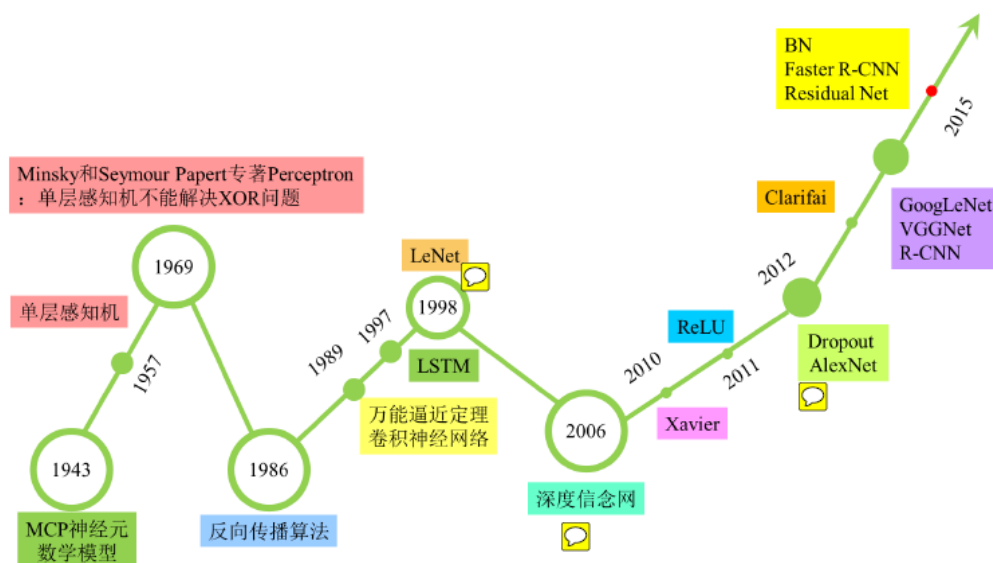


一名算法工程师的PyTorch学习之路

1. Pytorch入门简介

提及Pytorch就不得不先从深度学习开始讲起。从事数据相关岗位的都知道，深度学习是机器学习的一个子方向，其主要以神经网络为基础模块，通过灵活组合一定层数的网络实现特定的模型功能，尤其擅长于计算机视觉（CV）和自然语言处理（NLP）方向。其发展历史上，在经历了两次高潮和两次低谷之后，目前处于第三次高速发展的蓬勃期。



这里，深度学习的深度主要体现为构建的模型层数较多，故称之为“深”；但其实这里隐藏着一个重要假设，就是构建的模型都以神经元作为网络的最小单元，所以严谨的讲应叫做基于神经网络的深度学习。自然，也可以不基于神经网络，比如周志华团队前几年探索提出了深度随机森林模型，可谓提出了深度学习的一个新的研究思路。

从理论研究到工业应用，其中必然少不了成熟的工业级实现。以python语言为基础，对于经典的机器学习模型，那么必然人人皆知scikit-learn；而若提及深度学习，则相应的工具包则不那么“集中和统一”，甚至称得上是大厂纷争之地。其中，最具代表性和广泛使用的当属TensorFlow和Pytorch，前者源于google，后者发于Facebook；前者以工业应用居多，后者则流行于学术界。当然，单论学术界还是工业界而言，二者也没有明确的界限。

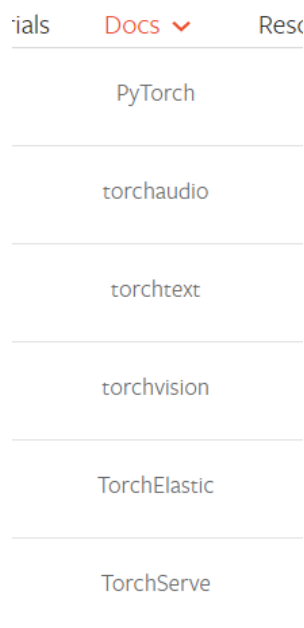
起初，在了解到TensorFlow广泛应用工业界，而自己早已远离院校所以就直接入坑了TensorFlow，当了一段时间的TF boy，尤其是了解到TensorFlow2.0克服了早期1.0版本饱受诟病的静态图问题，所以也不认为TensorFlow有啥缺点。但后来，随着学习的深入，加之通过周边同事的了解，发现Pytorch有着更为优秀的特点：比如与Numpy的设计更为接近，语法风格更加Pythonic等等。所以，个人也就果断转投Pytorch阵营。

本文作为第一篇，仅用来介绍Pytorch能干什么，以及对为什么这么设计的个人理解。

Torch是一个老牌的深度学习框架，最早是基于lua语言开发的，由于其开发语言的小众性，所以其发展和应用也是受到了很多限制。自从Facebook开源了Python生态圈的Torch工具包——Pytorch之后，其就一直一直是匹敌TensorFlow的一个重量级工具。目前Pytorch在GitHub上获得54k star（TensorFlow目前在GitHub上获得163k star，差距还是比较大的，大概有3倍之多）。

也正是由于深度学习最广泛的舞台在于图像和语音以及文本等应用方向，所以Pytorch框架生态还包含了其他常用工具，例如：

- torchvision
- torchtex
- torchaudio



当然，在上述torch生态框架中，Pytorch仍然是基础和核心

作为一个深度学习工具包，Pytorch能用来干什么呢？这里引用官方文档对其定位的描述，广义来说有两方面功能：

PyTorch is a Python package that provides two high-level features:

- Tensor computation (like NumPy) with strong GPU acceleration
- Deep neural networks built on a tape-based autograd system

即：

- 支持GPU加速的Tensor计算能力
- 支持自动求导的深度神经网络构建

那么问题来了：都说Pytorch是一个深度学习工具，为什么其核心功能设计为如上两点？对此，个人理解如下：

其一：Tensor是深度学习模型构建和训练的基础，其地位就好比是array之于Numpy、DataFrame之于Pandas，其本身是一种数据结构，但却构成了Pytorch的灵魂所在。这里，Tensor英文原义为“张量”，其实就是对应一个多维数组，本质上跟numpy的ndarray是一致的。

从这一角度来看，Pytorch可视为是numpy的升级版，这里的升级主要体现为可以利用GPU的强力并行计算能力。如果有Numpy基础，学习Pytorch其实可以很简单；另一方面，学Pytorch也完全可以作为是对Numpy的一个补充，而不去考虑构建深度学习模型的用途。







其二：Pytorch定位为一个深度学习工具，其更为主体的功能在于支持深度学习模型的构建和训练。与此同时，与经典机器学习中有成熟模型不同的是，深度学习网络大多没有固定的模型或范式，而一般由使用者将多个基础模块灵活搭配来组成（当然，其实也有一些成熟的模型，例如LeNet-5、AlexNet和VGGNet等，但更普遍的仍然需要使用者自己去定制），所以Pytorch对深度学习的支持不在于集成了多

少成熟的模型，而在于提供了基础的深度学习模块，这些就好似脚手架一般，可以任意组合搭配，从而实现更为自由定制化的功能。

Pytorch功能还是比较丰富和繁杂的，最好的学习平台是查阅其官方文档，<https://pytorch.org/>。源于Pytorch群体的广泛性，目前其文档支持多种语言，包括中文文档在内，这也为自学者快速入门提供了更多渠道。我个人也是受益其中，后续的推文也将以此作为重要参考框架。

COMMUNITY

Join the PyTorch developer community to contribute, learn, and get your questions answered.

 PyTorchDiscuss Browse and join discussions on deep learning with PyTorch.	 Slack Discuss advanced topics. Request access: https://bit.ly/ptslack	 中文文档 Docs and tutorials in Chinese, translated by the community.
 파이토치(PyTorch) Tutorials in Korean, translated by the community.	 日本語(PyTorch) Tutorials in Japanese, translated by the community.	 Contributors Stay up to date with the codebase and discover RFCs, PRs and more

本篇推文就写这么多，对标从工具入门到模型建模，后续将每周更新一篇Pytorch学习系列推文。

2. 何为Tensor?



作为Tensor的入门介绍篇，本文主要探讨三大"哲学"问题：

- 何为Tensor?
- Tensor如何创建?
- Tensor有哪些特性?

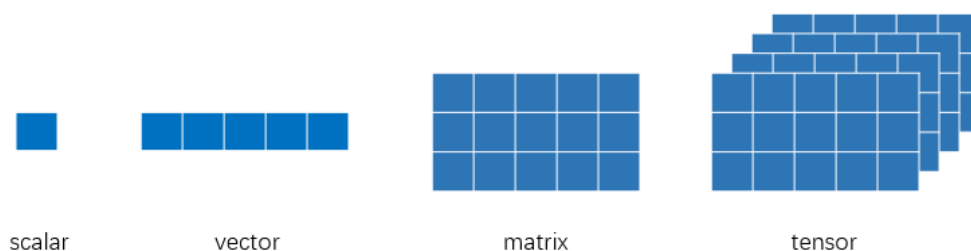
2.1 何为Tensor

什么是Tensor? Tensor英文原义是张量，在PyTorch官网中对其有如下介绍：

TORCH.TENSOR

A `torch.Tensor` is a multi-dimensional matrix containing elements of a single data type.

也就是说，一个Tensor是一个包含单一数据类型的高维矩阵，简言之Tensor其实和Numpy里的array是一样的。那么，高维矩阵，几维算高维呢（类似于深度学习，多深的网络才算是深度的？）？一般而言，描述Tensor的高维特性通常用三维及以上的矩阵来描述，例如下图所示：单个元素叫标量（scalar），一个序列叫向量（vector），多个序列组成的平面叫矩阵（matrix），多个平面组成的立方体叫张量（tensor）。当然，就像矩阵有一维矩阵和二维矩阵乃至多维矩阵一样，张量也无需严格限制在三维以上才叫张量，在深度学习的范畴内，标量、向量和矩阵都统称为张量。



熟悉机器学习的都知道，有监督机器学习模型的标准输入通常是多个特征列组成的输入矩阵和单个特征列组成的标签向量（多输出时，标签也可以是二维矩阵），用sklearn的约定规范就是训练数据集为 (X, y) ，其中大写X表示特征是一个二维的矩阵，小写y表示标签是一个一维的向量。那么，在深度学习领域中，为何要约定高维数组——Tensor呢？

其实这是出于应用方向的实际需要，以深度学习当前最成熟的两大应用方向莫过于CV和NLP两大领域，前者面向图像和视频，后者面向语音和文本，二者分别以卷积神经网络和循环神经网络为核心基础模块。而在这两个应用方向中，标准的输入数据集都至少是三维以上，例如：

- 图像数据集至少包含三个维度： $N \times H \times W$ ，即样本数 \times 图像高 \times 图像宽；如果是彩色图像，那么还要增加一个通道C，变为 $N \times C \times H \times W$ ；如果是视频图像，那么可能还要再增加一个维度T；
- 文本数据集典型的输入包含三个维度： $N \times L \times H$ ，即样本数 \times 序列长度 \times 特征数；

源于这些应用的需要，所以深度学习模型的输入数据结构一般都要3维以上，这也就直接促使了Tensor的出现。更进一步的，Tensor也不是PyTorch特有的定义，而是众多深度学习框架广泛使用的数据结构，例如TensorFlow更是形象的将Tensor加入了框架的命名之中。

小结一下：PyTorch中的Tensor是深度学习中广泛使用的数据结构，本质上就是一个高维的矩阵，甚至将其理解为NumPy中array的推广和升级也不为过。但由于其支持的一些特殊特性（详见后文第3小节），Tensor在用于支撑深度学习模型和训练时更为便利。

2.2 如何创建Tensor

前面介绍了何为Tensor，那么接下来就需要了解如何创建Tensor。一般而言，创建一个Tensor大体有三种方式：

- 从已有其他数据结构转化创建为Tensor
- 随机初始化一个Tensor
- 从已保存文件加载一个Tensor

当然，这大概也是一段计算机程序中所能创建数据的三种通用方式了，比如基于NumPy创建一个Array其实大体也是这三种方式。

下面依次予以介绍。

1.从已有其他数据结构转化创建为Tensor

这可能是实际应用中最常用的一种形式，比如从一个列表、从一个NumPy的array中读取数据，而后生成一个新的Tensor。为了实现这一目的，常用的有两种方式：

- torch.tensor
- torch.Tensor

没错，二者的区别就是前者用的是tensor函数（t是小写），后者用的是Tensor类（T是大写）。当然，二者也有一些区别，比如在创建Tensor的默认数据类型、支持传参以及个别细节的处理方面。举个例子，首先是创建的Tensor默认数据类型不同：

```
t = torch.tensor([1, 2])
T = torch.Tensor([1, 2])
print(t, T)
print(t.dtype, T.dtype)

tensor([1, 2]) tensor([1., 2.])
torch.int64 torch.float32
```

其次，应用Tensor类初始化输入一个整数将返回一个以此为长度的全零一维张量，而tensor函数则返回一个只有该元素的零维张量：

```
t = torch.tensor(3)
T = torch.Tensor(3)
print(t, T)

tensor(3) tensor([0., 0., 0.])
```

当然，上述有一个细节需要优先提及：应用Tensor类接收一个序列创建Tensor时，返回的数据类型为float型，这是因为Tensor是FloatTensor的等价形式，即除此之外还有ByteTensor，IntTensor，LongTensor以及DoubleTensor等不同的默认数据类型。

基于已有数据创建Tensor还有两个常用函数：

- from_numpy
- as_tensor

二者与上述方法最大的不同在于它们返回的Tensor与原有数据是共享内存的，而前述的tensor函数和Tensor类则是copy后创建一个新的对象。举个例子来说：

```
a = np.array([1, 2])
b = np.array([1, 2])
t1 = torch.tensor(a)
t2 = torch.from_numpy(b)
print("t1:", t1)
print("t2:", t2)

t1: tensor([1, 2], dtype=torch.int32)
t2: tensor([1, 2], dtype=torch.int32)
```

```
# 分别更改t1和t2中的数值
t1[0] = 3
t2[0] = 3
# a不变, b随之改变
print("a:", a)
print("b:", b)
```

```
a: [1 2]
b: [3 2]
```

2. 随机初始化一个Tensor

随机初始化也是一种常用的形式，比如在搭建一个神经网络模型中，其实在添加了一个模块的背后也自动随机初始化了该模块的权重。整体而言，这类方法大体分为两种形式：

- torch.xxxx，创建一个特定类型的tensor，例如torch.ones，torch.randn等等
- torch.xxx_like，即根据一个已有Tensor创建一个与其形状一致的特定类型tensor，例如torch.ones_like，torch.randn_like等等

```
[x for x in dir(torch) if x.endswith("like")]  
  
['empty_like',  
 'full_like',  
 'ones_like',  
 'rand_like',  
 'randint_like',  
 'randn_like',  
 'zeros_like']
```

例如，随机构建一个PyTorch中的全连接单元Linear，其会默认创建相应的权重系数和偏置（注意，由于网络参数一般是需要参与待后续模型训练，所以默认requires_grad=True）：

```
linear = torch.nn.Linear(2, 3)  
linear.weight, linear.bias  
  
(Parameter containing:  
 tensor([[ -0.3377, -0.6523],  
         [ 0.6873, -0.6760],  
         [-0.6482,  0.4089]], requires_grad=True),  
 Parameter containing:  
 tensor([-0.0853,  0.2903,  0.3780], requires_grad=True))
```

整体来看，这两类方法与NumPy中的相应方法特性几乎一致，基本可以从函数名中get到其相应的含义，这里也不再展开。

3. 从已保存文件加载一个Tensor

文件作为交互数据的常用形式，PyTorch中自然也不会缺席。实际上，PyTorch不会刻意区分要保存和加载的对象是何种形式，可以是训练好的网络，也可以是数据，这在Python中就是pickle。实现这一对互逆功能的函数是torch.save和torch.load。另外，值得指出的是，保存后的文件没有明确的后缀格式要求，常用的后缀格式有三种：

- .pkl
- .pth
- .pt

举个例子：

```
x = torch.tensor([0, 1, 2, 3, 4])  
torch.save(x, 'tensor.pkl')  
##### 生成文件: tensor.pkl  
torch.load('tensor.pkl')  
  
tensor([0, 1, 2, 3, 4])
```


小结一下：PyTorch中创建一个Tensor大体可分为三种方法，即：1) 从一个已有数据结构转换为Tensor，2) 随机初始化生成一个Tensor，3) 将已保存的文件加载为Tensor。其中，第一种方法主要用于构建训练数据集，第二种方法隐藏于网络模块参数的初始化，而第三种方法则可用于大型数据集的保存和跨环境使用。

2.3 Tensor的特性

PyTorch之所以定义了Tensor来支持深度学习，而没有直接使用Python中的一个list或者NumPy中的array，终究是因为Tensor被赋予了一些独有的特性。这里，我也将Tensor的特性概括为三个方面：

- 丰富的常用操作函数
- 灵活的dtype和CPU/GPU自由切换存储
- 自动梯度求解

下面分别予以介绍。

1.丰富的常用函数操作

Tensor本质上是一个由数值型元素组成的高维矩阵，而深度学习的过程其实也就是各种矩阵运算的过程，所以Tensor作为其基础数据结构，自然也就需要支持丰富的函数操作。构建一个Tensor实例，通过Python中的dir属性获取tensor实例支持的所有API，过滤以"_"开头的系统内置方法外（例如"str"这种），剩余结果仍有567种，其支持的函数操作之丰富程度可见一斑。

```
tensor = torch.tensor(3)
len([x for x in dir(tensor) if not x.startswith("_")])
```

567

这些函数有的是对自身进行操作，例如tensor.max(), tensor.abs()等等，有的是用于与其他tensor进行相关操作，例如tensor.add()用于与另一个tensor相加，tensor.mm()用于与另一个tensor进行矩阵乘法等等。当然，这里的相加和相乘对操作的两个tensor尺寸有所要求。

除了支持的函数操作足够丰富外，tensor的API函数还有另一个重要的便利特性：绝大多数函数都支持两个版本：带下划线版和不带下划线版，例如tensor.abs()和tensor.abs_()，二者均返回操作后的Tensor，但同时带下划线版本属于inplace操作，即调用后自身数据也随之改变。

```
[x for x in dir(tensor)
    if not x.startswith("_") and x.endswith("_")]
```

```
['abs_',
 'absolute_',
 'acos_',
 'acosh_',
 'add_',
 'addbmm_',
 'addcddiv_',
 'addcmul_',
 'addmm_',
 'addmv_',
 'addr_',
 'apply_',
 'arccos_']
```

inplace版函数共有159种

2.灵活的dtype和CPU/GPU自由切换

前面在介绍Tensor的创建时已提到了dtype的概念，其类似于NumPy和Pandas中的用法，用于指定待创建Tensor的数据结构。PyTorch中定义了10种不同的数据结构，包括不同长度的整型、不同长度的浮点型，整个Tensor的所有元素必须数据类型相同，且必须是数值类型（NumPy中的array也要求数组中的元素是同质的，但支持字符串类型的）：

Data types

Torch defines 10 tensor types with CPU and GPU variants which are as follows:

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	<code>torch.float32</code> or <code>torch.float</code>	<code>torch.FloatTensor</code>	<code>torch.cuda.FloatTensor</code>
64-bit floating point	<code>torch.float64</code> or <code>torch.double</code>	<code>torch.DoubleTensor</code>	<code>torch.cuda.DoubleTensor</code>
16-bit floating point ¹	<code>torch.float16</code> or <code>torch.half</code>	<code>torch.HalfTensor</code>	<code>torch.cuda.HalfTensor</code>
16-bit floating point ²	<code>torch.bfloat16</code>	<code>torch.BFloat16Tensor</code>	<code>torch.cuda.BFloat16Tensor</code>

Tensor中的10种数据类型（不完整版）

除了支持不同的数值数据类型外，Tensor的另一大特色是其支持不同的计算单元：CPU或GPU，支持GPU加速也是深度学习得以大规模应用的一大关键。为了切换CPU计算（数据存储于内存）或GPU计算（数据存储于显存），Tensor支持灵活的设置存储设备，包括如下两种方式：

- 创建tensor时，通过device参数直接指定
- 通过tensor.to()函数切换

```
tensor.to?
```

Docstring:

```
to(*args, **kwargs) -> Tensor
```

```
Performs Tensor dtype and/or device conversion. A :class:`torch.dtype` and :class:`torch.device` are inferred from the arguments of ``self.to(*args, **kwargs)``.
```

to()既可用于切换存储设备，也可切换数据类型

当然，能够切换到GPU的一大前提是运行环境带有独立显卡并已配置CUDA.....此外，除了dtype和device这两大特性之外，其实Tensor还有第三个特性，即layout，布局。主要包括strided和sparse_coo两种，该特性一般不需要额外考虑。

torch.layout

CLASS `torch.layout`

• WARNING

The `torch.layout` class is in beta and subject to change.

A `torch.layout` is an object that represents the memory layout of a `torch.Tensor`. Currently, we support `torch.strided` (dense Tensors) and have beta support for `torch.sparse_coo` (sparse COO Tensors).

`torch.strided` represents dense Tensors and is the memory layout that is most commonly used. Each strided tensor has an associated `torch.Storage`, which holds its data. These tensors provide multi-dimensional, **strided** view of a storage. Strides are a list of integers: the k-th stride represents the jump in the memory necessary to go from one element to the next one in the k-th dimension of the Tensor. This concept makes it possible to perform many tensor operations efficiently.

3.自动梯度求解

如果说支持丰富函数操作和灵活的特性，那么Tensor还不足以支撑深度学习的基石，关键是还需要自动梯度求解。

深度学习模型的核心是在于神经元的连接，而神经元之间连接的关键在于网络权重，也就是各个模块的参数。正因为网络参数的不同，所以才使得相同的网络结构能实现不同的模型应用价值。那么，如何学习最优网络参数呢？这就是深度学习中的优化利器：梯度下降法，而梯度下降法的一大前提就是支持自动梯度求解。

简言之，Tensor为了支持自动梯度求解，大体流程如下：

- Tensor支持grad求解，即`requires_grad=True`
- 根据Tensor参与计算的流程，将Tensor及相应的操作函数记录为一个树结构（或称之为有向无环图：DAG），计算的方向为从叶节点流向根节点
- 根据根节点Tensor与目标值计算相应的差值（loss），然后利用链式求导法则反向逐步计算梯度（也即梯度的反向传播）

Tensor中的自动梯度求导有很多细节值得展开，这里仅稍加介绍，并留待后续单独推文加以分享。

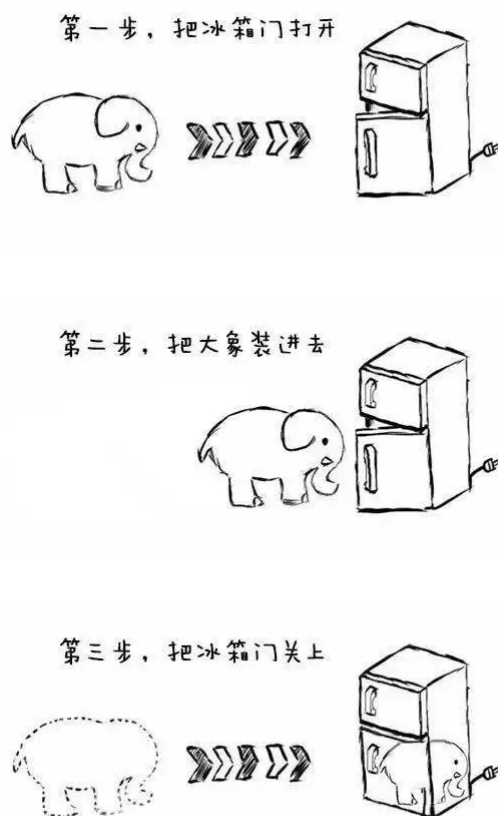
小结一下：Tensor具有很多特性，这使得其可以支撑深度学习的复杂操作，个人认为比较重要的包括三个方面，即：1) 丰富的操作函数，2) 三大特性（dtype、device和layout），以及3) 自动梯度求导。

2.4 小结

本文从何为Tensor—如何构建Tensor—Tensor有何特性三个方面入手，简要介绍了PyTorch中的核心数据结构——Tensor。理解并熟练运用Tensor的常用操作是深度学习的基石，就像“站在岸上学不会游泳”一样，学习Tensor的重点在于实践，毕竟“无他，唯手熟尔”！

3. 构建一个深度学习模型需要哪几步？

不同于经典的机器学习流程，深度学习模型的搭建和训练更为灵活和简单，称之为灵活是因为一般没有成熟和直接可用的模型，而更多需要使用者自己去设计和组装各个网络模块；称之为简单是因为深度学习往往实现端到端的训练，即直接从原始数据集到模型输出，而无需经典机器学习中的数据预处理、特征工程、特征选择等多阶段式的工作流。



类似于把大象装进冰箱需要3步一样，构建一个深度学习模型也可以将其分为三步：

- 数据集准备
- 模型定义
- 模型训练

本文就首先围绕这三个环节加以介绍，然后给出一个简单的应用案例。

3.1 数据集准备

理论上，深度学习中的数据集准备与经典机器学习中的数据集准备并无本质性差别，大体都是基于特定的数据构建样本和标签的过程，其中这里的样本依据应用场景的不同而有不同的样式，比如CV领域中典型的的就是图片，而NLP领域中典型的的就是一段段的文本。但无论原始样本如何，最终都要将其转化为数值型的Tensor。

当然，将数据集转化为Tensor之后理论上即可用于深度学习模型的输入和训练，但为了更好的支持模型训练以及大数据集下的分batch进行训练，PyTorch中提供了标准的数据集类型（Dataset），而我们则一般是要继承此类来提供这一格式。这里主要介绍3个常用的数据集相关的类：

- Dataset：所有自定义数据集的基类
- TensorDataset：Dataset的一个wrapper，用于快速构建Dataset
- DataLoader：Dataset的一个wrapper，将Dataset自动划分为多个batch

1.Dataset

Dataset是PyTorch中提供的一个数据集基类，首先查看Dataset的签名文档如下：

```
class Dataset(Generic[T_co]):
    r"""An abstract class representing a :class:`Dataset`.

    All datasets that represent a map from keys to data samples should subclass
    it. All subclasses should overwrite :meth:`~__getitem__`, supporting fetching a
    data sample for a given key. Subclasses could also optionally overwrite
    :meth:`~__len__`, which is expected to return the size of the dataset by many
    :class:`~torch.utils.data.Sampler` implementations and the default options
    of :class:`~torch.utils.data.DataLoader`.

    .. note::
        :class:`~torch.utils.data.DataLoader` by default constructs a index
        sampler that yields integral indices. To make it work with a map-style
        dataset with non-integral indices/keys, a custom sampler must be provided.
    """
```

从中可以看出，所有自定义的数据集都应继承此类，并重载其中的`getitem`和`len`两个方法即可。当然，还需通过类初始化方法`init`来设置要加载的数据。典型的自定义一个Dataset的实现如下：

```
class MyDataset(Dataset):
    def __init__(self, x, y):
        super().__init__()
        .....

    def __getitem__(self):
        return .....

    def __len__(self):
        return .....
```

2.TensorDataset

上述通过Dataset的方式可以实现一个标准自定义数据集的构建，但如果对于比较简单的数据集仍需八股文似的重载`getitem`和`len`两个方法，则难免有些繁杂和俗套。而TensorDataset就是对上述需求的一个简化，即当仅需将特定的tensor包裹为一个Dataset类型作为自定义数据集时，那么直接使用TensorDataset即可。这里仍然先给出其签名文档：

```
class TensorDataset(Dataset[Tuple[Tensor, ...]]):
    r"""Dataset wrapping tensors.

    Each sample will be retrieved by indexing tensors along the first dimension.

    Args:
        *tensors (Tensor): tensors that have the same size of the first dimension.
    """
    tensors: Tuple[Tensor, ...]
```

具体应用时，只需将若干个tensor格式的输入作为参数传入TensorDataset，而后返回结果即是一个标准的Dataset类型数据集。标准使用方式如下：

```
my_dataset = TensorDataset(tenso_x, tensor_y)
```

3.DataLoader

深度学习往往适用于大数据集场景，训练一个成熟的深度学习模型一般也需要足够体量的数据。所以，在深度学习训练过程中一般不会每次都把所有训练集数据一次性的喂给模型，而是小批量分批次的训练，其中每个批量叫做一个batch，完整的训练集参与一次训练叫做一个epoch。实现小批量多批次的方式有很多，比如完全可以通过随机取一个索引分片的方式来实现这一工作，但更为标准和优雅的方式则是使用DataLoader。其给出的签名文档如下：

```
class DataLoader(Generic[T_co]):  
    """  
    Data loader. Combines a dataset and a sampler, and provides an iterable over  
    the given dataset.  
  
    The :class:`~torch.utils.data.DataLoader` supports both map-style and  
    iterable-style datasets with single- or multi-process loading, customizing  
    loading order and optional automatic batching (collation) and memory pinning.
```

可见，DataLoader大体上可以等价为对一个Dataset实现随机采样（sampler），而后对指定数据集提供可迭代的类型。相应的，其使用方式也相对简单：直接将一个Dataset类型的数据集作为参数传入DataLoader即可。简单的使用样例如下：

```
dataloader = DataLoader(MyDataset, batch_size=128, shuffle=True)
```

以上是应用PyTorch构建数据集时常用的三种操作，基本可以覆盖日常使用的绝大部分需求，后面会结合实际案例加以完整演示。

3.2 网络架构定义

深度学习与经典机器学习的一个最大的区别在于模型结构方面，经典机器学习模型往往有着固定的范式和结构，例如：随机森林就是由指定数量的决策树构成，虽然这里的`n_estimators`可以任选，但整体来看随机森林模型的结构是确定的；而深度学习模型的基础在于神经网络，即由若干的神经网络层构成，每一层使用的神经网络模块类型可以不同（全连接层、卷积层等等），包含的神经元数量差异也会带来很大的不同。也正因如此，深度学习给使用者提供了更大的设计创新空间。

当然，网络架构（Architecture）的设计不需要从零开始，PyTorch这些深度学习框架的一大功能就是提供了基础的神经网络模块（Module），而使用者仅需根据自己的设计意图将其灵活组装起来即可——就像搭积木一般！PyTorch中所有网络模块均位于`torch.nn`模块下（nn=nueral network），总共包括以下模块：

```

__all__ = [
    'Module', 'Identity', 'Linear', 'Conv1d', 'Conv2d', 'Conv3d', 'ConvTranspose1d',
    'ConvTranspose2d', 'ConvTranspose3d', 'Threshold', 'ReLU', 'Hardtanh', 'ReLU6',
    'Sigmoid', 'Tanh', 'Softmax', 'Softmax2d', 'LogSoftmax', 'ELU', 'SELU', 'CELU', 'GLU', 'GELU', 'Hardshrink',
    'LeakyReLU', 'LogSigmoid', 'Softplus', 'Softshrink', 'MultiheadAttention', 'PReLU', 'Softsign', 'Softmin',
    'Tanhshrink', 'RRReLU', 'L1Loss', 'NLLLoss', 'KLDivLoss', 'MSELoss', 'BCELoss', 'BCEWithLogitsLoss',
    'NLLLoss2d', 'PoissonNLLLoss', 'CosineEmbeddingLoss', 'CTCLoss', 'HingeEmbeddingLoss', 'MarginRankingLoss',
    'MultiLabelMarginLoss', 'MultiLabelSoftMarginLoss', 'MultiMarginLoss', 'SmoothL1Loss', 'GaussianNLLLoss',
    'HuberLoss', 'SoftMarginLoss', 'CrossEntropyLoss', 'Container', 'Sequential', 'ModuleList', 'ModuleDict',
    'ParameterList', 'ParameterDict', 'AvgPool1d', 'AvgPool2d', 'AvgPool3d', 'MaxPool1d', 'MaxPool2d',
    'MaxPool3d', 'MaxUnpool1d', 'MaxUnpool2d', 'MaxUnpool3d', 'FractionalMaxPool2d', 'FractionalMaxPool3d',
    'LPPool1d', 'LPPool2d', 'LocalResponseNorm', 'BatchNorm1d', 'BatchNorm2d', 'BatchNorm3d', 'InstanceNorm1d',
    'InstanceNorm2d', 'InstanceNorm3d', 'LayerNorm', 'GroupNorm', 'SyncBatchNorm',
    'Dropout', 'Dropout2d', 'Dropout3d', 'AlphaDropout', 'FeatureAlphaDropout',
    'ReflectionPad1d', 'ReflectionPad2d', 'ReflectionPad3d', 'ReplicationPad2d', 'ReplicationPad1d', 'ReplicationPad3d',
    'CrossMapLRN2d', 'Embedding', 'EmbeddingBag', 'RNNBase', 'RNN', 'LSTM', 'GRU', 'RNNCellBase', 'RNNCell',
    'LSTMCell', 'GRUCell', 'PixelShuffle', 'PixelUnshuffle', 'Upsample', 'UpsamplingNearest2d', 'UpsamplingBilinear2d',
    'PairwiseDistance', 'AdaptiveMaxPool1d', 'AdaptiveMaxPool2d', 'AdaptiveMaxPool3d', 'AdaptiveAvgPool1d',
    'AdaptiveAvgPool2d', 'AdaptiveAvgPool3d', 'TripletMarginLoss', 'ZeroPad2d', 'ConstantPad1d', 'ConstantPad2d',
    'ConstantPad3d', 'Bilinear', 'CosineSimilarity', 'Unfold', 'Fold',
    'AdaptiveLogSoftmaxWithLoss', 'TransformerEncoder', 'TransformerDecoder',
    'TransformerEncoderLayer', 'TransformerDecoderLayer', 'Transformer',
    'LazyLinear', 'LazyConv1d', 'LazyConv2d', 'LazyConv3d',
    'LazyConvTranspose1d', 'LazyConvTranspose2d', 'LazyConvTranspose3d',
    'LazyBatchNorm1d', 'LazyBatchNorm2d', 'LazyBatchNorm3d',
    'LazyInstanceNorm1d', 'LazyInstanceNorm2d', 'LazyInstanceNorm3d',
    'Flatten', 'Unflatten', 'Hardsigmoid', 'Hardswish', 'SiLU', 'Mish', 'TripletMarginWithDistanceLoss', 'ChannelShuffle'
]

```

这些模块数量庞大，功能各异，构成了深度学习模型的核心。但就其功能而言，大体分为以下几类：

- 模型功能类：例如Linear、Conv2d，RNN等，分别对应全连接层、卷积层、循环神经网络，
- 激活函数：例如Sigmoid，Tanh，ReLU等，
- 损失函数：CrossEntropyLoss，MSELoss等，其中前者是分类常用的损失函数，后者是回归常用的损失函数
- 规范化：LayerNorm等，
- 防止过拟合：Dropout等
- 其他

某种程度上讲，学习深度学习的主体在于理解掌握这些基础的网络模块其各自的功能和使用方法，在此基础上可根据自己对数据和场景的理解来自定义设计网络架构，从而实现预期的模型效果。

该部分内容过于庞大，断不是一篇两篇文章能解释清楚的，自认当前自己也不足以完全理解，所以对这些模块的学习和介绍当徐徐图之、各个击破。

在这些单个网络模块的基础上，构建的完整网络模型则需继承PyTorch中的Module类来加以实现（这一过程类似于继承Dataset类实现自定义数据集），这里仍然给出Module的签名文档：

```

class Module:
    """Base class for all neural network modules.

    Your models should also subclass this class.

    Modules can also contain other Modules, allowing to nest them in
    a tree structure. You can assign the submodules as regular attributes::

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

        class Model(nn.Module):
            def __init__(self):
                super(Model, self).__init__()
                self.conv1 = nn.Conv2d(1, 20, 5)
                self.conv2 = nn.Conv2d(20, 20, 5)

            def forward(self, x):
                x = F.relu(self.conv1(x))
                return F.relu(self.conv2(x))

    Submodules assigned in this way will be registered, and will have their
    parameters converted too when you call :meth:`to`, etc.
    """

```

从中可以看出，所有自定义的网络模型均需继承Module类，并一般需要重写forward函数（用于实现神经网络的前向传播过程），而后模型即完成了注册，并拥有了相应的可训练参数等。

3.3 模型训练

仍然与经典机器学习模型的训练不同，深度学习模型由于其网络架构一般是自定义设计的，所以一般也不能简单的通过调用fit/predict的方式来实现简洁的模型训练/预测过程，而往往交由使用者自己去实现。

大体上，实现模型训练主要包含以下要素：

- 完成数据集的准备和模型定义
- 指定一个损失函数，用于评估当前模型在指定数据集上的表现
- 指定一个优化器，用于"指导"模型朝着预期方向前进
- 写一个循环调度，实现模型训练的迭代和进化

数据集的准备和模型定义部分就是前两小节所述内容；而损失函数，简单需求可以依据PyTorch提供的常用损失函数，而更为复杂和个性化的损失函数则继承Module类的方式来加以自定义实现；优化器部分则无太多“花样”可言，一般直接调用内置的优化器即可，例如Adam、SGD等等。

这些操作结合后续的实践案例一并介绍。

3.4 一个简单的深度学习案例

麻雀虽小五脏俱全，解剖一只麻雀，可有助于探悟内涵实质和基本规律。

有了前述小节的理论基础，就可以开始深度学习实践案例了，这里以sklearn中自带的手写数字分类作为目标来加以实践。

1. 首先给出应用sklearn中随机森林模型的实现方式和效果

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
X, y = load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y)
rf = RandomForestClassifier().fit(X_train, y_train)
rf.score(X_test, y_test)
# 输出: 0.9688888888888889
```

当然，该数据集分类的难度不大，即使在未经过调参的情况下也取得了很好的分类效果。

2. 基于PyTorch的深度学习模型训练实践，这里按照标准的深度学习训练流程，仍然使用上述手写数字分类数据集进行实验：

a. 构建Dataset类型数据集


```

import torch
from torch.utils.data import TensorDataset, DataLoader
X_train_tensor = torch.Tensor(X_train)
y_train_tensor = torch.Tensor(y_train).long() # 主要标签需要用整数形式，否则后续用于计算交叉熵损失时报错
dataset = TensorDataset(X_train_tensor, y_train_tensor) # 直接调用TensorDataset加以包裹使用
dataloader = DataLoader(dataset, batch_size=128, shuffle=True) # 每128个样本为一个batch，训练时设为随机
X_test_tensor = torch.Tensor(X_test) # 测试集只需转化为tensor即可
y_test_tensor = torch.Tensor(y_test).long()

```

b.自定义一个网络模型，仅使用Linear网络层

```

from torch import nn, optim
class Model(nn.Module): # 继承Module基类
    # 定义一个含有单隐藏层的全连接网络，其中输入64为手写数字数据集的特征数，输出10为类别数，隐藏层神经元数量设置32
    def __init__(self, n_input=64, n_hidden=32, n_output=10):
        super().__init__() # 使用全连接层和ReLU激活函数搭建网络模型
        self.dnn = nn.Sequential(
            nn.Linear(n_input, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_output)
        )

    def forward(self, x):
        # 重载forward函数，从输入到输出
        return self.dnn(x)

```

c.八股文式的深度学习训练流程

```

model = Model() # 初始化模型
criterion = nn.CrossEntropyLoss() # 选用交叉熵损失函数
optimizer = optim.Adam(model.parameters(), lr=0.001) # 选用Adam优化器，传入模型参数，设置学习率
for epoch in range(50): # 50个epoch
    for data, label in dataloader: # DataLoader是一个可迭代对象
        optimizer.zero_grad() # 待优化参数梯度清空
        prob = model(data) # 执行一次前向传播，计算预测结果
        loss = criterion(prob, label) # 评估模型损失
        loss.backward() # 损失反向传播，完成对待优化参数的梯度求解
        optimizer.step() # 参数更新
        if (epoch + 1) % 5 == 0: # 每隔5个epoch打印当前模型训练效果
            with torch.no_grad():
                train_prob = model(X_train_tensor)
                train_pred = train_prob.argmax(dim=1)
                acc_train = (train_pred==y_train_tensor).float().mean()
                test_prob = model(X_test_tensor)
                test_pred = test_prob.argmax(dim=1)
                acc_test = (test_pred==y_test_tensor).float().mean()
                print(f"epoch: {epoch}, train_accuracy: {acc_train}, test_accuracy: {acc_test} !")
    """ 输出
epoch: 4, train_accuracy: 0.8507795333862305, test_accuracy: 0.8577777743339539 !

```



```
epoch: 9, train_accuracy: 0.948775053024292, test_accuracy: 0.9200000166893005 !
epoch: 14, train_accuracy: 0.9717891812324524, test_accuracy: 0.9444444179534912
!
epoch: 19, train_accuracy: 0.9799554347991943, test_accuracy: 0.9577777981758118
!
epoch: 24, train_accuracy: 0.9866369962692261, test_accuracy: 0.9644444584846497
!
epoch: 29, train_accuracy: 0.9925761222839355, test_accuracy: 0.9644444584846497
!
epoch: 34, train_accuracy: 0.9925761222839355, test_accuracy: 0.9644444584846497
!
epoch: 39, train_accuracy: 0.9962880611419678, test_accuracy: 0.9666666388511658
!
epoch: 44, train_accuracy: 0.9970304369926453, test_accuracy: 0.9711111187934875
!
epoch: 49, train_accuracy: 0.9970304369926453, test_accuracy: 0.9711111187934875
!
""""
```

至此，就完成了深度学习模型训练的基本流程，从数据集准备到模型定义，直至最后的模型训练及输出。当然，由于该数据集分类任务比较简单，加之数据量不大，所以深度学习的优势并不明显。

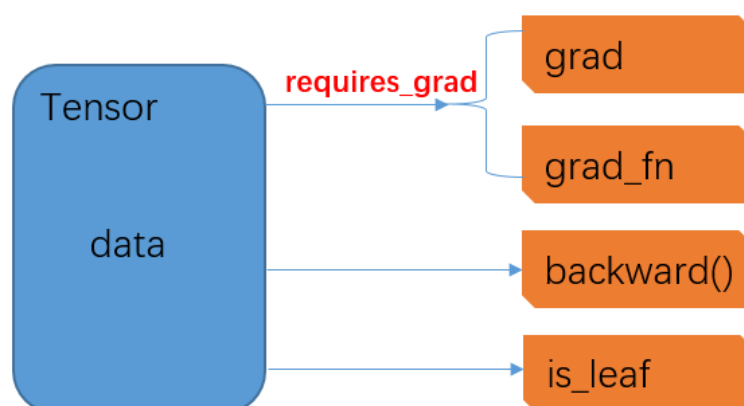
4. Tensor如何实现自动求导

讲解Tensor如何实现自动求导，本文分别从理论分析和代码实践的角度加以陈述：

- Tensor中的自动求导：与梯度相关的属性，前向传播和反向传播
- 自动求导探索实践：以线性回归为例，探索自动求导过程

4.1 Tensor中的自动求导分析

[Tensor](#)是PyTorch中的基础数据结构，构成了深度学习的基石，其本质上是一个高维数组。在前序推文中，实际上提到过在创建一个Tensor时可以指定其是否需要梯度。那么是否指定需要梯度(`requires_grad`)有什么区别呢？实际上，这个参数设置True/False将直接决定该Tensor是否支持自动求导并参与后续的梯度更新。具体来说，Tensor数据结构中，与梯度直接相关的几个重要属性间的关系如下：



原创拙图，权当意会

透过上面这个类图，大概想表达以下含义：

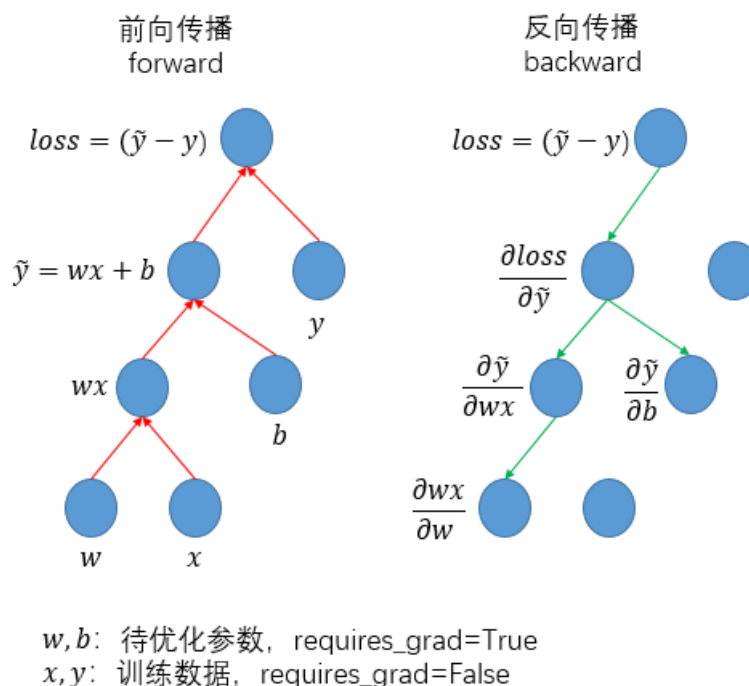
- 在一个Tensor数据结构中，最核心的属性是data，这里面存储了Tensor所代表的高维数组（当然，这里虽然称之为高维，但实际上可以从0维开始的任意维度）；
- 通过requires_grad参数控制两个属性，grad和grad_fn，其中前者代表当前Tensor的梯度，后者代表经过当前Tensor所需求导的梯度函数；当requires_grad=False时，grad和grad_fn都为None，且不会存在任何取值，而只有当requires_grad=True时，此时grad和grad_fn初始取值仍为None，但在后续反向传播中可以予以赋值更新
- backward()，是一个函数，仅适用于标量Tensor，即维度为0的Tensor
- is_leaf：标记了当前Tensor在所构建的计算图中的位置，其中计算图既可看做是一个有向无环图（DAG），也可视作是一个树结构。当Tensor是初始节点时，即为叶子节点，is_leaf=True，否则为False。

目前，Tensor支持自动求导功能对数据类型的要求是仅限于浮点型："As of now, we only support autograd for floating point `Tensor` types (half, float, double and bfloat16) and complex `Tensor` types (cfloat, cdouble)."——引自PyTorch官方文档

了解了Tensor所具有上述属性和方法，那么它是如何实现自动求导的呢？这就又要涉及到前向传播和反向传播这两个重要概念。简单来说，如果将神经网络的每层比作一系列函数映射（ f_1, f_2, \dots, f_n ）的话，那么：

- 前向传播，就是依据计算流程实现数据(data)的计算和计算图的构建：
- 反向传播，反向传播就是依据所构建计算图的反方向递归求导和赋值梯度：

而如果用图形化描述这一过程，则是：



其中，在前向传播过程中，是按照流程完成从初始输入（一般是训练数据+网络权重）直至最终输出（一般是损失函数）的计算过程，同步完成计算图的构建；而在反向传播过程中，则是通过调用 `loss.backward()` 函数，依据计算图的反方向递归完成各级求导（本质上就是求导的链式法则）。同时，对于 `requires_grad=False` 的tensor，在反向传播过程中实际不予以求导和更新，相应的反向链条被切断。

另外值得补充说明的是，在PyTorch早期版本中设计用于支持自动求导的数据类型为Variable，英文含义即为参数，特指网络中待优化的参数。其中，Variable与Tensor的关系是：Variable是对Tensor的二次封装，专门用于支持梯度求解。而在后来，Variable逐渐弃用（deprecated），并将其特有的自动求导功能与Tensor合并，并通过Tensor的requires_grad属性来区分一个Tensor是否支持求导。显然，这样的设计是更为合理的，对使用也更加方便统一。

Variable (deprecated)

• WARNING

The Variable API has been deprecated: Variables are no longer necessary to use autograd with tensors. Autograd automatically supports Tensors with `requires_grad` set to `True`. Below please find a quick guide on what has changed:

- `Variable(tensor)` and `Variable(tensor, requires_grad)` still work as expected, but they return Tensors instead of Variables.
- `var.data` is the same thing as `tensor.data`.
- Methods such as `var.backward()`, `var.detach()`, `var.register_hook()` now work on tensors with the same method names.

已进入历史舞台的Variable类型

4.2 Tensor中的自动求导实践

这里，我们以一个简单的单变量线性回归为例演示Tensor的自动求导过程。

1.创建训练数据x, y和初始权重w, b

```
# 训练数据，目标拟合线性回归  $y = 2 * x + 3$ 
x = torch.tensor([1., 2.])
y = torch.tensor([5., 7.])
# 初始权重, w=1.0, b=0.0
w = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(0.0, requires_grad=True)
```

此时查看w, b和x, y的梯度相关的各项属性，结果如下

```
# 1. 注意: x和y设置为requires_grad=False
x.grad, x.grad_fn, x.is_leaf, y.grad, y.grad_fn, y.is_leaf
# 输出: (None, None, True, None, None, True)
# 2. w和b初始梯度均为None，且二者均为叶子节点
w.grad, w.grad_fn, w.is_leaf, b.grad, b.grad_fn, b.is_leaf
# 输出: (None, None, True, None, None, True)
```

2.构建计算流程，实现前向传播

```
# 按计算流程逐步操作，实现前向传播
wx = w * x
wx_b = wx + b
loss = (wx_b - y)
loss2 = loss ** 2
loss2_sum = sum(loss2)
```

查看各中间变量的梯度相关属性：

```
# 1.查看是否叶子节点
wx.is_leaf, wx_b.is_leaf, loss.is_leaf, loss2.is_leaf, loss2_sum.is_leaf
# 输出: (False, False, False, False, False)

# 2.查看grad
wx.grad, wx_b.grad, loss.grad, loss2.grad, loss2_sum.grad
# 触发Warning
# UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is being
accessed. Its .grad attribute won't be populated during autograd.backward(). If
you indeed want the .grad field to be populated for a non-leaf Tensor, use
.retain_grad() on the non-leaf Tensor. If you access the non-leaf Tensor by
mistake, make sure you access the leaf Tensor instead. See
github.com/pytorch/pytorch/pull/30531 for more informations. (Triggered
internally at aten\src\ATen\core\TensorBody.h:417.) return self._grad
# 输出: (None, None, None, None, None)

# 3.查看grad_fn
wx.grad_fn, wx_b.grad_fn, loss.grad_fn, loss2.grad_fn, loss2_sum.grad_fn
# 输出: (<MulBackward0 at 0x23a875ee550>, <AddBackward0 at 0x23a875ee8e0>,
<SubBackward0 at 0x23a875ee4c0>, <PowBackward0 at 0x23a875ee490>, <AddBackward0
at 0x23a93dde040>)
```

3.对最终的loss调用backward，实现反向传播

```
loss2_sum.backward()
```

依次查看各中间变量和初始输入的梯度

```
# 1. 中间变量（非叶子节点）的梯度仅用于反向传播，但不对外暴露
wx.grad, wx_b.grad, loss.grad, loss2.grad, loss2_sum.grad
# 输出: (None, None, None, None, None)

# 2. 检查叶子节点是否获得梯度：w, b均获得梯度，x, y不支持求导，仍为None
w.grad, b.grad, x.grad, y.grad
# 输出: (tensor(-28.), tensor(-18.), None, None)
```

至此，即通过前向传播的计算图和反向传播的梯度传递，完成了初始权重参数的梯度赋值过程。注意，这里w和b是网络待优化参数，而一旦二者有了梯度，则可进一步应用梯度下降法予以更新。

那么进一步地，这里w.grad和b.grad的数值是如何得到的呢？我们实际手动求解一遍。首先分别推导loss对w和b的偏导公式：

而后，带入两组训练数据(x, y)=(1, 5)和(x, y)=(2, 7)，并将两组训练数据对应的梯度求和：

```
# w的梯度:  $2 * (1 * 1 + 0 - 5) * 1 + 2 * (1 * 2 + 0 - 7) * 2 = -28$ 
# b的梯度:  $2 * (1 * 1 + 0 - 5) + 2 * (1 * 2 + 0 - 7) = -18$ 
```

显然，手动计算结果与上述演示结果是一致的。

注意：在多个训练数据(batch_size)参与一次反向传播时，返回的参数梯度是在各训练数据上的求得的梯度之和。

`backward`

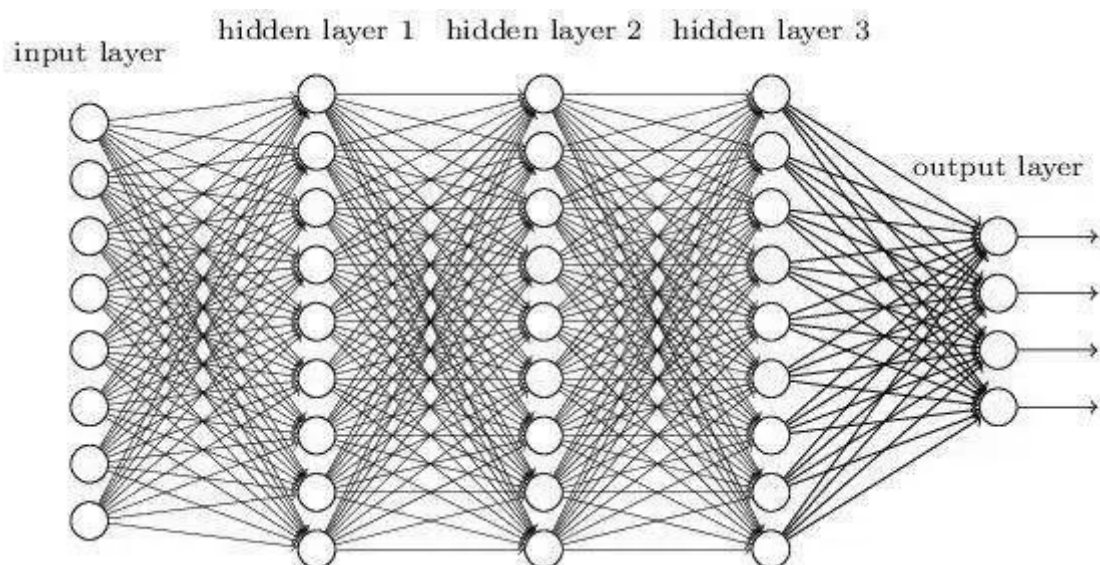
Computes the sum of gradients of given tensors with respect to graph leaves.

`grad`

Computes and returns the sum of gradients of outputs with respect to the inputs.

摘自PyTorch官网

5. 神经网络【DNN】



神经网络就是所谓的深度学习吗？

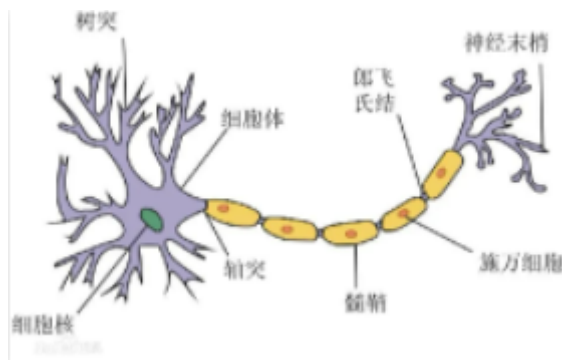
神经网络，英文Deep Neural Network，简写DNN，是研究最早也最为常用的神经网络模块，其本质上是一个多层感知机（Multi-Layer Perceptron，MLP）结构，只不过当层数较少时叫做MLP，层数更多时则叫DNN，但其实质是一致的。此外，MLP可算是传统机器学习模型的范畴，而DNN则归属于深度学习领域，所以从某种角度讲DNN也可算是传统机器学习和深度学习的结合部。

那么，除了以上源于对MLP的认知，DNN网络还有哪些形态？它为什么有效？它有什么特点或者适用场景？为了回答这些疑问，本文从以下几个方面加以介绍：

- 什么是DNN
- DNN为何有效
- DNN的适用场景
- 在PyTorch中的使用

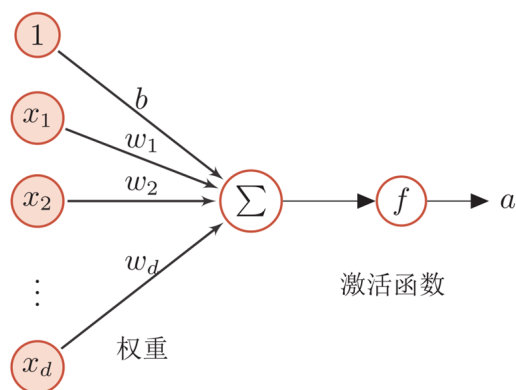
5.1 什么是DNN

DNN叫做深度神经网络，顾名思义，其包含两层含义：其一它是一个神经网络；其二它是一个层数较深的神经网络。而为了解释神经网络，那么就不得不先从生物神经元讲起（可能这也是所有介绍神经网络时，都会引用的一个例子）：



生物神经元示意图

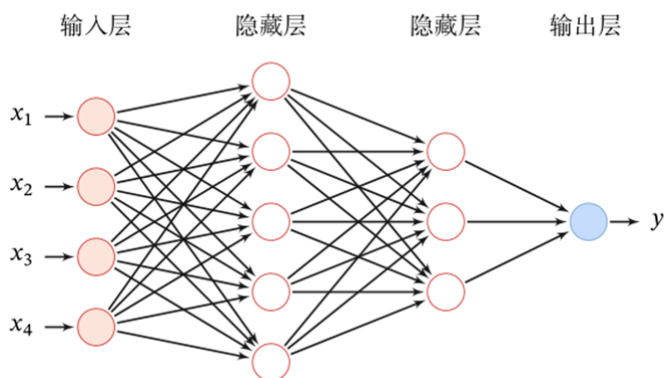
其实刚看到这张图时，我也很好奇伟大的先知者是怎样的脑洞大开受其启发而提出了神经网络——直到我们将这个神经元抽象为以下逻辑连接图：



神经网络中的一个神经元

也就是说，生物神经元虽然结构看上去错综复杂，但其实无外乎是多个像树枝一样的分支（这里的分支就是树突）汇聚到一个节点（也叫轴突），而后轴突根据所有分支汇入的能量之和大小来判断是否给上游一个刺激信号（这个根据能量大小决定输出的过程就是激活函数）。

有了单个神经元作为基础单元，很快就可以将其量产形成一个网络，这个由大量的神经元构成的网络就叫做神经网络。一个简单的神经网络示意图如下：



一个具有3层的DNN网络架构

当然，如果仔细对比这个3层的DNN和前述的单个神经元，那么严谨的说，这个DNN网络结构中省略了激活函数。

以上介绍了神经网络是怎么提出的，相当于回答了什么是神经网络的问题。那么进一步的，什么是深度神经网络呢？这个问题其实不难理解，但却没有统一的标准答案。说其不难理解，是因为深度神经网络无外乎就是层数比较深，例如前面的是一个3层的DNN网络，可以很容易将其拓展为4层、5层乃至更多层，但究竟多少层开始算深度，其实是没有确切答案的。一般而言，当网络层数 ≥ 3 时，就可以称之为深度神经网络了。

这里，还有几个细节值得注意：

- 神经网络的层数怎么算？以上述的网络为例，输入层就叫做输入层，不算做一层（如果习惯于计算机世界的从0开始，将其称之为第0层也可以），而后从第二列开始（也就是第一个隐藏层）开始算作网络的第一层，直至输出层（输出层也算作一层），所以上述的示意网络是一个3层的DNN模型；
- 相邻层之间的所有神经元均具有连接关系，即这是一个全连接结构。实际上，输入层的节点代表一个特征，之后的隐藏层和输出层的每个神经元代表一个信息提取结果，由于无法断言前一层的哪个节点对于后一层有作用或者没作用，所以最简单有效的办法就是相邻层的任意两个节点之间均建立连接。这在深度学习框架中就叫做网络权重（weight），是一个可训练的参数，网络的权重矩阵直接代表了模型；
- 相邻层之间的连接必须搭配适当的激活函数，否则增加层数无意义。个人以为，激活函数的提出可谓是深度学习中的救世主，虽然只是简单的提供了非线性关系，但却大大增强了网络承载信息的能力；换言之，如果不设置激活函数，那么任意多层的神经元线性组合的结果其实等价于单层的线性组合，结果就是增加层数是无意义的。

那么，为什么增加了激活函数就能保证神经网络的承载信息能力？什么又是激活函数？这就要引出下一话题：DNN为何有效？

5.2 DNN为何有效

DNN为何有效？其实提出神经网络的先知们也思考过这个问题，最终得出的答案是——通用近似定理。

定理 4.1 – 通用近似定理（Universal Approximation Theorem）

[Cybenko, 1989, Hornik et al., 1989]: 令 $\varphi(\cdot)$ 是一个非常数、有界、单调递增的连续函数， \mathcal{I}_d 是一个 d 维的单位超立方体 $[0, 1]^d$ ， $C(\mathcal{I}_d)$ 是定义在 \mathcal{I}_d 上的连续函数集合。对于任何一个函数 $f \in C(\mathcal{I}_d)$ ，存在一个整数 m ，和一组实数 $v_i, b_i \in \mathbb{R}$ 以及实数向量 $\mathbf{w}_i \in \mathbb{R}^d$ ， $i = 1, \dots, m$ ，以至于我们可以定义函数

$$F(\mathbf{x}) = \sum_{i=1}^m v_i \varphi(\mathbf{w}_i^T \mathbf{x} + b_i), \quad (4.33)$$

作为函数 f 的近似实现，即

$$|F(\mathbf{x}) - f(\mathbf{x})| < \epsilon, \forall \mathbf{x} \in \mathcal{I}_d. \quad (4.34)$$

其中 $\epsilon > 0$ 是一个很小的正数。

摘自《神经网络与深度学习》-邱锡鹏

不过，看了这一段极为绕口的理论之后，似乎竟不能理解这是要表达什么含义。简言之，通用近似定理阐述的内涵是：通过增加神经网络深度+激活函数（体现为上述定理中非常数、单调、有界的函数 $\varphi()$ ）带来的非线性效果，可以逼近任意函数（这里的逼近体现为上述的 ϵ ）。

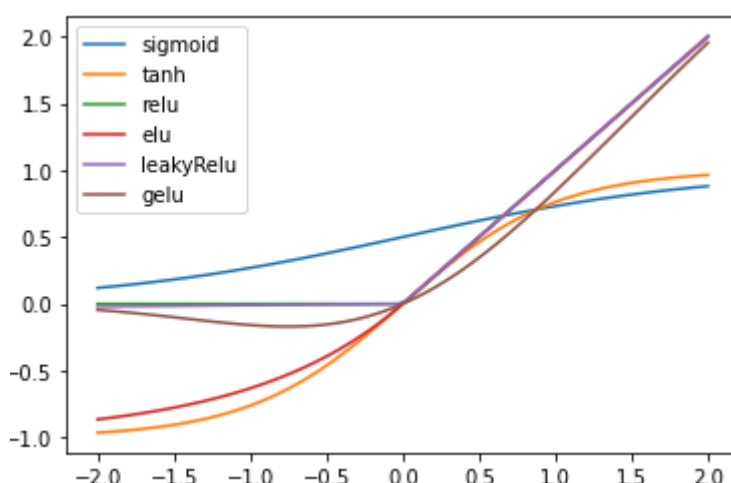
换言之：从机器学习的视角理解，模型经过在训练集上的学习过程，无非是**拟合了一个由输入到输出的映射函数**，只不过这个映射函数往往没有显式表达式，很可能是一个极为复杂且不连续的多段函数，那么经过足够层数的神经网络+激活函数之后，可以逼近这个映射。

关于这里的通用近似定理的介绍，参考书籍（邱锡鹏教授的《神经网络与深度学习》）中有一段非常贴切的描述，通过几次非线性的函数嵌套逼近了一个多段函数，以此来论证深度神经网络可以通用近似任意函数。

显然，也正如前面第一小节里提到的那样，这里之所以能够通用近似为任意函数，激活函数发挥了至关重要的作用。所以这里也简单的介绍一下几个常用的激活函数：

- sigmoid：最早使用和常用的激活函数之一
- tanh：sigmoid激活函数的缩放版， $\tanh(x) = 2(\text{sigmoid}(2x)-1)$ ，具有零点对称性和相对更好的梯度
- relu：最为常用的激活函数之一，具有单侧抑制特性，计算简单，梯度固定，可一定程度抑制梯度消失和梯度弥散
- elu：relu的改进版，在零点时梯度连续
- leakyRelu：relu的改进版，在负半轴梯度不再为0
- gelu：近年来论文中新提出的激活函数，不再具有单调性质。。

总的来说，激活函数的种类比较多，除了个别魔改的网络外，大体上用sigmoid、tanh和relu三者中的一种或几种组合是足够的，这也是最为常用的形式。



几种激活函数的曲线特性对比

小结一下，这里介绍DNN为什么是一个有效的神经网络模型，也就是why it works的问题。简言之：通过在层与层神经元的连接之间加入具有非线性特性的激活函数，增加神经元的层数，可以实现逼近任意函数的性质。这也是深度学习中的一个具有奠基地位的理论——通用近似定理。

5.3 DNN适用场景

DNN作为深度学习网络的早期代表，曾经在一段时间之内扮演着重要角色。但若论其适用的场景，其实答案是没有特别适用的场景：似乎其适用于任何深度学习场景，但又没有特殊专长的场景。相较于专长于图像视觉的卷积神经网络（CNN）和专长于序列建模的循环神经网络（RNN）来说，如果定要给DNN定下个适用场景，那么我个人觉得可以这样描述：

DNN适用于输入数据之间**具有同质的特性且又不体现特殊依赖**的场景，其中：数据同质的意义在于强调各输入数据间的含义和作用是一样的，而并非像传统机器学习中每列特征代表不同的含义；而不体现特殊依赖在于表明DNN不足以提取出输入数据间可能具有的方位组合或者顺序依赖信息——这是CNN和RNN分别擅长的场景。

当然，随着深度学习理论的迅猛发展，单纯的使用DNN进行网络建模的场景越来越少，更多的情况是与其他网络结构组合使用，例如在图像分类中，搭配若干个CNN单元之后配备一层或几层全连接单元用于最后的分类；在序列建模中的最后一层往往也是全连接单元。这或许才是DNN的真正价值和灵魂之所在吧！

5.4 在PyTorch中的使用

DNN作为深度学习中几乎是最常用的网络单元，在PyTorch中具有很好的封装结构。实际上，每个全连接层其实都在做一个线性变换，例如输入数据用矩阵 X 表示（ X 的维度为 $N \times D1$ ， N 为样本数量， $D1$ 为特征数量），下一层有 $D2$ 个神经元，那么描述这一变换只需要用一个矩阵乘法即可： $Y = X * W^T + b$ ，其中 W 为权重矩阵，维度为 $D2 \times D1$ ， b 为偏置向量，维度为 $D2$ 。

描述这一过程，在PyTorch中的模块即为`nn.Linear()`，其说明文档为：

```
nn.Linear?

Init signature:
nn.Linear(
    in_features: int,
    out_features: int,
    bias: bool = True,
    device=None,
    dtype=None,
) -> None
Docstring:
Applies a linear transformation to the incoming data:  $y = xA^T + b$ 

This module supports :ref:`TensorFloat32<tf32_on_ampere>`.

Args:
    in_features: size of each input sample
    out_features: size of each output sample
    bias: If set to False, the layer will not learn an additive bias.
        Default: True

Shape:
- Input:  $(*, H_{in})$  where  $*$  means any number of
  dimensions including none and  $H_{in} = \text{in\_features}$ .
- Output:  $(*, H_{out})$  where all but the last dimension
  are the same shape as the input and  $H_{out} = \text{out\_features}$ .

Attributes:
    weight: the learnable weights of the module of shape
       $(\text{out\_features}, \text{in\_features})$ . The values are
      initialized from  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ , where
       $k = \frac{1}{\text{in\_features}}$ 
    bias: the learnable bias of the module of shape  $(\text{out\_features})$ .
      If attr: 'bias' is True, the values are initialized from
       $\mathcal{U}(-\sqrt{k}, \sqrt{k})$  where
       $k = \frac{1}{\text{in\_features}}$ 
```

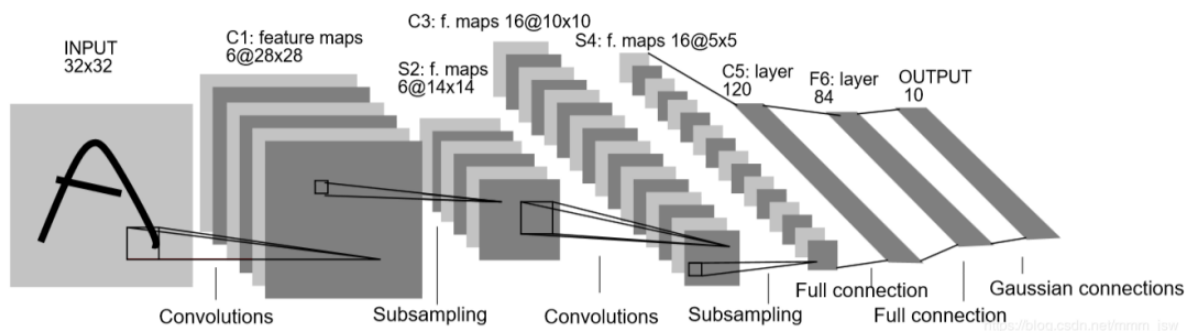
文档交代得也比较清晰了：

- 类的初始化参数：`in_features`、`out_features`分别表示全连接前后的神经元数量，`bias`表示是否拟合偏置项；
- 类的输入输出形状，输入数据维度为 $(in_features)$ ，输出数据维度为 $(out_features)$ ，即保持前序的维度不变，仅将最后一个维度由`in_features`维度变换为`out_features`；
- 类的属性：`weight`，拟合的权重矩阵，维度为 $(out_features, in_features)$ ；`bias`，拟合的偏置向量。

ok，上述就是关于深度学习中最为基础和常用的网络模型——DNN的介绍，虽然现在已经很少单纯使用这种类型的网络架构，但却仍然是众多高阶复杂模型中不可缺少的一部分；甚至说，理解了DNN的工作原理，对于后续理解其他网络结构也是很有必要的。

参考资料：《神经网络与深度学习》邱锡鹏 著。

6. 卷积神经网络【CNN】



LeNet5——CNN的开山之作

前篇介绍了DNN网络，理论上通过增加网络层数可以逼近任意复杂的函数，即通用近似定理。但在实践过程中，增加网络层数也带来了两个问题：其一是层数较深的网络容易可能存在梯度消失或梯度弥散问题，其二是网络层数的增加也带来了过多的权重参数，对训练数据集和算力资源都带来了更大的考验。与此同时，针对图像这类特殊的训练数据，应用DNN时需要将其具有二维矩阵结构的像素点数据拉平成一维向量，而后方可作为DNN的模型输入——这一过程实际上丢失了图片像素点数据的方位信息，所以针对图像数据应用DNN也不见得是最优解。

在这样的研究背景下，卷积神经网络应运而生，并开启了深度学习的新篇章。延续前文的行文思路，本篇从以下几个方面展开介绍：

- 什么是CNN
- CNN为何有效
- CNN的适用场景
- 在PyTorch中的使用

6.1 什么是CNN

卷积神经网络，应为Convolutional Neural Network，简称CNN，一句话来说就是应用了卷积滤波器和池化层两类模块的神经网络。显然，这里表达的重点在于CNN网络的典型网络模块是卷积滤波器和池化层。所以，这里有必要首先介绍这两类模块。

1.卷积滤波器

作为一名通信专业毕业人士，我对卷积一词并不陌生，最初在信号处理的课中就有所接触。当然，卷积操作本身应该是一个数学层面的操作，对两个函数 $f(x)$ 和 $g(x)$ 做卷积，其实就是求解以下积分：

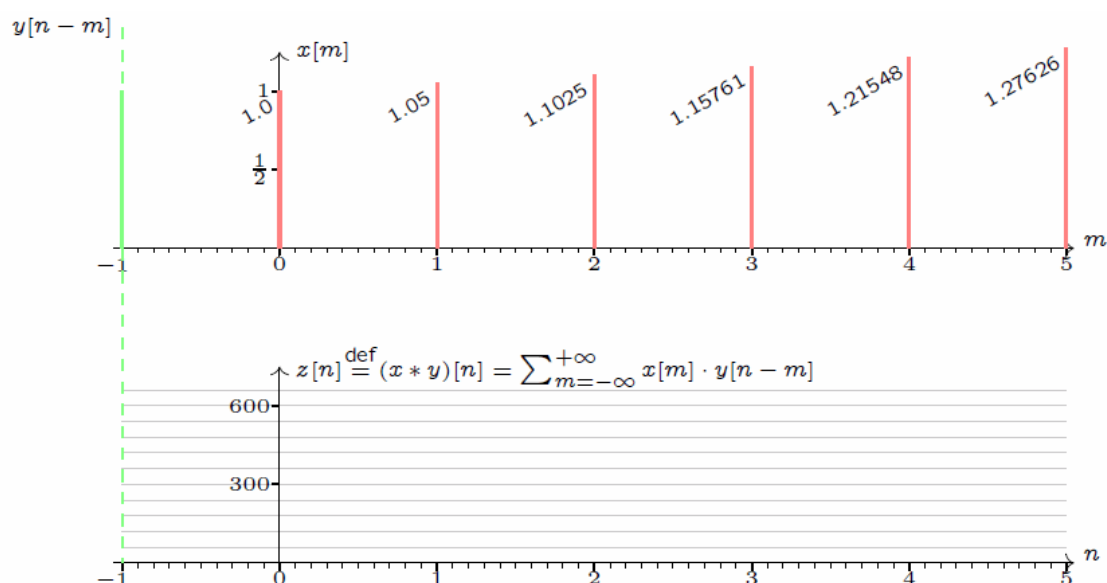
$$f(x) * g(x) = \int_{-\infty}^{\infty} f(\tau)g(x - \tau)d\tau$$

这个卷积的数学表达形式很优美，但其实有些过于抽象。从计算机的角度理解，一个连续的函数积分是不便操作的，因为计算机只能接受离散的输入，所以上述卷积操作的离散表达形式便是：

$$f * g = \sum_{-\infty}^{\infty} f(i)g(x - i)$$

当然了，上述形式也只是从连续的积分形式变成了离散的求和形式。其中一个值得关注的细节是这里的卷积操作符号用的" $*$ "——这也是后续所有卷积神经网络中沿用的卷积操作符号。

那么，这个卷积操作如何理解呢？这里引用网络上的一张信号处理卷积的动图：



补充说明：在上述卷积操作中，两个函数 $x()$ 和 $y()$ 均为单位脉冲函数，即有值的时刻均取值为1。而至于说这个卷积有什么功能和优势，其实在通信处理中其最大的价值在于用于时域和频域的变换——时域卷积等于频域卷积相乘，用公式表达就是 $\text{FFT}(f * g) = \text{FFT}(f) \times \text{FFT}(g)$ ，这里FFT表达信号处理领域常用的操作：快速傅里叶变换。换句话说，卷积和乘法构成了两个信号在时频域的交换操作。

ok，了解了卷积操作的功能和其用途之后，我们来看其在神经网络中能有什么应用，或者进一步说对于图像分类任务的神经网络有什么应用。

注意到，卷积操作适用于两个函数（连续积分形式），或者说两串序列数据（离散求和形式）；更进一步地，即可通过交错形式逐一得到二者对位相乘的求和，并得到一个新的序列结果。也就是说，两个序列卷积的结果是一个新的序列。将这对应到用于图像分类的神经网络中，有两个问题：

- 卷积操作的两个对象（或者说两串序列）分别是什么呢？一个应该是图像的像素数据，而另一个则是网络权重，也就是说卷积操作中进行滑动相乘求和的对象分别是图像像素数据和网络权重
- 卷积是两个一维序列在卷，那应用到图像数据呢？难道还是要将其展平为一维序列吗？——这又回到了DNN中丢失了空间依赖信息的问题。所以，这里卷积操作的范围又进一步由一维序列延伸为二维矩阵——很小的一处改动，但却是CNN的灵魂之处。

除了上述两个问题，其实还隐藏一个细节：卷积之所以用“卷”这个词，大概是因为卷积操作的两个序列是反向滑动的——一个向左，一个向右。但无论以什么方向滑动，但对于像素数据和网络权重来说，卷积其本质是将二者对位相乘求和。那么，正向对位是对位，反面对位也是对位，为何还要卷一下呢——直接正向对位不足够吗？当然是可以的，所以神经网络中的卷积都是直接对位相乘求和。

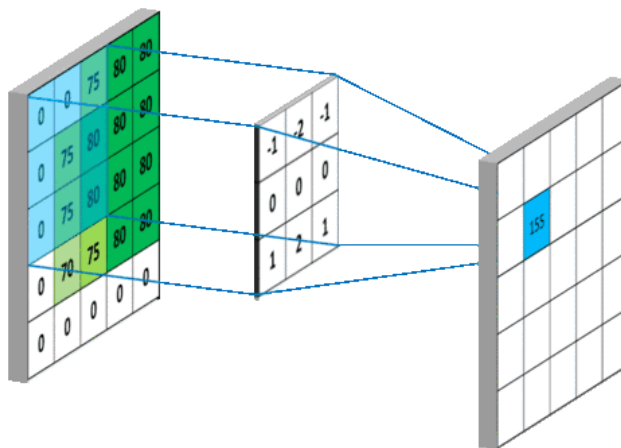
经过这样设计的卷积操作已基本实现了从数学中卷积到神经网络中卷积的衍变，但还有最后一处调整：数学中的卷积操作是输入两个序列，得到一个新的序列，同时这两个序列可以长度不同，如果两个序列长度分别记作M和N的话，那么卷积得到的新序列长度为M+N-1。但在神经网络中，似乎这种维度不可控的操作不够友好，所以就要求两个卷积对象尺寸一致，并只保留了对位相乘求和的一个结果作为卷积输出。具体来说，单个卷积完成的是以下操作：

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix} = -1$$

至此，算是真正完成了神经网络中单个卷积操作的讲解，小结一下，可概括为以下要点：

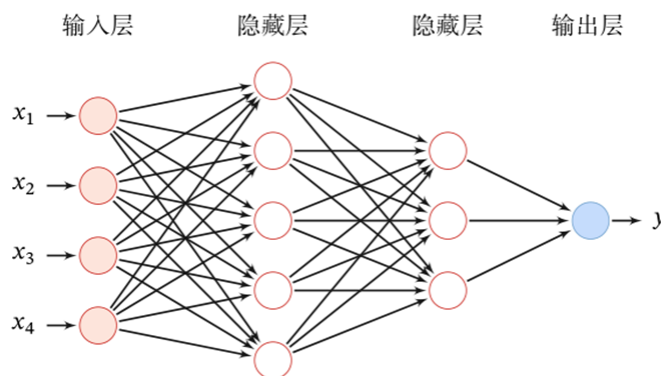
- 神经网络中的卷积操作源起于数学中的卷积，但取消了反向滑动的特点，而仅采用正向对位相乘的特性——从这个角度讲，**神经网络中的卷积叫做加权求和更贴切**
- 神经网络中卷积操作的两个对象是像素数据和网络权重，其中这里的网络权重也叫做一个卷积核（kernel），且要求二者尺寸相同
- 神经网络中的单词卷积操作只保留一个输出

类似于DNN网络中的神经元结构，在CNN网络中上述单个卷积核的操作应该叫做一个神经元。那么，有了单个神经元，就可以很容易的通过滑动的形式将其推广到整张图像：整张的含义既包括横向和纵向，也包括多个通道，例如彩色图片的RGB。所以，在一幅图像上做卷积操作，就是如下过程：



注：一组卷积模板组成的矩阵称作卷积核，一个卷积核仅作用于单个输入通道上，若前一层有M个通道，后一层输出N个通道，则需要M×N个卷积核。除原始图片输入数据外，后续经过卷积层提取的每个通道都叫做一个特征图。

这里再次贴出DNN的网络架构，方便我们对比：



对比DNN和CNN两种网络，可以窥探更深层的对比：

- CNN的网络结构体现的也是相邻网络层之间的连接关系，但这种连接仅考虑了小范围的输入，即局部连接而非全连接
- 与DNN中各神经元拥有不同的连接权重相比，CNN中的连接权重只有一套公共的模板，即权重共享

局部连接、权重共享，这是CNN的两大特性，也正是这两大特性，一方面大大降低了权重参数的数量，另一方面也更容易提取图像数据的局部特征！

2.池化层

池化层，英文为pooling，其实单纯从其英文是很难理解为何要在卷积神经网络中设计一个这样的结构。虽然目前我个人未能理解这个名字的含义，但其功能却是非常直观和简单的——如果说卷积滤波器是用于局部特征提取的话，那么池化层可以看做是局部特征降维。

举个例子，池化层典型的有三种类型，即MaxPooling, AvgPooling, MinPooling, SumPooling等，其中前两种更为常用。那么MaxPooling要干的事情就是将局部的一组像素求其最大值作为输出，相应的AvgPooling和MinPooling则是求均值或者最小值，SumPooling就是求和。举个例子：

Feature Map				Max Pooling	Average Pooling	Sum Pooling
6	6	6	6			
4	5	5	4			
2	4	4	2			
2	4	4	2			

池化层的功能是很容易理解的，那么设计的目的是什么呢？答案有两个：

- 数据降维，即将大尺寸图像数据变为小尺寸
- 特殊特征提取操作，即池化层其实也可看做是一种特殊的特征提取器（当然，与卷积核的滑动特征提取还是有显著的功能差异的）

至此，有了卷积层和池化层这两大模块的理解，即可用其堆叠出想要的卷积神经网络，例如在开篇给出的CNN开山之作——LeNet5中，则是一个含有两个卷积层、两个池化层和三个全连接层组成的网络。

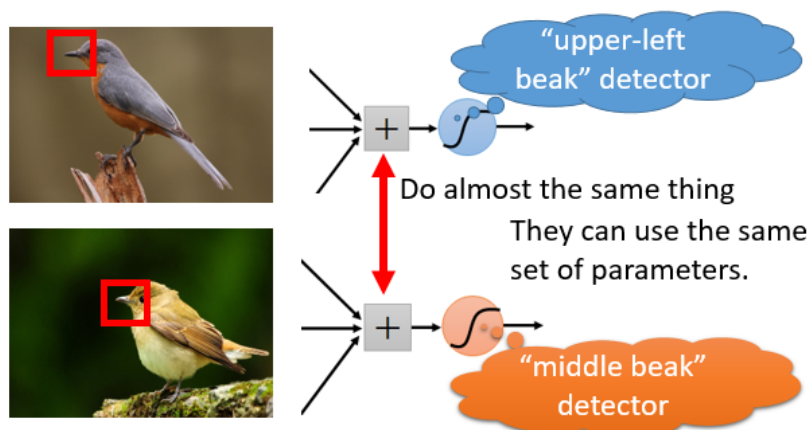
上面从数学中的卷积操作开始，介绍了卷积神经网络中的卷积是如何设计的。实际上，当理解了卷积的编码实现之后，会发现其实卷积的计算还是非常简单的，一句话概括就是——**卷积操作就是用卷积核中的权重矩阵通过滑窗的形式依次与图像像素数据进行相乘求和的过程。**

那么问题来了，为什么这样的设计是有效的？换言之，原始的图像数据经过卷积操作之后提取到了哪些特征？这就是接下来要介绍的内容。

6.2 CNN为何有效

CNN为何有效，回答这一问题的核心在于解释卷积操作为何有效，因为CNN网络中的标志性操作是卷积。

为了理解卷积操作是如何工作的，这里先给出一段形象的描述，然后再以LeNet5为例加以探索实践。



在上述图片中，我们可以看到同样是检测鸟嘴(beak)的局部特征，通过选用相同的卷积核与其滤波，通过多次变换可以用以分辨其是一个尖嘴还是短嘴，从而为最终鸟的分类任务提供一个特征。

当然，这只是一个示意描述，那么实际情况如何呢？我们选用LeNet5对手写数字分类任务加以尝试，看看模型是怎么利用这一卷积操作。

首先是mnist数据集的准备，可直接使用torchvision包在线下载：

```
from torchvision import datasets
from torch.utils.data import DataLoader, TensorDataset

train = datasets.MNIST('data/', download=True, train=True) # 在线下载
test = datasets.MNIST('data/', download=True, train=False)

X_train = train.data.unsqueeze(1)/255.0
y_train = train.targets
trainloader = DataLoader(TensorDataset(X_train, y_train), batch_size=256,
                          shuffle=True)

X_test = test.data.unsqueeze(1)/255.0
y_test = test.targets
```

然后是LeNet5的网络模型（torchvision中内置了部分经典模型，但LeNet5由于比较简单，不在其中）

```
import torch
from torch import nn

class LeNet5(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 5, padding=2)
        self.pool1 = nn.MaxPool2d((2, 2))
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.pool2 = nn.MaxPool2d((2, 2))
        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 = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = x.view(len(x), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

最后是模型的训练过程：

```
model = LeNet5()
optimizer = optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss()

for epoch in range(10):
```



```

for X, y in trainloader:
    pred = model(X)
    loss = criterion(pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    with torch.no_grad():
        y_pred = model(X_train)
        acc_train = (y_pred.argmax(dim=1) == y_train).float().mean().item()

        y_pred = model(X_test)
        acc_test = (y_pred.argmax(dim=1) == y_test).float().mean().item()
    print(epoch, acc_train, acc_test)

### 训练结果 ###
...
10%|██████████|
      | 1/10 [00:28<04:12, 28.05s/it] 0 0.9379666447639465 0.9406999945640564
20%|██████████|
      |
2/10 [00:56<03:48, 28.54s/it] 1 0.9663333296775818 0.9685999751091003
30%|██████████|
      |
3/10 [01:25<03:21, 28.76s/it] 2 0.975350022315979 0.9771000146865845
40%|██████████|
      |
4/10 [01:54<02:52, 28.78s/it] 3 0.9786166548728943 0.9787999987602234
50%|██████████|
      |
5/10 [02:22<02:22, 28.43s/it] 4 0.9850000143051147 0.9853000044822693
60%|██████████|
      |
6/10 [02:51<01:53, 28.49s/it] 5 0.9855666756629944 0.9843999743461609
70%|██████████|
      |
7/10 [03:20<01:26, 28.78s/it] 6 0.9882833361625671 0.9873999953269958
80%|██████████|
      |
8/10 [03:51<00:58, 29.37s/it] 7 0.9877333045005798 0.9872000217437744
90%|██████████|
      |
9/10 [04:21<00:29, 29.54s/it] 8 0.9905833601951599 0.9896000027656555
100%|██████████|
██████████|
██████|
10/10 [04:49<00:00, 28.93s/it] 9 0.9918666481971741 0.9886000156402588
...

```

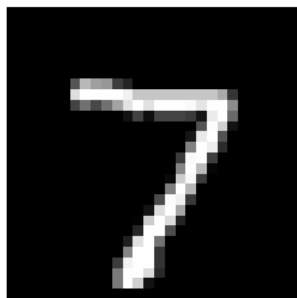
可见，短短通过10个epoch的训练，模型在训练集和测试集上均取得了很好的准确率得分，其中训练集高达99%以上，测试集上也接近99%，说明模型不存在过拟合。

那么接下来我们的重点来了：经过LeNet5模型中的两个卷积层操作之后，原本的手写数字图片变成了什么形态？换句话说，提取到了哪些特征？

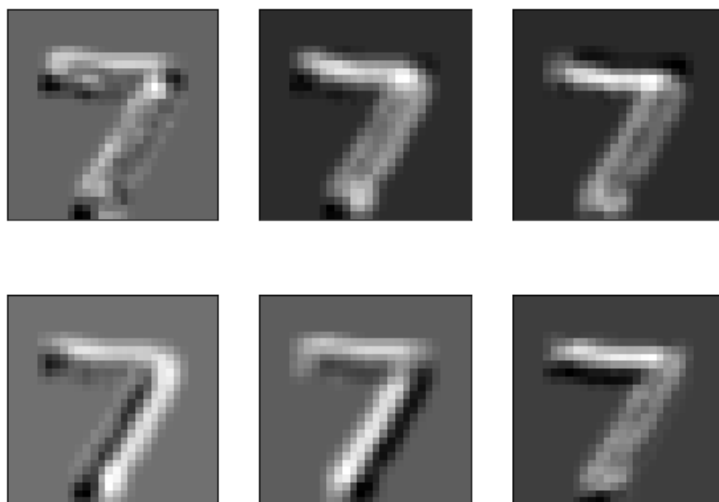
我们这里举两个特殊case研究一下：

case-1：测试集样本0，对应手写数字7

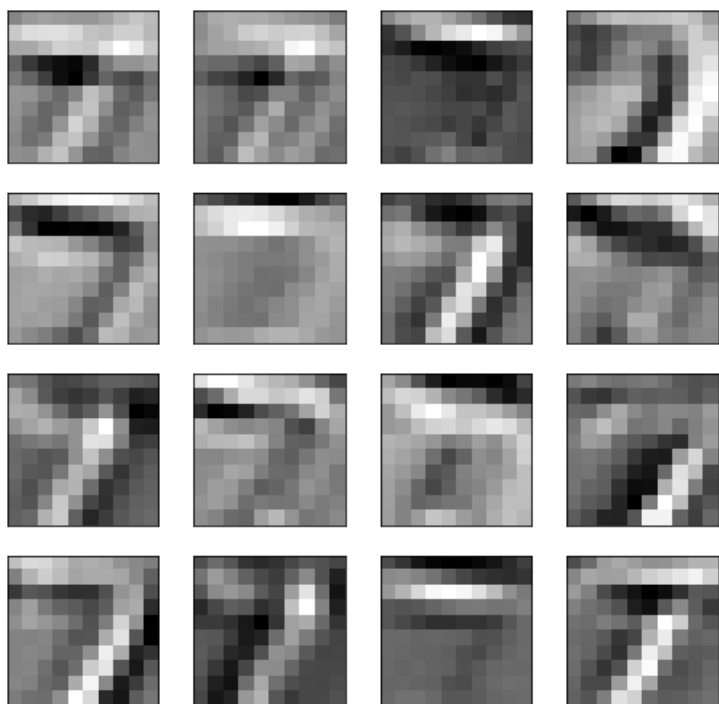
1-a: 原始图片:



1-b: 经过第一层卷积, 共提取6个通道的特征图



1-c: 经过第二层卷积, 提取16个通道的特征图

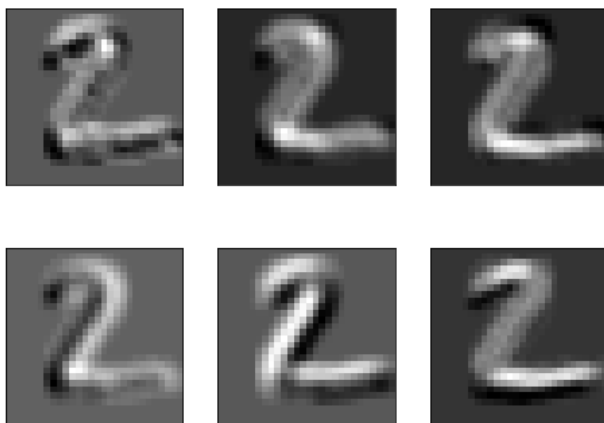


case-2: 测试集样本1, 对应手写数字2

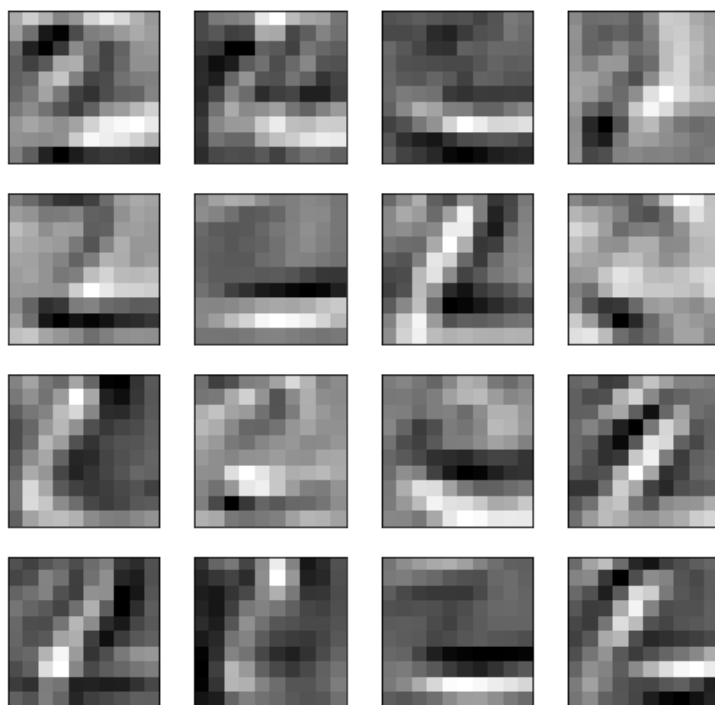
2-a: 原始图片



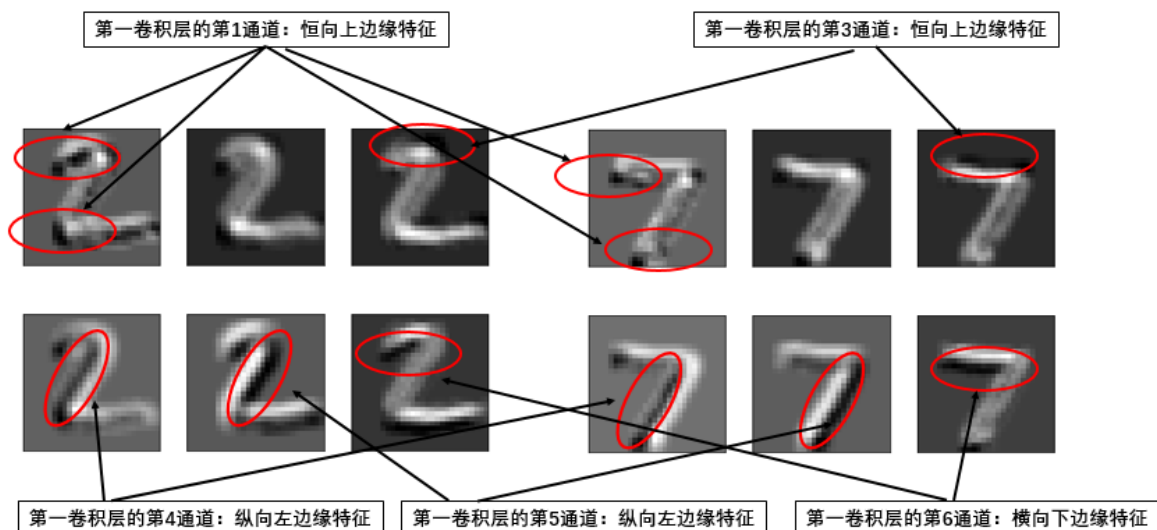
2-b: 经过第一层卷积，共提取6个通道的特征图



2-c: 经过第二层卷积，提取16个通道的特征图



对比两组案例的具体卷积提取结果，其实是能大体看出一些规律的，例如第一层卷积后的第1和第2通道更加注重提取手写数字的边角特征（轮廓），而第4和第5通道特征图注重手写数字的纵向特征，且分别提取左边缘和右边缘的特征；类似地，第3和第6通道特征图则注重提取手写数字的横向特征，且分别提取上边缘和下边缘特征：



而在此之后的第二卷积层中，则用于提取更为细节和丰富的特征，具体可以自行对比研究一下。至于说为什么提取了这些局部特征就可以完成手写数字的识别——即区分哪个是0，哪个是1等等？这里可以联想一下数字电路中逻辑判断的例子：对于由7个笔画组成的数字模板，当外圈全亮而中间不亮时为0，当右侧两个亮而其他不亮时为1。而现在LeNet5通过各个卷积核提取到的特征，就可根据取值大小对应到图片中各部分的亮暗情况，进而完成数字的分类。



当然，这个例子只是简单的举例，模型的实际处理逻辑会比这复杂得多——但全靠模型自己去训练和学习。

以上，我们首先通过识别鸟嘴的直观例子描述了卷积操作在CNN网络中扮演的角色——提取局部特征，而后用LeNet5模型在mnist手写数字数据集上的实际案例加以研究分析，证实了这一直观理解。所以，CNN模型之所以有效，其核心在于——卷积操作具有提取图像数据局部特征的能力。

此外，卷积操作配合池化层，其实还有更鲁棒的效果：包括图像伸缩不变性、旋转不变性等，这是普通的DNN所不具备的能力，此处不再展开。

6.3 CNN的适用场景

前面一直在以图像数据为例介绍CNN的原理和应用，当然图像数据也确实是CNN网络最为擅长的场景，反之亦然，即最擅长图像数据的网络结构是CNN。

除了图像数据，随着近年来研究的进展，CNN其实在更多的领域都有所突破和崭露头角，例如：

- 将一维卷积应用于序列数据建模，也可以提取相邻序列数据间的特征关系，从而很好的完成时序数据建模，例如TCN模型【参考文献：Temporal convolutional networks: A unified approach to action segmentation. 2016】
- 将二维卷积应用于空间数据建模，例如交通流量预测中，一个路口的流量往往与其周边路口的流量大小密切相关，此时卷积也是有效的

总而言之，以卷积和池化操作为核心的CNN网络，最为适用的场景是图像数据，也可推广到其他需要提取局部特征的场景。

6.4 在PyTorch中的使用

最后，简单介绍一下CNN网络中的两个关键单元：卷积模块和池化模块，在PyTorch中的基本使用。

1.卷积模块：Conv1d、Conv2d, Conv3d

PyTorch中卷积模块主要包括3个，即分别为1维卷积Conv1d、2维卷积Conv2d和3维卷积Conv3d，其中Conv2d即是最常用于图像数据的二维卷积，也是最早出现的模块；Conv1d则可用于时序数据中的卷积，而Conv3d目前个人还未接触到。这里以Conv2d为例展开介绍一下。

首先是类的初始化参数：

```
nn.Conv2d?
```

Init signature:

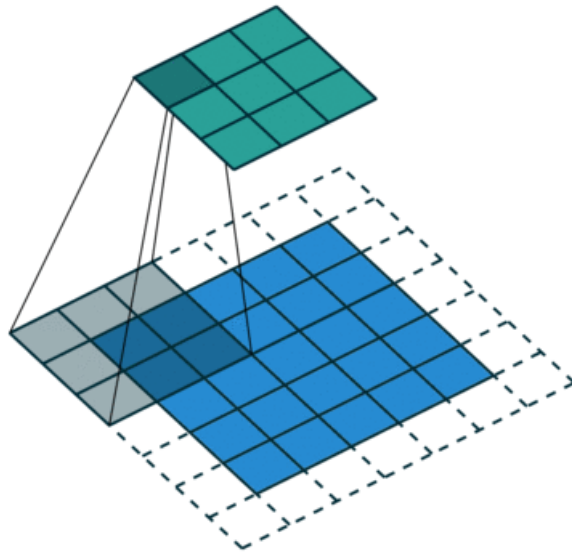
```
nn.Conv2d(
    in_channels: int,
    out_channels: int,
    kernel_size: Union[int, Tuple[int, int]],
    stride: Union[int, Tuple[int, int]] = 1,
    padding: Union[str, int, Tuple[int, int]] = 0,
    dilation: Union[int, Tuple[int, int]] = 1,
    groups: int = 1,
    bias: bool = True,
    padding_mode: str = 'zeros',
    device=None,
    dtype=None,
) -> None
```

Docstring:

Applies a 2D convolution over an input signal composed of several input planes.

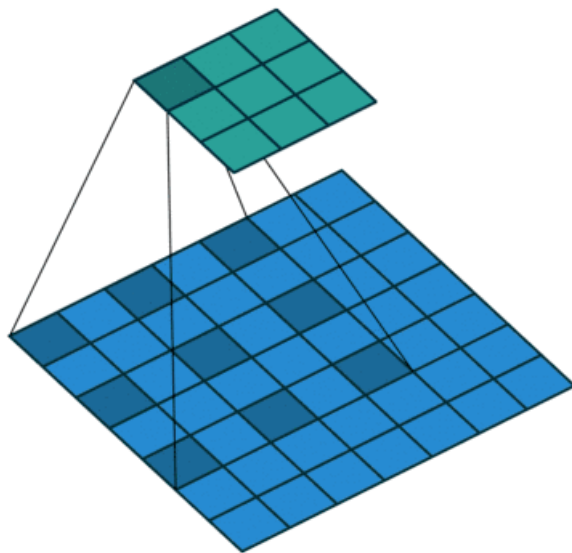
依次说明：

- in_channels：输入层的图像数据通道数
- out_channels：输出层的图像数据通道数
- kernel_size：卷积核尺寸，既可以是标量，代表一个正方形的卷积核；也可以是一个二元组，分别代表长和宽
- stride：卷积核滑动的步幅，默认情况下为1，即逐像素点移动，若设置大于1的数值，则可以实现跨步移动的效果
- padding：边缘填充的层数，默认为0，表示对原始图片数据不做填充。如果取值大于0，例如padding=1，则在原始图片数据的外圈添加一圈0值，注意这里是添加一圈，填充后的图片尺寸为原尺寸长和宽均+2。padding=1和stride=2的卷积示意图如下：



padding=1, stride=2的卷积

- dilation: 用于控制是否是空洞卷积，这是后续论文新提出的卷积改进，即由原始的稠密的卷积核变为空洞卷积核，用于减小卷积核参数同时增大感受野。空洞卷积示意图如下：



dilation=1的空洞卷积

细品一下stride和dilation两个参数对卷积操作影响的区别。

然后是Conv2d的输入和输出数据。Conv2d可以看做是一个特殊的神经网络层，所以其本质上是将一个输入的tensor变换为另一个tensor，其中输入和输出tensor的尺寸即含义分别如下：

- input: batch × in_channels × height × width
- output: batch × output_channels × height × width

即输入和输出tensor主要是图像通道数上的改变，图像的高和宽的大小则要取决于kernel、padding、stride和dilation四个参数的综合作用，这里不再给出具体的计算公式。

2.池化模块：MaxPool1d、MaxPool2d、MaxPool3d

池化模块在PyTorch中主要内置了最大池化和平均池化，每种池化又可细分为一维、二维和三维池化层。这里仍然以MaxPool2d简要介绍：

```
nn.MaxPool2d?
```

Init signature:

```
nn.MaxPool2d(
    kernel_size: Union[int, Tuple[int, ...]],
    stride: Union[int, Tuple[int, ...], NoneType] = None,
    padding: Union[int, Tuple[int, ...]] = 0,
    dilation: Union[int, Tuple[int, ...]] = 1,
    return_indices: bool = False,
    ceil_mode: bool = False,
) -> None
```

Docstring:

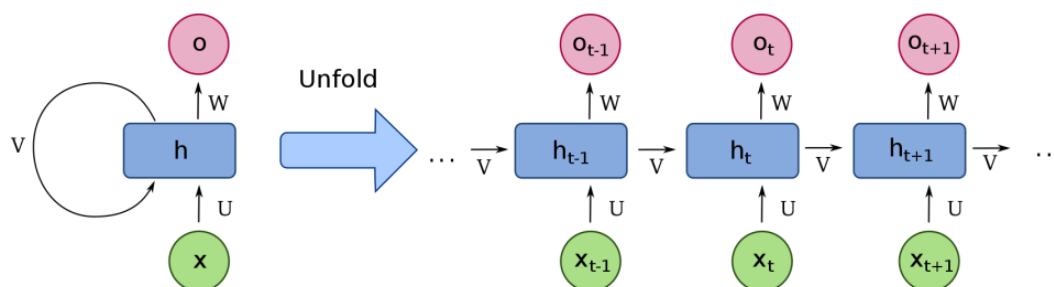
Applies a 2D max pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, H, W) , output (N, C, H_{out}, W_{out}) and `attr:kernel_size` (kH, kW) can be precisely described as:

可见，池化模块的初始化参数与卷积模块中的初始化参数有很多共通之处，包括kernel、stride、padding和dilation等4个参数的设计上。相应的，由于池化层仅仅是各通道上实现数据尺寸的降维，所以其输入和输出数据的通道数不变，而仅仅是尺寸的变化，这里也不再给出相应的计算公式。

以上，便是对卷积神经网络的一些介绍，从卷积操作的起源、到对卷积提取局部特征的理解，最后到在PyTorch中的模块使用，希望对读者有所帮助。

7. 循环神经网络【RNN】



标准的RNN模块结构

如果说从DNN到CNN的技术演进是为了面向图像数据解决提取空间依赖特征的问题，那么RNN的出现则是为了应对序列数据建模，提取时间依赖特征（这里的“时间”不一定要具有确切的时间信息，仅用于强调数据的先后性）。

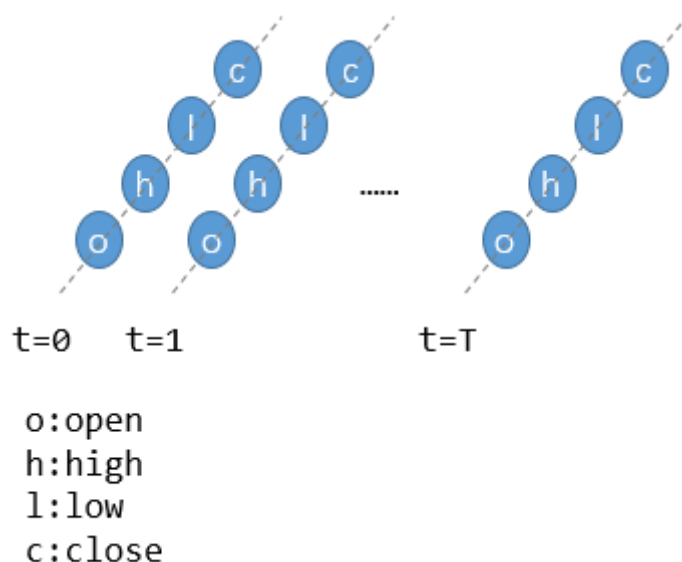
延续前文的行文思路，本文仍然从以下四个方面加以介绍：

- 什么是RNN
- RNN为何有效
- RNN的适用场景
- 在PyTorch中的使用

7.1 什么是RNN

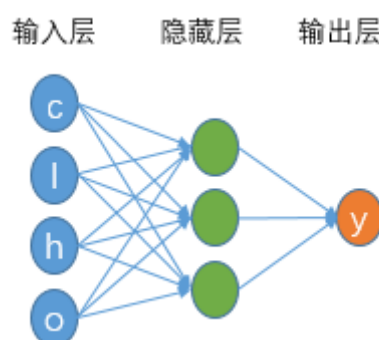
循环神经网络，英文Recurrent Neural Network，简写RNN。显然，这里的“循环”是最具特色的关键词。那么，如何理解“循环”二字呢？这首先要从RNN适用的任务——序列数据建模说起。

RNN适用于序列数据建模，典型的序列数据可以是时间序列数据，例如股票价格、天气预报等；也可以是文本序列数据，比如文本情感分析，语言翻译等。这些数据都有一个共同的特点，那就是输入数据除了具有特征维度外，还有一个先后顺序的维度。以股票数据为例，假设一支股票包含[open, high, low, close]四个维度特征，那么其数据结构的示意图为：



实际上，这就是一个二维的数据矩阵 $[T, 4]$ ，其中 T 为时序长度，4为特征个数。在机器学习中，单支股票数据只能算作一个样本，进一步考虑多支股票则可构成标准的序列数据集 $[N, T, 4]$ ，其中 N 为股票数量。

进一步地，针对这样的序列数据集，RNN是怎样进行特征提取的呢？这里，我们有必要先追溯RNN之前的模型——DNN，并给出一个简单的DNN网络架构如下：



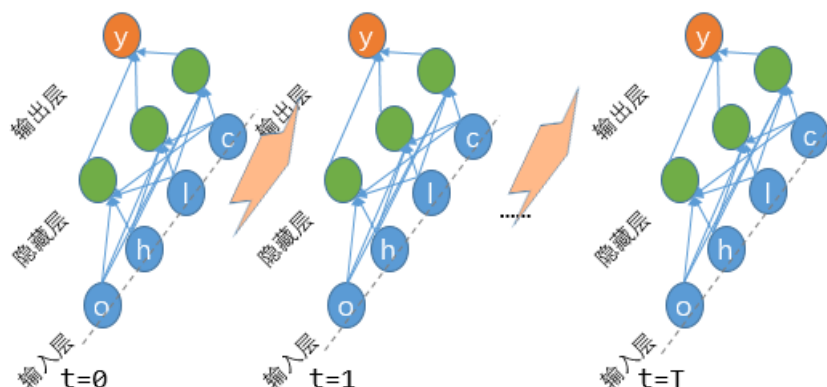
一个具有4个输入特征、单隐藏层的DNN架构

如果我们不考虑股票的时间特性（消除前述数据集的时间维度），则每支股票特征仅有4个，便可以直接利用上述的3层DNN架构完成特征处理，并得到最终的预测结果。上述这一过程可以抽象为：

$$y = f(XW^T + b)$$

这其实恰好就是前文提到的内容：神经网络本质上是在拟合一个复杂的复合函数，其中这个复合函数的输入是 X ，网络参数是 W 和 b 。

那么，当引入了时间维度，输入数据不再是4个特征，而是 $T \times 4$ 个特征，且这 T 组特征具有确切的先后顺序，那么RNN要如何处理呢？一个简单的思路是将上述DNN结构堆叠起来，并循环执行，例如网络结构可能长这样：



RNN处理序列数据示意图

如上述示意图所示，纵向上仍然是一个单纯的DNN网络进行数据处理的流程，而横向上则代表了新增的时间维度。也正因为这个时间维度的出现，所以时刻 t 对应DNN输入数据将来源于两部分：当前时刻 t 对应的4个输入特征，以及 $t-1$ 时刻的输出信息，即图中粉色横向宽箭头表示的部分。

是否好奇：为啥要将 $t-1$ 时刻的输出作为 t 时刻的输入呢？当然是因为要序列建模！如果不把相邻时刻的输入输出联系起来，那序列先后顺序又该如何体现？

用数学公式加以抽象表示，就是：

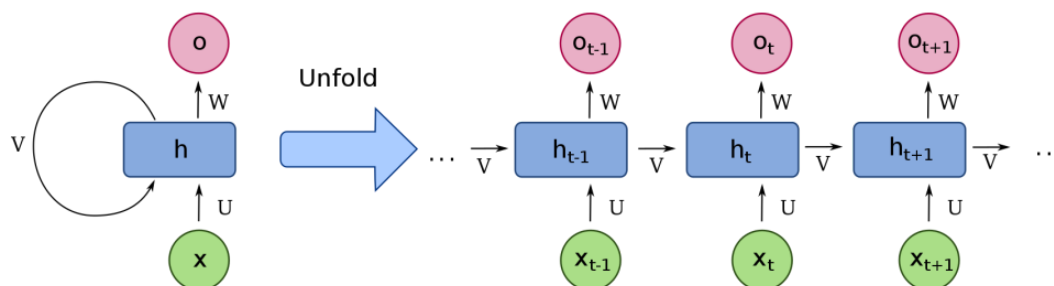
$$h_t = f(X_t W_i^T + h_{t-1} W_h^T + b)$$

上式中， W_i 表达当前输入信息的权重矩阵， W_h 表达对前一时刻输入的权重矩阵，且二者在各个时刻是相同的，可理解为面向时间维度的权值共享。

对比该公式和前面DNN中的公式，主要有两点区别：

- 映射函数的输入数据部分不止是 X ，还有前一时刻提供的信息 h_{t-1} ；
- 模型的直接输出变为中间状态 h_t ，而 h_t 与最终输出 y 的区别在于： y 可以是直接给出最终需要的信息，例如股票预测中的收盘价，但 h_t 为了兼顾相邻时刻之间的信息交互，往往不一定符合最终的输出结果，所以可能需要对 h_t 进一步使用一个DNN网络进行映射得到想要的输出

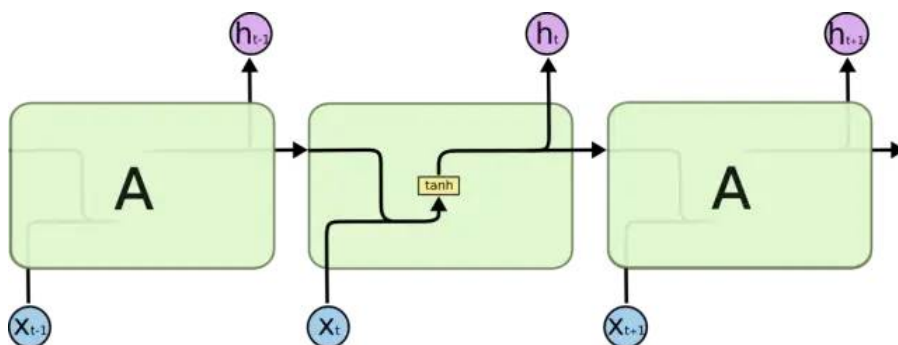
至此，我们再来看一下开篇中给出的标准RNN模块结构：



上图中的右侧（unfold部分），横向代表了沿时间维度传播的流程，纵向代表了单个时刻的信息处理流程（各时刻都是一个DNN），其中 X 代表各时刻的输入特征， h_t 代表各时刻对应的状态信息， U 和 V 分别为当前输入和前一时刻状态的神经网络权重， W 为由当前状态 h_t 拟合最终需要结果的神经网络权重。而左侧呢，其实就是把这个循环处理的流程抽象为一个循环结构，也就是那个指向自己的箭头。

这个用指向自己的箭头来表示神经网络的循环，乍一看还挺唬人的！

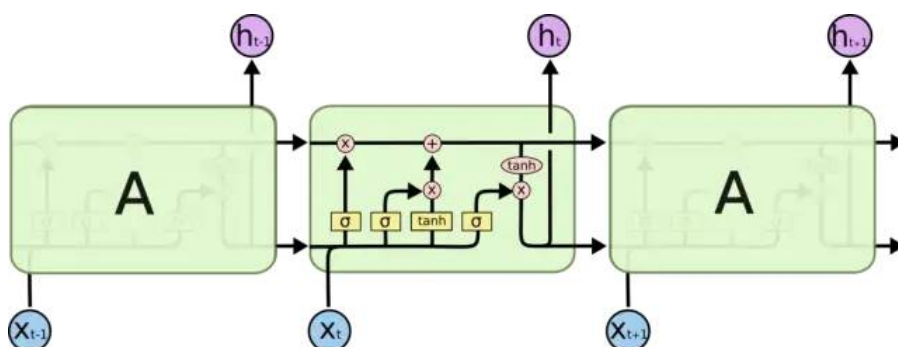
至此，其实就已经完成了标准RNN的结构介绍。用一个更为广泛使用且抽象的RNN单元结构示意图，表达如下：



标准RNN模块的内部结构

标准RNN结构非常简单，通常来说，在神经网络中过于简单的结构也意味着其表达能力有限。比如说，由于每经过一个时间节点的信息传递，都会将之前的历史信息 and 当前信息进行线性组合，并通过一个 tanh 激活函数。tanh 激活函数的输出值在 $(-1, 1)$ 之间，这也就意味着，如果时间链路较长时历史信息很可能会被淹没！这也是标准RNN结构最大的问题——不容易表达长期记忆——换句话说，就是时间链路较长的历史信息会变得很小。

于是，一个相对更为复杂、可以提供相对长期记忆的循环神经网络——LSTM应运而生。LSTM: Long-Short Term Memory network, 中文即为长短时记忆网络。顾名思义，这是一个能够兼顾长期和短期记忆的网络。内部是如何实现的呢？LSTM单元结构如下：



当然，除了上述这一单元结构示意图，LSTM还往往需要这样一组标准计算公式（这个等到后续择机再讲吧。。）：

$$\begin{aligned} i^{(t)} &= \sigma(W^{(i)}x^{(t)} + U^{(i)}h^{(t-1)}) \\ f^{(t)} &= \sigma(W^{(f)}x^{(t)} + U^{(f)}h^{(t-1)}) \\ o^{(t)} &= \sigma(W^{(o)}x^{(t)} + U^{(o)}h^{(t-1)}) \\ \tilde{c}^{(t)} &= \tanh(W^{(c)}x^{(t)} + U^{(c)}h^{(t-1)}) \\ c^{(t)} &= f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)} \\ h^{(t)} &= o^{(t)} \circ \tanh(c^{(t)}) \end{aligned}$$

宏观对照标准RNN和LSTM单元结构，可以概括二者间的主要异同点如下：

- 相同点：各单元结构的输入信息均包含两部分，即当前时刻的输入和前一时刻的输入；输出均为 h_t
- 不同点：
 - RNN中接收前一时刻的输入信息只有一种（这部分叫做 h_t ），体现为相邻单元间的单箭头；而在LSTM中接收前一时刻的输入则包含两部分（两部分分别是 h_t 和 c_t ， c_t 是新引入的部分），体现为相邻单元间的双箭头
 - RNN中的内部结构非常简单，就是两部分向量相加的结果；而LSTM的内部结构相对非常复杂

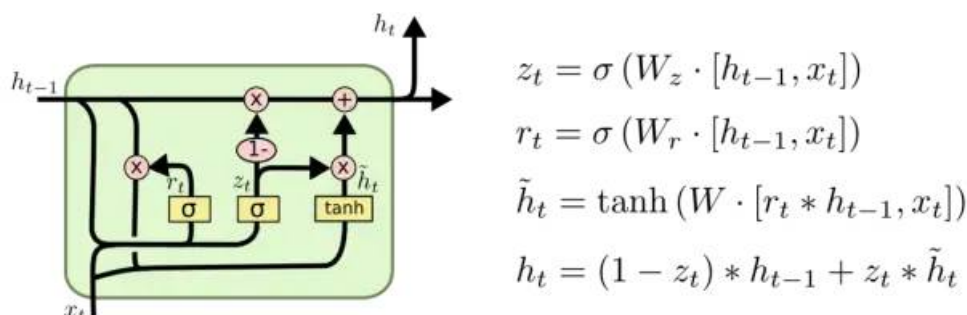
深入理解LSTM单元的内部结构，建议参考某国外作者的博客：<https://colah.github.io/posts/2015-08-Understanding-LSTMs/>。本文中的部分网络模块示意图素材也摘自于此（想必也是国内各种介绍循环神经网络时广泛传播的一组内容）。

这里不再班门弄斧，仅简单补充个人理解：

- 与标准RNN中简单地将前一状态信息与当前信息线性相加不同，LSTM中设计了三个门结构（所谓的门结构就是经过sigmoid处理后的权重矩阵，这个矩阵的取值在(0, 1)之间，越接近于1表示通过的信息越大，越接近于0表示对信息的消减越严重），即遗忘门、输入门和输出门，其中：
 - 遗忘门作用于历史数据输入上，用于控制历史信息对当前输出影响的大小；
 - 输入门作用于当前输入上，用于控制当前输入信息对当前输出影响的大小；
 - 输出门则进一步控制当前输出的大小；
- LSTM中之所以相较于标准RNN能提供更为长期的记忆，根本原因在于引入了从历史信息直接到达输出的通路（LSTM结构中的上侧贯通线），由于该通路可以不与当前时刻输入同步接受相同的门控作用，所以允许网络学习更为久远的记忆（只需将遗忘门的结果学习得大一些）；
- LSTM除了可以提供长期记忆，还可以提供短期记忆，原因在于专门提供了对当前输入信息的通路（LSTM结构中的下侧通路），但同时该信息又与部分历史信息会和，一并经过输入门的控制，这一部分子结构大体定位相当于标准RNN中的简单处理流程

进一步复盘由RNN到LSTM的改进：虽然LSTM设计的非常精妙，通过三个门结构很好的权衡了历史信息 and 当前信息对输出结果的影响，但也有一个细节问题——为了控制两部分信息的相对大小，我们是不是只需要1个参数就可以了呢？比如，计算x和y的加权平均时，我们无需为其分别提供两个系数 α 和 β ，计算 $z=\alpha x+\beta y$ ，而只需 $z=\alpha x+y$ 就可以，或者写成其归一化形式 $z=\alpha x+(1-\alpha)y$ 。正是基于这一朴素思想，LSTM的精简版——GRU单元结构顺利诞生！

具体来说，GRU就是将遗忘门和输入门整合为一个更新门，其单元结构如下：



对比下LSTM与GRU的异同点

所以概括一下：从RNN到LSTM的改进是为了增加网络容量，权衡长短期记忆；而从LSTM到GRU的演进则是为了精简模型，去除冗余结构。

上述大体介绍了循环神经网络的起源，并简要介绍了三种最常用的循环神经网络单元结构：RNN、LSTM和GRU。如果说卷积和池化是卷积神经网络中的标志性模块，那么这三个模块无疑就是循环神经网络中的典型代表。

7.2 RNN为何有效

DNN可以用通用近似定理论证其有效性（更准确地说，通用近似定理适用于所有神经网络，而不止是DNN），CNN也可以抽取若干个特征图直观的表达其卷积的操作结果，但RNN似乎并不容易直接说明其为何会有效。

总体而言，我个人从以下几个方面加以感性理解：

- 循环神经网络适用于序列数据建模场景，而相较于普通的DNN（包括CNN，其实也是不带有时间依赖信息的单时刻输入特征）而言，其最大的特点在于如何按顺序提取各时刻的新增信息，所以形式上必然是要将当前信息与历史信息做融合
- 为了保持对所有时刻信息处理流程的一致性，RNN中也有权值共享机制，即网络参数在随时间维度的传播过程中使用同一套网络权重（ W_i 和 W_h ），这保证了处理时序信息的公平性
- 适当的门机制。实际上，标准的RNN单元其记忆能力是有限的，所以谈不上有效；但为何LSTM却非常有效？其核心就在于门的设计上，即允许神经网络通过反向传播算法去自主学习：什么情况下应给历史信息较大的权重——记忆历史；什么情况下又该给当前输入信息较大的权重——更新现在，而这一切都交由网络通过训练集自己去训练

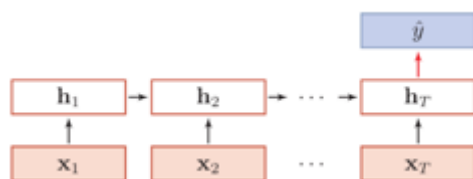
循环神经网络是一个精妙的设计，对于序列建模而言是非常有效的，其历史地位不亚于卷积神经网络。但也值得指出，目前循环神经网络似乎已经有了替代结构——注意力机制——这也是对序列建模非常有效的网络结构，而且无需按时间顺序执行，可以方便的实现并行化，从而提高执行效率——当然，这是后来的故事！

7.3 RNN适用的场景

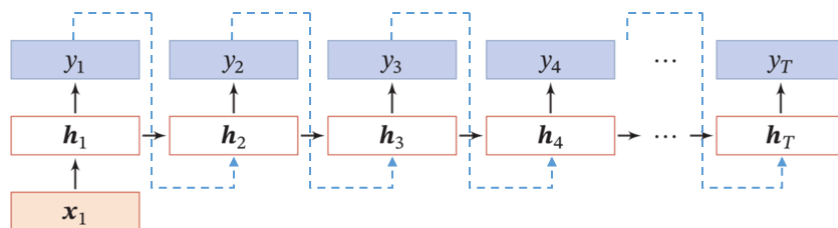
论及循环神经网络适用的场景，其实答案是相对明确的，即序列数据建模。进一步地，这里的序列数据既可以是**带有时间属性的时序数据**，也可以是**仅含有先后顺序关系的其他序列数据**，例如文本序列等。

与此同时，根据输入数据和预测结果各自序列长度的不同，又衍生了几种不同的任务场景，包括：

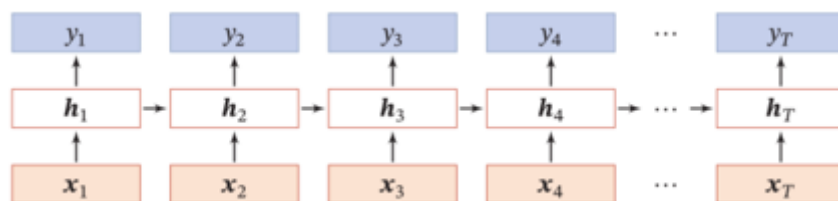
- N to 1：前述的例子都是N个时刻的输入对应1个时刻的输出，例如股票预测，天气预报、文本情感分类等



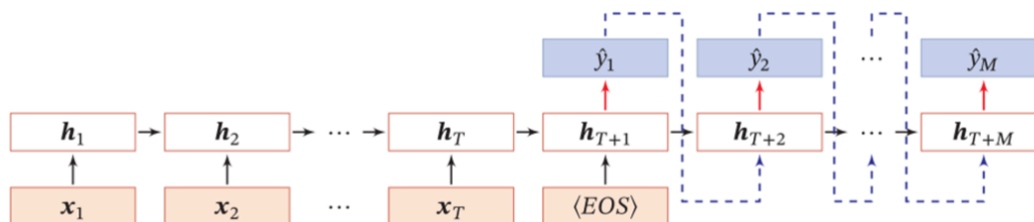
- 1 to N：典型应用是NLP中的作诗、写文章等，即仅提供标题或起始单词，交由模型输出一篇文章



- N to N：即给定N个历史时刻的输入数据，同步给出相应的N个输出（注意，这里的输入和输出是同步的），典型的应用场景是词性分析，即逐一输入一段文本中的各个单词，要求输出各单词的词性判断结果



- N to M：也就是给定N个历史的输入数据，完全处理后得到M个输出，其中M个输出与N个输入具有先后顺序，输入和输出是异步的（也叫Seq2Seq）。典型的场景是机器翻译：给定N个英文单词，翻译结果是M个中文词语，多步的股票预测也符合这种场景



7.4 在PyTorch中的使用

对于标准RNN、LSTM和GRU三种典型的循环神经网络单元，PyTorch中均有相应的实现。对使用者来说，无需过度关心各单元内在的结构，因为三者几乎是具有相近的封装形式，无论是类的初始化参数，还是对输入和输出数据的形式上。

所以，这里以LSTM为例加以阐述。首先来看其类的签名文档：

```
nn.LSTM?
```

```
Init signature: nn.LSTM(*args, **kwargs)
```

```
Docstring:
```

```
Applies a multi-layer long short-term memory (LSTM) RNN to an input
sequence.
```

```
For each element in the input sequence, each layer computes the following
function:
```

```
.. math::
    \begin{array}{ll}
        i_t = \sigma(W_{ii} x_t + b_{ii} + W_{hi} h_{t-1} + b_{hi}) \quad \backslash \backslash \\
        f_t = \sigma(W_{if} x_t + b_{if} + W_{hf} h_{t-1} + b_{hf}) \quad \backslash \backslash \\
        g_t = \tanh(W_{ig} x_t + b_{ig} + W_{hg} h_{t-1} + b_{hg}) \quad \backslash \backslash \\
        o_t = \sigma(W_{io} x_t + b_{io} + W_{ho} h_{t-1} + b_{ho}) \quad \backslash \backslash \\
        c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad \backslash \backslash \\
        h_t = o_t \odot \tanh(c_t) \quad \backslash \backslash
    \end{array}
```

```
where :math:`h_t` is the hidden state at time `t`, :math:`c_t` is the cell
state at time `t`, :math:`x_t` is the input at time `t`, :math:`h_{t-1}`
is the hidden state of the layer at time `t-1` or the initial hidden
state at time `0`, and :math:`i_t`, :math:`f_t`, :math:`g_t`,
:math:`o_t` are the input, forget, cell, and output gates, respectively.
:math:`\sigma` is the sigmoid function, and :math:`\odot` is the Hadamard product.
```

上述文档仅给出了LSTM的理论介绍，主要是陈列了其内部结构的几个相关理论计算公式。而对于类的初始化参数方面，LSTM并未直接给出形参列表，而是使用`*arg`和`**kwargs`来接受自定义的传入参数，其中，常用的参数包括：

Args:

- `input_size`: The number of expected features in the input `x`
- `hidden_size`: The number of features in the hidden state `h`
- `num_layers`: Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a 'stacked LSTM', with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- `bias`: If `False`, then the layer does not use bias weights `b_ih` and `b_hh`. Default: `True`
- `batch_first`: If `True`, then the input and output tensors are provided as `(batch, seq, feature)` instead of `(seq, batch, feature)`. Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- `dropout`: If non-zero, introduces a 'Dropout' layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `attr:'dropout'`. Default: 0
- `bidirectional`: If `True`, becomes a bidirectional LSTM. Default: `False`
- `proj_size`: If `> 0`, will use LSTM with projections of corresponding size. Default: 0

具体来说:

- `input_size`: 输入数据的特征维度, 例如在前面举的股票例子中包括open、high、low和close共4个特征, 即`input_size=4`
- `hidden_size`: 前面提到, 在每个时间截面循环神经单元其实都是一个DNN结构, 默认情况下该DNN只有单个隐藏层, `hidden_size`即为该隐藏层神经元的个数, 在前述的股票例子中隐藏层神经元数量为3, 即`hidden_size=3`
- `num_layers`: 虽然RNN、LSTM和GRU这些循环单元的重点是构建时间维度的序列依赖信息, 但在单个事件截面的特征处理也可以支持含有更多隐藏层的DNN结构, 默认状态下为1
- `bias`: 类似于`nn.Linear`中的`bias`参数, 用于控制是否拟合偏置项, 默认为`bias=True`, 即拟合偏置项
- `batch_first`: 用于控制输入数据各维度所对应的含义, 前面举例中一直用的示例维度是(N, T, 4), 即分别对应样本数量、时序长度和特征数量, 这种可能比较符合部分人的思维习惯(包括我自己也是如此), 但实际上LSTM更喜欢的方式是将序列维度放于第一个维度, 此时即为(T, N, 4)。`batch_first`默认为`False`, 即样本数量为第二个维度, 序列长度为第一个维度, (seq_len, batch, input_size)
- `dropout`: 用于控制全连接层后面是否设置dropout单元, 增加dropout有时是为了增强模型的泛化能力
- `bidirectional`: 上述所介绍LSTM等都是沿着序列的正向进行处理和传播, 正向传播更容易记住靠后的序列信息, 而忘记前面的信息; 所以LSTM的一种改进就是双向循环单元结构, 即首先沿正向处理一遍, 再逆向处理一遍。`bidirectional`参数即用于控制是单向还是双向, 默认为`bidirectional=False`, 即仅正向处理

接下来是输入和输出信息:

Inputs: input, (h_0, c_0)

- `**input**`: tensor of shape (L, H_{in}) for unbatched input, (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `func:torch.nn.utils.rnn.pack_padded_sequence` or `func:torch.nn.utils.rnn.pack_sequence` for details.
- `**h_0**`: tensor of shape $(D * \text{num_layers}, H_{out})$ for unbatched input or $(D * \text{num_layers}, N, H_{out})$ containing the initial hidden state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.
- `**c_0**`: tensor of shape $(D * \text{num_layers}, H_{cell})$ for unbatched input or $(D * \text{num_layers}, N, H_{cell})$ containing the initial cell state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.

```
Outputs: output, (h_n, c_n)
* **output**: tensor of shape :math:`(L, D * H_{out})` for unbatched input,
:math:`(L, N, D * H_{out})` when ``batch_first=False`` or
:math:`(N, L, D * H_{out})` when ``batch_first=True`` containing the output features
`h_t` from the last layer of the LSTM, for each `t`. If a
:class:`torch.nn.utils.rnn.PackedSequence` has been given as the input, the output
will also be a packed sequence.
* **h_n**: tensor of shape :math:`(D * \text{num\_layers}, H_{out})` for unbatched input or
:math:`(D * \text{num\_layers}, N, H_{out})` containing the
final hidden state for each element in the sequence.
* **c_n**: tensor of shape :math:`(D * \text{num\_layers}, H_{cell})` for unbatched input or
:math:`(D * \text{num\_layers}, N, H_{cell})` containing the
final cell state for each element in the sequence.
```

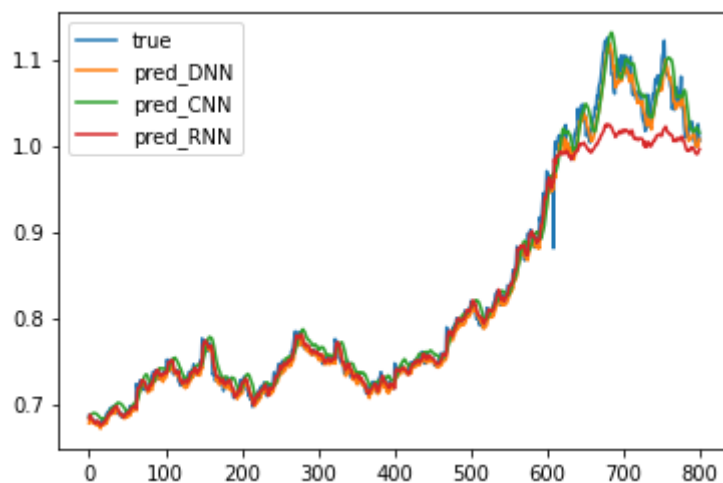
大体来看，输入和输出具有相近的形式（这也是为啥可以循环处理的原因），对于LSTM来说包含三部分，即：

- input/output: (L, N, H_in/H_out)，其中L为序列长度，N为样本数量，H_in和H_out分别为输入数据和输出结果的特征维度，即前面初始化中用到的input_size和hidden_size
- h_n和c_n，分别对应最后时刻循环单元对应的隐藏状态和细胞状态（LSTM的相邻单元之间有两条连接线，上面的代表细胞状态c_n，下面代表隐藏状态h_n），如果是RNN或者GRU则只有隐藏状态h_n
- 进一步地，output与h_n的联系和区别是什么呢？output是区分时间维度的输出序列，记录了各时刻所对应DNN的最终输出结果，L个序列长度对应了L个时刻的输出；而h_n则只记录最后一个序列所对应的隐藏层输出，所以只有一个时刻的结果，但如果num_layers>1或者bidirectional设置为True时，则也会有多个输出结果。当在默认情况下，即num_layers=1，bidirectional=False时，output[-1]=h_n

关于循环神经网络的介绍就到这里，后续将基于股票数据集提供一个实际案例。

8. 三大神经网络在股票数据集上的实战

DNN、CNN和RNN是深度学习中的三大经典神经网络，分别有各自的适用场景。但为了能够在同一任务下综合对比这三种网络，本文选择对股票预测这一任务开展实验，其中DNN可以将历史序列特征转化为全连接网络，而CNN则可利用一维卷积进行特征提取，RNN则天然适用于序列数据建模。



本文行文结构如下：

- 数据集准备
- DNN模型构建及训练

- CNN模型构建及训练
- RNN模型构建及训练
- 对比与小结

8.1 数据集准备

本次实战案例选择了某股票数据，时间范围为2005年1月至2021年7月间的所有交易日，共4027条记录，其中每条记录包含[Open, High, Low, Close, Vol]共5个特征字段。数据示意如下：

	Date	Open	High	Low	Close	Vol
0	2005/1/4	1150	1154	1143	1145	101322
1	2005/1/5	1141	1152	1141	1151	115404
2	2005/1/6	1151	1157	1151	1154	116530
3	2005/1/7	1162	1164	1156	1158	142096
4	2005/1/10	1158	1159	1152	1155	125086

显然，各字段的取值范围不同，为了尽可能适配神经网络中激活函数的最优特性区间，需要对特征字段进行归一化处理，这里选用sklearn中MinMaxScaler进行。同时，为了确保数据预处理时不造成信息泄露，在训练MinMaxScaler时，只能用训练集中的记录。所以，这里按照大体上8:2的比例切分，选择后800条记录用于提取测试集，之前的数据用作训练集。因此，做如下数据预处理：

```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()
mms.fit(df.iloc[:-800][["Open", "High", "Low", "Close", "Vol"]])
df[["Open", "High", "Low", "Close", "Vol"]] = mms.transform(df[["Open", "High", "Low", "Close", "Vol"]])
```

而后，查看预处理之后的数据：

	Open	High	Low	Close	Vol
count	4027.000000	4027.000000	4027.000000	4027.000000	4027.000000
mean	0.761073	0.568585	0.758978	0.761103	0.089978
std	0.154262	0.277390	0.154109	0.154221	0.088783
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.650391	0.368939	0.648337	0.650781	0.026120
50%	0.735547	0.521434	0.733464	0.735547	0.063542
75%	0.919141	0.851019	0.918200	0.919336	0.125015
max	1.123828	1.251581	1.124070	1.125391	1.000000

显然，除了Vol列字段的数据范围调整为[0, 1]外，其他4个字段的最大值均超过了1，这是因为测试集中的数据范围比训练集中的数据范围要大，但这更符合实际训练的要求。

而后，进行数据集的构建。既然是时序数据，我们的任务是基于当前及历史一段时间的数据，预测股票次日的收盘价（Close字段），我们大体将历史数据的时间长度设定为30，而后采用滑动窗口的形式依次构建数据集和标签列，构建过程如下：

```

x = []
y = []
for i in range(30, len(df)):
    x.append(df.iloc[i-30:i, 1:6].values) # 输入数据未取到i时刻
    y.append(df.iloc[i, 4]) # 标签数据为i时刻
x = torch.tensor(x, dtype=torch.float)
y = torch.tensor(y, dtype=torch.float).view(-1, 1)
x.shape, y.shape
## 输出
(torch.Size([3997, 30, 5]), torch.Size([3997, 1]))

```

而后，进行训练集和测试集的切分。由于是时序数据，仅能按时间顺序切分，这里沿用之前的设定，及选取后800条记录作为测试集，前面的作为训练集：

```

N = -800
x_train, x_test = x[:N], x[N:]
y_train, y_test = y[:N], y[N:]
trainloader = DataLoader(TensorDataset(x_train, y_train), 64, True)
x_train.shape, x_test.shape
## 输出
(torch.Size([3197, 30, 5]), torch.Size([800, 30, 5]))

```

至此，完成了数据集的准备和切分。注意，这里数据集维度为3，其含义为[batch, seq_len, input_size]，即[样本数, 序列长度, 特征数]。接下来开始使用三类神经网络进行建模。

8.2 DNN模型构建及训练

DNN是最早的神经网络，主要构成元素是若干个全连接层及相应的激活函数。这里为了多个时刻的历史特征一并加入到全连接训练，需要首先对三维的输入数据展平为二维，此处即为[batch, seq_len, input_size]变为[batch, seq_len*input_size]，而后即可应用全连接模块。这里我们对DNN添加3个隐藏层，且遵循神经元数量逐渐减少的节奏。具体来说，DNN模型设计如下：

```

class ModelDNN(nn.Module):
    def __init__(self, input_size, hiddens=[64, 32, 8]):
        super().__init__()
        self.hiddens = hiddens
        self.net = nn.Sequential(nn.Flatten())
        for pre, nxt in zip([input_size]+hiddens[:-1], hiddens):
            self.net.append(nn.Linear(pre, nxt))
            self.net.append(nn.ReLU())
        self.net.append(nn.Linear(hiddens[-1], 1))

    def forward(self, x):
        return self.net(x)

```

而后即可开始训练，其中模型优化器选择Adam，并保留默认学习率0.001，损失函数选用MSELoss，epoch设置为100，每10个epoch监控一下训练集损失和测试集损失。训练过程如下：

```

modelDNN = ModelDNN(30*5)
optimizer = optim.Adam(modelDNN.parameters())
criterion = nn.MSELoss()

for i in trange(100):

```

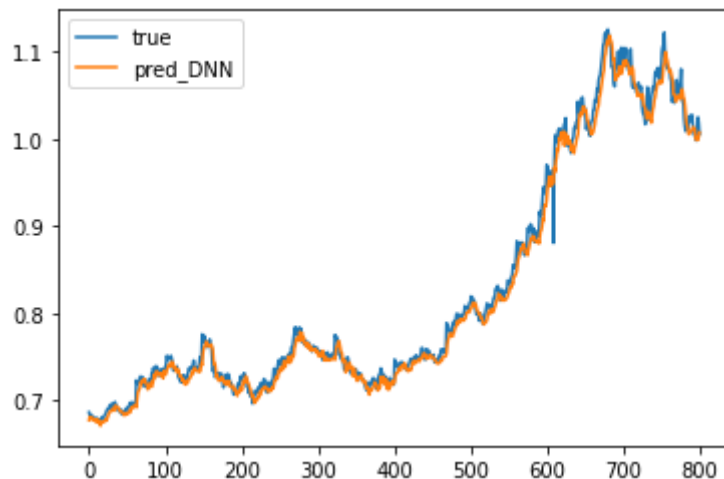
```

for X, y in trainloader:
    y_pred = modelDNN(X)
    loss = criterion(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
if (i+1) % 10 == 0:
    with torch.no_grad():
        train_pred = modelDNN(X_train)
        train_mse = criterion(train_pred, y_train)
        test_pred = modelDNN(X_test)
        test_mse = criterion(test_pred, y_test)
        print(i, train_mse.item(), test_mse.item())

## 输出
...
9 0.000504376832395792 0.0004596100770868361
19 0.00039938988629728556 0.00026557157980278134
29 0.0003091454564128071 0.00021832890342921019
39 0.0002735845628194511 0.00017248920630663633
49 0.0002678786404430866 0.00018119681044481695
59 0.00024609942920506 0.00013418315211310983
69 0.0002652891562320292 0.00018864106095861644
79 0.00023099897953215986 0.00011782139335991815
89 0.000230113830184564 0.0001355513377347961
99 0.00023369801056105644 0.0001391700643580407
...

```

整体来看，模型训练是比较有效的，损失下降得很快。用最终的模型预测一下测试集的输出，并绘制对照曲线：



看上去效果还不错！

8.3 CNN模型构建及训练

CNN模型的核心元素是卷积和池化，所以这里我们也对该序列数据应用这两个模块。值得注意的是，对于序列数据，特征数应对应卷积核的通道数，而卷积滑动的方向应该是在序列维度上。也就是，此处我们首先应将输入数据形状由[batch, seq_len, input_size]转化为[batch, input_size, seq_len]，而后再应用一维卷积和一维池化层。不失一般性，我们首先设置两个kernel_size=3的Conv1d和两个kernel_size=2的AvgPool1d，而后再将特征展平转变为2维数据，最后经过一个全连接得到预测输出。模型构建代码如下：

```

class ModelCNN(nn.Module):
    def __init__(self, in_channels, hidden_channels=[8, 4]):
        # input: N x C x L
        # C: 5->8->4
        # L: 30->28->14->12->6
        super().__init__()
        self.in_channels = in_channels
        self.hidden_channels = hidden_channels
        self.net = nn.Sequential()
        for in_channels, out_channels in zip([in_channels]+hidden_channels[:-1],
        hidden_channels):
            self.net.append(nn.Conv1d(in_channels, out_channels, kernel_size=3))
            self.net.append(nn.AvgPool1d(kernel_size=2))
        self.net.append(nn.Flatten())
        self.net.append(nn.Linear(6*hidden_channels[-1], 1))

    def forward(self, x):
        x = x.permute(0, 2, 1)
        return self.net(x)

```

接下来是训练过程，训练参数沿用DNN中的设定，代码及结果如下：

```

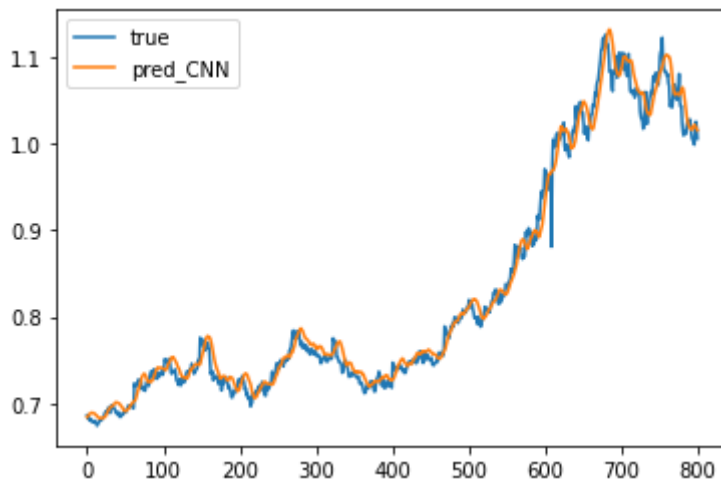
modelCNN = ModelCNN(5)
optimizer = optim.Adam(modelCNN.parameters())
criterion = nn.MSELoss()

for i in range(100):
    for X, y in trainloader:
        y_pred = modelCNN(X)
        loss = criterion(y_pred, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    if (i+1) % 10 == 0:
        with torch.no_grad():
            train_pred = modelCNN(X_train)
            train_mse = criterion(train_pred, y_train)
            test_pred = modelCNN(X_test)
            test_mse = criterion(test_pred, y_test)
            print(i, train_mse.item(), test_mse.item())

## 输出
...
9 0.0006343051209114492 0.0005071654450148344
19 0.0005506690358743072 0.0005008925218135118
29 0.00048158588469959795 0.0003848765918519348
39 0.0004702212754637003 0.00034184992546215653
49 0.00042759429197758436 0.00038456765469163656
59 0.0003927639627363533 0.00028791968361474574
69 0.00037852991954423487 0.0002683571365196258
79 0.00034848105860874057 0.00025418962468393147
89 0.00035096638021059334 0.00023561224224977195
99 0.0003425602917559445 0.00022362983145285398
...

```

绘图展示一下：



看上去效果也不错！

8.4 RNN模型构建及训练

RNN是天然适用于序列数据建模的，这里我们选用GRU实践一下，并只选择最基础的GRU结构，即num_layers=1, bidirectional=False。在最后时刻输出的隐藏状态hn的基础上，使用一个全连接得到预测输出。网络结构代码如下：

```
class ModelRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        # 注意，这里设置batch_first=True
        self.gru = nn.GRU(input_size=input_size, hidden_size=hidden_size,
                           batch_first=True)
        self.activation = nn.ReLU()
        self.output = nn.Linear(hidden_size, 1)

    def forward(self, x):
        _, hidden = self.gru(x)
        hidden = hidden.squeeze(0)
        hidden = self.activation(hidden)
        return self.output(hidden)
```

RNN模型训练仍沿用前序训练参数设定，训练过程及结果如下：

```
modelRNN = ModelRNN(5, 4)
optimizer = optim.Adam(modelRNN.parameters())
criterion = nn.MSELoss()

for i in range(100):
    for x, y in trainloader:
        y_pred = modelRNN(x)
        loss = criterion(y_pred, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    if (i+1) % 10 == 0:
```

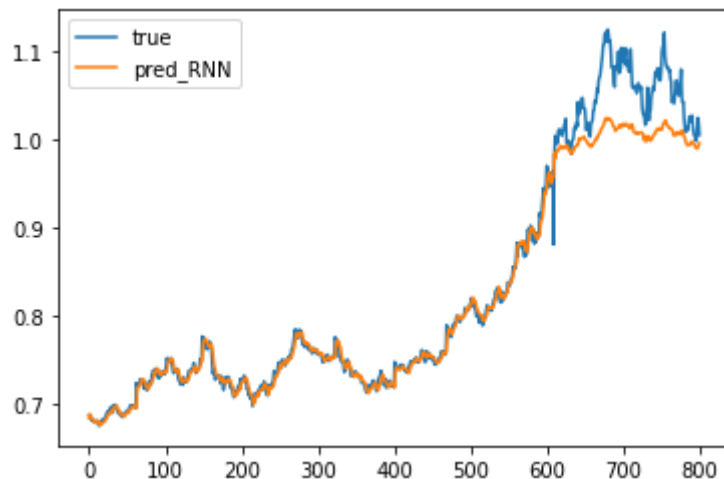
```

with torch.no_grad():
    train_pred = modelRNN(X_train)
    train_mse = criterion(train_pred, y_train)
    test_pred = modelRNN(X_test)
    test_mse = criterion(test_pred, y_test)
    print(i, train_mse.item(), test_mse.item())

## 输出
'''
9 0.00942652765661478 0.014085204340517521
19 0.0008321275236085057 0.003637362737208605
29 0.0002676403964869678 0.001723079476505518
39 0.00024218574981205165 0.0013364425394684076
49 0.00022829060617368668 0.0011184979230165482
59 0.000223998453700915 0.0009826462483033538
69 0.00021638070757035166 0.0008919781539589167
79 0.00021416762319859117 0.0008065822767093778
89 0.00022308445477392524 0.0007256607641465962
99 0.00021087308414280415 0.0006862917798571289
'''

```

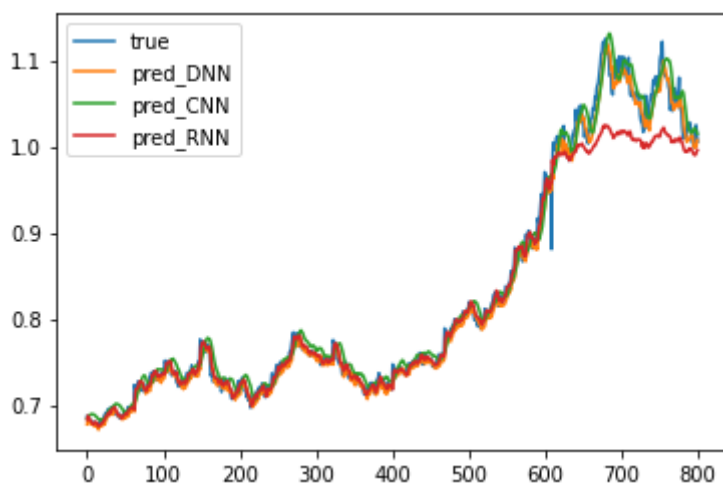
查看一下输出，并绘制预测结果曲线如下：



前面预测的都还是比较准的，只是最后一点预测误差较大，这可能是由于测试集标签真实值超出了1，而这种情况是模型在训练集上所学不到的信息.....

8.5 对比与小结

最后，我们综合对比一下三大神经网络模型在该股票预测任务上的表现。首先来看各自的预测结果对比曲线：



整体来看，DNN和CNN在全部测试集上的表现要略胜于RNN一些。然后再从评价指标和训练速度方面对比一下：

模型	训练耗时	R2_score	MSE
DNN	11s	0.9923	0.00014
CNN	12s	0.9878	0.00022
RNN	44s	0.9625	0.00068

好吧，DNN几乎呈现碾压态势——模型训练速度快，预测结果精度高！这大体可以体现两个结论：

- 机器学习界广泛受用的“天下没有免费的午餐”定理，即不存在一种确切的模型在所有数据集上均表现较好；
- 虽然RNN是面向序列数据建模而生，但DNN和CNN对这类任务也有一定的适用性，巧妙设计网络结构也能带来不错的效果。

以上案例及结论仅供参考！