训练有一定的基础，我以前训练网络一直都是用的 TensorFlow，后面需要把模型和数据迁移到 Pytorch 平台上去，发现很多里面有很多知识点需要注意，写这篇文章一方面是给自己做个笔记，总结下自己的经验，另一方面是为了方便想要快速上手 Pytorch 的同学。这篇文章主要内容有：

- Tensorflow 的 Playground
- Pytorch 介绍和安装
- Torch 和 Torchvision 里的常用包
- Variable、Tensor、Numpy 之间的关系
- CPU 与 GPU
- 示例--GAN 生成 MINIST 数据

Tensorflow 的 Playground

PlayGround 是一个在线演示、实验的神经网络平台，是一个入门神经网络非常直观的网站。这个图形化平台非常强大，将神经网络的训练过程直接可视化。假若有的同学刚刚想入门深度学习这一领域，可以去看看：

PlayGround 地址：<http://playground.tensorflow.org>

这里也有一篇 PlayGround 介绍写的非常详细的文章：

参考地址：<https://finthon.com/tensorflow-playground-nn/>

Pytorch 介绍和安装



2017 年 1 月，由 Facebook 人工智能研究院（FAIR）基于 Torch 推出了 PyTorch。Pytorch 和 Torch 底层实现都用的是 C 语言，但是 Torch 的调用需要掌握 Lua 语言，相比而言使用 Python 的人更多，根本不是一个数量级，所以 Pytorch 基于 Torch 做了些底层修改、优化并且支持 Python 语言调用。它是一个基于 Python 的可续计算包，目标用户有两类：

1. 使用 GPU 来运算 numpy
2. 一个深度学习平台，提供最大的灵活型和速度

如何安装 Pytorch 呢？

- 基础环境
 - 一台 PC 设备、一张高性能 NVIDIA 显卡(可选)、Ubuntu 系统
- 安装步骤
 1. Anaconda(可选)和 Python
 2. 显卡驱动和 CUDA
 3. 运行 Pytorch 的安装命令

- 相关资料
详细的安装教程 <https://blog.csdn.net/zzlyw/article/details/78674543>
Pytorch 中文网 <https://www.pytorchtutorial.com>

Torch 和 Torchvision 里的常用包

Torch

- torch: 张量相关的运算, 例如创建、索引、切片、连接、转置、加减乘除等
- torch.nn: 包含搭建网络层的模块 (Modules) 和一系列的 loss 函数, 例如全连接、卷积、池化、BN 批处理、dropout、CrossEntropyLoss、MSELoss 等
- torch.nn.functional: 常用的激活函数 relu、leaky_relu、sigmoid 等
- torch.autograd: 提供 Tensor 所有操作的自动求导方法
- torch.optim: 各种参数优化方法, 例如 SGD、AdaGrad、RMSProp、Adam 等
- torch.nn.init: 可以用它更改 nn.Module 的默认参数初始化方式
- torch.utils.data: 用于加载数据

Torchvision

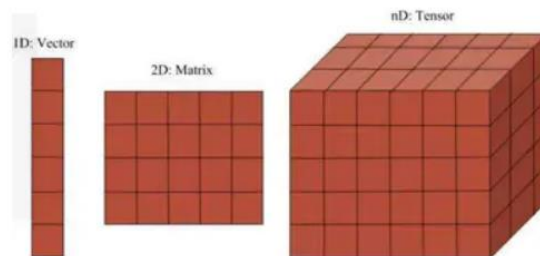
- torchvision.datasets: 常用数据集, MNIST、COCO、CIFAR10、Imagenet 等
- torchvision.models: 常用模型, AlexNet、VGG、ResNet、DenseNet 等
- torchvision.transforms: 图片相关处理, 裁剪、尺寸缩放、归一化等
- torchvision.utils: 将给定的 Tensor 保存成 image 文件

Variable、Tensor、Numpy 之间的关系

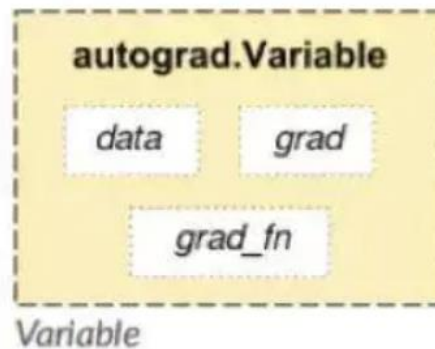
- Numpy
NumPy 是 Python 语言的一个扩充程序库。支持高级大量的维度数组与矩阵运算, 此外也针对数组运算提供大量的数学函数库。
例子:

```
>>> import numpy as np
>>> x=np.array([[1,2,3],[9,8,7],[6,5,4]])
```

- Tensor
PyTorch 提供一种类似 NumPy 的抽象方法来表征张量 (或多维数组), 它可以利用 GPU 来加速训练。



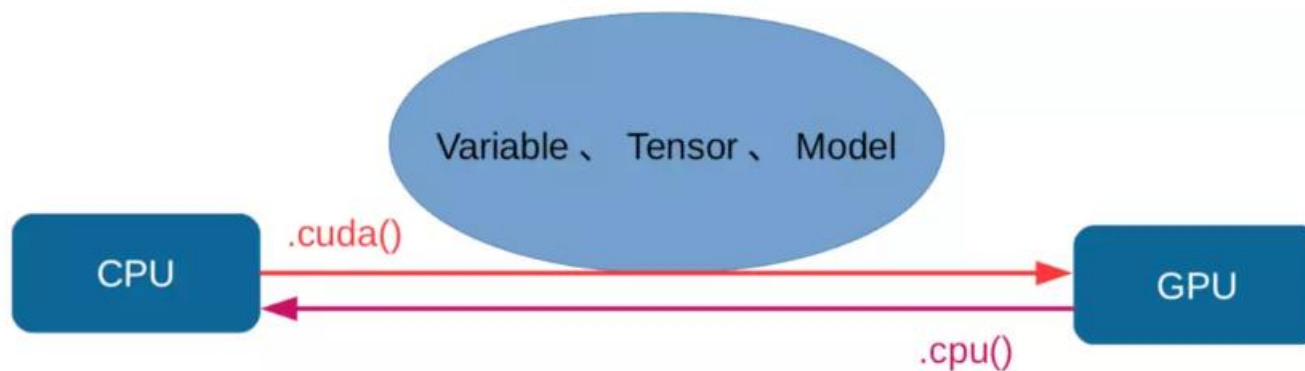
- Variable



1. PyTorch 张量的简单封装
 2. 帮助建立计算图
 3. Autograd（自动微分库）的必要部分
 4. 将关于这些变量的梯度保存在 `.grad` 中
- Tensor、Variable、Numpy 之间相互转化
1. 将 Numpy 矩阵转换为 Tensor 张量
`sub_ts = torch.from_numpy(sub_img)`
 2. 将 Tensor 张量转化为 Numpy 矩阵
`sub_npl = sub_ts.numpy()`
 3. 将 Tensor 转换为 Variable
`sub_va = Variable(sub_ts)`
 4. 将 Variable 转换为 Tensor
`sub_np2 = sub_va.data`

CPU 与 GPU

Pytorch 支持 CPU 运行，但是速度非常慢，一张好的 NVIDIA 显卡能够大大减少网络训练时间，以我自己经验来看，15 年 MacBook Pro 与戴尔工作站附加一张显存 11GB 的 1080ti 显卡相比，后者速度是前者速度的 224 倍，尤其训练复杂网络一定要在 GPU 上跑。Pytorch 中把数据和模型从 CPU 迁移到 GPU 非常简单：



直接对变量、张量、模型使用 `.cuda()` 即可把他们迁移到 GPU 上，反过来迁移到 CPU 上，使用 `.cpu()`。当有多行显卡时，想充分利用它们，则可使用 `model = nn.DataParallel(model)` 命令：

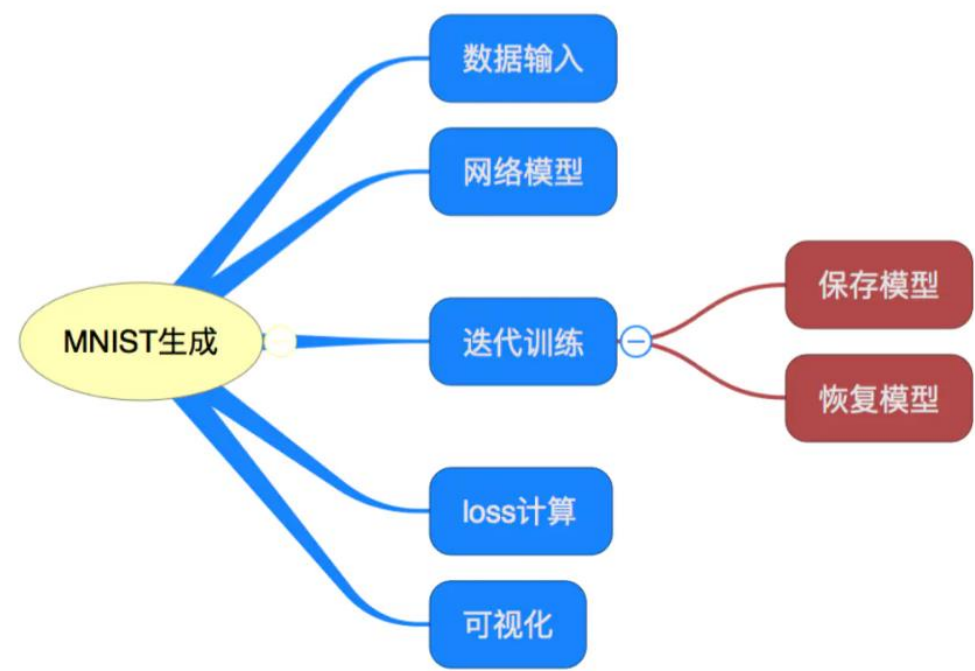


常见问题

- 这里的不同位置包含 GPU 与 CPU，还包含不同 GPU 之间
- 不同位置的 Variable 之间不能直接相互运算
- 不同位置的 Tensor 直接不能直接相互运算
- 不同位置的 Variable 和模型不能直接训练
- 使用指定显卡： `.cuda(<显卡号数>)`

示例—GAN 生成 MINIST 数据

最后看个实例，如何使用 GAN 网络生成 MNIST 数据，主要内容有：




MNIST 数据集

MNIST 数据集是一个手写体数据集，图片大小都是 28x28，包含 0-9 共 10 个数字，各种风格：

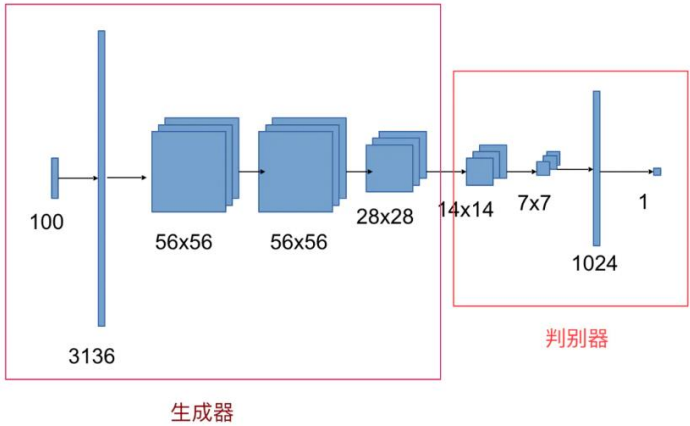


下载好的数据集：

	t10k-images-idx3-ubyte	7.8 MB
	t10k-labels-idx1-ubyte	10.0 KB
	train-images-idx3-ubyte	47.0 MB
	train-labels-idx1-ubyte	60.0 KB

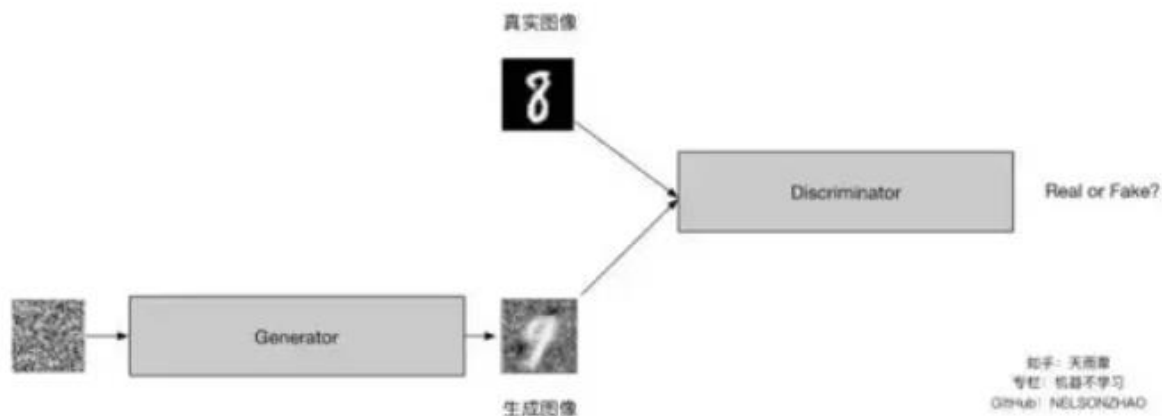
测试集 t10k 开头，训练集 train 开头，images 是图片，labels 是标签

GAN 网络模型



输入 100 长度的噪声向量，经过一个全连接，两个卷积层，一个下采样之后生成成 28x28 大小的图片，这一部分是生成器
生成的假图片和 MNIST 里的真图片经过两个卷积层下采样之后，再次经历两个全连接层后输出一个 1 长度的单位向量，1 代表输入图片为真，0 代表输入图片为假

GAN 训练和 Loss



$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]$$

训练判别器 D 时，要使得 V 整体变大，训练生成器 G 时，要使得 V 整体变小。

这是一个博弈的过程，就像制造假钱的犯罪团伙和验钞机的关系，犯罪团伙需要努力提高技术，让验钞机无法识别出来其制造的假币，而验钞机要能够正确的分辨出真正的纸币还有假币。

理论上当判别器 D 只有一半的概率 0.5 能识别出假图片时，就已经收敛了，实际上达不到一半的概率，没关系，使得假图片概率尽量高就行了，最终看上去效果不错。

这是一张由生成器生成的假图片，你能区分出来吗？



可视化

可视化方式有两种，一种是利用 torchvision 里面的包 torchvision.utils，另外一种是利用 visdom 插件，下面是二者的对比：

方式	torchvision.utils	visdom
优点	直接对张量进行存储，并且支持多张图片合并，简单易用	网页在线直观，更多可视化功能，高大上
缺点	功能有限	图片只支持 numpy，需要一直开启服务进程

上面那张生成的假图片就是利用 torchvision.utils 里的 save_image 函数来存储在本地的。而以下这张图是利用 visdom，在浏览器中看到的效果：



visdom 不光可以查看图片，还可以查看 loss 变化曲线图等各种功能。

具体的代码实现去工程里查看，这里给出分享地址：

https://github.com/gcfrun/GAN_MNIST_Pytorch

mnist_data.py：数据输入模块

mnist_net.py：网络模型模块

mnist_loss.py：Loss 计算模块

mnist_train.py: 迭代训练模块

mnist_visual.py: 可视化模块

PyTorch 入门教程

介绍

PyTorch 是一个非常有可能改变深度学习领域前景的 Python 库。我尝试使用了几星期 PyTorch，然后被它的易用性所震惊，在我使用过的各种深度学习库中，PyTorch 是最灵活、最容易掌握的。

在本文中，我们将讲解如何入门 PyTorch，包括基础知识和案例研究。还将分别在 numpy 和 PyTorch 中从零开始构建神经网络，以了解它们在实际中的相似处与区别。

目录

- PyTorch 的概述
- 深入研究技术细节
- 在 Numpy 和 PyTorch 中分别构建神经网络并进行对比
- 与其它深度学习库比较
- 案例研究——用 PyTorch 解决图像识别问题

PyTorch 的概述

PyTorch 的创始人说过他们创作的一个准则——他们想成为当务之急。这意味着我们可以立即执行计算。这正好符合 Python 的编程方法，不需要完成全部代码才能运行，可以轻松的运行部分代码并实时检查。对于我来说把它作为一个神经网络调试器是一件非常幸福的事。

PyTorch 是一个基于 Python 的库，用来提供一个具有灵活性的深度学习开发平台。PyTorch 的工作流程非常接近 Python 的科学计算库——numpy。

现在你可能会问，为什么我们要用 PyTorch 来建立深度学习模型呢？我可以列出三件有助于回答的事情：

- **易于使用的 API**——它就像 Python 一样简单。
- **Python 的支持**——如上所述，PyTorch 可以顺利地 Python 数据科学栈集成。它非常类似于 numpy，甚至注意不到它们的差别。
- **动态计算图**——取代了具有特定功能的预定义图形，PyTorch 为我们提供了一个框架，以便可以在运行时构建计算图，甚至在运行时更改它们。在不知道创建神经网络需要多少内存的情况下这非常有价值。

PyTorch 的其他一些优点还包括：多 gpu 支持，自定义数据加载器和简化的预处理器。

自从 2016 年 1 月发布以来，许多研究人员将其作为一种“go-to”库，因为它可以轻松地构建新颖的甚至是极其复杂的图形。虽说如此，PyTorch 仍有一段时间没有被大多数数据科学实践者采用，因为它是新的而且处于“正在建设”的状态。

深入技术细节

在深入讨论细节之前，让我们先看看 PyTorch 的工作流程。

PyTorch 使用了命令式/热切的范例。也就是说，在构建一个图形时，每一行代码都定义了改图的一个组件。我们甚至能在图形构建完成前，独立的对这些组件进行计算。这就是所谓的“逐运行”方法。

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```



安装 PyTorch 非常简单。您可以按照官方文档中提到的步骤操作，并根据您的系统规格运行命令。例如，这是我根据我选择的选项使用的命令：

Get Started.

Select your preferences, then run the PyTorch install command.

Please ensure that you are on the latest pip and numpy packages.
Anaconda is our recommended package manager

OS	<input checked="" type="radio"/> Linux	<input type="radio"/> OSX		
Package Manager	<input checked="" type="radio"/> conda	<input type="radio"/> pip	<input type="radio"/> Source	
Python	<input type="radio"/> 2.7	<input type="radio"/> 3.5	<input checked="" type="radio"/> 3.6	
CUDA	<input type="radio"/> 8	<input type="radio"/> 9.0	<input checked="" type="radio"/> 9.1	<input type="radio"/> None

Run this command:

```
conda install pytorch torchvision cuda91 -c pytorch
```

[Click here for previous versions of PyTorch](#)

在开始使用 PyTorch 时应该了解的主要元素：

- PyTorch 张量
- 数学运算
- Autograd 模块
- Optim 模块
- 神经网络模块

下面让我们依次介绍这些元素吧。

PyTorch 张量

张量只是多维数组。PyTorch 中的张量类似于 numpy 的 ndarrays，另外，张量也可以在 GPU 上使用。PyTorch 支持各种类型的张量。你可以如下定义一个简单的二维矩阵：

```
# import pytorch
import torch

# define a tensor
torch.FloatTensor([2])
```

```
2
[torch.FloatTensor of size 1]
```

数学运算

与 numpy 一样，科学计算库非常重要的一点是能够实现高效的数学功能。而 PyTorch 提供了一个类似的借口，可以使用 200 个以上的数学运算。

下面是在 PyTorch 中实现一个简单的添加操作的例子：

```
a = torch.FloatTensor([2])
b = torch.FloatTensor([3])

a + b
```

```
5
[torch.FloatTensor of size 1]
```

这和基本的 python 方法非常相似。我们还可以在定义的 PyTorch 张量上执行各种矩阵运算。例如，我们要转置一个二维矩阵：

```
matrix = torch.randn(3, 3)
```

```
matrix
```

```
-1.3531 -0.5394  0.8934
```

```
 1.7457 -0.6291 -0.0484
```

```
-1.3502 -0.6439 -1.5652
```

```
[torch.FloatTensor of size 3x3]
```

```
matrix.t()
```

```
-2.1139  1.8278  0.1976
```

```
 0.6236  0.3525  0.2660
```

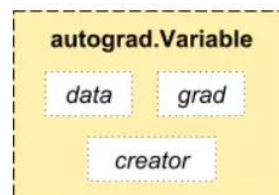
```
-1.4604  0.8982  0.0428
```

```
[torch.FloatTensor of size 3x3]
```

云栖社区 yq.aliyun

Autograd 模块

PyTorch 使用了一种叫做自动微分的技术。也就是说，它会有一个记录我们所有执行操作的记录器，之后再回放记录来计算我们的梯度。这一技术在构建神经网络时尤其有效，因为我们可以通过计算前路参数的微分来节省时间。



```
from torch.autograd import Variable
```

```
x = Variable(train_x)
```

```
y = Variable(train_y, requires_grad=False)
```

云栖社区 yq.aliyun

Optim 模块

Torch.optim 是一个实现各种优化算法的模块，用于构建神经网络。它支持大多数常用的方法，因此我们不必从头开始构建它们。下面是使用 Adam 优化器的代码：

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

神经网络模块

虽然 PyTorch Autograd 可以很容易的定义计算图形和使用梯度，但是对于定义复杂的神经网络来说可能有点太低级了。而这就需要神经网络模块来提供帮助。

nn 包定义了一组模块，我们可以把它看作是一个神经网络层，它产生输入输出，并且可能有一些可训练的权重。

你可以把 nn 模块看作是 PyTorch 的内核！

```
import torch

# define model
model = torch.nn.Sequential(
    torch.nn.Linear(input_num_units, hidden_num_units),
    torch.nn.ReLU(),
    torch.nn.Linear(hidden_num_units, output_num_units),
)
loss_fn = torch.nn.CrossEntropyLoss()
```

现在您已经了解了 PyTorch 的基本组件，那么可以轻松地从头构建自己的神经网络了。如果想知道怎么做，就继续往下看吧。

分别在 Numpy 和 PyTorch 中构建神经网络并比较

我之前提到过 PyTorch 和 Numpy 非常相似，现在让我们看看原因。在本节中，我们将通过实现一个简单的神经网络来解决二进制分类问题。

```
#Input array
X=np.array([[1,0,1,0],[1,0,1,1],[0,1,0,1]])

#Output
y=np.array([[1],[1],[0]])

#Sigmoid Function
def sigmoid (x):
    return 1/(1 + np.exp(-x))

#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
    return x * (1 - x)

#Variable initialization
epoch=5000 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = X.shape[1] #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer
```

```
#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))
```

云栖社区 yq.aliyun

```
for i in range(epoch):
    #Forward Propagation
    hidden_layer_input1=np.dot(X,wh)
    hidden_layer_input=hidden_layer_input1 + bh
    hiddenlayer_activations = sigmoid(hidden_layer_input)
    output_layer_input1=np.dot(hiddenlayer_activations,wout)
    output_layer_input= output_layer_input1+ bout
    output = sigmoid(output_layer_input)

    #Backpropagation
    E = y-output
    slope_output_layer = derivatives_sigmoid(output)
    slope_hidden_layer = derivatives_sigmoid(hiddenlayer_activations)
    d_output = E * slope_output_layer
    Error_at_hidden_layer = d_output.dot(wout.T)
    d_hiddenlayer = Error_at_hidden_layer * slope_hidden_layer
    wout += hiddenlayer_activations.T.dot(d_output) *lr
    bout += np.sum(d_output, axis=0,keepdims=True) *lr
    wh += X.T.dot(d_hiddenlayer) *lr
    bh += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr

print('actual :\n', y, '\n')
print('predicted :\n', output)
```

云栖

现在，试着在 PyTorch 中以超级简单的方式发现差异（在下面的代码中用粗体表示差异）。

```
## neural network in pytorch
import torch

#Input array
X = torch.Tensor([[1,0,1,0],[1,0,1,1],[0,1,0,1]])

#Output
y = torch.Tensor([[1],[1],[0]])

#Sigmoid Function
def sigmoid (x):
    return 1/(1 + torch.exp(-x))

#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
    return x * (1 - x)

#Variable initialization
epoch=5000 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = X.shape[1] #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer
```

```
#weight and bias initialization
wh=torch.randn(inputlayer_neurons, hiddenlayer_neurons).type(torch.FloatTensor)
bh=torch.randn(1, hiddenlayer_neurons).type(torch.FloatTensor)
wout=torch.randn(hiddenlayer_neurons, output_neurons)
bout=torch.randn(1, output_neurons)
```



```
for i in range(epoch):

    #Forward Propogation
    hidden_layer_input1 = torch.mm(X, wh)
    hidden_layer_input = hidden_layer_input1 + bh
    hidden_layer_activations = sigmoid(hidden_layer_input)

    output_layer_input1 = torch.mm(hidden_layer_activations, wout)
    output_layer_input = output_layer_input1 + bout
    output = sigmoid(output_layer_input1)

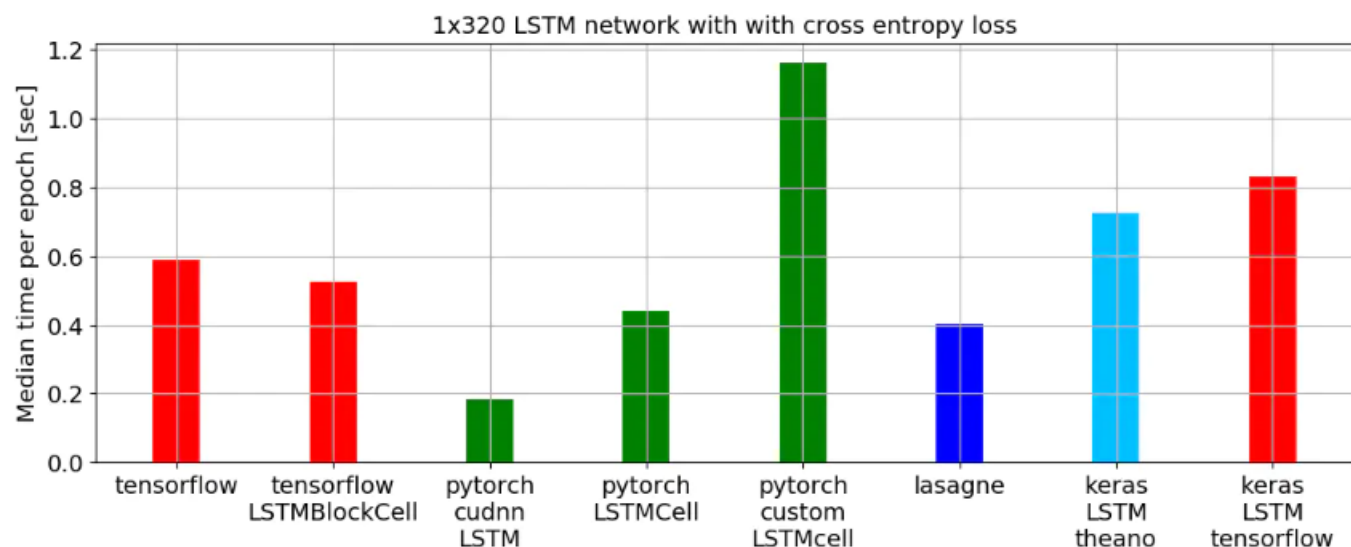
    #Backpropagation
    E = y-output
    slope_output_layer = derivatives_sigmoid(output)
    slope_hidden_layer = derivatives_sigmoid(hidden_layer_activations)
    d_output = E * slope_output_layer
    Error_at_hidden_layer = torch.mm(d_output, wout.t())
    d_hiddenlayer = Error_at_hidden_layer * slope_hidden_layer
    wout += torch.mm(hidden_layer_activations.t(), d_output) *lr
    bout += d_output.sum() *lr
    wh += torch.mm(X.t(), d_hiddenlayer) *lr
    bh += d_output.sum() *lr

print('actual :\n', y, '\n')
print('predicted :\n', output)
```

云栖社区 yq.c

与其它深度学习库比较

通过这个基准测试脚本中可以看出，PyTorch 训练一个长短期记忆网络（LSTM）的过程比其他所有主要的深度学习库都要出色，因为它在每个时代的中位时间都最低（参考下图）。



PyTorch 中用于数据加载的 API 设计的非常好，它的接口可以在数据集、采样器和数据加载器中指定。并且通过与 TensorFlow（读取器、队列等）数据加载工具比较发现，PyTorch 的数据加载模块非常容易使用。此外，PyTorch 在构建神经网络时是无缝的，所以不必依赖像 Keras 这样的第三方高层库。

另一方面，我也不建议使用 PyTorch 进行部署。因为它还尚未发展完美。正如 PyTorch 开发者说：“我们能够看到，用户会首先创建一个 PyTorch 模型，当要把模型投入生产时会将其转换为 Caffe2 模型，之后再运送到移动平台或其他平台。”

案例研究——解决 PyTorch 中的图像识别问题

为了更加熟悉 PyTorch，我们将实践解决分析 Vidhya 的深度学习问题——识别数字。让我们看看我们的问题陈述：

我们的问题是图像识别问题，从一个给定的 28x28 图像中识别数字。一部分图像用于训练，其余的用于测试模型。

首先下载 train 和测试文件。该数据集包含所有图像的压缩文件，以及具有相应 train 和测试图像名称的 train.csv 和 test.csv 文件。数据集只提供 png 格式原始图像，不提供其它附加功能。

现在让我们开始吧：

步骤 0：准备

a) 导入所有必要的库。

```
# import modules
%pylab inline
import os
import numpy as np
import pandas as pd
from scipy.misc import imread
from sklearn.metrics import accuracy_score
```

b) 设置一个种子值，这样我们就可以控制模型的随机性。

```
# To stop potential randomness
seed = 128
rng = np.random.RandomState(seed)
```

c) 安全起见，第一步设置目录路径。

```
root_dir = os.path.abspath('.')
data_dir = os.path.join(root_dir, 'data')

# check for existence
os.path.exists(root_dir), os.path.exists(data_dir)
```

步骤 1: 数据加载和预处理

A) 现在让我们看看这些数据集。它们都有相应标签文件名，并且是.csv 格式。

```
# load dataset
train = pd.read_csv(os.path.join(data_dir, 'Train', 'train.csv'))
test = pd.read_csv(os.path.join(data_dir, 'Test.csv'))

sample_submission = pd.read_csv(os.path.join(data_dir, 'Sample_Submission.csv'))

train.head()
```

	filename	label
0	0.png	4
1	1.png	9
2	2.png	1
3	3.png	7
4	4.png	3

云栖社区 yq.aliy

B) 让我们看看数据是什么样的，现在读取图像并显示。

```
# print an image
img_name = rng.choice(train.filename)
filepath = os.path.join(data_dir, 'Train', 'Images', 'train', img_name)

img = imread(filepath, flatten=True)

pylab.imshow(img, cmap='gray')
pylab.axis('off')
pylab.show()
```

云栖社区

C) 为了更容易操作，让我们把所以图像存储为 numpy 数组。

```
# load images to create train and test set
temp = []
for img_name in train.filename:
    image_path = os.path.join(data_dir, 'Train', 'Images', 'train', img_name)
    img = imread(image_path, flatten=True)
    img = img.astype('float32')
    temp.append(img)

train_x = np.stack(temp)

train_x /= 255.0
train_x = train_x.reshape(-1, 784).astype('float32')

temp = []
for img_name in test.filename:
    image_path = os.path.join(data_dir, 'Train', 'Images', 'test', img_name)
    img = imread(image_path, flatten=True)
    img = img.astype('float32')
    temp.append(img)

test_x = np.stack(temp)

test_x /= 255.0
test_x = test_x.reshape(-1, 784).astype('float32')
```

D) 由于这是一个典型的机器语言（ML）问题，为了测试模型的正常运行，我们创建了一个验证集。训练集与验证集比例为 70：30。

```
# create validation set
split_size = int(train_x.shape[0]*0.7)

train_x, val_x = train_x[:split_size], train_x[split_size:]
train_y, val_y = train_y[:split_size], train_y[split_size:]
```

步骤 2: 构建模型

A) 这是最重要的部分！首先定义神经网络架构。我们定义了一个具有输入、隐藏和输出三层的神经网络。输入和输出中的神经元数目是固定的，因为输入是 28x28 的图像，输出是一个 10x1 向量的代表类，而在隐藏层我们采用了 50 个神经元。在这里，我们用 Adam 作为优化算法，这是梯度下降算法的有效变体。

```
import torch
from torch.autograd import Variable
```

```
# number of neurons in each layer
input_num_units = 28*28
hidden_num_units = 500
output_num_units = 10

# set remaining variables
epochs = 5
batch_size = 128
learning_rate = 0.001
```

B) 训练模型。

```
# define model
model = torch.nn.Sequential(
    torch.nn.Linear(input_num_units, hidden_num_units),
    torch.nn.ReLU(),
    torch.nn.Linear(hidden_num_units, output_num_units),
)
loss_fn = torch.nn.CrossEntropyLoss()

# define optimization algorithm
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

```
## helper functions
# preprocess a batch of dataset
def preproc(unclean_batch_x):
    """Convert values to range 0-1"""
    temp_batch = unclean_batch_x / unclean_batch_x.max()

    return temp_batch

# create a batch
def batch_creator(batch_size):
    dataset_name = 'train'
    dataset_length = train_x.shape[0]

    batch_mask = rng.choice(dataset_length, batch_size)

    batch_x = eval(dataset_name + '_x')[batch_mask]
    batch_x = preproc(batch_x)

    if dataset_name == 'train':
        batch_y = eval(dataset_name).ix[batch_mask, 'label'].values

    return batch_x, batch_y
```

```
# train network
total_batch = int(train.shape[0]/batch_size)

for epoch in range(epochs):
    avg_cost = 0
    for i in range(total_batch):
        # create batch
        batch_x, batch_y = batch_creator(batch_size)

        # pass that batch for training
        x, y = Variable(torch.from_numpy(batch_x)), Variable(torch.from_numpy(batch_y), requires_grad=False)
        pred = model(x)

        # get loss
        loss = loss_fn(pred, y)

        # perform backpropagation
        loss.backward()
        optimizer.step()
        avg_cost += loss.data[0]/total_batch

    print(epoch, avg_cost)
```

```
# get training accuracy
x, y = Variable(torch.from_numpy(preproc(train_x))), Variable(torch.from_numpy(train_y), requires_grad=False)
pred = model(x)

final_pred = np.argmax(pred.data.numpy(), axis=1)

accuracy_score(train_y, final_pred)
```



```
# get validation accuracy
x, y = Variable(torch.from_numpy(preproc(val_x))), Variable(torch.from_numpy(val_y), requires_grad=False)
pred = model(x)
final_pred = np.argmax(pred.data.numpy(), axis=1)

accuracy_score(val_y, final_pred)
```

训练成绩如下：

0.8779008746355685

而验证分数为：

0.867482993197279

这是一个相当令人印象深刻的分数，尤其是这个非常简单的神经网络我们只训练了 5 次。

希望这篇文章能让您看到 PyTorch 是如何改变构建深度学习模型的。在本文中，我们只是触及了表面。要想深入研究，您可以从 PyTorch 官网下载相关文档和教程。

本文由阿里云云栖社区组织翻译。

文章原标题《An Introduction to PyTorch - A Simple yet Powerful Deep Learning Library》

Pytorch 框架使用

介绍

相比 TensorFlow 的静态图开发，Pytorch 的动态图特性使得开发起来更加人性化，选择 Pytorch 的理由可以参考：

<https://www.jianshu.com/p/c1d9cdb52548>，这里也顺便介绍一下 TensorFlow 静态图和 Pytorch 动态图开发的区别：

总的来说，在 TensorFlow 里你只能通过定义数据、网络等，然后直接训练、预测啥的，中间过程到底发生了什么对我们来说都是未知的，只能等待训练完毕后查看结果如何，想要 debug 都 debug 不了，于是在学了一段时间以后还是一脸懵：训练的过程到底发生了什么？数据怎么变化的？

直到接触了 pytorch 框架以后，这些疑问就渐渐地减少了，因为在 pytorch 里，一切就像操作 numpy 数组一样（很多方法名甚至都一样），只是类型变成了张量，如何训练、训练多少次等等一切都是我们自己来决定，过程发生了什么？print 或者 debug 一下就知道了

拿最简单的线性拟合举例，通过 pytorch 操作，你会发现整个拟合的过程就是：提供输入和输出值，然后网络层（假如就一个全连接层）就相当于一个矩阵（这里暂时忽略偏置值，原本全连接层的计算是与矩阵相乘后再加上一个偏置值），里面的参数（这里称为权值）随着每一次计算，再根据求导之类的操作不断更新参数的值，最终使得输入的值与这个矩阵相乘后的值不断贴近于输出值，而这计算和修改权值的过程就称为训练。

比如一个函数 $y = 2x + 1$ ，然后定义一个 1×1 的矩阵（全连接层），然后矩阵里的权值一开始都是随机的，比如 $((1))$ ，那么要拟合这个函数的话，最终理想的结果肯定是传入 x ，输出的结果是 $2x + 1$ ，所以目标就是让一个数与这个矩阵相乘尽量等价于这个数和 2 相乘后再加 1，也就是说最终 x 乘以矩阵和 $2x$ 的差最好等于 0，比如在 $x=-1$ 时， $y=-1$ ，那么矩阵的权值理想结果就是 1。但是当结果越来越大以后，会发现 y/x 的值逐渐与 2 相近，所以最终矩阵经过多次训练以后，结果肯定就是 2 左右的数，而刚才说的这些可以通过一段 pytorch 代码来演示：

```
import torch
x = torch.unsqueeze(torch.linspace(-100, 100, 1000), dim=1)
# 生成-100 到 100 的 1000 个数的等差数列
y = 2*x + 1
# 定义 y=2x+1 函数
matrix = torch.nn.Linear(1, 1)
# 定义一个 1x1 的矩阵
optimizer = torch.optim.Adam(matrix.parameters(), lr=0.1)
# 使用优化器求导更新矩阵权重
for _ in range(100):
    # 训练 100 次
    value = matrix(x)
    # value 是 x 与矩阵相乘后的值
    score = torch.mean((value - y) ** 2)
    # 目标偏差，值为(value-y)的平方取均值，越接近 0 说明结果越准确
    matrix.zero_grad()
    score.backward()
    optimizer.step()
    # 根据求导结果更新权值
    print("第 {} 次训练权值结果: {}, 结果偏差: {}".format(_, matrix.weight.data.numpy(), score))
# 输出结果：
# 第 0 次训练权值结果: [[0.9555]], 结果偏差: 4377.27294921875
```

```
# ...  
# 第 99 次训练权值结果:[[2.0048]], 结果偏差: 0.10316929966211319
```

从这段代码的结果可以看到最开始权值初始值为 0.9555，偏差为 4377.27294921875，经过 100 次训练后，权值为 2.0048，偏差为 0.1 那样，从而可以证实我们前面的思路是对的。

其实上面的代码里不止更新了权值，也更新了偏置值 bias，只不过这里为了更加简单的解释，而没有进行说明，可以通过 `matrix.bias` 可以调用查看，最终会发现偏置值接近 1 左右（假设是 0.99），而偏置值就是在与矩阵相乘后加上的值，所以可以看出通过训练以后，矩阵和偏置拟合成了函数： $y = 2.0048x + 0.99$ 。

上面介绍的是线性拟合的例子。即数学相关的示例，而放在生活应用当中，其就可以理解成拟合一个特定要求的函数，比如识别图片中人脸的场景中就是 $F(\text{图片}) = \text{人脸坐标}$

即输入图片到函数当中，输出的就是人脸对应的坐标，而我们的目标就是要实现这个函数（经过不断训练，修正模型中的参数）

通过上面的介绍，想必你应该对 pytorch 以及深度学习有了一点入门的了解了，而本文接下来也将针对于 pytorch 的基本使用作出介绍

安装

分为 CPU 和 GPU 版本：

- CPU 版本下载：

直接 pip 下载容易出问题，因此这里推荐离线下载：

1. 进入网址：<https://www.lfd.uci.edu/~gohlke/pythonlibs/#pytorch> 或者 https://download.pytorch.org/whl/torch_stable.html，下载对应版本的 pytorch

2. pip 安装对应的下载文件

- GPU 下载：

1. 先安装 cuda：<https://developer.nvidia.com/cuda-downloads>

2. 下载并安装 GPU 版本 pytorch 文件

简单示例-线性拟合

```
import torch  
import matplotlib.pyplot as plt
```

```
w = 2  
b = 1  
noise = torch.rand(100, 1)
```

```
x = torch.unsqueeze(torch.linspace(-1, 1, 100), dim=1)
# 因为输入层格式要为(-1, 1)，所以这里将(100)的格式转成(100, 1)
y = w*x + b + noise
# 拟合分布在 y=2x+1 上并且带有噪声的散点
model = torch.nn.Sequential(
    torch.nn.Linear(1, 16),
    torch.nn.Tanh(),
    torch.nn.Linear(16, 1),
)
# 自定义的网络，带有 2 个全连接层和一个 tanh 层
loss_fun = torch.nn.MSELoss()
# 定义损失函数为均方差
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
# 使用 adam 作为优化器更新网络模型的权重，学习率为 0.01

plt.ion()
# 图形交互
for _ in range(1000):
    ax = plt.axes()
    output = model(x)
    # 数据向后传播（经过网络层的一次计算）
    loss = loss_fun(output, y)
    # 计算损失值
    # print("before zero_grad: {}".format(list(model.children())[0].weight.grad))
    # print("-"*100)
    model.zero_grad()
    # 优化器清空梯度
    # print("before zero_grad: {}".format(list(model.children())[0].weight.grad))
    # print("-"*100)
    # 通过注释地方可以对比发现执行 zero_grad 方法以后倒数梯度将会被清 0
    # 如果不清空梯度的话，则会不断累加梯度，从而影响到当前梯度的计算
    loss.backward()
    # 向后传播，计算当前梯度，如果这步不执行，那么优化器更新时则会找不到梯度
    optimizer.step()
```

```

# 优化器更新梯度参数，如果这步不执行，那么因为梯度没有发生改变，loss 会一直计算最开始的那个梯度
if _ % 100 == 0:
    plt.cla()
    plt.scatter(x.data.numpy(), y.data.numpy())
    plt.plot(x.data.numpy(), output.data.numpy(), 'r-', lw=5)
    plt.text(0.5, 0, 'Loss=%.4f' % loss.data.numpy(), fontdict={'size': 20, 'color': 'red'})
    plt.pause(0.1)
    # print("w:", list(model.children())[0].weight.t() @ list(model.children())[-1].weight.t())
    # 通过这句可以查看权值变化，可以发现最后收敛到 2 附近

```

```

plt.ioff()
plt.show()

```

从上面的示例里不知道你有没有看出来，其实 pytorch 的使用和 numpy 特别相似，只是 numpy 是基于数组（numpy.ndarray），而 pytorch 是基于张量（torch.Tensor），但是在使用上很多都是一样的，包括很多方法名等。所以如果学习过 numpy 的话，会感觉 pytorch 特别的亲切，如果没学过的 numpy 话，通过学习 pytorch，也将顺便给你将来的 numpy 学习奠定一定的基础

数据类型

标量/张量

pytorch 里的基本单位，个人理解是 0 维、没有方向的（比如单个数字那样）称为标量，有方向的称为张量（比如一串数字），通过 torch.tensor 定义，举例：

```

>>> torch.tensor(1.)
tensor(1.)

```

pytorch 中还提供了以下数据类型的张量：

```

通用类型  Tensor
float 型   torch.FloatTensor
double 型  torch.DoubleTensor
int 型     torch.IntTensor
long 型    torch.LongTensor

```

byte 型 torch.ByteTensor

这些数据类型的张量使用方法和 tensor 有点不同，tensor 是自定义数据，而标定数据类型的将会随机生成一个该类型的数据，举例：

```
>>> torch.tensor(1.)
tensor(1.)
# 生成值为 1 的张量
>>> torch.FloatTensor(1)
tensor([-3.0147e-21])
# 生成 1 个随机的 float 型张量
>>> torch.FloatTensor(2, 3)
tensor([[ -7.3660e-21,  4.5914e-41,  9.6495e+20],
        [ 8.2817e-43,  0.0000e+00,  0.0000e+00]])
# 生成随机指定尺寸的 float 型张量
```

并且可以发现如果是大写开头的张量都是传入形状生成对应尺寸的张量，只有 tensor 是传入自己定义的数据，举例：

```
>>> torch.Tensor(2, 3)
tensor([[ -7.3660e-21,  4.5914e-41,  0.0000e+00],
        [ 0.0000e+00,  0.0000e+00,  0.0000e+00]])
# 创建一个格式为 2 行 3 列的张量
>>> torch.tensor(3)
tensor(3)
# 创建一个值为 3 的张量
```

当然如果希望大写开头的张量传入自己定义的数据，则传入一个列表或者数组，举例：

```
>>> torch.Tensor([2, 3])
tensor([2., 3.])
# 创建一个张量，值为自定义的
```

注 1：

上面提供的张量类型是在 CPU 下的，如果是在 GPU 下的，则在 torch.cuda 下，举例：

float 型 torch.cuda.FloatTensor

double 型 torch.cuda.DoubleTensor
int 型 torch.cuda.IntTensor
long 型 torch.cuda.LongTensor
byte 型 torch.cuda.ByteTensor

注 2:

pytorch 框架里没有提供 string 这样的数据类型，所以为了表示某个标记之类的，我们可以使用 one-hot 编码，或者使用 Embedding（常用的 Word2vec/glove）

张量属性/方法

索引

张量可以像数组那样进行索引，举例：

```
>>> a = torch.rand(2, 3)
# 生成 2 行 3 列随机数
>>> a
tensor([[0.2488, 0.2794, 0.2949],
        [0.1818, 0.1950, 0.3803]])
>>> a[0]
tensor([0.2488, 0.2794, 0.2949])
# 索引第一行
>>> a[0, 1]
tensor(0.2794)
# 索引第一行第二列
>>> x[:, [0, 2]]
tensor([[0.2488, 0.2949],
        [0.1818, 0.3803]])
# 索引第一列和第三列
```

也可以使用内置的 index_select 方法进行索引，使用该方法可以索引多个自定义的行列（比如取第 1/3/4 列），该方法传入两个参数分别为张量维度以及张量中的索引（传入类型为张量），举例：

```
>>> a = torch.rand(2, 3)
```

```
>>> a
tensor([[0.7470, 0.8258, 0.5929],
        [0.7803, 0.7016, 0.4281]])
>>> a.index_select(0, torch.tensor([1]))
tensor([[0.7803, 0.7016, 0.4281]])
# 对数据第1维（整体）取第2个数据（第二行）
# 第二个参数为张量
>>> a.index_select(1, torch.tensor([1, 2]))
tensor([[0.8258, 0.5929],
        [0.7016, 0.4281]])
# 对数据第1维取2/3列
```

切片

张量可以像数组那样进行切片，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.7470, 0.8258, 0.5929],
        [0.7803, 0.7016, 0.4281]])
>>> a[0, :2]
tensor([0.7470, 0.8258])
# 第一行前两列
>>> a[0, ::2]
tensor([0.7470, 0.5929])
# 第一行从第一个起隔两个取一个，因为这里只有3个，所以取第一列和第三列
```

还有...代表对这部分取全部，举例：

```
>>> a = torch.rand(2, 2, 2, 2)
>>> a
tensor([[[[0.0882, 0.8744],
          [0.9916, 0.6415]],
        [[0.7247, 0.4012],
```



```
[0.5703, 0.9776]]],
```

```
[[[0.1076, 0.0710],  
   [0.1275, 0.5045]],
```

```
   [[0.7833, 0.6519],  
    [0.3394, 0.2560]]]])
```

```
>>> a[1, ..., 1]  
tensor([[0.0710, 0.5045],  
        [0.6519, 0.2560]])  
# 相当于 a[1, :, :, 1]
```

逻辑操作

可以像数组那样进行逻辑操作，举例：

```
>>> a = torch.rand(2, 3)  
>>> a  
tensor([[0.7679, 0.1081, 0.3601],  
        [0.0661, 0.8539, 0.3079]])  
>>> a>0.5  
tensor([[1, 0, 0],  
        [0, 1, 0]], dtype=torch.uint8)  
# 大于 0.5 的变成 1，否则变成 0  
# 在数组里是变成 True 或 False
```

shape

获取张量形状，举例：

```
>>> a = torch.tensor(1.)  
>>> a.shape  
torch.Size([1])  
>>> a = torch.tensor((1., 2.))
```

```
>>> a.shape
torch.Size([2])
>>> a = torch.tensor((1., 2.), (3, 4))
>>> a
tensor([[1., 2.],
        [3., 4.]])
>>> a.shape
torch.Size([2, 2])
>>> a.shape[0]
2
# 一维的尺寸
```

也可以通过内置方法 `size()` 来获取，举例：

```
>>> a = torch.tensor((1., 2.), (3, 4))
>>> a.size()
torch.Size([2, 2])
>>> a.size(0)
2
# 一维的尺寸
>>> a.size(1)
2
# 二维的尺寸
```

data

获取张量

item()

获取数据，仅当只有一个数据时才能用

dim()

获取张量维度，举例：

```
>>> a = torch.tensor(1.)
>>> a.dim()
0
# 标量数据, 0 维
>>> a = torch.tensor([1.])
>>> a.dim()
1
# 一维张量
>>> a = torch.tensor((1., 2.), (3, 4))
>>> a.dim()
2
```

```
numel()
```

获取张量大小, 举例:

```
>>> a = torch.tensor((1., 2.), (3, 4))
>>> a.numel()
4
# a 里总共有 4 个数
```

```
type()
```

获取张量类型, 举例:

```
>>> x = torch.IntTensor(1)
>>> x
tensor([1065353216], dtype=torch.int32)
>>> x.type()
'torch.IntTensor'
>>> type(x)
<class 'torch.Tensor'>
# 使用内置的 type 函数只能知道是个 torch 下的张量
```

```
cuda()
```

CPU 数据转 GPU，举例：

```
>>> x = torch.IntTensor(1)
>>> x = x.cuda()
# 转成 GPU 数据，这条语句得在 GPU 环境下才能运行
```

注：

判断是否可用 GPU 可以通过 `torch.cuda.is_available()` 判断，举例：

```
>>> torch.cuda.is_available()
False
```

下面是一个 GPU 常用操作：

```
>>> torch.cuda.device_count()
1
# 获取可使用 GPU 数量
>>> torch.cuda.set_device(0)
# 使用编号为 0 的 GPU

cpu()
```

GPU 数据转 CPU，举例：

```
>>> a = torch.cuda.FloatTensor(1)
>>> a.cpu()
tensor(1)
```

注：

上面的 GPU 和 CPU 数据互换是在低版本的 pytorch 上使用的，使用起来可能不太方便，之后的版本推出了简单切换的版本：先通过 `torch.device()` 方法选择一个 GPU/CPU 设备，然后对需要使用该设备的数据通过 `to()` 方法调用，举例：

```
>>> device = torch.device('cuda:0')
# 选择 GPU 设备
>>> net = torch.nn.Linear(10, 100).to(device)
```

这里定义了一个全连接网络层，并使用该 GPU 设备

`view()/reshape()`

这两个方法一样，修改张量形状，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.7824, 0.0911, 0.5798],
        [0.4280, 0.2592, 0.4978]])
>>> a.reshape(3, 2)
tensor([[0.7824, 0.0911],
        [0.5798, 0.4280],
        [0.2592, 0.4978]])
>>> a.view(3, 2)
tensor([[0.7824, 0.0911],
        [0.5798, 0.4280],
        [0.2592, 0.4978]])
```

`t()`

转置操作，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.4620, 0.9787, 0.3998],
        [0.4092, 0.1320, 0.5631]])
>>> a.t()
tensor([[0.4620, 0.4092],
        [0.9787, 0.1320],
        [0.3998, 0.5631]])
>>> a.t().shape
torch.Size([3, 2])
```

`pow()`

对张量进行幂运算，也可以用**代替，举例：

```
>>> a = torch.full([2, 2], 3)
>>> a
tensor([[3., 3.],
        [3., 3.]])
>>> a.pow(2)
tensor([[9., 9.],
        [9., 9.]])
>>> a**2
tensor([[9., 9.],
        [9., 9.]])
```

sqrt()

对张量取平方根，举例：

```
>>> a = torch.full([2, 2], 9)
>>> a
tensor([[9., 9.],
        [9., 9.]])
>>> a.sqrt()
tensor([[3., 3.],
        [3., 3.]])
>>> a**(0.5)
tensor([[3., 3.],
        [3., 3.]])
# 可以看出结果一样
```

rsqrt()

取平方根的倒数，举例：

```
>>> a = torch.full([2, 2], 9)
>>> a
```

```
tensor([[9., 9.],
        [9., 9.]])
>>> a.rsqrt()
tensor([[0.3333, 0.3333],
        [0.3333, 0.3333]])
```

exp()

取 e 为底的幂次方，举例：

```
>>> a = torch.full([2, 2], 2)
>>> a.exp()
tensor([[7.3891, 7.3891],
        [7.3891, 7.3891]])
```

log()

取 log 以 e 为底的对数，举例：

```
>>> a = torch.full([2, 2], 2)
>>> a.exp()
tensor([[7.3891, 7.3891],
        [7.3891, 7.3891]])
>>> a.log()
tensor([[0.6931, 0.6931],
        [0.6931, 0.6931]])
```

对应的还有以 2 为底的 log2、以 10 为底的 log10 等，举例：

```
>>> a = torch.full([2, 2], 2)
>>> a.log2()
tensor([[1., 1.],
        [1., 1.]])
```

round()/floor()/ceil()

四舍五入、向下取整和向上取整

```
trunc()/frac()
```

取整数/小数部分，举例：

```
>>> a = torch.tensor(1.2)
>>> a.trunc()
tensor(1.)
>>> a.frac()
tensor(0.2000)
```

```
max()/min()/median()/mean()
```

取最大值、最小值、中位数和平均值，举例：

```
>>> a = torch.tensor([1., 2., 3., 4., 5.])
>>> a.max()
tensor(5.)
>>> a.min()
tensor(1.)
>>> a.median()
tensor(3.)
>>> a.mean()
tensor(3.)
```

注意的是该方法默认会将全部数据变成一维的，并取整个数据里的最大/最小之类的值，因此可以通过 dim 参数设置取值维度，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.0042, 0.5913, 0.0104],
        [0.1673, 0.9443, 0.1303]])
>>> a.max()
tensor(0.9443)
# 默认返回总体的最大值
```



```
>>> a.max(dim=0)
(tensor([0.1673, 0.9443, 0.1303]), tensor([1, 1, 1]))
# 在第 1 维选择, 返回每一列最大值, 并且对应索引为 1,1,1
>>> a.max(dim=1)
(tensor([0.5913, 0.9443]), tensor([1, 1]))
# 在第 2 维选择, 返回每一行最大值, 并且对应索引为 1,1
```

还有一个 `keepdim` 参数, 可以控制返回的格式和之前相同, 举例:

```
>>> a = torch.rand(2, 3)
>>> a.max(dim=0, keepdim=True)
(tensor([[0.1673, 0.9443, 0.1303]]), tensor([[1, 1, 1]]))
>>> a.max(dim=1, keepdim=True)
(tensor([[0.5913,
          [0.9443]]]), tensor([[1],
          [1]]))
# 和之前的对比, 可以发现设置该参数后格式变成一样的了
```

`sum()/prod()`

求累加、累乘, 举例:

```
>>> a = torch.tensor([1., 2., 3., 4., 5.])
>>> a.sum()
tensor(15.)
>>> a.prod()
tensor(120.)
```

该方法也可以使用 `dim` 参数对某维度进行运算

`argmax()/argmin()`

返回最大/小值的索引, 举例:

```
>>> a = torch.tensor([1., 2., 3., 4., 5.])
```

```
>>> a.argmax()
tensor(4)
>>> a.argmin()
tensor(0)
```

该方法同样默认会将全部数据变成一维的，并计算整个数据里最大值的索引，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.2624, 0.2925, 0.0866],
        [0.0545, 0.8841, 0.9959]])
>>> a.argmax()
tensor(5)
# 可以看出所有数据默认转到 1 维上，最大值在第 6 个
```

但可以通过输入维度来控制索引的判断基准，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.7365, 0.4280, 0.6650],
        [0.6988, 0.9839, 0.8990]])
>>> a.argmax(dim=0)
tensor([0, 1, 1])
# 在第 1 维下判断每一列的最大值索引，第一列是 0.7365 索引为 0，第二列是 0.9839 索引为 1，第三列是 0.8990 索引为 1
>>> a.argmax(dim=1)
tensor([0, 1])
# 在第 2 维下判断每一行的最大值索引，第一行是 0.7365 索引为 0，第二行是 0.9839 索引为 1
```

argsort()

返回从小到大（默认，可以通过 descending 参数设置）的索引，举例：

```
>>> a.argsort()
tensor([0, 1, 2, 3, 4])
>>> a = torch.tensor([1., 5., 4., 2., 5., 3.])
```

```
>>> a.argsort()
tensor([0, 3, 5, 2, 1, 4])
# 从小到大的索引
>>> a.argsort(descending=True)
tensor([1, 4, 2, 5, 3, 0])
# 从大到小的索引
```

topk0

返回前几大的值（取前几小的值设置参数 `largest` 为 `False` 就行），并且还是按从大到小（取前几小就是从小到大）排序好的，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.0831, 0.3135, 0.2989],
        [0.2959, 0.6371, 0.9715]])
>>> a.topk(3)
(tensor([[0.3135, 0.2989, 0.0831],
        [0.9715, 0.6371, 0.2959]]), tensor([[1, 2, 0],
        [2, 1, 0]]))
# 取前三大的，可以看到结果也从大到小排序好
>>> a.topk(3, dim=1)
(tensor([[0.3135, 0.2989, 0.0831],
        [0.9715, 0.6371, 0.2959]]), tensor([[1, 2, 0],
        [2, 1, 0]]))
# 在第 2 维取前大三的数
>>> a.topk(3, largest=False)
(tensor([[0.0831, 0.2989, 0.3135],
        [0.2959, 0.6371, 0.9715]]), tensor([[0, 2, 1],
        [0, 1, 2]]))
# 取前三小的，可以看到结果也从小到大排序好
```

kthvalue0

返回第几小的数，举例：

```

>>> a = torch.rand(2, 3)
>>> a
tensor([[0.2052, 0.1159, 0.8533],
        [0.3335, 0.3922, 0.7414]])
>>> a.sort()
(tensor([[0.1159, 0.2052, 0.8533],
        [0.3335, 0.3922, 0.7414]]), tensor([[1, 0, 2],
        [0, 1, 2]]))
>>> a
tensor([[0.2052, 0.1159, 0.8533],
        [0.3335, 0.3922, 0.7414]])
>>> a.kthvalue(2, dim=0)
(tensor([0.3335, 0.3922, 0.8533]), tensor([1, 1, 0]))
# 在第 1 维返回第二小的数，可以看到每一列第二小的数被返回
>>> a.kthvalue(2, dim=1)
(tensor([0.2052, 0.3922]), tensor([0, 1]))
# 在第 2 维返回第二小的数，可以看到每一行第二小的数被返回

```

norm()

计算张量的范数（可以理解成张量长度的模），默认是 12 范数（即欧氏距离），举例：

```

>>> a = torch.tensor((-3, 4), dtype=torch.float32)
>>> a.norm(2)
tensor(5.)
# 12 范数：((-3)**2 + 4**2)**0.5
>>> a.norm(1)
tensor(7.)
# 11 范数（曼哈顿距离）：|-3| + |4|
>>> a.norm(2, dim=0)
>>> a = torch.tensor((-3, 4), (6, 8)), dtype=torch.float32)
>>> a
tensor([[ -3.,  4.],
        [ 6.,  8.]])

```

```
>>> a.norm(2, dim=0)
tensor([6.7082, 8.9443])
# 在第 1 维算范数:  $((-3)**2 + 6**2)**0.5$ ,  $(4**2 + 8**2)**0.5$ 
>>> a.norm(2, dim=1)
tensor([ 5., 10.])
# 在第 2 维算范数:  $((-3)**2 + 4**2)**0.5$ ,  $(6**2 + 8**2)**0.5$ 
```

clamp()

设置张量值范围，举例：

```
>>> a = torch.tensor([1, 2, 3, 4, 5])
>>> a.clamp(3)
tensor([3, 3, 3, 4, 5])
# 范围控制在 3~
>>> a.clamp(3, 4)
tensor([3, 3, 3, 4, 4])
# 范围控制在 3~4
```

transpose()

将指定维度对调，如果在 2 维情况，就相当于转置，举例：

```
>>> a = torch.rand(2, 3, 1)
>>> a
tensor([[[0.8327],
         [0.7932],
         [0.7497]],
        [[0.2347],
         [0.7611],
         [0.5529]]])
>>> a.transpose(0, 2)
tensor([[[0.8327, 0.2347],
         [0.7932, 0.7611],
         [0.7497, 0.5529]]])
```

```
# 将第 1 维和第 3 维对调
>>> a.transpose(0, 2).shape
torch.Size([1, 3, 2])
```

permute()

和 transpose 类似也是对调维度，但使用不太一样，举例：

```
>>> a = torch.rand(2, 3, 1)
>>> a
tensor([[[[0.6857],
           [0.4819],
           [0.3992]],

         [[0.7477],
           [0.8073],
           [0.1939]]]])
>>> a.permute(2, 0, 1)
tensor([[[[0.6857, 0.4819, 0.3992],
          [0.7477, 0.8073, 0.1939]]]])
# 将 a 修改成原来第 3 个维度放在第 1 个维度，第 1 个维度放在第 2 个维度，第 2 个维度放在第 3 个维度
>>> a.permute(2, 0, 1).shape
torch.Size([1, 2, 3])
```

squeeze()

在指定索引位置删减维度，如果不传入索引，将会把所有能删减的维度（值为 1）都删减了，举例：

```
>>> a = torch.rand(1, 2, 1, 1)
>>> a
tensor([[[[0.3160],
           [0.5993]]]])
>>> a.squeeze()
tensor([0.3160, 0.5993])
>>> a.squeeze().shape
```

```
torch.Size([2])
# 可以看出删减了所有能删减的维度
>>> a.squeeze(0)
tensor([[0.3160]],
        [[0.5993]])
>>> a.squeeze(0).shape
torch.Size([2, 1, 1])
# 删减了第1维
>>> a.squeeze(1)
tensor([[[[0.3160]],
          [[0.5993]]]])
>>> a.squeeze(1).shape
torch.Size([1, 2, 1, 1])
# 第2维因为无法删减，所以没有变化
```

unsqueeze()

在指定索引位置增加维度，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.8979, 0.5201, 0.2911],
        [0.8355, 0.2032, 0.9345]])
>>> a.unsqueeze(0)
tensor([[[0.8979, 0.5201, 0.2911],
          [0.8355, 0.2032, 0.9345]]])
# 在第1维增加维度
>>> a.unsqueeze(0).shape
torch.Size([1, 2, 3])
# 可以看出(2, 3)→(1, 2, 3)
>>> a.unsqueeze(-1)
tensor([[[0.8979],
          [0.5201],
          [0.2911]],
```

```
        [[0. 8355],
         [0. 2032],
         [0. 9345]])
# 在最后 1 维增加维度
>>> a.unsqueeze(-1).shape
torch.Size([2, 3, 1])
# 可以看出 (2, 3)→(2, 3, 1)
```

expand()

扩展数据，但仅限于维度是 1 的地方，举例：

```
>>> a = torch.rand(1, 2, 1)
>>> a
tensor([[[0. 5487],
         [0. 9694]]])
>>> a.expand([2, 2, 1])
tensor([[[0. 5487],
         [0. 9694]],
        [[0. 5487],
         [0. 9694]]])
# 扩展了第 1 维度的数据
>>> a.expand([2, 2, 2])
tensor([[[0. 5487, 0. 5487],
         [0. 9694, 0. 9694]],
        [[0. 5487, 0. 5487],
         [0. 9694, 0. 9694]]])
# 扩展了第 1/3 维度的数据
>>> a.expand([1, 4, 1])
Traceback (most recent call last):
  File "<pyshell#242>", line 1, in <module>
    a.expand([1, 4, 1])
```


RuntimeError: The expanded size of the tensor (4) must match the existing size (2) at non-singleton dimension 1. Target sizes: [1, 4, 1]. Tensor sizes: [1, 2, 1]
因为第 2 维度不为 1，所以不能扩展

`repeat()`

复制数据，对指定维度复制指定倍数，举例：

```
>>> a = torch.rand(1, 2, 1)
>>> a.repeat([2, 2, 1]).shape
torch.Size([2, 4, 1])
# 将 1/2 维变成原来 2 倍，第 3 维不变
>>> a.repeat([2, 2, 2]).shape
torch.Size([2, 4, 2])
```

`split()`

根据长度切分数据，举例：

```
>>> a = torch.rand(2, 2, 3)
>>> a
tensor([[[0.6913, 0.3448, 0.5107],
         [0.5714, 0.1821, 0.2043]],
        [[0.9937, 0.4512, 0.8015],
         [0.9622, 0.3952, 0.6199]]])
>>> a.split(1, dim=0)
(tensor([[[0.6913, 0.3448, 0.5107],
         [0.5714, 0.1821, 0.2043]]]), tensor([[[0.9937, 0.4512, 0.8015],
         [0.9622, 0.3952, 0.6199]]]))
# 可以看出根据第 1 维将数据按长度 1 切分成了 2/1 份（第 1 维长度是 2）
>>> a.split(1, dim=0).shape
>>> x, y = a.split(1, dim=0)
>>> x.shape, y.shape
(torch.Size([1, 2, 3]), torch.Size([1, 2, 3]))
```

`chunk`

根据数量切分数据，也就是自定义要切成多少份，举例：

```
>>> a = torch.rand(3,3)
>>> a
tensor([[0.3355, 0.0770, 0.1840],
        [0.0844, 0.4452, 0.8723],
        [0.9296, 0.4290, 0.4051]])
>>> a.chunk(3, dim=0)
(tensor([[0.3355, 0.0770, 0.1840]]), tensor([[0.0844, 0.4452, 0.8723]]), tensor([[0.9296, 0.4290, 0.4051]]))
# 把数据切成 3 份
>>> a.chunk(2, dim=0)
(tensor([[0.3355, 0.0770, 0.1840],
        [0.0844, 0.4452, 0.8723]]), tensor([[0.9296, 0.4290, 0.4051]]))
# 切成 2 份，可以看到最后一个被独立出来了
```

常用方法

数组/张量转换

`torch.from_numpy`

数组转张量，举例：

```
>>> a = np.array([1, 2, 3])
>>> torch.from_numpy(a)
tensor([1, 2, 3], dtype=torch.int32)
```

张量转数组则通过 `data.numpy()` 转，若为 GPU 数据，则先转成 CPU 的，也可以通过 `np.array(tensor)` 强行转成数组，举例：

```
>>> a = torch.tensor([1., 2., 3.])
>>> a
tensor([1., 2., 3.])
```

```
>>> a.cpu().data.numpy()
array([1., 2., 3.], dtype=float32)
# 转成 CPU 数据，然后转数组
>>> np.array(a)
array([1., 2., 3.], dtype=float32)
# 强制转成数组
```

基本运算

torch.add

加法运算，一般情况下也可以用+号代替，举例：

```
>>> a = torch.rand(2, 2)
>>> a
tensor([[0.5643, 0.4722],
        [0.5939, 0.6289]])
>>> torch.add(a, b)
tensor([[1.5643, 1.4722],
        [1.5939, 1.6289]])
>>> a + b
tensor([[1.5643, 1.4722],
        [1.5939, 1.6289]])
# 可以看出结果一样
```

torch.sub

张量减法

torch.mul

张量乘法

torch.div

张量除法

`torch.matmul`

矩阵乘法，举例：

```
>>> a = torch.tensor([[1, 1], [1, 1]])
>>> b = torch.tensor([[1, 1], [1, 1]])
>>> torch.matmul(a, b)
tensor([[2, 2],
        [2, 2]])
>>> a*b
tensor([[1, 1],
        [1, 1]])
# 别把数组乘法和矩阵乘法弄混了
```

矩阵乘法还可以用@代替，举例：

```
>>> a@b
tensor([[2, 2],
        [2, 2]])
# 结果和前面矩阵乘法一样
```

逻辑操作

`torch.equal`

判断两个张量是否完全相等，返回 True 或者 False，而==符号返回的只是一个由 0 和 1 组成的张量，举例：

```
>>> a = torch.rand(2, 2)
>>> a
tensor([[0.8146, 0.1331],
        [0.6715, 0.4594]])
>>> b = a
>>> a == b
```

```
tensor([[1, 1],
        [1, 1]], dtype=torch.uint8)
# 判断两个张量每个元素是否相等，并用 0 和 1 来表示是否相等
>>> torch.equal(a, b)
True
# 判断两个张量是否相等并返回结果
```

torch.all

逻辑与操作，判断是否全为 1，举例：

```
>>> a = torch.tensor([1, 2, 3])
>>> b = torch.tensor([1, 2, 4])
>>> a
tensor([1, 2, 3])
>>> b
tensor([1, 2, 4])
>>> torch.all(a==b)
tensor(0, dtype=torch.uint8)
>>> a==b
tensor([1, 1, 0], dtype=torch.uint8)
```

torch.any

逻辑或操作，判断是否存在 1

torch.where

有三个参数 a, b, c，对数据 a 进行逻辑判断，为 1 的取数据 b 上对应位置的值，为 0 取 c 上对应值，举例：

```
>>> a = torch.tensor([[1., 2.], [3., 4.]])
>>> b = torch.tensor([[5., 6.], [7., 8.]])
>>> c = torch.rand(2, 2)
>>> c
tensor([[0.3821, 0.6138],
```

```
[0.2323, 0.2675]])
>>> torch.where(c>0.3, a, b)
tensor([[1., 2.],
        [7., 8.]])
# 位置(0,0)为1, 所以取 b 上(0,0)的值, 即 1
# 位置(0,1)为1, 所以取 b 上(0,1)的值, 即 2
# 位置(1,0)为0, 所以取 c 上(1,0)的值, 即 7
# 位置(1,1)为0, 所以取 c 上(1,1)的值, 即 8
```

数据生成

`torch.zeros`

生成固定尺寸的全 0 张量, 举例:

```
>>> torch.zeros(2,2)
tensor([[0., 0.],
        [0., 0.]])
```

`torch.ones`

生成固定尺寸的全 1 张量, 举例:

```
>>> torch.ones(2,2)
tensor([[1., 1.],
        [1., 1.]])
```

`torch.full`

生成固定尺寸的值全为指定值的张量, 举例:

```
>>> torch.full([2, 3], 2)
tensor([[2., 2., 2.],
        [2., 2., 2.]])
# 指定格式且值全为 2
```

torch.eye

生成对角线上值全为 1 的张量，举例：

```
>>> torch.eye(3)
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
# 生成 3*3 张量
>>> torch.eye(3,3)
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
# 和上面一样
>>> torch.eye(3,4)
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.]])
>>> torch.eye(3,2)
tensor([[1., 0.],
        [0., 1.],
        [0., 0.]])
```

torch.arange

生成指定等差数列的张量，举例：

```
>>> torch.arange(10)
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# 只传一个参数 n 则默认为从 0-n-1 的 n 个数
>>> torch.arange(-1, 1, 0.1)
tensor([-1.0000, -0.9000, -0.8000, -0.7000, -0.6000, -0.5000, -0.4000, -0.3000,
        -0.2000, -0.1000,  0.0000,  0.1000,  0.2000,  0.3000,  0.4000,  0.5000,
         0.6000,  0.7000,  0.8000,  0.9000])
# 生成-1 到 1，且距离为 0.1 的等差数列
```

`torch.linspace`

也是生成等差数列的张量，用法和 `arrange` 稍有不同，举例：

```
>>> torch.linspace(-1, 1, steps=21)
tensor([-1.0000, -0.9000, -0.8000, -0.7000, -0.6000, -0.5000, -0.4000, -0.3000,
        -0.2000, -0.1000,  0.0000,  0.1000,  0.2000,  0.3000,  0.4000,  0.5000,
         0.6000,  0.7000,  0.8000,  0.9000,  1.0000])
# 生成一个长度为 21 的等差数列，且值为-1 到 1
# 可以看出和上面等价，但是这个方便设置数据量，上面的方便设置距离
```

`torch.logspace`

和 `linspace` 用法相似，可以理解成再 `linspace` 的基础上对其求 10 的 n 次方值，举例：

```
>>> torch.logspace(0, 10, steps=11)
tensor([1.0000e+00, 1.0000e+01, 1.0000e+02, 1.0000e+03, 1.0000e+04, 1.0000e+05,
        1.0000e+06, 1.0000e+07, 1.0000e+08, 1.0000e+09, 1.0000e+10])
# 10 的 0 次方、1 次方、2 次方、...
```

`torch.rand`

随机生成一个固定尺寸的张量，并且数值范围都在 $0 \sim 1$ 之间，举例：

```
>>> torch.rand((2, 3))
tensor([[0.6340, 0.4699, 0.3745],
        [0.5066, 0.3480, 0.7346]])
```

`torch.randn`

也是随机生成固定尺寸的张量，数值符合正态分布

`torch.randint`

随机生成一个固定尺寸的张量，并且数值为自定义范围的整数，举例：


```
>>> torch.randint(0, 10, (2, 3))
tensor([[4, 3, 1],
        [6, 3, 9]])
# 0~9 的固定格式整数
```

torch.randperm

生成指定个数的张量（范围为 0~个数-1）并打乱，举例：

```
>>> torch.randperm(10)
tensor([0, 4, 9, 5, 1, 7, 3, 6, 2, 8])
# 生成 0~9 的数，并打乱
```

torch.rand_like

传入一个张量，并根据该张量 shape 生成一个新的随机张量，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.4079, 0.9071, 0.9304],
        [0.0641, 0.0043, 0.0429]])
>>> torch.rand_like(a)
tensor([[0.2936, 0.4585, 0.7674],
        [0.4049, 0.0707, 0.0456]])
# 生成一个和 a 格式相同的张量
```

注：

有好多 xxx_like 的方法，原理都是一样的：传入一个张量，根据张量的 shape 生成新的张量

数据处理

torch.masked_select

取出指定条件数据，举例：

```
>>> a = torch.tensor([0, 0.5, 1, 2])
>>> torch.masked_select(a, a>0.5)
tensor([1., 2.])
# 取出所有大于 0.5 的数据
```

torch.cat

在指定维度合并数据，但要求两个数据维度相同，并且指定维度以外的维度尺寸相同，举例：

```
>>> a = torch.rand(1, 2, 3)
>>> b = torch.rand(2, 2, 3)
>>> torch.cat((a, b))
tensor([[[[0.0132, 0.4118, 0.5814],
          [0.8034, 0.8765, 0.8404]],
        [[0.7860, 0.6115, 0.4745],
          [0.0846, 0.4158, 0.3805]],
        [[0.9454, 0.3390, 0.3802],
          [0.6526, 0.0319, 0.7155]]]])
>>> torch.cat((a, b)).shape
torch.Size([3, 2, 3])
# 可以看出默认在第 1 维合并数据，合并过程可以看成：[1, 2, 3] + [2, 2, 3] = [3, 2, 3]
>>> torch.cat((a, b), dim=2)
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    torch.cat((a, b), dim=2)
RuntimeError: invalid argument 0: Sizes of tensors must match except in dimension 2. Got 1 and 2 in dimension 0 at
d:\build\pytorch\pytorch-1.0.1\aten\src\th\generic\thtensormoremath.cpp:1307
# 在第 3 维合并数据时因为 1 维（非 3 维部分）不一样所以报错
```

torch.stack

在指定维度创建一个新维度合并两个数据，举例：

```
>>> a = torch.rand(1, 2, 3)
>>> b = torch.rand(1, 2, 3)
```

```

>>> a
tensor([[[0.5239, 0.0540, 0.0213],
         [0.9713, 0.5983, 0.1413]]])
>>> b
tensor([[[0.3397, 0.0976, 0.3744],
         [0.5080, 0.7520, 0.1759]]])
>>> torch.stack((a, b))
tensor([[[[0.5239, 0.0540, 0.0213],
          [0.9713, 0.5983, 0.1413]]],
        [[0.3397, 0.0976, 0.3744],
          [0.5080, 0.7520, 0.1759]]]])
>>> torch.stack((a, b)).shape
torch.Size([2, 1, 2, 3])
# 合并过程可以看成: [1, 1, 2, 3] + [1, 1, 2, 3] = [1+1, 1, 2, 3] = [2, 1, 2, 3]
>>> torch.stack((a, b), dim=2)
tensor([[[[0.5239, 0.0540, 0.0213],
          [0.3397, 0.0976, 0.3744]],
        [[0.9713, 0.5983, 0.1413],
          [0.5080, 0.7520, 0.1759]]]])
>>> torch.stack((a, b), dim=2).shape
torch.Size([1, 2, 2, 3])
# 合并过程可以看成: [1, 2, 1, 3] + [1, 2, 1, 3] = [1, 2, 1+1, 3] = [1, 2, 2, 3]

```

其他操作

one-hot 编码

通过结合 `torch.zeros()` 方法和 `tensor.scatter_()` 方法实现，举例：

```

>>> label = torch.tensor([[0], [1], [2], [3]])
# 标签内容
>>> label_number = len(label)
# 标签长度
>>> label_range = 10

```

标签总数

```
>>> torch.zeros(label_number, label_range).scatter_(1, label, 1)
tensor([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]])
```

通过 torch.zeros 生成 4 行, 10 列的全零矩阵, 通过 torch.scatter_ 以标签第一维为基准, 用 1 来覆盖对应的位置

带下划线的方法

在 pytorch 里面可以看到很多方法有两种版本: 带下划线和不带下划线的, 这两者的区别就是: 不带下划线的方法操作数据会先新建一个相同的数据, 然后对其进行操作后返回; 带下划线的则直接对该数据进行操作并返回 (声明该 tensor 是个 in-place 类型), 举例:

```
>>> a = torch.tensor((1., -2.))
>>> a.abs()
tensor([1., 2.])
>>> a
tensor([ 1., -2.])
# 不带下划线的取绝对值后, 查看原来的数据, 发现没有变
>>> a.abs_()
tensor([1., 2.])
>>> a
tensor([1., 2.])
# 带下划线的取绝对值后, 查看原来的数据, 发现已经变了
```

更多关于 in-place 类型的参考:

<https://blog.csdn.net/hbhhhxs/article/details/93886525>

工具集

在 torch.utils 下提供了很多 API 工具方便我们的使用

随机切分数据集

通过 `torch.utils.data.random_split()` 方法随机切分数据集，然后通过 `torch.utils.data.DataLoader` 来载入数据，举例：

```
>>> a = torch.rand(100)
>>> a
tensor([0.8579, 0.6903, 0.8042, 0.6803, 0.5619, 0.4721, 0.3132, 0.6476, 0.6644,
        0.1822, 0.8333, 0.6207, 0.5666, 0.3410, 0.9760, 0.1522, 0.5908, 0.4049,
        0.8710, 0.3284, 0.7598, 0.1615, 0.2269, 0.7273, 0.5658, 0.7861, 0.4562,
        0.4225, 0.0466, 0.2845, 0.2759, 0.0649, 0.7345, 0.7406, 0.0044, 0.2111,
        0.5922, 0.1108, 0.8785, 0.5843, 0.3432, 0.1751, 0.8386, 0.8131, 0.5848,
        0.3727, 0.4079, 0.3207, 0.7192, 0.5415, 0.2176, 0.3019, 0.9200, 0.1222,
        0.1771, 0.7479, 0.1213, 0.7306, 0.7951, 0.6702, 0.4286, 0.6684, 0.4392,
        0.5319, 0.8701, 0.5307, 0.0664, 0.6950, 0.8652, 0.8842, 0.1940, 0.5079,
        0.1927, 0.3511, 0.6232, 0.5951, 0.7436, 0.3113, 0.8578, 0.6422, 0.6670,
        0.5569, 0.4681, 0.3848, 0.5463, 0.2438, 0.7747, 0.2718, 0.8766, 0.3523,
        0.1736, 0.9693, 0.6800, 0.6727, 0.9430, 0.5596, 0.7665, 0.8402, 0.3828,
        0.6339])
>>> m, n = torch.utils.data.random_split(a, [10, 90])
# 将数据随机分成 10 和 90 个
>>> list(m)
[tensor(0.6803), tensor(0.8386), tensor(0.4079), tensor(0.8652), tensor(0.8333), tensor(0.8402), tensor(0.7861), tensor(0.8710),
 tensor(0.7306), tensor(0.5848)]
# 查看数据集 m
>>> m.indices
tensor([ 3, 42, 46, 68, 10, 97, 25, 18, 57, 44])
# 可以看到其存放的是随机的 10 个索引，然后寻找对应下标数据
>>> data_m = torch.utils.data.DataLoader(m, batch_size=2)
# 载入 m 数据
>>> data_n = torch.utils.data.DataLoader(n, batch_size=2)
```

函数式 API 和类 API

在 pytorch 当中，大部分神经网络层、激活函数、损失函数等都提供了两种 API 调用方式，分别是函数式 API 和类 API，前者基本都在 `torch.nn.functional` 下，后者基本都在 `torch.nn` 下，前者一般直接调用即可，适合函数式编程；后者一般是先实例化，然后通过其内置的方法进行调用，适合面向对象编程。当然这些 API 功能基本都可以自定义实现，只是这里提供了 API 简化了操作，并且还提供了 GPU 加速等功能

网络层

全连接层

说白了就是单纯的矩阵相乘然后有偏置则加上偏置（公式： $\text{input} @ \mathbf{w} + \mathbf{b}$ ），函数式 API：`torch.nn.functional.linear`，类 API：`torch.nn.Linear`，类 API 举例：

```
>>> layer1 = torch.nn.Linear(100, 10)
# 这里使用类 API
# 定义一个全连接层，输入 100 个单元，输出 10 个，可以理解成初始化的一个 (100, 10) 的矩阵
>>> layer2 = torch.nn.Linear(10, 1)
>>> x = torch.rand(1, 100)
# 定义一个 (1, 100) 的矩阵
>>> x = layer1(x)
# x 经过 layer1 全连接层的运算
>>> x
tensor([[ -0.1354,  0.1530,  0.1946, -0.1349,  0.6149, -0.0482,  0.1025, -0.8483,
          -1.0567, -0.5853]], grad_fn=<AddmmBackward>)
>>> x.shape
torch.Size([1, 10])
# 可以发现乘完以后变成了 (1, 10) 的矩阵
>>> x = layer2(x)
# x 再经过 layer2 层运算
>>> x
tensor([[ -0.2182]], grad_fn=<AddmmBackward>)
>>> x.shape
torch.Size([1, 1])
>>> layer1.weight.shape
torch.Size([10, 100])
```

可以通过 weight 属性查看当前层的权值，计算的时候会将权值矩阵进行转置后才进行运算，所以是 (10, 100) 而不是 (100, 10)

```
>>> layer1.bias
```

Parameter containing:

```
tensor([ 0.0049, -0.0081, -0.0541, -0.0301,  0.0320, -0.0621,  0.0072, -0.0024,
        -0.0339,  0.0456], requires_grad=True)
```

可以通过 bias 属性查看当前层的偏置值

函数式 API 举例：

```
>>> x = torch.rand(1, 100)
```

```
>>> w = torch.rand(10, 100)
```

```
>>> x = torch.nn.functional.linear(x, w)
```

可以看出函数式 API 需要我们自己定义初始化权值，然后直接调用即可

```
>>> x
```

```
tensor([[25.9789, 23.4787, 24.2929, 25.8615, 22.0681, 23.1044, 22.0457, 22.0386,
        23.0654, 24.6127]])
```

通过上面对比我们可以发现对于类 API，我们只需实例化对应的类，然后在其的__init__方法里会对权值之类的数据进行初始化，然后我们传入自己的数据进行调用运算；而在函数式 API 当中，首先我们需要自己定义初始化的权值，然后通过往 API 接口传入数据和权值等数据进行运算

注：

通过上面我们可以看出每一层操作的时候可以实时打印查看其权值之类的变化，以供我们观察，这也是 pytorch 作为动态图和 TensorFlow 最大的区别

Dropout

随机选取一部分节点使用，忽略一部分节点，函数式 API：torch.nn.functional.dropout，类 API：torch.nn.Dropout，举例：

```
>>> a = torch.rand(20)
```

```
>>> torch.nn.functional.dropout(a, 0.2)
```

```
tensor([1.2178, 1.0375, 0.0555, 0.0307, 0.3235, 0.0000, 0.5209, 0.0000, 0.3346,
        1.2383, 0.3606, 1.0937, 0.0000, 0.2957, 0.9463, 0.2932, 0.8088, 0.4445,
        0.5565, 0.0241])
```

随机将百分之 20 的节点转成 0

批标准化层

函数式 API: `torch.nn.functional.batch_norm`, 类 API: `torch.nn.BatchNorm2d` (对应的有 1d、2d 等等), 类 API 举例:

```
>>> x1 = torch.rand(1, 3, 784)
# 3 通道的 1d 数据
>>> layer1 = torch.nn.BatchNorm1d(3)
# 1d 批标准化层, 3 通道
>>> layer1.weight
Parameter containing:
tensor([1., 1., 1.], requires_grad=True)
# 可以看出 batch_norm 层的权值全是 1
>>> layer1(x1)
tensor([[[[-0.0625, -0.1859, -0.3823, ..., 0.6668, -0.7487, 0.8913],
          [ 0.0115, -0.1149, 0.1470, ..., -0.1546, 0.3012, 0.2472],
          [ 1.5185, -0.4740, -0.8664, ..., 0.6266, 0.2797, -0.2975]]],
        grad_fn=<NativeBatchNormBackward>])
# 可以看到数据都被标准化了
>>> layer1(x1).shape
torch.Size([1, 3, 784])
>>> x2 = torch.rand(1, 3, 28, 28)
# 3 通道的 2d 数据
>>> layer2 = torch.nn.BatchNorm2d(3)
>>> layer2(x2)
tensor([[[[[-0.0378, -0.3922, 0.2255, ..., -0.1469, -0.3016, 0.2384],
          [-0.3901, -0.0220, -0.3118, ..., -0.2492, 0.1705, -0.0599],
          [-0.1309, -0.3064, -0.2001, ..., -0.0613, -0.1838, 0.1335],
          ...,
          [ 0.9022, -0.3031, 1.0695, ..., -0.8257, -0.6438, -0.2672],
          [-0.1015, 1.1482, 1.0834, ..., 0.6641, -0.8632, -0.2418],
          [-1.2068, -0.7443, 0.8346, ..., 0.1213, 0.4528, -0.5756]]]],
        grad_fn=<NativeBatchNormBackward>])
>>> layer2(x2).shape
# 经过 batch_norm 只是将数据变得符合高斯分布, 并不会改变数据形状
torch.Size([1, 3, 28, 28])
>>> x2.mean()
```



```
tensor(0.4942)
# 原来数据的平均值
>>> x2.std()
tensor(0.2899)
# 原来数据的标准差
>>> layer2(x2).mean()
tensor(-2.1211e-08, grad_fn=<MeanBackward0>)
# 经过 batch_norm 的平均值，可以看出经过 batch_norm 层后数据平均值变成接近 0
>>> layer2(x2).std()
tensor(1.0002, grad_fn=<StdBackward0>)
# 经过 batch_norm 的标准差，可以看出经过 batch_norm 层后数据标准差变成接近 1
```

卷积层

函数式 API: `torch.nn.functional.conv2d`, 类 API: `torch.nn.Conv2d`, 类 API 举例:

```
>>> x = torch.rand(1, 1, 28, 28)
>>> layer = torch.nn.Conv2d(1, 3, kernel_size=3, stride=1, padding=0)
# 设置输入通道为 1, 输出通道为 3, filter 大小为 3x3, 步长为 1, 边框不补 0
>>> layer.weight
Parameter containing:
tensor([[[[-0.1893,  0.1177, -0.2837],
          [ 0.1116,  0.0348,  0.3011],
          [-0.1871, -0.0722, -0.1843]]],
        ...,
        [[ [ 0.0083, -0.0784,  0.1592],
          [-0.1896,  0.0082, -0.0146],
          [-0.2069, -0.0147, -0.1899]]]], requires_grad=True)
# 可以查看初始化权值
>>> layer.weight.shape
torch.Size([3, 1, 3, 3])
# 格式分别代表输出通道 3, 输入通道 1, 尺寸为 3x3
>>> layer.bias.shape
torch.Size([3])
```

查看初始化偏置

```
>>> layer(x)
tensor([[[[-0.0494, -0.1396, -0.0690, ..., -0.1382, -0.0539, -0.1876],
          [-0.2185, -0.0116, -0.1287, ..., 0.1233, -0.0091, 0.0407],
          [-0.0648, 0.0506, -0.1971, ..., -0.2013, 0.1151, -0.0026],
          ...,
          [-0.4974, -0.5449, -0.4583, ..., -0.7153, -0.1890, -0.7381],
          [-0.4254, -0.6051, -0.2578, ..., -0.4957, -0.4128, -0.4875],
          [-0.5392, -0.4214, -0.5671, ..., -0.2785, -0.6113, -0.3150]]]],
        grad_fn=<ThnnConv2DBackward>)
```

进行一次卷积运算，实际是魔法方法__call__里调用了 forward 方法

```
>>> layer(x).shape
```

```
torch.Size([1, 3, 26, 26])
```

可以看到计算后由于边框不补 0，而滤波器大小为 3x3，所以结果的长宽就变成了 (height-3+1, weight-3+1)

```
>>> layer1 = torch.nn.Conv2d(1, 3, kernel_size=3, stride=1, padding=1)
```

这里边缘补 0

```
>>> layer1(x).shape
```

```
torch.Size([1, 3, 28, 28])
```

可以看到由于边缘补 0，所以大小没变

```
>>> layer2 = torch.nn.Conv2d(1, 3, kernel_size=3, stride=2, padding=0)
```

这里步长改成 2

```
>>> layer2(x).shape
```

```
torch.Size([1, 3, 13, 13])
```

结果的长宽就变成了 $((\text{height}-3+1)/2, (\text{weight}-3+1)/2)$

```
>>> layer3 = torch.nn.Conv2d(1, 3, kernel_size=3, stride=2, padding=1)
```

这里边缘补 0，且步长改成 2

```
>>> layer3(x).shape
```

```
torch.Size([1, 3, 14, 14])
```

可以看到结果的长宽就变成了 $(\text{height}/2, \text{weight}/2)$

函数式 API 举例：

```
>>> x = torch.rand(1, 1, 28, 28)
```

```
>>> w = torch.rand(3, 1, 3, 3)
```

```
# 输出 3 通道，输入 1 通道，尺寸 3x3
>>> b = torch.rand(3)
# 偏置长度要和通道数一样
>>> layer = torch.nn.functional.conv2d(x, w, b, stride=1, padding=1)
>>> layer
tensor([[[[2.1963, 2.6321, 3.4186, ..., 3.2495, 3.1609, 2.5473],
          [2.5637, 3.4892, 4.0079, ..., 4.1167, 4.4497, 3.1637],
          [2.7618, 3.2788, 3.2314, ..., 4.7185, 4.3128, 2.6393],
          ...,
          [1.3735, 2.3738, 1.8388, ..., 2.9912, 2.6638, 1.5941],
          [2.1967, 2.0466, 2.0095, ..., 3.3192, 2.9521, 2.2673],
          [1.6091, 2.1341, 1.5108, ..., 2.1684, 2.4585, 1.7931]]]])
>>> layer.shape
torch.Size([1, 3, 28, 28])
```

池化层

函数式 API: `torch.nn.functional.max_pool2d`, 类 API: `torch.nn.MaxPool2d`, 类 API 举例:

```
>>> x = torch.rand(1, 1, 28, 28)
>>> layer = torch.nn.MaxPool2d(3, stride=2)
# 尺寸 3x3, 步长为 2
>>> layer(x)
tensor([[[[0.9301, 0.9342, 0.9606, 0.9922, 0.9754, 0.9055, 0.7142, 0.9882,
          0.9803, 0.8054, 0.9903, 0.9903, 0.9426],
          ...,
          [0.8873, 0.8873, 0.9324, 0.9876, 0.9566, 0.9225, 0.9673, 0.9675,
          0.9977, 0.9977, 0.9552, 0.9552, 0.8689]]]])
>>> layer(x).shape
torch.Size([1, 1, 13, 13])
```

还有个 avgpool (函数式 API: `torch.nn.functional.avg_pool2d`, 类 API: `torch.nn.AvgPool2d`), 和 maxpool 不一样的是: maxpool 是取最大值, 而 avgpool 取的是平均值, 举例:

```
>>> torch.nn.functional.avg_pool2d(x, 3, stride=2)
```

```

tensor([[[[0.5105, 0.6301, 0.5491, 0.4691, 0.5788, 0.4525, 0.3903, 0.5718,
          0.6259, 0.3388, 0.4169, 0.6122, 0.4760],
          ...,
          [0.4705, 0.5332, 0.4150, 0.5000, 0.5686, 0.5325, 0.6241, 0.4926,
          0.4646, 0.3121, 0.2975, 0.5203, 0.5701]]]])
>>> torch.nn.functional.avg_pool2d(x, 3, stride=2).shape
torch.Size([1, 1, 13, 13])

```

flatten

将张量转成一维，举例：

```

>>> torch.flatten(torch.rand(2, 2))
tensor([0.1339, 0.5694, 0.9034, 0.6025])
>>> torch.flatten(torch.rand(2, 2)).shape
torch.Size([4])

```

向上取样

将图片放大/缩小成原来的几倍，函数式 API：torch.nn.functional.interpolate，举例：

```

>>> x = torch.rand(1, 1, 2, 2)
# 可以理解成 1 张 1 通道的 2x2 图片
>>> x
tensor([[[[0.9098, 0.7948],
          [0.0670, 0.3906]]]])
>>> torch.nn.functional.interpolate(x, scale_factor=2, mode='nearest').shape
torch.Size([1, 1, 4, 4])
# 可以看到数据被放大了一倍
>>> torch.nn.functional.interpolate(x, scale_factor=2, mode='nearest')
tensor([[[[0.9098, 0.9098, 0.7948, 0.7948],
          [0.9098, 0.9098, 0.7948, 0.7948],
          [0.0670, 0.0670, 0.3906, 0.3906],
          [0.0670, 0.0670, 0.3906, 0.3906]]]])
# 可以看到是往横纵向都复制成原来的对应倍数

```

```
>>> torch.nn.functional.interpolate(x, scale_factor=0.5, mode='nearest')
tensor([[[[0.9098]]]])
# 将数据缩小一倍，可以看到取那一部分的第一个数据
```

注：

还有如 Upsampling、UpsamplingNearest2d 也是向上采样，现在已经逐渐被 interpolate 给取代。上面 interpolate 示例参数中 mode='nearest' 时，相当于该 UpsamplingNearest2d，函数式 API：torch.nn.functional.upsample_nearest，函数式 API：torch.nn.UpsamplingNearest2d，类 API 举例：

```
>>> x = torch.rand(1, 1, 2, 2)
>>> x
tensor([[[[0.9977, 0.9778],
          [0.4167, 0.6936]]]])
>>> torch.nn.functional.upsample_nearest(x, scale_factor=2).shape
torch.Size([1, 1, 4, 4])
>>> torch.nn.functional.upsample_nearest(x, scale_factor=2)
tensor([[[[0.9977, 0.9977, 0.9778, 0.9778],
          [0.9977, 0.9977, 0.9778, 0.9778],
          [0.4167, 0.4167, 0.6936, 0.6936],
          [0.4167, 0.4167, 0.6936, 0.6936]]]])
```

嵌入层

常用于定义词向量，可以理解 embedding 层定义了一个词典用来存储和表示所有的词向量，而传入的数据则会根据索引找到对应的词向量，函数式 API：torch.nn.functional.embedding，类 API：torch.nn.Embedding，类 API 举例：

```
>>> embed = torch.nn.Embedding(10, 2)
# 定义了 10 个词向量，每个词向量用格式为 (1, 2) 的 tensor 表示
>>> words = torch.tensor([0, 1, 2, 0])
# 定义一句话，里面有 4 个词，那么可以看出第一个和最后一个词相同
>>> embed(words)
# 经过嵌入层索引可以看到 4 个词对应的词向量如下，也可以看出第一个和最后一个词索引相同，所以值是一样的
tensor([[[-0.0019,  1.6786],
         [ 0.3118, -1.6250],
         [ 1.6038,  1.5044],
```

```
[-0.0019, 1.6786]], grad_fn=<EmbeddingBackward>)
>>> embed.weight
# 再看 embedding 层的权重，可以发现这就是定义了一个词向量表，并且会随着训练而更新，从而找出词与词之间的关系
Parameter containing:
tensor([[ -1.8939e-03,  1.6786e+00],
        [ 3.1179e-01, -1.6250e+00],
        [ 1.6038e+00,  1.5044e+00],
        [-6.2278e-01, -2.5135e-01],
        [ 1.6210e+00, -5.6379e-01],
        [-7.3388e-02, -2.0099e+00],
        [ 8.7655e-01,  2.4011e-01],
        [-2.5685e+00,  2.6756e-01],
        [ 4.9723e-01, -8.3784e-01],
        [ 4.2338e-01, -1.9839e+00]], requires_grad=True)
```

更多参考: <https://blog.csdn.net/tommorrow12/article/details/80896331>

RNN 层

类 API: `torch.nn.RNN`，会返回计算后总体的输出，以及最后一个时间戳上的输出，通过下面代码可以证明最后一个时间戳的输出和总体输出的最后一个是一样的，类 API 举例：

```
>>> x = torch.randn(10, 3, 100)
# 模拟句子序列：有 10 个单词（序列长度是 10），共 3 句话，每个单词用 100 维向量表示
# input: [seq_len, batch, input_size]，如果希望 batch_size 放第一个，可以设置 batch_first=True
>>> layer = torch.nn.RNN(input_size=100, hidden_size=20, num_layers=4)
>>> layer
RNN(100, 20, num_layers=4)
>>> out, h = layer(x)
# 返回 output 和 hidden
>>> out.shape
torch.Size([10, 3, 20])
# 所有时间戳上的状态
# output: [seq_len, batch, hidden_size]
```

```

>>> h.shape
torch.Size([4, 3, 20])
# 最后一个时间戳上的 hidden
# hidden: [num_layers, batch, hidden_size]
>>> h[-1]
# 最后一层的最后一个时间戳上的输出（因为 num_layers 的值为 4，所以要取第四个，对于 num_layers 参数的解释，下面会说）
tensor([[ 3.5205e-01,  3.6580e-01, -5.6378e-01, -9.9363e-02,  3.8728e-03,
         -5.0282e-01,  1.4762e-01, -2.5631e-01, -8.8786e-03,  1.2912e-01,
          4.7565e-01, -8.8090e-02, -3.9374e-02,  3.1736e-02,  3.1264e-01,
          2.8091e-01,  5.0764e-01,  2.9722e-01, -3.6929e-01, -5.1096e-02],
        ...
        [ 5.4770e-01,  4.8047e-01, -5.2541e-01,  2.5208e-01, -4.0260e-04,
         -2.3619e-01, -2.1128e-01, -1.1262e-01, -6.2672e-02,  3.5301e-01,
         -4.1065e-02, -3.5043e-02, -4.3008e-01, -1.8410e-01,  2.5826e-01,
          3.5430e-02,  2.5651e-01,  4.5170e-01, -5.4705e-01, -2.4720e-01]],
        grad_fn=<SelectBackward>)
>>> out[-1]
# 所有状态的最后一个输出，可以看到是一样的
tensor([[ 3.5205e-01,  3.6580e-01, -5.6378e-01, -9.9363e-02,  3.8728e-03,
         -5.0282e-01,  1.4762e-01, -2.5631e-01, -8.8786e-03,  1.2912e-01,
          4.7565e-01, -8.8090e-02, -3.9374e-02,  3.1736e-02,  3.1264e-01,
          2.8091e-01,  5.0764e-01,  2.9722e-01, -3.6929e-01, -5.1096e-02],
        ...
        [ 5.4770e-01,  4.8047e-01, -5.2541e-01,  2.5208e-01, -4.0260e-04,
         -2.3619e-01, -2.1128e-01, -1.1262e-01, -6.2672e-02,  3.5301e-01,
         -4.1065e-02, -3.5043e-02, -4.3008e-01, -1.8410e-01,  2.5826e-01,
          3.5430e-02,  2.5651e-01,  4.5170e-01, -5.4705e-01, -2.4720e-01]],
        grad_fn=<SelectBackward>)

```

num_layers 参数的理解：rnn 的基本参数都挺好理解因为其他深度学习框架基本也都一样，而比较特殊的就是 num_layers 参数，其实也很简单，顾名思义就是代表着有几层 rnn，直接一口气帮你定义好直接计算，省的你再去定义一堆 rnn，然后一层一层的算过去，比如上面的示例代码设置 num_layers=4，那么上面的代码可以替换成下面这种：

```

>>> x = torch.randn(10, 3, 100)

```

```
>>> layer1 = torch.nn.RNN(input_size=100, hidden_size=20, num_layers=1)
# 把上面示例代码中 num_layers=4 的 rnn 改成 4 个为 1 的 rnn
>>> layer2 = torch.nn.RNN(input_size=20, hidden_size=20, num_layers=1)
# 因为第一层的 hidden 是 20, 所以后几层的输入都是 20
>>> layer3 = torch.nn.RNN(input_size=20, hidden_size=20, num_layers=1)
>>> layer4 = torch.nn.RNN(input_size=20, hidden_size=20, num_layers=1)
>>> out, h = layer1(x)
>>> out, h = layer2(out)
>>> out, h = layer3(out)
>>> out, h = layer4(out)
>>> out.shape
torch.Size([10, 3, 20])
```

RNN 参数理解参考:

<https://blog.csdn.net/rogerfang/article/details/84500754>

LSTM 层

类 API: torch.nn.LSTM, 因为 LSTM 是基于 RNN 并添加了门控制, 因此返回的时候比 RNN 要多返回一个 cell 单元, 格式和 hidden 一样, 举例:

```
>>> x = torch.randn(10, 3, 100)
>>> layer = torch.nn.LSTM(input_size=100, hidden_size=20, num_layers=4)
>>> layer
LSTM(100, 20, num_layers=4)
>>> out, (h, c) = layer(x)
# 返回 output、hidden 和 cell
>>> out.shape
torch.Size([10, 3, 20])
>>> h.shape
torch.Size([4, 3, 20])
>>> c.shape
torch.Size([4, 3, 20])
# 可以看出和 hidden 格式一样
# cell: [num_layers, batch_size, hidden_size]
```


这里给一个通过前三个数预测后一个数的模型代码示例：

```
# -----  
# 模块导入  
import numpy  
import torch  
from torch import nn  
  
# -----  
# 数据预处理  
data_length = 30  
# 定义 30 个数，通过前三个预测后一个，比如：1, 2, 3->4  
seq_length = 3  
# 通过上面可知序列长度为 3  
  
number = [i for i in range(data_length)]  
li_x = []  
li_y = []  
for i in range(0, data_length - seq_length):  
    x = number[i: i + seq_length]  
    y = number[i + seq_length]  
    li_x.append(x)  
    li_y.append(y)  
#     print(x, '->', y)  
  
data_x = numpy.reshape(li_x, (len(li_x), 1, seq_length))  
# 输入数据格式：seq_len, batch, input_size  
# 这里可能会有误解，seq_len 不是步长，而是你的样本有多少组，即 sample  
# 而 input_size 就是你数据的维度，比如用三个预测一个，就是 3 维  
data_x = torch.from_numpy(data_x / float(data_length)).float()  
# 将输入数据归一化  
data_y = torch.zeros(len(li_y), data_length).scatter_(1, torch.tensor(li_y).unsqueeze_(dim=1), 1).float()  
# 将输出数据设置为 one-hot 编码
```

```

# print(data_x.shape)
# # 格式: torch.Size([27, 1, 3]), 代表: 27 组数据 (batch)、序列步长为 3 (sequence)
# print(data_y.shape)
# # 格式: torch.Size([27, 30]), 代表: 27 组数据, 30 个特征 (features)

# -----
# 定义网络模型
class net(nn.Module):
    # 模型结构: LSTM + 全连接 + Softmax
    def __init__(self, input_size, hidden_size, output_size, num_layer):
        super(net, self).__init__()
        self.layer1 = nn.LSTM(input_size, hidden_size, num_layer)
        self.layer2 = nn.Linear(hidden_size, output_size)
        self.layer3 = nn.Softmax()
    def forward(self, x):
        x, _ = self.layer1(x)
        sample, batch, hidden = x.size()
        # 格式: [27, 1, 32], 代表样本数量, batch 大小以及隐藏层尺寸
        x = x.reshape(-1, hidden)
        # 转成二维矩阵后与全连接进行计算
        x = self.layer2(x)
        x = self.layer3(x)
        return x

model = net(seq_length, 32, data_length, 4)

# -----
# 定义损失函数和优化器
loss_fun = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# -----
# 训练模型

```

```
# 训练前可以先看看初始化的参数预测的结果差距
# result = model(data_x)
# for target, pred in zip(data_y, result):
#     print("{} -> {}".format(target.argmax().data, pred.argmax().data))

# 开始训练 1000 轮
for _ in range(500):
    output = model(data_x)
    loss = loss_fun(data_y, output)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (_ + 1) % 50 == 0:
        print('Epoch: {}, Loss: {}'.format(_, loss.data))

# -----
# 预测结果
result = model(data_x)
for target, pred in zip(data_y, result):
    print("正确结果: {}, 预测: {}".format(target.argmax().data, pred.argmax().data))

# 结果:
# 正确结果: 3, 预测: 3
# 正确结果: 4, 预测: 4
# 正确结果: 5, 预测: 5
# 正确结果: 6, 预测: 6
# 正确结果: 7, 预测: 7
# 正确结果: 8, 预测: 8
# 正确结果: 9, 预测: 9
# 正确结果: 10, 预测: 10
# 正确结果: 11, 预测: 11
# 正确结果: 12, 预测: 12
# 正确结果: 13, 预测: 13
```

```
# 正确结果: 14, 预测: 14
# 正确结果: 15, 预测: 15
# 正确结果: 16, 预测: 16
# 正确结果: 17, 预测: 21
# 正确结果: 18, 预测: 18
# 正确结果: 19, 预测: 27
# 正确结果: 20, 预测: 21
# 正确结果: 21, 预测: 21
# 正确结果: 22, 预测: 21
# 正确结果: 23, 预测: 21
# 正确结果: 24, 预测: 24
# 正确结果: 25, 预测: 25
# 正确结果: 26, 预测: 26
# 正确结果: 27, 预测: 27
# 正确结果: 28, 预测: 28
# 正确结果: 29, 预测: 29
```

当然这个示例使用到的数据极少，只是一个能快速跑来玩玩的程序而已，不必当真...

LSTM 原理理解: https://blog.csdn.net/gzj_1101/article/details/79376798

LSTM 计算过程参考: <https://blog.csdn.net/qyk2008/article/details/80225986>

序列化模型

即用来定义神经网络模型（每一层都要是继承于 `torch.nn` 下的网络），类 API: `torch.nn.Sequential`（该模型就是针对面向对象模式编写，因此不提供函数式 API），举例：

```
>>> net = torch.nn.Sequential(
    torch.nn.Linear(100, 10),
    torch.nn.Dropout(0.7),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 1)
)
>>> net
Sequential(
```

```
(0): Linear(in_features=100, out_features=10, bias=True)
(1): Dropout(p=0.7)
(2): ReLU()
(3): Linear(in_features=10, out_features=1, bias=True)
# 可以直接查看网络结构
```

序列化模型修改

序列化模型可以理解成一个列表，里面按顺序存放了所有的网络层，官方也提供了添加往序列化模型里添加网络层的方法 `add_module(name, layer)`，而修改则可以索引到对应的层直接修改，删除可以通过 `del` 关键字删除，举例：

```
>>> seq = nn.Sequential(nn.Linear(10, 20), nn.Linear(20, 1))
>>> seq
Sequential(
  (0): Linear(in_features=10, out_features=20, bias=True)
  (1): Linear(in_features=20, out_features=1, bias=True)
)
>>> seq.add_module("tanh", nn.Tanh())
# 在最后面添加一个 tanh 激活层
>>> seq
# 可以看到添加成功
Sequential(
  (0): Linear(in_features=10, out_features=20, bias=True)
  (1): Linear(in_features=20, out_features=1, bias=True)
  (tanh): Tanh()
)
>>> seq[2]
Tanh()
>>> seq[2] = nn.ReLU()
# 修改第三层为 relu
>>> seq
# 可以看到修改成功
Sequential(
  (0): Linear(in_features=10, out_features=20, bias=True)
```

```

    (1): Linear(in_features=20, out_features=1, bias=True)
    (tanh): ReLU()
)
>>> del seq[2]
# 删除第三层
>>> seq
# 可以看到删除成功
Sequential(
  (0): Linear(in_features=10, out_features=20, bias=True)
  (1): Linear(in_features=20, out_features=1, bias=True)
)

```

修改网络参数

对于网络层中的参数，其是一个 Parameter 类型，因此如果我们需要手动修改其参数时，可以通过定义一个该类的数据来赋值修改，举例：

```

>>> layer = nn.Linear(2, 1)
>>> layer.weight
Parameter containing:
tensor([[0.6619, 0.2653]], requires_grad=True)
>>> type(layer.weight)
# 参数的数据类型
<class 'torch.nn.parameter.Parameter'>
>>> layer.weight = torch.rand(2, 1, requires_grad=True)
# 直接赋值张量会报错
Traceback (most recent call last):
  File "<pyshell#150>", line 1, in <module>
    layer.weight = torch.rand(2, 1)
  File "D:\python\lib\site-packages\torch\nn\modules\module.py", line 604, in __setattr__
    .format(torch.typename(value), name))
TypeError: cannot assign 'torch.FloatTensor' as parameter 'weight' (torch.nn.Parameter or None expected)
>>> layer.weight = nn.parameter.Parameter(torch.rand(2, 1))
# 赋值 Parameter 类型的则可以
>>> layer.weight

```

```
# 可以看到修改成功
Parameter containing:
tensor([[0.7412],
        [0.9723]], requires_grad=True)
```

自定义神经网络

当需要自己定义神经网络层的时候，首先需要继承于 `torch.nn.Module`，并在初始化时调用父类的初始化方法，同时也要在 `forward` 方法里实现数据的前向传播，举例：

```
import torch
```

```
class Dense(torch.nn.Module):
    # 实现一个自定义全连接+relu 层，继承 torch.nn.Module
    def __init__(self, input_shape, output_shape):
        super(Dense, self).__init__()
        # 首先初始化时执行父类的初始化，这句话可以看
        # 在父类初始化中会初始化很多变量
        self.w = torch.nn.Parameter(torch.randn(output_shape, input_shape))
        # 初始化权重和偏置参数
        # 使用 Parameter 其会自动将参数设置为需要梯度信息，并且可以通过内置的 parameters 方法返回这些参数
        self.b = torch.nn.Parameter(torch.rand(output_shape))
        self.relu = torch.nn.ReLU()
        # 初始化 relu 层

    def forward(self, x):
        # 定义前向传播方法
        x = x @ self.w.t() + self.b
        # 全连接层的功能就是矩阵相乘计算
        x = self.relu(x)
        # 进行 relu 层计算
        return x

    def __call__(self, x):
```

```
# 调用该类对象执行时，调用前向传播方法
# 这个可以不写，直接通过调用 forward 方法也一样
return self.forward(x)
```

```
layer = Dense(10, 1)
x = torch.rand(2, 10)
output = layer(x)
print(output)
# 输出结果:
# tensor([[0.1780],
#         [0.0000]], grad_fn=<ThresholdBackward0>)
```

冻结网络层

如果希望训练过程当中，对某些网络层的权重不进行训练的话（该场景在迁移学习当中比较常见），可以设置该层的权重、偏差等属性为 False，举例：

```
>>> net = torch.nn.Sequential(
    torch.nn.Linear(100, 10),
    torch.nn.Dropout(0.7),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 1)
)
>>> net
Sequential(
  (0): Linear(in_features=100, out_features=10, bias=True)
  (1): Dropout(p=0.7, inplace=False)
  (2): ReLU()
  (3): Linear(in_features=10, out_features=1, bias=True)
)
>>> for name, value in net.named_parameters():
    print(name, value.requires_grad)
# 可以看到网络层的两个全连接层的权重和偏置都可求导
0.weight True
```



```

0.bias True
3.weight True
3.bias True
>>> net[0].weight.requires_grad = False
# 冻结第一个全连接的权重
>>> net[0].bias.requires_grad = False
>>> for name, value in net.named_parameters():
    print(name, value.requires_grad)
# 可以看到第一个全连接的权重和偏置都被冻结
0.weight False
0.bias False
3.weight True
3.bias True

```

保存和载入网络

对于所有继承自 `torch.nn.Module` 下的网络，保存时首先通过内置的方法 `state_dict()` 返回当前模型的所有参数，然后通过 `torch.save()` 方法保存成文件（也可以不保存参数，直接保存模型，但这样可控性低，不推荐）；载入时通过 `torch.load()` 方法载入文件，并通过内置的 `load_state_dict()` 方法载入所有的参数，举例：

```

>>> layer = torch.nn.Linear(10, 1)
>>> layer.state_dict()
OrderedDict([('weight', tensor([[ -0.1597,  0.0573,  0.0976, -0.1028, -0.1264, -0.0400,  0.0308,  0.2192,
                               -0.0150, -0.3148]])), ('bias', tensor([0.0557]))])
# 可以看到 layer 里定义的参数配置
>>> torch.save(layer.state_dict(), "ckpt.mdl")
# 现在保存这个网络参数
>>> layer1 = torch.nn.Linear(10, 1)
# 新建一个网络
>>> layer1.state_dict()
OrderedDict([('weight', tensor([[ -0.2506, -0.2960, -0.3083,  0.0629,  0.1707,  0.3018,  0.2345, -0.1922,
                               -0.0527, -0.1894]])), ('bias', tensor([-0.0069]))])
# 显然 layer1 参数和 layer 的不一样
>>> layer1.load_state_dict(torch.load("ckpt.mdl"))

```

```
# layer1 载入前面的 layer 网络参数
>>> layer1.state_dict()
OrderedDict([('weight', tensor([[[-0.1597,  0.0573,  0.0976, -0.1028, -0.1264, -0.0400,  0.0308,  0.2192,
        -0.0150, -0.3148]]])), ('bias', tensor([0.0557]))])
# 可以发现 layer1 的参数变得和 layer 保存的参数一样
```

优化器

`torch.optim`

定义了各种优化器，将优化器实例化后（传入需要求导的参数和学习率），通过 `step()` 方法进行梯度下降

- Adam
- SGD

动态调整优化器学习率

对于实例化的优化器，其参数都将存放到一个属性 `param_groups[0]` 里，举例：

```
optim = torch.optim.Adam(model.parameters(), lr=0.01)
print(optim)
```

```
# 结果：
# Adam (
# Parameter Group 0
#   amsgrad: False
#   betas: (0.9, 0.999)
#   eps: 1e-08
#   lr: 0.01
#   weight_decay: 0
# )
```

而 `param_groups[0]` 是一个字典对象，所以要动态修改学习率等参数，可以通过下面代码实现：

```
optim.param_groups[0]['lr'] = new_lr
```

上面介绍的是修改优化器的学习率，如果希望对网络的不同层采用不同的学习率可以参考：

<https://blog.csdn.net/jdzwanghao/article/details/83239111>

激活函数

sigmoid

公式：

$$1 / (1 + e^{-x})$$

注：该公式可以将值控制在 0~1 之间，适合概率之类的问题，但因为当 x 特别大时，导数几乎为 0，容易发生梯度弥散（梯度长时间得不到更新，损失值不下降）之类的问题

函数式 API：torch.nn.functional.sigmoid，类 API：torch.nn.Sigmoid，pytorch 自带：torch.sigmoid，举例：

```
>>> a = torch.linspace(-100, 100, 10)
>>> a
tensor([-100.0000,  -77.7778,  -55.5556,  -33.3333,  -11.1111,   11.1111,
         33.3333,   55.5555,   77.7778,  100.0000])
>>> torch.sigmoid(a)
tensor([0.0000e+00,  1.6655e-34,  7.4564e-25,  3.3382e-15,  1.4945e-05,  9.9999e-01,
        1.0000e+00,  1.0000e+00,  1.0000e+00,  1.0000e+00])
# 可以看到值都被映射在 0~1 之间
```

tanh

公式：

$$(e^x - e^{-x}) / (e^x + e^{-x})$$

注：该公式可以将值控制在 -1~1 之间

函数式 API：torch.nn.functional.tanh，类 API：torch.nn.Tanh，pytorch 自带：torch.tanh

relu

公式:

$$0, x \leq 0$$

$$x, x > 0$$

注: 该公式可以将值控制在 $0 \sim +\infty$ 之间

函数式 API: `torch.nn.functional.relu`, 类 API: `torch.nn.ReLU`

leakyrelu

公式:

$$ax, x \leq 0$$

$$x, x > 0$$

注: a 是一个很小的参数值, 该公式在 `relu` 的基础上, 使负区间的数不为 0, 而是保持在一个很小的梯度上

函数式 API: `torch.nn.functional.leaky_relu`, 类 API: `torch.nn.LeakyReLU`

softmax

公式:

$$e^{x_i} / \sum e^{x_i}$$

注: 该公式可以将值控制在 $0 \sim 1$ 之间, 并且所有的值总和为 1, 适合分类之类的问题, 并且可以发现通过 e 阶函数, 其还会把大的值 (特征) 放大, 小的缩小

函数式 API: `torch.nn.functional.softmax`, 类 API: `torch.nn.Softmax`, pytorch 自带: `torch.softmax`

损失函数

均方差

就是计算的 y 与实际 y 之差的平方取平均值，函数式 API: `torch.nn.functional.mse_loss`，类 API: `torch.nn.MSELoss`，第一个参数是 y ，第二个参数是 y 对应的公式，举例：

```
>>> x = torch.tensor(1.)
>>> w = torch.tensor(2.)
>>> b = torch.tensor(0.)
>>> y = torch.tensor(1.)
>>> mse = torch.nn.functional.mse_loss(y, w*x+b)
# 计算 y=wx+b 在点 (1, 1) 时的均方差:  $(1 - (2*1+0))^2 = 1$ 
>>> mse
tensor(1.)
```

交叉熵

通过含有的信息量大小来进行判断，一般用于分类，函数式 API: `torch.nn.functional.cross_entropy`，类 API: `torch.nn.CrossEntropyLoss`，要注意的是目标 y 的格式要求为 Long 类型，值为每个 one-hot 数据对应的 argmax 处，举例：

```
>>> output = torch.rand(5, 10)
# 输出结果，假设 5 条数据，每条数据有 10 个特征
>>> target = torch.tensor([0, 2, 9, 9, 2]).long()
# 目标 y，数据必须为 long 型，值分别为每条数据的特征，比如第一个 0 代表第一条数据的第一个特征
>>> output.shape
torch.Size([5, 10])
>>> target.shape
torch.Size([5])
# 目标 y 的要求还要求是 1 维的
>>> loss = torch.nn.CrossEntropyLoss()
>>> loss(output, target)
tensor(2.2185)
```

注：

对于分类问题一般会选择最后一层激活函数用 softmax，并且损失函数使用交叉熵，但是在 pytorch 的交叉熵中已经内置了 softmax，所以在使用交叉熵时就不需要再自己使用 softmax 了

二分类交叉熵

顾名思义，是一种特殊的交叉熵，专门在 2 分类时使用，比如 GAN 的判别器里，函数式 API: `torch.nn.functional.binary_cross_entropy`，类 API: `torch.nn.BCELoss`，使用举例：

```
>>> batch_size = 5
# 5 条数据
>>> output = torch.rand(batch_size, 1)
>>> output
# 每个数据的正确率
tensor([[0.3546],
        [0.9064],
        [0.0617],
        [0.2839],
        [0.3106]])
>>> target = torch.ones(batch_size, 1)
>>> target
# 正确的概率是 1
tensor([[1.],
        [1.],
        [1.],
        [1.],
        [1.]])
>>> loss = torch.nn.BCELoss()
>>> loss(output, target)
tensor(1.2697)
```

更多参考

<https://blog.csdn.net/shanglianlm/article/details/85019768>

<https://blog.csdn.net/jackel21/article/details/82812218>

求导机制

在 pytorch 中定义了自动求导的机制，方便了我们在反向传播时更新参数等操作

`torch.autograd.grad`

定义了自动求导，传入第一个参数是对应的公式，第二个参数是一个列表，里面存放所有要求导的变量，并且在求导前的变量需要通过 `require_grad()` 方法来声明该公式的某个变量是需要求导的（或者在定义时就设置 `requires_grad` 参数为 `True`），返回一个元组，里面是对应每个变量的求导信息，举例：

```
>>> x = torch.tensor(1.)
>>> w = torch.tensor(2.)
>>> b = torch.tensor(0.)
>>> y = torch.tensor(1.)
>>> mse = torch.nn.functional.mse_loss(y, w*x+b)
# 模拟一个 y=wx+b 的函数
>>> mse
tensor(1.)
>>> torch.autograd.grad(mse, [w])
# 此时没有变量声明过是需要求导的
>>> w.requires_grad_()
tensor(2., requires_grad=True)
# 声明 w 需要求导，可以看到一开始就定义 w = torch.tensor(2., requires_grad=True) 效果也是一样的
# 加下划线代表为 in-place 类型，直接对 w 进行修改，也可以替换成：w = w.requires_grad_()
>>> torch.autograd.grad(mse, [w])
# 报错，因为 mse 里的 w 还是之前的 w，需要更新一下 mse 里的 w
>>> mse = torch.nn.functional.mse_loss(y, w*x+b)
# 更新 mse 里的 w 为声明了需要求导的 w
>>> torch.autograd.grad(mse, [w])
(tensor(2.),)
# 可以看出 mse 对第一个变量 w 求偏导的结果为 2，计算过程：
# mse 为：(y-(wx+b))^2
# 对 w 求偏导为：-2x(y-(wx+b))
# 代入数值：-2*1*(1-(2*1+0)) = 2
```

`torch.backward`

反向传播，也能实现求导，通过设置该 tensor 允许求导，那么该方法会对计算过程中所有声明了求导信息的变量进行求导，然后在对应的变量上通过 grad 属性获取求导结果，举例：

```
>>> x = torch.tensor(1.)
>>> b = torch.tensor(0.)
>>> w = torch.tensor(2., requires_grad=True)
>>> y = torch.tensor(1.)
>>> mse = torch.nn.functional.mse_loss(y, w*x+b)
>>> mse.backward()
# 对 mse 公式里需要求导的变量都进行求导
>>> w.grad
tensor(2.)
# w 的求导结果为 2
>>> x.grad
# 因为 x 没有声明需要求导，所以为空
```

注：

在反向传播当中，存在梯度累加问题，即第一次进行反向传播以后的值会进行保留，当第二次再进行反向传播时，则会将梯度和前面的进行累加，导致越来越大，因此在大多情况下，为了避免这种情况，需要我们手动进行梯度清零，举例：

```
>>> x = torch.tensor(5., requires_grad=True)
>>> y = x + 1
>>> y.backward()
# 第一次反向传播
>>> x.grad
# 求导结果（梯度）为 1
tensor(1.)
>>> y.backward()
# 第二次反向传播，如果报错（比如 x 的前面乘了一个数值），那么在反向传播里加上参数：retain_graph=True，即：
y.backward(retain_graph=True)
>>> x.grad
# 发现梯度在原来 1 的基础上又加上了 1
tensor(2.)
>>> del x.grad
```



```
# 梯度清零
>>> y.backward()
# 再次反向传播
>>> x.grad
# 可以看到又变回 1 了
tensor(1.)
```

基于上面的情况，在实际模型训练当中，我们首先会用优化器来进行梯度清零，然后再对 loss 进行反向传播，最后再用优化器来进行梯度下降，举例：

```
>>> x = torch.tensor([2.])
# 定义输入
>>> y = torch.tensor([4.])
# 定义输出
>>> layer = nn.Linear(1, 1)
# 定义网络参数
>>> layer.weight
# 可以看到权重 w 为 0.8162
Parameter containing:
tensor([[0.8162]], requires_grad=True)
>>> layer.bias
# 偏置 y 为-0.4772
# 所以可以得出初始化的函数为： $f(x) = 0.8162 * x - 0.4772$ 
Parameter containing:
tensor([-0.4772], requires_grad=True)
>>> optim = torch.optim.SGD(layer.parameters(), lr=0.1)
# 定义随机梯度下降优化器，要更新的是网络层的参数，学习率为 0.1
>>> loss = y - layer(x)
# 定义目标函数，经过网络层后的数和 y 越接近越好
>>> loss
# 计算后可以看出 y 和计算的结果相差 2.8448
tensor([2.8448], grad_fn=<SubBackward0>)
>>> loss.backward()
# 反向传播
```

```
>>> layer.weight.grad
# 权重求导梯度为负，说明正向是在下降
tensor([[ -2. ]])
>>> layer.bias.grad
# 偏置同理
tensor([ -1. ])
>>> optim.step()
# 梯度下降，更新权重和偏置
>>> layer.weight
# 可以看到权重减了-2*0.1
Parameter containing:
tensor([[1.0162]], requires_grad=True)
>>> layer.bias
# 偏置减了-1*0.1
Parameter containing:
tensor([-0.3772], requires_grad=True)
>>> loss = y - layer(x)
>>> loss
# 可以看到 loss 减小了，所以更新网络层参数后，计算后的值和 y 更加接近
tensor([2.3448], grad_fn=<SubBackward0>)
```

可视化操作

tensorboardX

在 TensorFlow 里有 tensorboard 可以进行训练过程的可视化，而在 Pytorch 里也提供了 tensorboardX 几乎和 tensorboard 一样，适合习惯了 tensorboard 的使用者，具体使用可以参考：

<https://www.jianshu.com/p/713c1bc4cf8a>

<https://www.jianshu.com/p/46eb3004beca>

visdom

pytorch 提供的另一个可视化，个人更喜欢这种界面样式（~~tensorboard 的黄色底色看着不太习惯~~）

安装

```
pip install visdom
```

使用步骤

1. 通过命令 `python -m visdom.server` 启动 visdom 服务器进行监听
2. 导入可视化类: `from visdom import Visdom`
3. 实例化可视化类, 并通过 `line/bar/text` 等提供的 API 进行绘图

具体使用可以参考:

https://blog.csdn.net/wen_fei/article/details/82979497

<https://ptorch.com/news/77.html>

torchvision 模块

提供了很多计算视觉相关的数据集, 以及较流行的模型等, 参考: <https://blog.csdn.net/zhenaoxi1077/article/details/80955607>

5 人点赞

[深度学习](#)

作者: dawsonenjoy

链接: <https://www.jianshu.com/p/5460b7fa3ec4>

来源: 简书

著作权归作者所有。商业转载请联系作者获得授权, 非商业转载请注明出处。

Pytorch 框架使用

介绍

相比 TensorFlow 的静态图开发, Pytorch 的动态图特性使得开发起来更加人性化, 选择 Pytorch 的理由可以参考:

<https://www.jianshu.com/p/c1d9cdb52548>, 这里也顺便介绍一下 TensorFlow 静态图和 Pytorch 动态图开发的区别:

总的来说，在 TensorFlow 里你只能通过定义数据、网络等，然后直接训练、预测啥的，中间过程到底发生了什么对我们来说都是未知的，只能等待训练完毕后查看结果如何，想要 debug 都 debug 不了，于是在学了一段时间以后还是一脸懵：训练的过程到底发生了什么？数据怎么变化的？

直到接触了 pytorch 框架以后，这些疑问就渐渐地减少了，因为在 pytorch 里，一切就像操作 numpy 数组一样（很多方法名甚至都一样），只是类型变成了张量，如何训练、训练多少次等等一切都是我们自己来决定，过程发生了什么？print 或者 debug 一下就知道了

拿最简单的线性拟合举例，通过 pytorch 操作，你会发现整个拟合的过程就是：提供输入和输出值，然后网络层（假如就一个全连接层）就相当于一个矩阵（这里暂时忽略偏置值，原本全连接层的计算是与矩阵相乘后再加上一个偏置值），里面的参数（这里称为权值）随着每一次计算，再根据求导之类的操作不断更新参数的值，最终使得输入的值与这个矩阵相乘后的值不断贴近于输出值，而这计算和修改权值的过程就称为训练。

比如一个函数 $y = 2x + 1$ ，然后定义一个 1×1 的矩阵（全连接层），然后矩阵里的权值一开始都是随机的，比如 $((1))$ ，那么要拟合这个函数的话，最终理想的结果肯定是传入 x ，输出的结果是 $2x + 1$ ，所以目标就是让一个数与这个矩阵相乘尽量等价于这个数和 2 相乘后再加 1，也就是说最终 x 乘以矩阵和 $2x$ 的差最好等于 0，比如在 $x=-1$ 时， $y=-1$ ，那么矩阵的权值理想结果就是 1。但是当结果越来越大以后，会发现 y/x 的值逐渐与 2 相近，所以最终矩阵经过多次训练以后，结果肯定就是 2 左右的数，而刚才说的这些可以通过一段 pytorch 代码来演示：

```
import torch

x = torch.unsqueeze(torch.linspace(-100, 100, 1000), dim=1)
# 生成-100 到 100 的 1000 个数的等差数列
y = 2*x + 1
# 定义 y=2x+1 函数
matrix = torch.nn.Linear(1, 1)
# 定义一个 1x1 的矩阵
optimizer = torch.optim.Adam(matrix.parameters(), lr=0.1)
# 使用优化器求导更新矩阵权重

for _ in range(100):
    # 训练 100 次
    value = matrix(x)
    # value 是 x 与矩阵相乘后的值
    score = torch.mean((value - y) ** 2)
    # 目标偏差，值为(value-y)的平方取均值，越接近 0 说明结果越准确
    matrix.zero_grad()
    score.backward()
```

```
optimizer.step()
# 根据求导结果更新权值
print("第{}次训练权值结果: {}, 结果偏差: {}".format(_, matrix.weight.data.numpy(), score))

# 输出结果:
# 第 0 次训练权值结果:[[0.9555]], 结果偏差: 4377.27294921875
# ...
# 第 99 次训练权值结果:[[2.0048]], 结果偏差: 0.10316929966211319
```

从这段代码的结果可以看到最开始权值初始值为 0.9555，偏差为 4377.27294921875，经过 100 次训练后，权值为 2.0048，偏差为 0.1 那样，从而可以证实我们前面的思路是对的。

其实上面的代码里不止更新了权值，也更新了偏置值 bias，只不过这里为了更加简单的解释，而没有进行说明，可以通过 `matrix.bias` 可以调用查看，最终会发现偏置值接近 1 左右（假设是 0.99），而偏置值就是在与矩阵相乘后加上的值，所以可以看出通过训练以后，矩阵和偏置拟合成了函数： $y = 2.0048x + 0.99$ 。

上面介绍的是线性拟合的例子，即数学相关的示例，而放在生活应用当中，其就可以理解成拟合一个特定要求的函数，比如识别图片中人脸的场景中就是：

即输入图片到函数当中，输出的就是人脸对应的坐标，而我们的目标就是要实现这个函数（经过不断训练，修正模型中的参数）

通过上面的介绍，想必你应该对 pytorch 以及深度学习有了一点入门的了解了，而本文接下来也将针对于 pytorch 的基本使用作出介绍

安装

分为 CPU 和 GPU 版本：

- CPU 版本下载：
直接 pip 下载容易出问题，因此这里推荐离线下载：
1. 进入网址：<https://www.lfd.uci.edu/~gohlke/pythonlibs/#pytorch> 或者 https://download.pytorch.org/whl/torch_stable.html，下载对应版本的 pytorch
2. pip 安装对应的下载文件

- GPU 下载:
 1. 先安装 cuda: <https://developer.nvidia.com/cuda-downloads>
 2. 下载并安装 GPU 版本 pytorch 文件

简单示例-线性拟合

```
import torch
import matplotlib.pyplot as plt

w = 2
b = 1
noise = torch.rand(100, 1)
x = torch.unsqueeze(torch.linspace(-1, 1, 100), dim=1)
# 因为输入层格式要为(-1, 1)，所以这里将(100)的格式转成(100, 1)
y = w*x + b + noise
# 拟合分布在 y=2x+1 上并且带有噪声的散点
model = torch.nn.Sequential(
    torch.nn.Linear(1, 16),
    torch.nn.Tanh(),
    torch.nn.Linear(16, 1),
)
# 自定义的网络，带有 2 个全连接层和一个 tanh 层
loss_fun = torch.nn.MSELoss()
# 定义损失函数为均方差
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
# 使用 adam 作为优化器更新网络模型的权重，学习率为 0.01

plt.ion()
# 图形交互
for _ in range(1000):
    ax = plt.axes()
    output = model(x)
    # 数据向后传播（经过网络层的一次计算）
```

```

loss = loss_fun(output, y)
# 计算损失值
# print("before zero_grad: {}".format(list(model.children())[0].weight.grad))
# print("-"*100)
model.zero_grad()
# 优化器清空梯度
# print("before zero_grad: {}".format(list(model.children())[0].weight.grad))
# print("-"*100)
# 通过注释地方可以对比发现执行 zero_grad 方法以后倒数梯度将会被清 0
# 如果不清空梯度的话，则会不断累加梯度，从而影响到当前梯度的计算
loss.backward()
# 向后传播，计算当前梯度，如果这步不执行，那么优化器更新时则会找不到梯度
optimizer.step()
# 优化器更新梯度参数，如果这步不执行，那么因为梯度没有发生改变，loss 会一直计算最开始的那个梯度
if _ % 100 == 0:
    plt.cla()
    plt.scatter(x.data.numpy(), y.data.numpy())
    plt.plot(x.data.numpy(), output.data.numpy(), 'r-', lw=5)
    plt.text(0.5, 0, 'Loss=%.4f' % loss.data.numpy(), fontdict={'size': 20, 'color': 'red'})
    plt.pause(0.1)
    # print("w:", list(model.children())[0].weight.t() @ list(model.children())[1].weight.t())
    # 通过这句可以查看权值变化，可以发现最后收敛到 2 附近

plt.ioff()
plt.show()

```

从上面的示例里不知道你有没有看出来，其实 pytorch 的使用和 numpy 特别相似，只是 numpy 是基于数组（numpy.ndarray），而 pytorch 是基于张量（torch.Tensor），但是在使用上很多都是一样的，包括很多方法名等。所以如果学习过 numpy 的话，会感觉 pytorch 特别的亲切，如果没学过的 numpy 话，通过学习 pytorch，也将顺便给你将来的 numpy 学习奠定一定的基础

数据类型

标量/张量

pytorch 里的基本单位，个人理解是 0 维、没有方向的（比如单个数字那样）称为标量，有方向的称为张量（比如一串数字），通过 `torch.tensor` 定义，举例：

```
>>> torch.tensor(1.)
tensor(1.)
```

pytorch 中还提供了以下数据类型的张量：

```
通用类型  Tensor
float 型   torch.FloatTensor
double 型  torch.DoubleTensor
int 型     torch.IntTensor
long 型    torch.LongTensor
byte 型    torch.ByteTensor
```

这些数据类型的张量使用方法和 `tensor` 有点不同，`tensor` 是自定义数据，而标定数据类型的将会随机生成一个该类型的数据，举例：

```
>>> torch.tensor(1.)
tensor(1.)
# 生成值为 1 的张量
>>> torch.FloatTensor(1)
tensor([-3.0147e-21])
# 生成 1 个随机的 float 型张量
>>> torch.FloatTensor(2, 3)
tensor([[ -7.3660e-21,   4.5914e-41,   9.6495e+20],
        [ 8.2817e-43,   0.0000e+00,   0.0000e+00]])
# 生成随机指定尺寸的 float 型张量
```

并且可以发现如果是大写开头的张量都是传入形状生成对应尺寸的张量，只有 `tensor` 是传入自己定义的数据，举例：

```
>>> torch.Tensor(2, 3)
tensor([[ -7.3660e-21,   4.5914e-41,   0.0000e+00],
        [ 0.0000e+00,   0.0000e+00,   0.0000e+00]])
# 创建一个格式为 2 行 3 列的张量
>>> torch.tensor(3)
```



```
tensor(3)
# 创建一个值为 3 的张量
```

当然如果希望大写开头的张量传入自己定义的数据，则传入一个列表或者数组，举例：

```
>>> torch.Tensor([2, 3])
tensor([2., 3.])
# 创建一个张量，值为自定义的
```

注 1:

上面提供的张量类型是在 CPU 下的，如果是在 GPU 下的，则在 torch.cuda 下，举例：

```
float 型 torch.cuda.FloatTensor
double 型 torch.cuda.DoubleTensor
int 型 torch.cuda.IntTensor
long 型 torch.cuda.LongTensor
byte 型 torch.cuda.ByteTensor
```

注 2:

pytorch 框架里没有提供 string 这样的数据类型，所以为了表示某个标记之类的，我们可以使用 one-hot 编码，或者使用 Embedding（常用的 Word2vec/glove）

张量属性/方法

索引

张量可以像数组那样进行索引，举例：

```
>>> a = torch.rand(2, 3)
# 生成 2 行 3 列随机数
>>> a
tensor([[0.2488, 0.2794, 0.2949],
        [0.1818, 0.1950, 0.3803]])
>>> a[0]
tensor([0.2488, 0.2794, 0.2949])
```

```
# 索引第一行
>>> a[0, 1]
tensor(0.2794)
# 索引第一行第二列
>>> x[:, [0, 2]]
tensor([[0.2488, 0.2949],
        [0.1818, 0.3803]])
# 索引第一列和第三列
```

也可以使用内置的 `index_select` 方法进行索引，使用该方法可以索引多个自定义的行列（比如取第 1/3/4 列），该方法传入两个参数分别为张量维度以及张量中的索引（传入类型为张量），举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.7470, 0.8258, 0.5929],
        [0.7803, 0.7016, 0.4281]])
>>> a.index_select(0, torch.tensor([1]))
tensor([[0.7803, 0.7016, 0.4281]])
# 对数据第 1 维（整体）取第 2 个数据（第二行）
# 第二个参数为张量
>>> a.index_select(1, torch.tensor([1, 2]))
tensor([[0.8258, 0.5929],
        [0.7016, 0.4281]])
# 对数据第 1 维取 2/3 列
```

切片

张量可以像数组那样进行切片，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.7470, 0.8258, 0.5929],
        [0.7803, 0.7016, 0.4281]])
>>> a[0, :2]
tensor([0.7470, 0.8258])
```

```
# 第一行前两列
>>> a[0, ::2]
tensor([0.7470, 0.5929])
# 第一行从第一个起隔两个取一个，因为这里只有 3 个，所以取第一列和第三列
```

还有... 代表对这部分取全部，举例：

```
>>> a = torch.rand(2, 2, 2, 2)
>>> a
tensor([[[[0.0882, 0.8744],
          [0.9916, 0.6415]],

        [[0.7247, 0.4012],
          [0.5703, 0.9776]]],

       [[0.1076, 0.0710],
        [0.1275, 0.5045]],

        [[0.7833, 0.6519],
          [0.3394, 0.2560]]]])
>>> a[1, ..., 1]
tensor([[0.0710, 0.5045],
        [0.6519, 0.2560]])
# 相当于 a[1, :, :, 1]
```

逻辑操作

可以像数组那样进行逻辑操作，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.7679, 0.1081, 0.3601],
        [0.0661, 0.8539, 0.3079]])
>>> a>0.5
```

```
tensor([[1, 0, 0],
        [0, 1, 0]], dtype=torch.uint8)
# 大于 0.5 的变成 1，否则变成 0
# 在数组里是变成 True 或 False
```

shape

获取张量形状，举例：

```
>>> a = torch.tensor(1.)
>>> a.shape
torch.Size([1])
>>> a = torch.tensor((1., 2.))
>>> a.shape
torch.Size([2])
>>> a = torch.tensor(((1., 2.), (3, 4)))
>>> a
tensor([[1., 2.],
        [3., 4.]])
>>> a.shape
torch.Size([2, 2])
>>> a.shape[0]
2
# 一维的尺寸
```

也可以通过内置方法 `size()` 来获取，举例：

```
>>> a = torch.tensor(((1., 2.), (3, 4)))
>>> a.size()
torch.Size([2, 2])
>>> a.size(0)
2
# 一维的尺寸
>>> a.size(1)
2
```

二维的尺寸

data

获取张量

item()

获取数据，仅当只有一个数据时才能用

dim()

获取张量维度，举例：

```
>>> a = torch.tensor(1.)
```

```
>>> a.dim()
```

```
0
```

标量数据，0 维

```
>>> a = torch.tensor([1.])
```

```
>>> a.dim()
```

```
1
```

一维张量

```
>>> a = torch.tensor((1., 2.), (3, 4))
```

```
>>> a.dim()
```

```
2
```

numel()

获取张量大小，举例：

```
>>> a = torch.tensor((1., 2.), (3, 4))
```

```
>>> a.numel()
```

```
4
```

a 里总共有 4 个数

```
type()
```

获取张量类型，举例：

```
>>> x = torch.IntTensor(1)
>>> x
tensor([1065353216], dtype=torch.int32)
>>> x.type()
'torch.IntTensor'
>>> type(x)
<class 'torch.Tensor'>
# 使用内置的 type 函数只能知道是个 torch 下的张量
```

```
cuda()
```

CPU 数据转 GPU，举例：

```
>>> x = torch.IntTensor(1)
>>> x = x.cuda()
# 转成 GPU 数据，这条语句得在 GPU 环境下才能运行
```

注：

判断是否可用 GPU 可以通过 `torch.cuda.is_available()` 判断，举例：

```
>>> torch.cuda.is_available()
False
```

下面是一个 GPU 常用操作：

```
>>> torch.cuda.device_count()
1
# 获取可使用 GPU 数量
>>> torch.cuda.set_device(0)
# 使用编号为 0 的 GPU
```

`cpu()`

GPU 数据转 CPU，举例：

```
>>> a = torch.cuda.FloatTensor(1)
>>> a.cpu()
tensor(1)
```

注：

上面的 GPU 和 CPU 数据互换是在低版本的 pytorch 上使用的，使用起来可能不太方便，之后的版本推出了简单切换的版本：先通过 `torch.device()` 方法选择一个 GPU/CPU 设备，然后对需要使用该设备的数据通过 `to()` 方法调用，举例：

```
>>> device = torch.device('cuda:0')
# 选择 GPU 设备
>>> net = torch.nn.Linear(10, 100).to(device)
# 这里定义了一个全连接网络层，并使用该 GPU 设备
```

`view()/reshape()`

这两个方法一样，修改张量形状，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.7824, 0.0911, 0.5798],
        [0.4280, 0.2592, 0.4978]])
>>> a.reshape(3, 2)
tensor([[0.7824, 0.0911],
        [0.5798, 0.4280],
        [0.2592, 0.4978]])
>>> a.view(3, 2)
tensor([[0.7824, 0.0911],
        [0.5798, 0.4280],
        [0.2592, 0.4978]])
```

`t()`

转置操作，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.4620, 0.9787, 0.3998],
        [0.4092, 0.1320, 0.5631]])
>>> a.t()
tensor([[0.4620, 0.4092],
        [0.9787, 0.1320],
        [0.3998, 0.5631]])
>>> a.t().shape
torch.Size([3, 2])
```

pow()

对张量进行幂运算，也可以用**代替，举例：

```
>>> a = torch.full([2, 2], 3)
>>> a
tensor([[3., 3.],
        [3., 3.]])
>>> a.pow(2)
tensor([[9., 9.],
        [9., 9.]])
>>> a**2
tensor([[9., 9.],
        [9., 9.]])
```

sqrt()

对张量取平方根，举例：

```
>>> a = torch.full([2, 2], 9)
>>> a
tensor([[9., 9.],
```



```
        [9., 9.]])
>>> a.sqrt()
tensor([[3., 3.],
        [3., 3.]])
>>> a**(0.5)
tensor([[3., 3.],
        [3., 3.]])
# 可以看出结果一样
```

rsqrt()

取平方根的倒数，举例：

```
>>> a = torch.full([2, 2], 9)
>>> a
tensor([[9., 9.],
        [9., 9.]])
>>> a.rsqrt()
tensor([[0.3333, 0.3333],
        [0.3333, 0.3333]])
```

exp()

取 e 为底的幂次方，举例：

```
>>> a = torch.full([2, 2], 2)
>>> a.exp()
tensor([[7.3891, 7.3891],
        [7.3891, 7.3891]])
```

log()

取 log 以 e 为底的对数，举例：

```
>>> a = torch.full([2, 2], 2)
```

```
>>> a.exp()
tensor([[7.3891, 7.3891],
        [7.3891, 7.3891]])
>>> a.log()
tensor([[0.6931, 0.6931],
        [0.6931, 0.6931]])
```

对应的还有以 2 为底的 \log_2 、以 10 为底的 \log_{10} 等，举例：

```
>>> a = torch.full([2, 2], 2)
>>> a.log2()
tensor([[1., 1.],
        [1., 1.]])
```

`round()/floor()/ceil()`

四舍五入、向下取整和向上取整

`trunc()/frac()`

取整数/小数部分，举例：

```
>>> a = torch.tensor(1.2)
>>> a.trunc()
tensor(1.)
>>> a.frac()
tensor(0.2000)
```

`max()/min()/median()/mean()`

取最大值、最小值、中位数和平均值，举例：

```
>>> a = torch.tensor([1., 2., 3., 4., 5.])
>>> a.max()
tensor(5.)
```

```
>>> a.min()
tensor(1.)
>>> a.median()
tensor(3.)
>>> a.mean()
tensor(3.)
```

注意的是该方法默认会将全部数据变成一维的，并取整个数据里的最大/最小之类的值，因此可以通过 dim 参数设置取值维度，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.0042, 0.5913, 0.0104],
        [0.1673, 0.9443, 0.1303]])
>>> a.max()
tensor(0.9443)
# 默认返回总体的最大值
>>> a.max(dim=0)
(tensor([0.1673, 0.9443, 0.1303]), tensor([1, 1, 1]))
# 在第 1 维选择，返回每一列最大值，并且对应索引为 1,1,1
>>> a.max(dim=1)
(tensor([0.5913, 0.9443]), tensor([1, 1]))
# 在第 2 维选择，返回每一行最大值，并且对应索引为 1,1
```

还有一个 keepdim 参数，可以控制返回的格式和之前相同，举例：

```
>>> a = torch.rand(2, 3)
>>> a.max(dim=0, keepdim=True)
(tensor([[0.1673, 0.9443, 0.1303]]), tensor([[1, 1, 1]]))
>>> a.max(dim=1, keepdim=True)
(tensor([[0.5913],
        [0.9443]]), tensor([[1],
        [1]]))
# 和之前的对比，可以发现设置该参数后格式变成一样的了
```

`sum()/prod()`

求累加、累乘，举例：

```
>>> a = torch.tensor([1., 2., 3., 4., 5.])
>>> a.sum()
tensor(15.)
>>> a.prod()
tensor(120.)
```

该方法也可以使用 dim 参数对某维度进行运算

argmax()/argmin()

返回最大/小值的索引，举例：

```
>>> a = torch.tensor([1., 2., 3., 4., 5.])
>>> a.argmax()
tensor(4)
>>> a.argmin()
tensor(0)
```

该方法同样默认会将全部数据变成一维的，并计算整个数据里最大值的索引，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.2624, 0.2925, 0.0866],
        [0.0545, 0.8841, 0.9959]])
>>> a.argmax()
tensor(5)
# 可以看出所有数据默认转到 1 维上，最大值在第 6 个
```

但可以通过输入维度来控制索引的判断基准，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.7365, 0.4280, 0.6650],
```

```

        [0.6988, 0.9839, 0.8990]])
>>> a.argmax(dim=0)
tensor([0, 1, 1])
# 在第1维下判断每一列的最大值索引，第一列是 0.7365 索引为 0，第二列是 0.9839 索引为 1，第三列是 0.8990 索引为 1
>>> a.argmax(dim=1)
tensor([0, 1])
# 在第2维下判断每一行的最大值索引，第一行是 0.7365 索引为 0，第二行是 0.9839 索引为 1

```

argsort()

返回从小到大（默认，可以通过 `descending` 参数设置）的索引，举例：

```

>>> a.argsort()
tensor([0, 1, 2, 3, 4])
>>> a = torch.tensor([1., 5., 4., 2., 5., 3.])
>>> a.argsort()
tensor([0, 3, 5, 2, 1, 4])
# 从小到大的索引
>>> a.argsort(descending=True)
tensor([1, 4, 2, 5, 3, 0])
# 从大到小的索引

```

topk()

返回前几大的值（取前几小的值设置参数 `largest` 为 `False` 就行），并且还是按从大到小（取前几小就是从小到大）排序好的，举例：

```

>>> a = torch.rand(2, 3)
>>> a
tensor([[0.0831, 0.3135, 0.2989],
        [0.2959, 0.6371, 0.9715]])
>>> a.topk(3)
(tensor([[0.3135, 0.2989, 0.0831],
        [0.9715, 0.6371, 0.2959]]), tensor([[1, 2, 0],
        [2, 1, 0]]))
# 取前三大的，可以看到结果也从大到小排序好

```

```
>>> a.topk(3, dim=1)
(tensor([[0.3135, 0.2989, 0.0831],
        [0.9715, 0.6371, 0.2959]]), tensor([[1, 2, 0],
        [2, 1, 0]]))
# 在第2维取前大的数
>>> a.topk(3, largest=False)
(tensor([[0.0831, 0.2989, 0.3135],
        [0.2959, 0.6371, 0.9715]]), tensor([[0, 2, 1],
        [0, 1, 2]]))
# 取前三小的, 可以看到结果也从小到大排序好
```

kthvalue()

返回第几小的数, 举例:

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.2052, 0.1159, 0.8533],
        [0.3335, 0.3922, 0.7414]])
>>> a.sort()
(tensor([[0.1159, 0.2052, 0.8533],
        [0.3335, 0.3922, 0.7414]]), tensor([[1, 0, 2],
        [0, 1, 2]]))
>>> a
tensor([[0.2052, 0.1159, 0.8533],
        [0.3335, 0.3922, 0.7414]])
>>> a.kthvalue(2, dim=0)
(tensor([0.3335, 0.3922, 0.8533]), tensor([1, 1, 0]))
# 在第1维返回第二小的数, 可以看到每一列第二小的数被返回
>>> a.kthvalue(2, dim=1)
(tensor([0.2052, 0.3922]), tensor([0, 1]))
# 在第2维返回第二小的数, 可以看到每一行第二小的数被返回
```

norm()

计算张量的范数（可以理解成张量长度的模），默认是 12 范数（即欧氏距离），举例：

```
>>> a = torch.tensor((-3, 4), dtype=torch.float32)
>>> a.norm(2)
tensor(5.)
# 12 范数:  $((-3)**2 + 4**2)**0.5$ 
>>> a.norm(1)
tensor(7.)
# 11 范数（曼哈顿距离）:  $|-3| + |4|$ 
>>> a.norm(2, dim=0)
>>> a = torch.tensor((-3, 4), (6, 8)), dtype=torch.float32)
>>> a
tensor([[ -3.,  4.],
        [ 6.,  8.]])
>>> a.norm(2, dim=0)
tensor([6.7082, 8.9443])
# 在第 1 维算范数:  $((-3)**2 + 6**2)**0.5$ ,  $(4**2 + 8**2)**0.5$ 
>>> a.norm(2, dim=1)
tensor([ 5., 10.])
# 在第 2 维算范数:  $((-3)**2 + 4**2)**0.5$ ,  $(6**2 + 8**2)**0.5$ 
```

`clamp()`

设置张量值范围，举例：

```
>>> a = torch.tensor([1, 2, 3, 4, 5])
>>> a.clamp(3)
tensor([3, 3, 3, 4, 5])
# 范围控制在 3~
>>> a.clamp(3, 4)
tensor([3, 3, 3, 4, 4])
# 范围控制在 3~4
```

`transpose()`

将指定维度对调，如果在 2 维情况，就相当于转置，举例：

```
>>> a = torch.rand(2, 3, 1)
>>> a
tensor([[[0.8327],
          [0.7932],
          [0.7497]],
        [[0.2347],
          [0.7611],
          [0.5529]]])
>>> a.transpose(0, 2)
tensor([[[0.8327, 0.2347],
          [0.7932, 0.7611],
          [0.7497, 0.5529]]])
# 将第 1 维和第 3 维对调
>>> a.transpose(0, 2).shape
torch.Size([1, 3, 2])
```

permute()

和 transpose 类似也是对调维度，但使用不太一样，举例：

```
>>> a = torch.rand(2, 3, 1)
>>> a
tensor([[[0.6857],
          [0.4819],
          [0.3992]],
        [[0.7477],
          [0.8073],
          [0.1939]]])
>>> a.permute(2, 0, 1)
tensor([[[0.6857, 0.4819, 0.3992],
          [0.7477, 0.8073, 0.1939]]])
```



```
# 将 a 修改成原来第 3 个维度放在第 1 个维度，第 1 个维度放在第 2 个维度，第 2 个维度放在第 3 个维度
>>> a.permute(2, 0, 1).shape
torch.Size([1, 2, 3])
```

squeeze()

在指定索引位置删减维度，如果不传入索引，将会把所有能删减的维度（值为 1）都删减了，举例：

```
>>> a = torch.rand(1, 2, 1, 1)
>>> a
tensor([[[[0.3160]],
          [[0.5993]]]])
>>> a.squeeze()
tensor([0.3160, 0.5993])
>>> a.squeeze().shape
torch.Size([2])
# 可以看出删减了所有能删减的维度
>>> a.squeeze(0)
tensor([[[0.3160]],
          [[0.5993]]])
>>> a.squeeze(0).shape
torch.Size([2, 1, 1])
# 删减了第 1 维
>>> a.squeeze(1)
tensor([[[[0.3160]],
          [[0.5993]]]])
>>> a.squeeze(1).shape
torch.Size([1, 2, 1, 1])
# 第 2 维因为无法删减，所以没有变化
```

unsqueeze()

在指定索引位置增加维度，举例：

```
>>> a = torch.rand(2, 3)
```

```

>>> a
tensor([[0.8979, 0.5201, 0.2911],
        [0.8355, 0.2032, 0.9345]])
>>> a.unsqueeze(0)
tensor([[[0.8979, 0.5201, 0.2911],
         [0.8355, 0.2032, 0.9345]]])
# 在第 1 维增加维度
>>> a.unsqueeze(0).shape
torch.Size([1, 2, 3])
# 可以看出 (2, 3)→(1, 2, 3)
>>> a.unsqueeze(-1)
tensor([[[0.8979],
         [0.5201],
         [0.2911]],

        [[0.8355],
         [0.2032],
         [0.9345]]])
# 在最后 1 维增加维度
>>> a.unsqueeze(-1).shape
torch.Size([2, 3, 1])
# 可以看出 (2, 3)→(2, 3, 1)

```

expand()

扩展数据，但仅限于维度是 1 的地方，举例：

```

>>> a = torch.rand(1, 2, 1)
>>> a
tensor([[[0.5487],
         [0.9694]]])
>>> a.expand([2, 2, 1])
tensor([[[0.5487],
         [0.9694]],

```

```

        [[0.5487],
         [0.9694]])
# 扩展了第 1 维度的数据
>>> a.expand([2, 2, 2])
tensor([[[0.5487, 0.5487],
         [0.9694, 0.9694]],
        [[0.5487, 0.5487],
         [0.9694, 0.9694]]])
# 扩展了第 1/3 维度的数据
>>> a.expand([1, 4, 1])
Traceback (most recent call last):
  File "<pyshell#242>", line 1, in <module>
    a.expand([1, 4, 1])
RuntimeError: The expanded size of the tensor (4) must match the existing size (2) at non-singleton dimension 1. Target sizes: [1, 4, 1]. Tensor sizes: [1, 2, 1]
# 因为第 2 维度不为 1，所以不能扩展

```

repeat()

复制数据，对指定维度复制指定倍数，举例：

```

>>> a = torch.rand(1, 2, 1)
>>> a.repeat([2, 2, 1]).shape
torch.Size([2, 4, 1])
# 将 1/2 维变成原来 2 倍，第 3 维不变
>>> a.repeat([2, 2, 2]).shape
torch.Size([2, 4, 2])

```

split()

根据长度切分数据，举例：

```

>>> a = torch.rand(2, 2, 3)
>>> a
tensor([[[0.6913, 0.3448, 0.5107],

```

```

        [0.5714, 0.1821, 0.2043]],
        [[0.9937, 0.4512, 0.8015],
         [0.9622, 0.3952, 0.6199]]])
>>> a.split(1, dim=0)
(tensor([[[0.6913, 0.3448, 0.5107],
          [0.5714, 0.1821, 0.2043]]]), tensor([[[0.9937, 0.4512, 0.8015],
          [0.9622, 0.3952, 0.6199]]]))
# 可以看出根据第 1 维将数据按长度 1 切分成了 2/1 份（第 1 维长度是 2）
>>> a.split(1, dim=0).shape
>>> x, y = a.split(1, dim=0)
>>> x.shape, y.shape
(torch.Size([1, 2, 3]), torch.Size([1, 2, 3]))

```

chunk

根据数量切分数据，也就是自定义要切成多少份，举例：

```

>>> a = torch.rand(3,3)
>>> a
tensor([[0.3355, 0.0770, 0.1840],
        [0.0844, 0.4452, 0.8723],
        [0.9296, 0.4290, 0.4051]])
>>> a.chunk(3, dim=0)
(tensor([[[0.3355, 0.0770, 0.1840]]]), tensor([[[0.0844, 0.4452, 0.8723]]]), tensor([[[0.9296, 0.4290, 0.4051]]]))
# 把数据切成 3 份
>>> a.chunk(2, dim=0)
(tensor([[[0.3355, 0.0770, 0.1840],
          [0.0844, 0.4452, 0.8723]]]), tensor([[[0.9296, 0.4290, 0.4051]]]))
# 切成 2 份，可以看到最后一个被独立出来了

```

常用方法

数组/张量转换

`torch.from_numpy`

数组转张量，举例：

```
>>> a = np.array([1, 2, 3])
>>> torch.from_numpy(a)
tensor([1, 2, 3], dtype=torch.int32)
```

张量转数组则通过 `data.numpy()` 转，若为 GPU 数据，则先转成 CPU 的，也可以通过 `np.array(tensor)` 强行转成数组，举例：

```
>>> a = torch.tensor([1., 2., 3.])
>>> a
tensor([1., 2., 3.])
>>> a.cpu().data.numpy()
array([1., 2., 3.], dtype=float32)
# 转成 CPU 数据，然后转数组
>>> np.array(a)
array([1., 2., 3.], dtype=float32)
# 强制转成数组
```

基本运算

`torch.add`

加法运算，一般情况下也可以用+号代替，举例：

```
>>> a = torch.rand(2, 2)
>>> a
tensor([[0.5643, 0.4722],
        [0.5939, 0.6289]])
>>> torch.add(a, b)
tensor([[1.5643, 1.4722],
        [1.5939, 1.6289]])
>>> a + b
tensor([[1.5643, 1.4722],
```

```
        [1.5939, 1.6289]])  
# 可以看出结果一样
```

```
torch.sub
```

张量减法

```
torch.mul
```

张量乘法

```
torch.div
```

张量除法

```
torch.matmul
```

矩阵乘法，举例：

```
>>> a = torch.tensor([[1, 1], [1, 1]])  
>>> b = torch.tensor([[1, 1], [1, 1]])  
>>> torch.matmul(a, b)  
tensor([[2, 2],  
        [2, 2]])  
>>> a*b  
tensor([[1, 1],  
        [1, 1]])  
# 别把数组乘法和矩阵乘法弄混了
```

矩阵乘法还可以用@代替，举例：

```
>>> a@b  
tensor([[2, 2],  
        [2, 2]])  
# 结果和前面矩阵乘法一样
```

逻辑操作

`torch.equal`

判断两个张量是否完全相等，返回 True 或者 False，而==符号返回的只是一个由 0 和 1 组成的张量，举例：

```
>>> a = torch.rand(2, 2)
>>> a
tensor([[0.8146, 0.1331],
        [0.6715, 0.4594]])
>>> b = a
>>> a == b
tensor([[1, 1],
        [1, 1]], dtype=torch.uint8)
# 判断两个张量每个元素是否相等，并用 0 和 1 来表示是否相等
>>> torch.equal(a, b)
True
# 判断两个张量是否相等并返回结果
```

`torch.all`

逻辑与操作，判断是否全为 1，举例：

```
>>> a = torch.tensor([1, 2, 3])
>>> b = torch.tensor([1, 2, 4])
>>> a
tensor([1, 2, 3])
>>> b
tensor([1, 2, 4])
>>> torch.all(a==b)
tensor(0, dtype=torch.uint8)
>>> a==b
tensor([1, 1, 0], dtype=torch.uint8)
```

`torch.any`

逻辑或操作，判断是否存在 1

torch.where

有三个参数 a, b, c，对数据 a 进行逻辑判断，为 1 的取数据 b 上对应位置的值，为 0 取 c 上对应值，举例：

```
>>> a = torch.tensor([[1., 2.], [3., 4.]])
>>> b = torch.tensor([[5., 6.], [7., 8.]])
>>> c = torch.rand(2, 2)
>>> c
tensor([[0.3821, 0.6138],
        [0.2323, 0.2675]])
>>> torch.where(c>0.3, a, b)
tensor([[1., 2.],
        [7., 8.]])
# 位置(0,0)为1，所以取 b 上(0,0)的值，即 1
# 位置(0,1)为1，所以取 b 上(0,1)的值，即 2
# 位置(1,0)为0，所以取 c 上(1,0)的值，即 7
# 位置(1,1)为0，所以取 c 上(1,1)的值，即 8
```

数据生成

torch.zeros

生成固定尺寸的全 0 张量，举例：

```
>>> torch.zeros(2,2)
tensor([[0., 0.],
        [0., 0.]])
```

torch.ones

生成固定尺寸的全 1 张量，举例：

```
>>> torch.ones(2,2)
```



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

torch.full

生成固定尺寸的值全为指定值的张量，举例：

```
>>> torch.full([2, 3], 2)
tensor([[2., 2., 2.],
        [2., 2., 2.]])
# 指定格式且值全为 2
```

torch.eye

生成对角线上值全为 1 的张量，举例：

```
>>> torch.eye(3)
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
# 生成 3*3 张量
>>> torch.eye(3, 3)
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
# 和上面一样
>>> torch.eye(3, 4)
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.]])
>>> torch.eye(3, 2)
tensor([[1., 0.],
        [0., 1.],
        [0., 0.]])
```

torch.arange

生成指定等差数列的张量，举例：

```
>>> torch.arange(10)
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# 只传一个参数 n 则默认为从 0-n-1 的 n 个数
>>> torch.arange(-1, 1, 0.1)
tensor([-1.0000, -0.9000, -0.8000, -0.7000, -0.6000, -0.5000, -0.4000, -0.3000,
        -0.2000, -0.1000,  0.0000,  0.1000,  0.2000,  0.3000,  0.4000,  0.5000,
         0.6000,  0.7000,  0.8000,  0.9000])
# 生成-1 到 1，且距离为 0.1 的等差数列
```

torch.linspace

也是生成等差数列的张量，用法和 `arrange` 稍有不同，举例：

```
>>> torch.linspace(-1, 1, steps=21)
tensor([-1.0000, -0.9000, -0.8000, -0.7000, -0.6000, -0.5000, -0.4000, -0.3000,
        -0.2000, -0.1000,  0.0000,  0.1000,  0.2000,  0.3000,  0.4000,  0.5000,
         0.6000,  0.7000,  0.8000,  0.9000,  1.0000])
# 生成一个长度为 21 的等差数列，且值为-1 到 1
# 可以看出和上面等价，但是这个方便设置数据量，上面的方便设置距离
```

torch.logspace

和 `linspace` 用法相似，可以理解成再 `linspace` 的基础上对其求 10 的 n 次方值，举例：

```
>>> torch.logspace(0, 10, steps=11)
tensor([1.0000e+00, 1.0000e+01, 1.0000e+02, 1.0000e+03, 1.0000e+04, 1.0000e+05,
        1.0000e+06, 1.0000e+07, 1.0000e+08, 1.0000e+09, 1.0000e+10])
# 10 的 0 次方、1 次方、2 次方、...
```

torch.rand

随机生成一个固定尺寸的张量，并且数值范围都在 $0\sim 1$ 之间，举例：

```
>>> torch.rand((2, 3))
tensor([[0.6340, 0.4699, 0.3745],
        [0.5066, 0.3480, 0.7346]])
```

torch.randn

也是随机生成固定尺寸的张量，数值符合正态分布

torch.randint

随机生成一个固定尺寸的张量，并且数值为自定义范围的整数，举例：

```
>>> torch.randint(0, 10, (2, 3))
tensor([[4, 3, 1],
        [6, 3, 9]])
# 0~9 的固定格式整数
```

torch.randperm

生成指定个数的张量（范围为 $0\sim$ 个数-1）并打乱，举例：

```
>>> torch.randperm(10)
tensor([0, 4, 9, 5, 1, 7, 3, 6, 2, 8])
# 生成 0~9 的数，并打乱
```

torch.rand_like

传入一个张量，并根据该张量 shape 生成一个新的随机张量，举例：

```
>>> a = torch.rand(2, 3)
>>> a
tensor([[0.4079, 0.9071, 0.9304],
        [0.0641, 0.0043, 0.0429]])
```

```
>>> torch.rand_like(a)
tensor([[0.2936, 0.4585, 0.7674],
        [0.4049, 0.0707, 0.0456]])
# 生成一个和 a 格式相同的张量
```

注：

有好多 xxx_like 的方法，原理都是一样的：传入一个张量，根据张量的 shape 生成新的张量

数据处理

`torch.masked_select`

取出指定条件数据，举例：

```
>>> a = torch.tensor([0, 0.5, 1, 2])
>>> torch.masked_select(a, a>0.5)
tensor([1., 2.])
# 取出所有大于 0.5 的数据
```

`torch.cat`

在指定维度合并数据，但要求两个数据维度相同，并且指定维度以外的维度尺寸相同，举例：

```
>>> a = torch.rand(1, 2, 3)
>>> b = torch.rand(2, 2, 3)
>>> torch.cat((a, b))
tensor([[[[0.0132, 0.4118, 0.5814],
          [0.8034, 0.8765, 0.8404]],
        [[0.7860, 0.6115, 0.4745],
          [0.0846, 0.4158, 0.3805]],
        [[0.9454, 0.3390, 0.3802],
          [0.6526, 0.0319, 0.7155]]]])
>>> torch.cat((a, b)).shape
torch.Size([3, 2, 3])
# 可以看出默认在第 1 维合并数据，合并过程可以看成：[1, 2, 3] + [2, 2, 3] = [3, 2, 3]
```

```
>>> torch.cat((a,b), dim=2)
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    torch.cat((a,b), dim=2)
RuntimeError: invalid argument 0: Sizes of tensors must match except in dimension 2. Got 1 and 2 in dimension 0 at
d:\build\pytorch\pytorch-1.0.1\aten\src\th\generic\thtensormoremath.cpp:1307
# 在第3维合并数据时因为1维（非3维部分）不一样所以报错
```

torch.stack

在指定维度创建一个新维度合并两个数据，举例：

```
>>> a = torch.rand(1,2,3)
>>> b = torch.rand(1,2,3)
>>> a
tensor([[[[0.5239, 0.0540, 0.0213],
          [0.9713, 0.5983, 0.1413]]]])
>>> b
tensor([[[[0.3397, 0.0976, 0.3744],
          [0.5080, 0.7520, 0.1759]]]])
>>> torch.stack((a, b))
tensor([[[[0.5239, 0.0540, 0.0213],
          [0.9713, 0.5983, 0.1413]]],
        [[[0.3397, 0.0976, 0.3744],
          [0.5080, 0.7520, 0.1759]]]])
>>> torch.stack((a, b)).shape
torch.Size([2, 1, 2, 3])
# 合并过程可以看成：[1, 1, 2, 3] + [1, 1, 2, 3] = [1+1, 1, 2, 3] = [2, 1, 2, 3]
>>> torch.stack((a, b), dim=2)
tensor([[[[0.5239, 0.0540, 0.0213],
          [0.3397, 0.0976, 0.3744]],
        [[0.9713, 0.5983, 0.1413],
          [0.5080, 0.7520, 0.1759]]]])
>>> torch.stack((a, b), dim=2).shape
```

```
torch.Size([1, 2, 2, 3])
# 合并过程可以看成: [1, 2, 1, 3] + [1, 2, 1, 3] = [1, 2, 1+1, 3] = [1, 2, 2, 3]
```

其他操作

one-hot 编码

通过结合 `torch.zeros()` 方法和 `tensor.scatter_()` 方法实现, 举例:

```
>>> label = torch.tensor([[0], [1], [2], [3]])
# 标签内容
>>> label_number = len(label)
# 标签长度
>>> label_range = 10
# 标签总数
>>> torch.zeros(label_number, label_range).scatter_(1, label, 1)
tensor([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]])
# 通过 torch.zeros 生成 4 行, 10 列的全零矩阵, 通过 torch.scatter_ 以标签第一维为基准, 用 1 来覆盖对应的位置
```

带下划线的方法

在 pytorch 里面可以看到很多方法有两种版本: 带下划线和不带下划线的, 这两者的区别就是: 不带下划线的方法操作数据会先新建一个相同的数据, 然后对其进行操作后返回; 带下划线的则直接对该数据进行操作并返回 (声明该 tensor 是个 in-place 类型), 举例:

```
>>> a = torch.tensor((1., -2.))
>>> a.abs()
tensor([1., 2.])
>>> a
tensor([ 1., -2.])
# 不带下划线的取绝对值后, 查看原来的数据, 发现没有变
>>> a.abs_()
tensor([1., 2.])
```

```
>>> a
tensor([1., 2.])
# 带下划线的取绝对值后，查看原来的数据，发现已经变了
```

更多关于 in-place 类型的参考：

<https://blog.csdn.net/hbhhhxs/article/details/93886525>

工具集

在 torch.utils 下提供了很多 API 工具方便我们的使用

随机切分数据集

通过 torch.utils.data.random_split() 方法随机切分数据集，然后通过 torch.utils.data.DataLoader 来载入数据，举例：

```
>>> a = torch.rand(100)
>>> a
tensor([0.8579, 0.6903, 0.8042, 0.6803, 0.5619, 0.4721, 0.3132, 0.6476, 0.6644,
        0.1822, 0.8333, 0.6207, 0.5666, 0.3410, 0.9760, 0.1522, 0.5908, 0.4049,
        0.8710, 0.3284, 0.7598, 0.1615, 0.2269, 0.7273, 0.5658, 0.7861, 0.4562,
        0.4225, 0.0466, 0.2845, 0.2759, 0.0649, 0.7345, 0.7406, 0.0044, 0.2111,
        0.5922, 0.1108, 0.8785, 0.5843, 0.3432, 0.1751, 0.8386, 0.8131, 0.5848,
        0.3727, 0.4079, 0.3207, 0.7192, 0.5415, 0.2176, 0.3019, 0.9200, 0.1222,
        0.1771, 0.7479, 0.1213, 0.7306, 0.7951, 0.6702, 0.4286, 0.6684, 0.4392,
        0.5319, 0.8701, 0.5307, 0.0664, 0.6950, 0.8652, 0.8842, 0.1940, 0.5079,
        0.1927, 0.3511, 0.6232, 0.5951, 0.7436, 0.3113, 0.8578, 0.6422, 0.6670,
        0.5569, 0.4681, 0.3848, 0.5463, 0.2438, 0.7747, 0.2718, 0.8766, 0.3523,
        0.1736, 0.9693, 0.6800, 0.6727, 0.9430, 0.5596, 0.7665, 0.8402, 0.3828,
        0.6339])
>>> m, n = torch.utils.data.random_split(a, [10, 90])
# 将数据随机分成 10 和 90 个
>>> list(m)
[tensor(0.6803), tensor(0.8386), tensor(0.4079), tensor(0.8652), tensor(0.8333), tensor(0.8402), tensor(0.7861), tensor(0.8710),
 tensor(0.7306), tensor(0.5848)]
```

```
# 查看数据集 m
>>> m.indices
tensor([ 3, 42, 46, 68, 10, 97, 25, 18, 57, 44])
# 可以看到其存放的是随机的 10 个索引，然后寻找对应下标数据
>>> data_m = torch.utils.data.DataLoader(m, batch_size=2)
# 载入 m 数据
>>> data_n = torch.utils.data.DataLoader(n, batch_size=2)
```

函数式 API 和类 API

在 pytorch 当中，大部分神经网络层、激活函数、损失函数等都提供了两种 API 调用方式，分别是函数式 API 和类 API，前者基本都在 `torch.nn.functional` 下，后者基本都在 `torch.nn` 下，前者一般直接调用即可，适合函数式编程；后者一般是先实例化，然后通过其内置的方法进行调用，适合面向对象编程。当然这些 API 功能基本都可以自定义实现，只是这里提供了 API 简化了操作，并且还提供了 GPU 加速等功能

网络层

全连接层

说白了就是单纯的矩阵相乘然后有偏置则加上偏置（公式： $\text{input} @ w + b$ ），函数式 API：`torch.nn.functional.linear`，类 API：`torch.nn.Linear`，类 API 举例：

```
>>> layer1 = torch.nn.Linear(100, 10)
# 这里使用类 API
# 定义一个全连接层，输入 100 个单元，输出 10 个，可以理解成初始化的一个 (100, 10) 的矩阵
>>> layer2 = torch.nn.Linear(10, 1)
>>> x = torch.rand(1, 100)
# 定义一个 (1, 100) 的矩阵
>>> x = layer1(x)
# x 经过 layer1 全连接层的运算
>>> x
tensor([[ -0.1354,  0.1530,  0.1946, -0.1349,  0.6149, -0.0482,  0.1025, -0.8483,
          -1.0567, -0.5853]], grad_fn=<AddmmBackward>)
>>> x.shape
```



```

torch.Size([1, 10])
# 可以发现乘完以后变成了(1, 10)的矩阵
>>> x = layer2(x)
# x 再经过 layer2 层运算
>>> x
tensor([[ -0.2182]], grad_fn=<AddmmBackward>)
>>> x.shape
torch.Size([1, 1])
>>> layer1.weight.shape
torch.Size([10, 100])
# 可以通过 weight 属性查看当前层的权值，计算的时候会将权值矩阵进行转置后才进行运算，所以是(10, 100)而不是(100, 10)
>>> layer1.bias
Parameter containing:
tensor([ 0.0049, -0.0081, -0.0541, -0.0301,  0.0320, -0.0621,  0.0072, -0.0024,
        -0.0339,  0.0456], requires_grad=True)
# 可以通过 bias 属性查看当前层的偏置值

```

函数式 API 举例：

```

>>> x = torch.rand(1, 100)
>>> w = torch.rand(10, 100)
>>> x = torch.nn.functional.linear(x, w)
# 可以看出函数式 API 需要我们自己定义初始化权值，然后直接调用即可
>>> x
tensor([[25.9789, 23.4787, 24.2929, 25.8615, 22.0681, 23.1044, 22.0457, 22.0386,
         23.0654, 24.6127]])

```

通过上面对比我们可以发现对于类 API，我们只需实例化对应的类，然后在其的 `__init__` 方法里会对权值之类的数据进行初始化，然后我们传入自己的数据进行调用运算；而在函数式 API 当中，首先我们需要自己定义初始化的权值，然后通过往 API 接口传入数据和权值等数据进行运算

注：

通过上面我们可以看出每一层操作的时候可以实时打印查看其权值之类的变化，以供我们观察，这也是 pytorch 作为动态图和 TensorFlow 最大的区别

Dropout

随机选取一部分节点使用，忽略一部分节点，函数式 API: `torch.nn.functional.dropout`，类 API: `torch.nn.Dropout`，举例：

```
>>> a = torch.rand(20)
>>> torch.nn.functional.dropout(a, 0.2)
tensor([1.2178, 1.0375, 0.0555, 0.0307, 0.3235, 0.0000, 0.5209, 0.0000, 0.3346,
        1.2383, 0.3606, 1.0937, 0.0000, 0.2957, 0.9463, 0.2932, 0.8088, 0.4445,
        0.5565, 0.0241])
# 随机将百分之 20 的节点转成 0
```

批标准化层

函数式 API: `torch.nn.functional.batch_norm`，类 API: `torch.nn.BatchNorm2d`（对应的有 1d、2d 等等），类 API 举例：

```
>>> x1 = torch.rand(1, 3, 784)
# 3 通道的 1d 数据
>>> layer1 = torch.nn.BatchNorm1d(3)
# 1d 批标准化层，3 通道
>>> layer1.weight
Parameter containing:
tensor([1., 1., 1.], requires_grad=True)
# 可以看出 batch_norm 层的权值全是 1
>>> layer1(x1)
tensor([[[[-0.0625, -0.1859, -0.3823, ..., 0.6668, -0.7487, 0.8913],
         [ 0.0115, -0.1149, 0.1470, ..., -0.1546, 0.3012, 0.2472],
         [ 1.5185, -0.4740, -0.8664, ..., 0.6266, 0.2797, -0.2975]]],
        grad_fn=<NativeBatchNormBackward>])
# 可以看到数据都被标准化了
>>> layer1(x1).shape
torch.Size([1, 3, 784])
>>> x2 = torch.rand(1, 3, 28, 28)
# 3 通道的 2d 数据
>>> layer2 = torch.nn.BatchNorm2d(3)
>>> layer2(x2)
tensor([[[[[-0.0378, -0.3922, 0.2255, ..., -0.1469, -0.3016, 0.2384],
```

```

        [-0.3901, -0.0220, -0.3118, ..., -0.2492,  0.1705, -0.0599],
        [-0.1309, -0.3064, -0.2001, ..., -0.0613, -0.1838,  0.1335],
        ...,
        [ 0.9022, -0.3031,  1.0695, ..., -0.8257, -0.6438, -0.2672],
        [-0.1015,  1.1482,  1.0834, ...,  0.6641, -0.8632, -0.2418],
        [-1.2068, -0.7443,  0.8346, ...,  0.1213,  0.4528, -0.5756]]]],
    grad_fn=<NativeBatchNormBackward>)
>>> layer2(x2).shape
# 经过 batch_norm 只是将数据变得符合高斯分布，并不会改变数据形状
torch.Size([1, 3, 28, 28])
>>> x2.mean()
tensor(0.4942)
# 原来数据的平均值
>>> x2.std()
tensor(0.2899)
# 原来数据的标准差
>>> layer2(x2).mean()
tensor(-2.1211e-08, grad_fn=<MeanBackward0>)
# 经过 batch_norm 的平均值，可以看出经过 batch_norm 层后数据平均值变成接近 0
>>> layer2(x2).std()
tensor(1.0002, grad_fn=<StdBackward0>)
# 经过 batch_norm 的标准差，可以看出经过 batch_norm 层后数据标准差变成接近 1

```

卷积层

函数式 API: `torch.nn.functional.conv2d`, 类 API: `torch.nn.Conv2d`, 类 API 举例:

```

>>> x = torch.rand(1, 1, 28, 28)
>>> layer = torch.nn.Conv2d(1, 3, kernel_size=3, stride=1, padding=0)
# 设置输入通道为 1, 输出通道为 3, filter 大小为 3x3, 步长为 1, 边框不补 0
>>> layer.weight
Parameter containing:
tensor([[[[-0.1893,  0.1177, -0.2837],
          [ 0.1116,  0.0348,  0.3011],

```

```

        [-0.1871, -0.0722, -0.1843]]],
        ...,
        [[[ 0.0083, -0.0784,  0.1592],
          [-0.1896,  0.0082, -0.0146],
          [-0.2069, -0.0147, -0.1899]]]], requires_grad=True)
# 可以查看初始化权值
>>> layer.weight.shape
torch.Size([3, 1, 3, 3])
# 格式分别代表输出通道 3, 输入通道 1, 尺寸为 3x3
>>> layer.bias.shape
torch.Size([3])
# 查看初始化偏置
>>> layer(x)
tensor([[[[-0.0494, -0.1396, -0.0690,  ..., -0.1382, -0.0539, -0.1876],
          [-0.2185, -0.0116, -0.1287,  ...,  0.1233, -0.0091,  0.0407],
          [-0.0648,  0.0506, -0.1971,  ..., -0.2013,  0.1151, -0.0026],
          ...,
          [-0.4974, -0.5449, -0.4583,  ..., -0.7153, -0.1890, -0.7381],
          [-0.4254, -0.6051, -0.2578,  ..., -0.4957, -0.4128, -0.4875],
          [-0.5392, -0.4214, -0.5671,  ..., -0.2785, -0.6113, -0.3150]]]],
        grad_fn=<ThnnConv2DBackward>))
# 进行一次卷积运算, 实际是魔法方法__call__里调用了 forward 方法
>>> layer(x).shape
torch.Size([1, 3, 26, 26])
# 可以看到计算后由于边框不补 0, 而滤波器大小为 3x3, 所以结果的长宽就变成了(height-3+1, weight-3+1)
>>> layer1 = torch.nn.Conv2d(1, 3, kernel_size=3, stride=1, padding=1)
# 这里边缘补 0
>>> layer1(x).shape
torch.Size([1, 3, 28, 28])
# 可以看到由于边缘补 0, 所以大小没变
>>> layer2 = torch.nn.Conv2d(1, 3, kernel_size=3, stride=2, padding=0)
# 这里步长改成 2
>>> layer2(x).shape
torch.Size([1, 3, 13, 13])

```

```
# 结果的长宽就变成了((height-3+1)/2, (weight-3+1)/2)
>>> layer3 = torch.nn.Conv2d(1, 3, kernel_size=3, stride=2, padding=1)
# 这里边缘补 0, 且步长改成 2
>>> layer3(x).shape
torch.Size([1, 3, 14, 14])
# 可以看到结果的长宽就变成了(height/2, weight/2)
```

函数式 API 举例:

```
>>> x = torch.rand(1, 1, 28, 28)
>>> w = torch.rand(3, 1, 3, 3)
# 输出 3 通道, 输入 1 通道, 尺寸 3x3
>>> b = torch.rand(3)
# 偏置长度要和通道数一样
>>> layer = torch.nn.functional.conv2d(x, w, b, stride=1, padding=1)
>>> layer
tensor([[[[2.1963, 2.6321, 3.4186, ..., 3.2495, 3.1609, 2.5473],
          [2.5637, 3.4892, 4.0079, ..., 4.1167, 4.4497, 3.1637],
          [2.7618, 3.2788, 3.2314, ..., 4.7185, 4.3128, 2.6393],
          ...,
          [1.3735, 2.3738, 1.8388, ..., 2.9912, 2.6638, 1.5941],
          [2.1967, 2.0466, 2.0095, ..., 3.3192, 2.9521, 2.2673],
          [1.6091, 2.1341, 1.5108, ..., 2.1684, 2.4585, 1.7931]]]])
>>> layer.shape
torch.Size([1, 3, 28, 28])
```

池化层

函数式 API: `torch.nn.functional.max_pool2d`, 类 API: `torch.nn.MaxPool2d`, 类 API 举例:

```
>>> x = torch.rand(1, 1, 28, 28)
>>> layer = torch.nn.MaxPool2d(3, stride=2)
# 尺寸 3x3, 步长为 2
>>> layer(x)
tensor([[[[0.9301, 0.9342, 0.9606, 0.9922, 0.9754, 0.9055, 0.7142, 0.9882,
```

```

        0.9803, 0.8054, 0.9903, 0.9903, 0.9426],
        ...,
        [0.8873, 0.8873, 0.9324, 0.9876, 0.9566, 0.9225, 0.9673, 0.9675,
         0.9977, 0.9977, 0.9552, 0.9552, 0.8689]]]])
>>> layer(x).shape
torch.Size([1, 1, 13, 13])

```

还有个 avgpool（函数式 API：torch.nn.functional.avg_pool2d，类 API：torch.nn.AvgPool2d），和 maxpool 不一样的是：maxpool 是取最大值，而 avgpool 取的是平均值，举例：

```

>>> torch.nn.functional.avg_pool2d(x, 3, stride=2)
tensor([[[[0.5105, 0.6301, 0.5491, 0.4691, 0.5788, 0.4525, 0.3903, 0.5718,
          0.6259, 0.3388, 0.4169, 0.6122, 0.4760],
          ...,
          [0.4705, 0.5332, 0.4150, 0.5000, 0.5686, 0.5325, 0.6241, 0.4926,
          0.4646, 0.3121, 0.2975, 0.5203, 0.5701]]]])
>>> torch.nn.functional.avg_pool2d(x, 3, stride=2).shape
torch.Size([1, 1, 13, 13])

```

flatten

将张量转成一维，举例：

```

>>> torch.flatten(torch.rand(2,2))
tensor([0.1339, 0.5694, 0.9034, 0.6025])
>>> torch.flatten(torch.rand(2,2)).shape
torch.Size([4])

```

向上取样

将图片放大/缩小成原来的几倍，函数式 API：torch.nn.functional.interpolate，举例：

```

>>> x = torch.rand(1, 1, 2, 2)
# 可以理解成 1 张 1 通道的 2x2 图片
>>> x

```

```

tensor([[[[0.9098, 0.7948],
          [0.0670, 0.3906]]]])
>>> torch.nn.functional.interpolate(x, scale_factor=2, mode='nearest').shape
torch.Size([1, 1, 4, 4])
# 可以看到数据被放大了一倍
>>> torch.nn.functional.interpolate(x, scale_factor=2, mode='nearest')
tensor([[[[0.9098, 0.9098, 0.7948, 0.7948],
          [0.9098, 0.9098, 0.7948, 0.7948],
          [0.0670, 0.0670, 0.3906, 0.3906],
          [0.0670, 0.0670, 0.3906, 0.3906]]]])
# 可以看到是往横纵向都复制成原来的对应倍数
>>> torch.nn.functional.interpolate(x, scale_factor=0.5, mode='nearest')
tensor([[[[0.9098]]]])
# 将数据缩小一倍，可以看到取那一部分的第一个数据

```

注：

还有如 Upsampling、UpsamplingNearest2d 也是向上采样，现在已经逐渐被 interpolate 给取代。上面 interpolate 示例参数中 mode='nearest' 时，相当于该 UpsamplingNearest2d，函数式 API：torch.nn.functional.upsample_nearest，函数式 API：torch.nn.UpsamplingNearest2d，类 API 举例：

```

>>> x = torch.rand(1, 1, 2, 2)
>>> x
tensor([[[[0.9977, 0.9778],
          [0.4167, 0.6936]]]])
>>> torch.nn.functional.upsample_nearest(x, scale_factor=2).shape
torch.Size([1, 1, 4, 4])
>>> torch.nn.functional.upsample_nearest(x, scale_factor=2)
tensor([[[[0.9977, 0.9977, 0.9778, 0.9778],
          [0.9977, 0.9977, 0.9778, 0.9778],
          [0.4167, 0.4167, 0.6936, 0.6936],
          [0.4167, 0.4167, 0.6936, 0.6936]]]])

```

常用于定义词向量，可以理解 embedding 层定义了一个词典用来存储和表示所有的词向量，而传入的数据则会根据索引找到对应的词向量，函数式 API: `torch.nn.functional.embedding`，类 API: `torch.nn.Embedding`，类 API 举例：

```
>>> embed = torch.nn.Embedding(10, 2)
# 定义了 10 个词向量，每个词向量用格式为 (1, 2) 的 tensor 表示
>>> words = torch.tensor([0, 1, 2, 0])
# 定义一句话，里面有 4 个词，那么可以看出第一个和最后一个词相同
>>> embed(words)
# 经过嵌入层索引可以看到 4 个词对应的词向量如下，也可以看出第一个和最后一个词索引相同，所以值是一样的
tensor([[ -0.0019,  1.6786],
        [ 0.3118, -1.6250],
        [ 1.6038,  1.5044],
        [-0.0019,  1.6786]], grad_fn=<EmbeddingBackward>)
>>> embed.weight
# 再看 embedding 层的权重，可以发现这就是定义了一个词向量表，并且会随着训练而更新，从而找出词与词之间的关系
Parameter containing:
tensor([[ -1.8939e-03,  1.6786e+00],
        [ 3.1179e-01, -1.6250e+00],
        [ 1.6038e+00,  1.5044e+00],
        [-6.2278e-01, -2.5135e-01],
        [ 1.6210e+00, -5.6379e-01],
        [-7.3388e-02, -2.0099e+00],
        [ 8.7655e-01,  2.4011e-01],
        [-2.5685e+00,  2.6756e-01],
        [ 4.9723e-01, -8.3784e-01],
        [ 4.2338e-01, -1.9839e+00]], requires_grad=True)
```

更多参考：<https://blog.csdn.net/tommorrow12/article/details/80896331>

RNN 层

类 API: `torch.nn.RNN`，会返回计算后总体的输出，以及最后一个时间戳上的输出，通过下面代码可以证明最后一个时间戳的输出和总体输出的最后一个是一样的，类 API 举例：

```
>>> x = torch.randn(10, 3, 100)
```



```

# 模拟句子序列：有 10 个单词（序列长度是 10），共 3 句话，每个单词用 100 维向量表示
# input: [seq_len, batch, input_size], 如果希望 batch_size 放第一个，可以设置 batch_first=True
>>> layer = torch.nn.RNN(input_size=100, hidden_size=20, num_layers=4)
>>> layer
RNN(100, 20, num_layers=4)
>>> out, h = layer(x)
# 返回 output 和 hidden
>>> out.shape
torch.Size([10, 3, 20])
# 所有时间戳上的状态
# output: [seq_len, batch, hidden_size]
>>> h.shape
torch.Size([4, 3, 20])
# 最后一个时间戳上的 hidden
# hidden: [num_layers, batch, hidden_size]
>>> h[-1]
# 最后一层的最后一个时间戳上的输出（因为 num_layers 的值为 4，所以要取第四个，对于 num_layers 参数的解释，下面会说）
tensor([[ 3.5205e-01,  3.6580e-01, -5.6378e-01, -9.9363e-02,  3.8728e-03,
        -5.0282e-01,  1.4762e-01, -2.5631e-01, -8.8786e-03,  1.2912e-01,
         4.7565e-01, -8.8090e-02, -3.9374e-02,  3.1736e-02,  3.1264e-01,
         2.8091e-01,  5.0764e-01,  2.9722e-01, -3.6929e-01, -5.1096e-02],
        ...
        [ 5.4770e-01,  4.8047e-01, -5.2541e-01,  2.5208e-01, -4.0260e-04,
        -2.3619e-01, -2.1128e-01, -1.1262e-01, -6.2672e-02,  3.5301e-01,
        -4.1065e-02, -3.5043e-02, -4.3008e-01, -1.8410e-01,  2.5826e-01,
         3.5430e-02,  2.5651e-01,  4.5170e-01, -5.4705e-01, -2.4720e-01]])
grad_fn=<SelectBackward>)
>>> out[-1]
# 所有状态的最后一个输出，可以看到是一样的
tensor([[ 3.5205e-01,  3.6580e-01, -5.6378e-01, -9.9363e-02,  3.8728e-03,
        -5.0282e-01,  1.4762e-01, -2.5631e-01, -8.8786e-03,  1.2912e-01,
         4.7565e-01, -8.8090e-02, -3.9374e-02,  3.1736e-02,  3.1264e-01,
         2.8091e-01,  5.0764e-01,  2.9722e-01, -3.6929e-01, -5.1096e-02],
        ...

```

```
[ 5.4770e-01,  4.8047e-01, -5.2541e-01,  2.5208e-01, -4.0260e-04,
 -2.3619e-01, -2.1128e-01, -1.1262e-01, -6.2672e-02,  3.5301e-01,
 -4.1065e-02, -3.5043e-02, -4.3008e-01, -1.8410e-01,  2.5826e-01,
  3.5430e-02,  2.5651e-01,  4.5170e-01, -5.4705e-01, -2.4720e-01]],
grad_fn=<SelectBackward>)
```

num_layers 参数的理解：rnn 的基本参数都挺好理解因为其他深度学习框架基本也都一样，而比较特殊的就是 num_layers 参数，其实也很简单，顾名思义就是代表着有几层 rnn，直接一口气帮你定义好直接计算，省的你自已再去定义一堆 rnn，然后一层一层的算过去，比如上面的示例代码设置 num_layers=4，那么上面的代码可以替换成下面这种：

```
>>> x = torch.randn(10, 3, 100)
>>> layer1 = torch.nn.RNN(input_size=100, hidden_size=20, num_layers=1)
# 把上面示例代码中 num_layers=4 的 rnn 改成 4 个为 1 的 rnn
>>> layer2 = torch.nn.RNN(input_size=20, hidden_size=20, num_layers=1)
# 因为第一层的 hidden 是 20，所以后几层的输入都是 20
>>> layer3 = torch.nn.RNN(input_size=20, hidden_size=20, num_layers=1)
>>> layer4 = torch.nn.RNN(input_size=20, hidden_size=20, num_layers=1)
>>> out, h = layer1(x)
>>> out, h = layer2(out)
>>> out, h = layer3(out)
>>> out, h = layer4(out)
>>> out.shape
torch.Size([10, 3, 20])
```

RNN 参数理解参考：

<https://blog.csdn.net/rogerfang/article/details/84500754>

LSTM 层

类 API：torch.nn.LSTM，因为 LSTM 是基于 RNN 并添加了门控制，因此返回的时候比 RNN 要多返回一个 cell 单元，格式和 hidden 一样，举例：

```
>>> x = torch.randn(10, 3, 100)
>>> layer = torch.nn.LSTM(input_size=100, hidden_size=20, num_layers=4)
>>> layer
LSTM(100, 20, num_layers=4)
```

```
>>> out, (h, c) = layer(x)
# 返回 output、hidden 和 cell
>>> out.shape
torch.Size([10, 3, 20])
>>> h.shape
torch.Size([4, 3, 20])
>>> c.shape
torch.Size([4, 3, 20])
# 可以看出和 hidden 格式一样
# cell: [num_layers, batch_size, hidden_size]
```

这里给一个通过前三个数预测后一个数的模型代码示例：

```
# -----
# 模块导入
import numpy
import torch
from torch import nn

# -----
# 数据预处理
data_length = 30
# 定义 30 个数，通过前三个预测后一个，比如：1, 2, 3->4
seq_length = 3
# 通过上面可知序列长度为 3

number = [i for i in range(data_length)]
li_x = []
li_y = []
for i in range(0, data_length - seq_length):
    x = number[i: i + seq_length]
    y = number[i + seq_length]
    li_x.append(x)
    li_y.append(y)
```

```

#     print(x, '->', y)

data_x = numpy.reshape(li_x, (len(li_x), 1, seq_length))
# 输入数据格式: seq_len, batch, input_size
# 这里可能会有误解, seq_len 不是步长, 而是你的样本有多少组, 即 sample
# 而 input_size 就是你数据的维度, 比如用三个预测一个, 就是 3 维
data_x = torch.from_numpy(data_x / float(data_length)).float()
# 将输入数据归一化
data_y = torch.zeros(len(li_y), data_length).scatter_(1, torch.tensor(li_y).unsqueeze_(dim=1), 1).float()
# 将输出数据设置为 one-hot 编码

# print(data_x.shape)
# # 格式: torch.Size([27, 1, 3]), 代表: 27 组数据 (batch)、序列步长为 3 (sequence)
# print(data_y.shape)
# # 格式: torch.Size([27, 30]), 代表: 27 组数据, 30 个特征 (features)

# -----
# 定义网络模型
class net(nn.Module):
    # 模型结构: LSTM + 全连接 + Softmax
    def __init__(self, input_size, hidden_size, output_size, num_layer):
        super(net, self).__init__()
        self.layer1 = nn.LSTM(input_size, hidden_size, num_layer)
        self.layer2 = nn.Linear(hidden_size, output_size)
        self.layer3 = nn.Softmax()
    def forward(self, x):
        x, _ = self.layer1(x)
        sample, batch, hidden = x.size()
        # 格式: [27, 1, 32], 代表样本数量, batch 大小以及隐藏层尺寸
        x = x.reshape(-1, hidden)
        # 转成二维矩阵后与全连接进行计算
        x = self.layer2(x)
        x = self.layer3(x)
        return x

```

```

model = net(seq_length, 32, data_length, 4)

# -----
# 定义损失函数和优化器
loss_fun = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# -----
# 训练模型

# 训练前可以先看看初始化的参数预测的结果差距
# result = model(data_x)
# for target, pred in zip(data_y, result):
#     print("{} -> {}".format(target.argmax().data, pred.argmax().data))

# 开始训练 1000 轮
for _ in range(500):
    output = model(data_x)
    loss = loss_fun(data_y, output)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (_ + 1) % 50 == 0:
        print('Epoch: {}, Loss: {}'.format(_, loss.data))

# -----
# 预测结果
result = model(data_x)
for target, pred in zip(data_y, result):
    print("正确结果: {}, 预测: {}".format(target.argmax().data, pred.argmax().data))

# 结果:

```

```
# 正确结果: 3, 预测: 3
# 正确结果: 4, 预测: 4
# 正确结果: 5, 预测: 5
# 正确结果: 6, 预测: 6
# 正确结果: 7, 预测: 7
# 正确结果: 8, 预测: 8
# 正确结果: 9, 预测: 9
# 正确结果: 10, 预测: 10
# 正确结果: 11, 预测: 11
# 正确结果: 12, 预测: 12
# 正确结果: 13, 预测: 13
# 正确结果: 14, 预测: 14
# 正确结果: 15, 预测: 15
# 正确结果: 16, 预测: 16
# 正确结果: 17, 预测: 21
# 正确结果: 18, 预测: 18
# 正确结果: 19, 预测: 27
# 正确结果: 20, 预测: 21
# 正确结果: 21, 预测: 21
# 正确结果: 22, 预测: 21
# 正确结果: 23, 预测: 21
# 正确结果: 24, 预测: 24
# 正确结果: 25, 预测: 25
# 正确结果: 26, 预测: 26
# 正确结果: 27, 预测: 27
# 正确结果: 28, 预测: 28
# 正确结果: 29, 预测: 29
```

当然这个示例使用到的数据极少，只是一个能快速跑来玩玩的程序而已，不必当真...

LSTM 原理理解: https://blog.csdn.net/gzj_1101/article/details/79376798

LSTM 计算过程参考: <https://blog.csdn.net/qyk2008/article/details/80225986>

序列化模型

即用来定义神经网络模型（每一层都要是继承于 `torch.nn` 下的网络），类 API: `torch.nn.Sequential`（该模型就是针对面向对象模式编写，因此不提供函数式 API），举例：

```
>>> net = torch.nn.Sequential(
    torch.nn.Linear(100, 10),
    torch.nn.Dropout(0.7),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 1)
)
>>> net
Sequential(
  (0): Linear(in_features=100, out_features=10, bias=True)
  (1): Dropout(p=0.7)
  (2): ReLU()
  (3): Linear(in_features=10, out_features=1, bias=True)
# 可以直接查看网络结构
```

序列化模型修改

序列化模型可以理解成一个列表，里面按顺序存放了所有的网络层，官方也提供了添加往序列化模型里添加网络层的方法 `add_module(name, layer)`，而修改则可以索引到对应的层直接修改，删除可以通过 `del` 关键字删除，举例：

```
>>> seq = nn.Sequential(nn.Linear(10, 20), nn.Linear(20, 1))
>>> seq
Sequential(
  (0): Linear(in_features=10, out_features=20, bias=True)
  (1): Linear(in_features=20, out_features=1, bias=True)
)
>>> seq.add_module("tanh", nn.Tanh())
# 在最后面添加一个 tanh 激活层
>>> seq
# 可以看到添加成功
Sequential(
  (0): Linear(in_features=10, out_features=20, bias=True)
```

```

    (1): Linear(in_features=20, out_features=1, bias=True)
    (tanh): Tanh()
)
>>> seq[2]
Tanh()
>>> seq[2] = nn.ReLU()
# 修改第三层为 relu
>>> seq
# 可以看到修改成功
Sequential(
  (0): Linear(in_features=10, out_features=20, bias=True)
  (1): Linear(in_features=20, out_features=1, bias=True)
  (tanh): ReLU()
)
>>> del seq[2]
# 删除第三层
>>> seq
# 可以看到删除成功
Sequential(
  (0): Linear(in_features=10, out_features=20, bias=True)
  (1): Linear(in_features=20, out_features=1, bias=True)
)

```

修改网络参数

对于网络层中的参数，其是一个 Parameter 类型，因此如果我们需要手动修改其参数时，可以通过定义一个该类的数据来赋值修改，举例：

```

>>> layer = nn.Linear(2, 1)
>>> layer.weight
Parameter containing:
tensor([[0.6619, 0.2653]], requires_grad=True)
>>> type(layer.weight)
# 参数的数据类型
<class 'torch.nn.parameter.Parameter'>

```



```
>>> layer.weight = torch.rand(2, 1, requires_grad=True)
# 直接赋值张量会报错
Traceback (most recent call last):
  File "<pyshell#150>", line 1, in <module>
    layer.weight = torch.rand(2, 1)
  File "D:\python\lib\site-packages\torch\nn\modules\module.py", line 604, in __setattr__
    .format(torch.typename(value), name))
TypeError: cannot assign 'torch.FloatTensor' as parameter 'weight' (torch.nn.Parameter or None expected)
>>> layer.weight = nn.parameter.Parameter(torch.rand(2, 1))
# 赋值 Parameter 类型的则可以
>>> layer.weight
# 可以看到修改成功
Parameter containing:
tensor([[0.7412],
        [0.9723]], requires_grad=True)
```

自定义神经网络

当需要自己定义神经网络层的时候，首先需要继承于 `torch.nn.Module`，并在初始化时调用父类的初始化方法，同时也要在 `forward` 方法里实现数据的前向传播，举例：

```
import torch

class Dense(torch.nn.Module):
    # 实现一个自定义全连接+relu层，继承 torch.nn.Module
    def __init__(self, input_shape, output_shape):
        super(Dense, self).__init__()
        # 首先初始化时执行父类的初始化，这句话可以看
        # 在父类初始化中会初始化很多变量
        self.w = torch.nn.Parameter(torch.randn(output_shape, input_shape))
        # 初始化权重和偏置参数
        # 使用 Parameter 其会自动将参数设置为需要梯度信息，并且可以通过内置的 parameters 方法返回这些参数
        self.b = torch.nn.Parameter(torch.randn(output_shape))
        self.relu = torch.nn.ReLU()
```

```
# 初始化 relu 层
```

```
def forward(self, x):  
    # 定义前向传播方法  
    x = x @ self.w.t() + self.b  
    # 全连接层的功能就是矩阵相乘计算  
    x = self.relu(x)  
    # 进行 relu 层计算  
    return x  
  
def __call__(self, x):  
    # 调用该类对象执行时，调用前向传播方法  
    # 这个可以不写，直接通过调用 forward 方法也一样  
    return self.forward(x)
```

```
layer = Dense(10, 1)  
x = torch.rand(2, 10)  
output = layer(x)  
print(output)  
# 输出结果:  
# tensor([[0.1780],  
#         [0.0000]], grad_fn=<ThresholdBackward0>)
```

冻结网络层

如果希望训练过程当中，对某些网络层的权重不进行训练的话（该场景在迁移学习当中比较常见），可以设置该层的权重、偏差等属性为 False，举例：

```
>>> net = torch.nn.Sequential(  
    torch.nn.Linear(100, 10),  
    torch.nn.Dropout(0.7),  
    torch.nn.ReLU(),  
    torch.nn.Linear(10, 1)  
)
```

```
>>> net
Sequential(
  (0): Linear(in_features=100, out_features=10, bias=True)
  (1): Dropout(p=0.7, inplace=False)
  (2): ReLU()
  (3): Linear(in_features=10, out_features=1, bias=True)
)
```

```
>>> for name, value in net.named_parameters():
    print(name, value.requires_grad)
```

可以看到网络层的两个全连接层的权重和偏置都可求导

0.weight True

0.bias True

3.weight True

3.bias True

```
>>> net[0].weight.requires_grad = False
```

冻结第一个全连接的权重

```
>>> net[0].bias.requires_grad = False
```

```
>>> for name, value in net.named_parameters():
    print(name, value.requires_grad)
```

可以看到第一个全连接的权重和偏置都被冻结

0.weight False

0.bias False

3.weight True

3.bias True

保存和载入网络

对于所有继承自 `torch.nn.Module` 下的网络，保存时首先通过内置的方法 `state_dict()` 返回当前模型的所有参数，然后通过 `torch.save()` 方法保存成文件（也可以不保存参数，直接保存模型，但这样可控性低，不推荐）；载入时通过 `torch.load()` 方法载入文件，并通过内置的 `load_state_dict()` 方法载入所有的参数，举例：

```
>>> layer = torch.nn.Linear(10, 1)
```

```
>>> layer.state_dict()
```

```
OrderedDict([('weight', tensor([[ -0.1597,  0.0573,  0.0976, -0.1028, -0.1264, -0.0400,  0.0308,  0.2192,
```

```

        -0.0150, -0.3148]])), ('bias', tensor([0.0557]))))
# 可以看到 layer 里定义参数配置
>>> torch.save(layer.state_dict(), "ckpt.mdl")
# 现在保存这个网络参数
>>> layer1 = torch.nn.Linear(10, 1)
# 新建一个网络
>>> layer1.state_dict()
OrderedDict([('weight', tensor([[ -0.2506, -0.2960, -0.3083,  0.0629,  0.1707,  0.3018,  0.2345, -0.1922,
        -0.0527, -0.1894]])), ('bias', tensor([-0.0069]))])
# 显然 layer1 参数和 layer 的不一样
>>> layer1.load_state_dict(torch.load("ckpt.mdl"))
# layer1 载入前面的 layer 网络参数
>>> layer1.state_dict()
OrderedDict([('weight', tensor([[ -0.1597,  0.0573,  0.0976, -0.1028, -0.1264, -0.0400,  0.0308,  0.2192,
        -0.0150, -0.3148]])), ('bias', tensor([0.0557]))])
# 可以发现 layer1 的参数变得和 layer 保存的参数一样

```

优化器

torch.optim

定义了各种优化器，将优化器实例化后（传入需要求导的参数和学习率），通过 `step()` 方法进行梯度下降

- Adam
- SGD

动态调整优化器学习率

对于实例化的优化器，其参数都将存放到一个属性 `param_groups[0]` 里，举例：

```

optim = torch.optim.Adam(model.parameters(), lr=0.01)
print(optim)

```

结果：

```
# Adam (  
# Parameter Group 0  
#     amsgrad: False  
#     betas: (0.9, 0.999)  
#     eps: 1e-08  
#     lr: 0.01  
#     weight_decay: 0  
# )
```

而 `param_groups[0]` 是一个字典对象，所以要动态修改学习率等参数，可以通过下面代码实现：

```
optim.param_groups[0]['lr'] = new_lr
```

上面介绍的是修改优化器的学习率，如果希望对网络的不同层采用不同的学习率可以参考：

<https://blog.csdn.net/jdzwanghao/article/details/83239111>

激活函数

`sigmoid`

公式：

$$1 / (1 + e^{-x})$$

注：该公式可以将值控制在 $0 \sim 1$ 之间，适合概率之类的问题，但因为当 x 特别大时，导数几乎为 0，容易发生梯度弥散（梯度长时间得不到更新，损失值不下降）之类的问题

函数式 API: `torch.nn.functional.sigmoid`，类 API: `torch.nn.Sigmoid`，pytorch 自带: `torch.sigmoid`，举例：

```
>>> a = torch.linspace(-100, 100, 10)  
>>> a  
tensor([-100.0000,  -77.7778,  -55.5556,  -33.3333,  -11.1111,   11.1111,  
         33.3333,   55.5555,   77.7778,  100.0000])  
>>> torch.sigmoid(a)  
tensor([0.0000e+00,  1.6655e-34,  7.4564e-25,  3.3382e-15,  1.4945e-05,  9.9999e-01,
```

1.0000e+00, 1.0000e+00, 1.0000e+00, 1.0000e+00])

可以看到值都被映射在 $0 \sim 1$ 之间

tanh

公式:

$$(e^x - e^{-x}) / (e^x + e^{-x})$$

注: 该公式可以将值控制在 $-1 \sim 1$ 之间

函数式 API: `torch.nn.functional.tanh`, 类 API: `torch.nn.Tanh`, pytorch 自带: `torch.tanh`

relu

公式:

$$0, x \leq 0$$

$$x, x > 0$$

注: 该公式可以将值控制在 $0 \sim +\infty$ 之间

函数式 API: `torch.nn.functional.relu`, 类 API: `torch.nn.ReLU`

leakyrelu

公式:

$$ax, x \leq 0$$

$$x, x > 0$$

注: a 是一个很小的参数值, 该公式在 `relu` 的基础上, 使负区间的数不为 0, 而是保持在一个很小的梯度上

函数式 API: `torch.nn.functional.leaky_relu`, 类 API: `torch.nn.LeakyReLU`

softmax

公式:

$$e^{x_i} / \sum e^{x_i}$$

注：该公式可以将值控制在 $0 \sim 1$ 之间，并且所有的值总和为 1，适合分类之类的问题，并且可以发现通过 e 阶函数，其还会把大的值（特征）放大，小的缩小

函数式 API: `torch.nn.functional.softmax`，类 API: `torch.nn.Softmax`，pytorch 自带: `torch.softmax`

损失函数

均方差

就是计算的 y 与实际 y 之差的平方取平均值，函数式 API: `torch.nn.functional.mse_loss`，类 API: `torch.nn.MSELoss`，第一个参数是 y ，第二个参数是 y 对应的公式，举例：

```
>>> x = torch.tensor(1.)
>>> w = torch.tensor(2.)
>>> b = torch.tensor(0.)
>>> y = torch.tensor(1.)
>>> mse = torch.nn.functional.mse_loss(y, w*x+b)
# 计算 y=wx+b 在点(1, 1)时的均方差: (1 - (2*1+0))^2 = 1
>>> mse
tensor(1.)
```

交叉熵

通过含有的信息量大小来进行判断，一般用于分类，函数式 API: `torch.nn.functional.cross_entropy`，类 API: `torch.nn.CrossEntropyLoss`，要注意的是目标 y 的格式要求为 Long 类型，值为每个 one-hot 数据对应的 argmax 处，举例：

```
>>> output = torch.rand(5, 10)
# 输出结果，假设 5 条数据，每条数据有 10 个特征
>>> target = torch.tensor([0, 2, 9, 9, 2]).long()
# 目标 y，数据必须为 long 型，值分别为每条数据的特征，比如第一个 0 代表第一条数据的第一个特征
>>> output.shape
torch.Size([5, 10])
>>> target.shape
```

```
torch.Size([5])
# 目标 y 的要求还要求是 1 维的
>>> loss = torch.nn.CrossEntropyLoss()
>>> loss(output, target)
tensor(2.2185)
```

注:

对于分类问题一般会选择最后一层激活函数用 softmax，并且损失函数使用交叉熵，但是在 pytorch 的交叉熵中已经内置了 softmax，所以在使用交叉熵时就不需要再自己使用 softmax 了

二分类交叉熵

顾名思义，是一种特殊的交叉熵，专门在 2 分类时使用，比如 GAN 的判别器里，函数式 API: `torch.nn.functional.binary_cross_entropy`，类 API: `torch.nn.BCELoss`，使用举例：

```
>>> batch_size = 5
# 5 条数据
>>> output = torch.rand(batch_size, 1)
>>> output
# 每个数据的正确率
tensor([[0.3546],
        [0.9064],
        [0.0617],
        [0.2839],
        [0.3106]])
>>> target = torch.ones(batch_size, 1)
>>> target
# 正确的概率是 1
tensor([[1.],
        [1.],
        [1.],
        [1.],
        [1.]])
>>> loss = torch.nn.BCELoss()
```



```
>>> loss(output, target)
tensor(1.2697)
```

更多参考

<https://blog.csdn.net/shanglianlm/article/details/85019768>

<https://blog.csdn.net/jackel21/article/details/82812218>

求导机制

在 pytorch 中定义了自动求导的机制，方便了我们在反向传播时更新参数等操作

torch.autograd.grad

定义了自动求导，传入第一个参数是对应的公式，第二个参数是一个列表，里面存放所有要求导的变量，并且在求导前的变量需要通过 `require_grad()` 方法来声明该公式的某个变量是需要求导的（或者在定义时就设置 `requires_grad` 参数为 `True`），返回一个元组，里面是对应每个变量的求导信息，举例：

```
>>> x = torch.tensor(1.)
>>> w = torch.tensor(2.)
>>> b = torch.tensor(0.)
>>> y = torch.tensor(1.)
>>> mse = torch.nn.functional.mse_loss(y, w*x+b)
# 模拟一个 y=wx+b 的函数
>>> mse
tensor(1.)
>>> torch.autograd.grad(mse, [w])
# 此时没有变量声明过是需要求导的
>>> w.requires_grad_()
tensor(2., requires_grad=True)
# 声明 w 需要求导，可以看到一开始就定义 w = torch.tensor(2., requires_grad=True) 效果也是一样的
# 加下划线代表为 in-place 类型，直接对 w 进行修改，也可以替换成：w = w.requires_grad_()
>>> torch.autograd.grad(mse, [w])
# 报错，因为 mse 里的 w 还是之前的 w，需要更新一下 mse 里的 w
```

```
>>> mse = torch.nn.functional.mse_loss(y, w*x+b)
# 更新 mse 里的 w 为声明了需要求导的 w
>>> torch.autograd.grad(mse, [w])
(tensor(2.),)
# 可以看出 mse 对第一个变量 w 求偏导的结果为 2，计算过程：
# mse 为：  $(y - (wx + b))^2$ 
# 对 w 求偏导为：  $-2x(y - (wx + b))$ 
# 代入数值：  $-2 * 1 * (1 - (2 * 1 + 0)) = 2$ 
```

torch.backward

反向传播，也能实现求导，通过设置该 tensor 允许求导，那么该方法会对计算过程中所有声明了求导信息的变量进行求导，然后在对应的变量上通过 grad 属性获取求导结果，举例：

```
>>> x = torch.tensor(1.)
>>> b = torch.tensor(0.)
>>> w = torch.tensor(2., requires_grad=True)
>>> y = torch.tensor(1.)
>>> mse = torch.nn.functional.mse_loss(y, w*x+b)
>>> mse.backward()
# 对 mse 公式里需要求导的变量都进行求导
>>> w.grad
tensor(2.)
# w 的求导结果为 2
>>> x.grad
# 因为 x 没有声明需要求导，所以为空
```

注：

在反向传播当中，存在梯度累加问题，即第一次进行反向传播以后的值会进行保留，当第二次再进行反向传播时，则会将梯度和前面的进行累加，导致越来越大，因此在大多情况下，为了避免这种情况，需要我们手动进行梯度清零，举例：

```
>>> x = torch.tensor(5., requires_grad=True)
>>> y = x + 1
>>> y.backward()
# 第一次反向传播
```

```

>>> x.grad
# 求导结果（梯度）为 1
tensor(1.)
>>> y.backward()
# 第二次反向传播，如果报错（比如 x 的前面乘了一个数值），那么在反向传播里加上参数：retain_graph=True，即：
y.backward(retain_graph=True)
>>> x.grad
# 发现梯度在原来 1 的基础上又加上了 1
tensor(2.)
>>> del x.grad
# 梯度清零
>>> y.backward()
# 再次反向传播
>>> x.grad
# 可以看到又变回 1 了
tensor(1.)

```

基于上面的情况，在实际模型训练当中，我们首先会用优化器来进行梯度清零，然后再对 loss 进行反向传播，最后再用优化器来进行梯度下降，举例：

```

>>> x = torch.tensor([2.])
# 定义输入
>>> y = torch.tensor([4.])
# 定义输出
>>> layer = nn.Linear(1, 1)
# 定义网络参数
>>> layer.weight
# 可以看到权重 w 为 0.8162
Parameter containing:
tensor([[0.8162]], requires_grad=True)
>>> layer.bias
# 偏置 y 为 -0.4772
# 所以可以得出初始化的函数为： $f(x) = 0.8162 * x - 0.4772$ 
Parameter containing:

```

```

tensor([-0.4772], requires_grad=True)
>>> optim = torch.optim.SGD(layer.parameters(), lr=0.1)
# 定义随机梯度下降优化器，要更新的是网络层的参数，学习率为 0.1
>>> loss = y - layer(x)
# 定义目标函数，经过网络层后的数和 y 越接近越好
>>> loss
# 计算后可以看出 y 和计算的结果相差 2.8448
tensor([2.8448], grad_fn=<SubBackward0>)
>>> loss.backward()
# 反向传播
>>> layer.weight.grad
# 权重求导梯度为负，说明正向是在下降
tensor([[ -2. ]])
>>> layer.bias.grad
# 偏置同理
tensor([-1. ])
>>> optim.step()
# 梯度下降，更新权重和偏置
>>> layer.weight
# 可以看到权重减了-2*0.1
Parameter containing:
tensor([[1.0162]], requires_grad=True)
>>> layer.bias
# 偏置减了-1*0.1
Parameter containing:
tensor([-0.3772], requires_grad=True)
>>> loss = y - layer(x)
>>> loss
# 可以看到 loss 减小了，所以更新网络层参数后，计算后的值和 y 更加接近
tensor([2.3448], grad_fn=<SubBackward0>)

```

可视化操作

tensorboardX

在 TensorFlow 里有 tensorboard 可以进行训练过程的可视化，而在 Pytorch 里也提供了 tensorboardX 几乎和 tensorboard 一样，适合习惯了 tensorboard 的使用者，具体使用可以参考：

<https://www.jianshu.com/p/713c1bc4cf8a>

<https://www.jianshu.com/p/46eb3004beca>

visdom

pytorch 提供的另一个可视化，个人更喜欢这种界面样式（~~tensorboard 的黄色底色看着不太习惯~~）

安装

```
pip install visdom
```

使用步骤

1. 通过命令 `python -m visdom.server` 启动 visdom 服务器进行监听
2. 导入可视化类：`from visdom import Visdom`
3. 实例化可视化类，并通过 `line/bar/text` 等提供的 API 进行绘图

具体使用可以参考：

https://blog.csdn.net/wen_fei/article/details/82979497

<https://ptorch.com/news/77.html>

torchvision 模块

提供了很多计算视觉相关的数据集，以及较流行的模型等，参考：<https://blog.csdn.net/zhenaoxil077/article/details/80955607>

Pytorch 基础的基本概念

1. 什么是 Pytorch，为什么选择 Pytorch？

PyTorch 是一个基于 Python 的科学计算包，主要定位两类人群：

NumPy 的替代品，可以利用 GPU 的性能进行计算。

深度学习研究平台拥有足够的灵活性和速度

2. Pytorch 的安装

3.配置 Python 环境

4.准备 Python 管理器

5.通过命令行安装 PyTorch

买过某机构的 GPU 云服务器，默认提供了 pytorch 等深度学习环境，不用本地装，偷个懒！

环境安装配置也可以参考 <http://pytorchchina.com/>

6. PyTorch 基础概念

Tensors 类似于 NumPy 的 ndarrays，同时 Tensors 可以使用 GPU 进行计算。

7.通用代码实现流程(实现一个深度学习的代码流程)

一个神经网络的通用训练过程如下：

- ### 1.定义一个包含可训练参数的神经网络

- ## 2.迭代整个输入

- ### 3.通过神经网络处理输入

- #### 4.计算损失(loss)

- ## 5.反向传播梯度到神经网络的参数

- ## 6.更新网络的参数

#加载并归一化 CIFAR10

#使用 torchvision ,用它来加载 CIFAR10 数据非常简单。

```
import torch
```

```
import torchvision
```

```
import torchvision.transforms as transforms
```

#torchvision 数据集的输出是范围在 $[0, 1]$ 之间的 PILImage，我们将他们转换成归一化范围为 $[-1, 1]$ 之间的张量 Tensors。

```
transform = transforms.Compose(
```

`[transforms.ToTensor(),`

```
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
```

```
download=True, transform=transform)
```

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
```

```
shuffle=True, num_workers=2)
```

```
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
```

```
download=True, transform=transform)
```

```
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
```

```
shuffle=False, num workers=2)
```

```
classes = ('plane', 'car', 'bird', 'cat',  
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

输出

```
In [2]: transform = transforms.Compose(  
...:     [transforms.ToTensor(),  
...:      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])  
...:  
...: trainset = torchvision.datasets.CIFAR10(root='./data', train=True,  
...:                                         download=True, transform=transfo  
...: rm)  
...: trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,  
...:                                           shuffle=True, num_workers=2)  
...:  
...: testset = torchvision.datasets.CIFAR10(root='./data', train=False,  
...:                                         download=True, transform=transfor  
...: m)  
...: testloader = torch.utils.data.DataLoader(testset, batch_size=4,  
...:                                           shuffle=False, num_workers=2)  
...:  
...: classes = ('plane', 'car', 'bird', 'cat',  
...:            'deer', 'dog', 'frog', 'horse', 'ship', 'truck')  
Files already downloaded and verified  
Files already downloaded and verified
```

展示其中的某些训练图片。

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

```
# functions to show an image
```

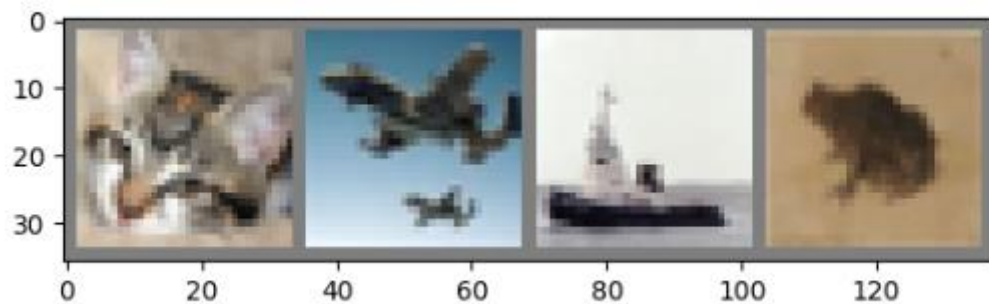
```
def imshow(img):  
    img = img / 2 + 0.5     # unnormalize  
    npimg = img.numpy()  
    plt.imshow(np.transpose(npimg, (1, 2, 0)))  
    plt.show()
```

```

# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))

```



```

...: def imshow(img):
...:     img = img / 2 + 0.5     # unnormalize
...:     npimg = img.numpy()
...:     plt.imshow(np.transpose(npimg, (1, 2, 0)))
...:     plt.show()
...:
...: # get some random training images
...: dataiter = iter(trainloader)
...: images, labels = dataiter.next()
...:
...: # show images
...: imshow(torchvision.utils.make_grid(images))
...: # print labels
...: print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
cat    car    ship    cat

```


自定义卷积神经网络

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()

定义一个损失函数和优化器
import torch.optim as optim
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

训练网络
for epoch in range(2): # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data
        # zero the parameter gradients
        optimizer.zero_grad()
```

```

# forward + backward + optimize
outputs = net(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
# print statistics
running_loss += loss.item()
if i % 2000 == 1999:    # print every 2000 mini-batches
    print('[%d, %5d] loss: %.3f' %
          (epoch + 1, i + 1, running_loss / 2000))
    running_loss = 0.0
print('Finished Training')

```

```

[1, 2000] loss: 2.145
[1, 4000] loss: 1.827
[1, 6000] loss: 1.645
[1, 8000] loss: 1.569
[1, 10000] loss: 1.514
[1, 12000] loss: 1.482
[2, 2000] loss: 1.394
[2, 4000] loss: 1.393
[2, 6000] loss: 1.350
[2, 8000] loss: 1.341
[2, 10000] loss: 1.324
[2, 12000] loss: 1.285
Finished Training

```

在测试集上测试网络

```

correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))

```

```
In [9]: correct = 0
....: total = 0
....: with torch.no_grad():
....:     for data in testloader:
....:         images, labels = data
....:         outputs = net(images)
....:         _, predicted = torch.max(outputs.data, 1)
....:         total += labels.size(0)
....:         correct += (predicted == labels).sum().item()
....:
....: print('Accuracy of the network on the 10000 test images: %d %%' % (
....:     100 * correct / total))
Accuracy of the network on the 10000 test images: 53 %
```

分类: Pytorch