

大语言模型与信息决策——补充资料

刘梓豪 2100011704

2024 年 3 月 3 日

目录

1	Introduction to Jupyter Notebook & PyTorch	2
1.1	Jupyter Notebook: 功能简介	2
1.2	Jupyter Notebook: 试用与安装	2
1.3	Jupyter Notebook: 使用与调试	4
1.4	Jupyter Notebook: 参考资料与高级用法	5
1.5	PyTorch: 功能简介与安装	5
1.6	PyTorch: 基本代码实现	7
1.6.1	Tensor 操作	7
1.6.2	自动求导	8
1.7	PyTorch 高级用法: 训练神经网络/数据集处理/全连接神经网络实战	8
1.7.1	训练神经网络原理简介	8
1.7.2	数据集处理	9
1.7.3	全连接神经网络实战	10

1 Introduction to Jupyter Notebook & PyTorch

1.1 Jupyter Notebook: 功能简介

Jupyter Notebook (简称为 Jupyter) 是一种开源的交互式计算环境，广泛用于数据分析、数据可视化、机器学习、科学计算等领域。它提供了一个基于网页浏览器的界面，允许用户创建和共享包含代码、文本、图像和数学公式的文档，这些文档被称为“笔记本”。



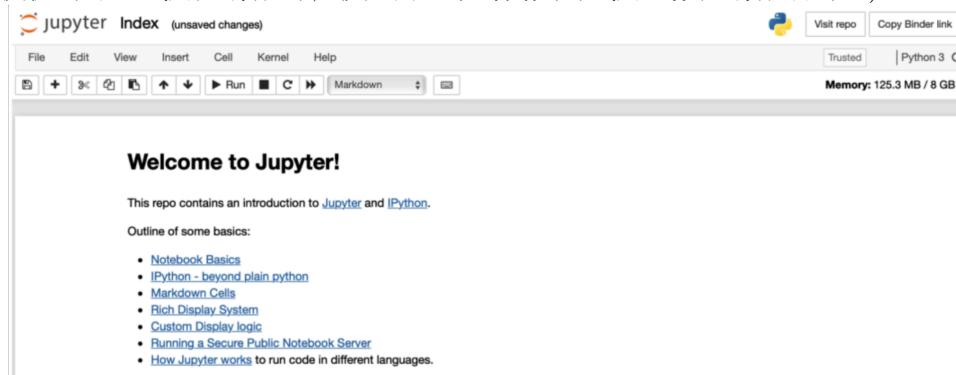
简而言之，Jupyter Notebook 是以网页的形式打开，可以在网页页面中直接编写代码和运行代码，代码的运行结果也会直接在代码块下方显示。如在编程过程中需要编写说明文档，可在同一个页面中直接编写，便于作及时的说明和解释。下面给大家详细介绍一下该工具的安装过程及使用方法。

Jupyter Notebook 有以下的主要特点：

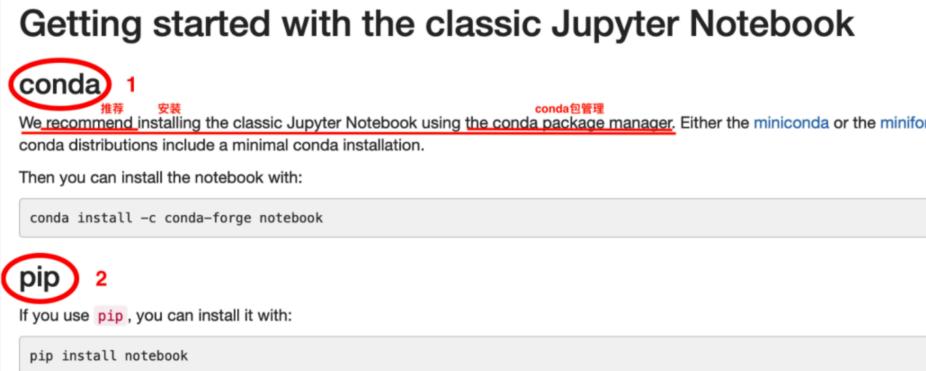
1. 编程时具有语法高亮、缩进、tab 补全的功能.
2. 可直接通过浏览器运行代码，同时在代码块下方展示运行结果.
3. 以富媒体格式展示计算结果。富媒体格式包括：HTML, LaTeX, PNG, SVG 等.
4. 对代码编写说明文档或语句时，支持 Markdown 语法.
5. 支持使用 LaTeX 编写数学性说明.

1.2 Jupyter Notebook: 试用与安装

打开官网<https://jupyter.org>, 你就可以看到 Jupyter Notebook，它给出了两个链接:1. 试用，2. 安装. 你可以先免安装试用体验一下，提前感受下 Jupyter Notebook 的特殊之处 (值得注意的是，有时官方提供的并发试用是有限的，就是同一个时间点的试用人数是有限制的。)



按照 Jupyter Notebook 官方文档的安装方式，有两种：1. conda 2. pip。但是官方还是推荐使用 conda 包管理方式安装：



为什么要推荐是 conda，总结了这几点：

1、解决繁琐的环境与包的问题

在 Python 中，最常用的包管理工具是 pip，它可以很方便帮我们解决依赖问题。但是在某些情况下安装一些包，你会遇到各种安装错误使你焦头烂额。而使用 Anaconda 很多包都已经集成好了（还内置 conda 包管理工具，类似 pip），直接用就行了。省事又省心。而且还可以通过可视化的界面进行包的管理。

2、可以方便地配置多个环境

在不同的需求下，有时候你需要在不同的环境中使用 Python 的不同版本，如果你不会使用虚拟环境，也不会使用 Pycharm 的话，可以试一试 Anaconda，它可以很方便地实现环境的隔离和切换。

3、内置很多数据分析实用工具

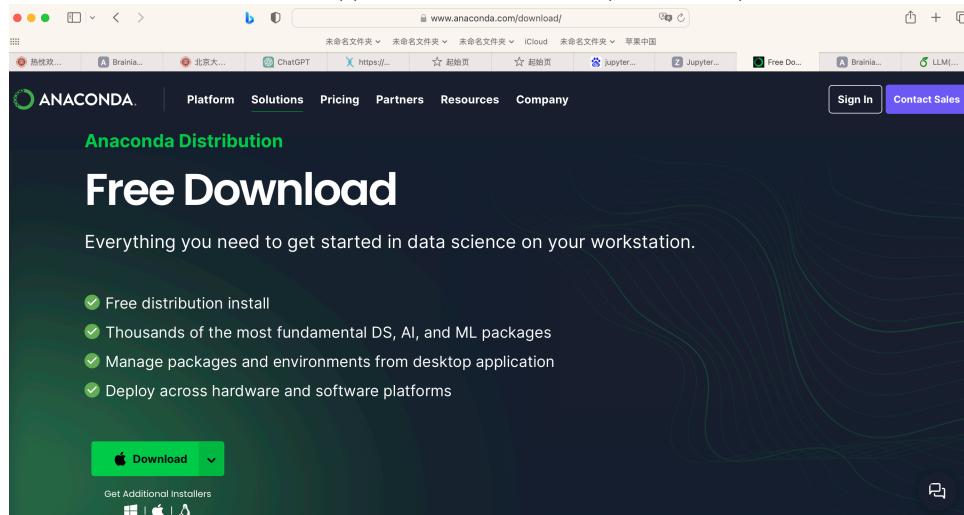
Anaconda 号称是适用于企业级大数据分析的 Python 工具。它包含了 720 多个数据科学相关的开源包（如 Pandas，Scipy 等），在数据可视化、机器学习、深度学习等多方面都有涉及。

4、内置了 Jupyter NoteBook 工具

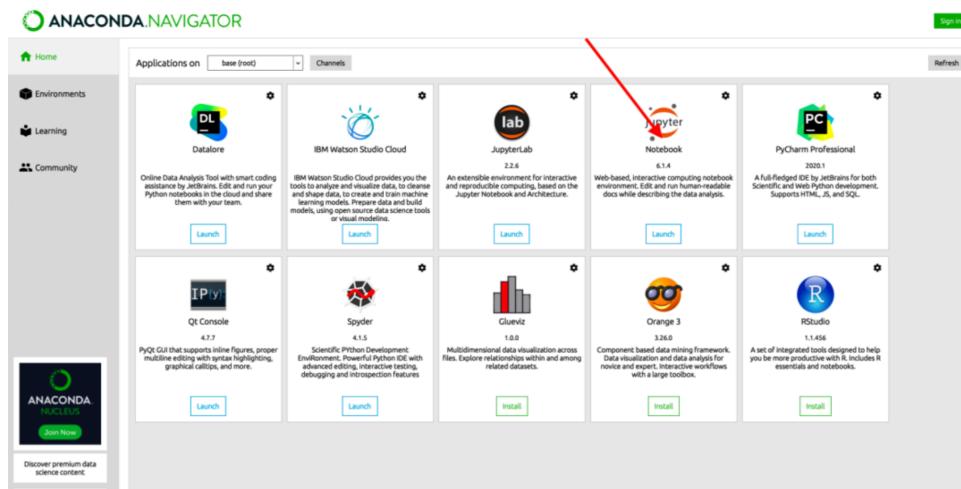
有了这个工具，以后你不管是开发，调试，还是记录学习笔记，都可以在上面完成。

安装步骤（Anaconda 方式）：

1. 到官网下载 Anaconda：<https://www.anaconda.com/download/>，选择对应操作系统的文件。



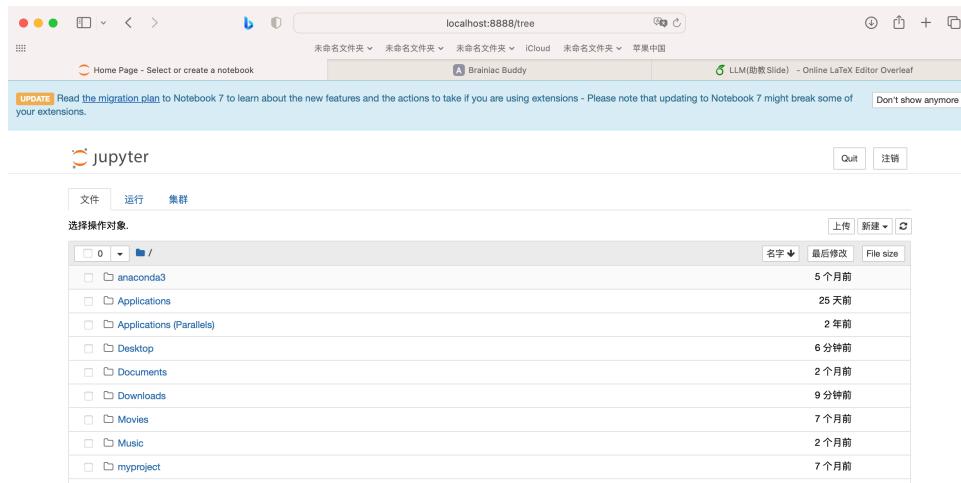
2. 安装完成之后程序部分就会出现一个 Anaconda 的启动器，启动之后页面如下，选择 Jupyter Notebook，点击 Launch.



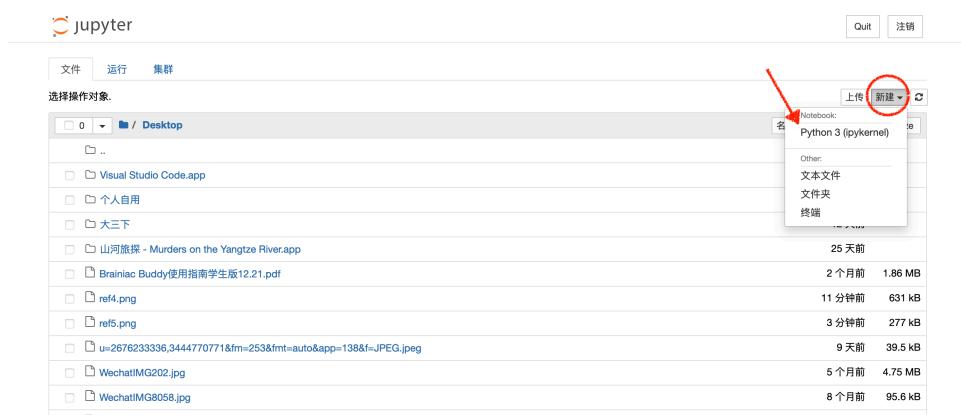
当然也可以在终端直接输入 `jupyter notebook` 进行启动，默认终端打开页面在哪里，打开的网页页面就在哪里。

1.3 Jupyter Notebook: 使用与调试

登陆进来后，我们将看到如下的界面，这便是 Jupyter Notebook 的主界面（文件夹树），一般对应的是用户目录的文档文件夹，可以根据自己的使用习惯在对应的文件夹中新建项目（推荐 Desktop）。



点击“新建 python3(ipykernel)”按钮，将新建一个拓展名为”.ipynb”的文档文件。

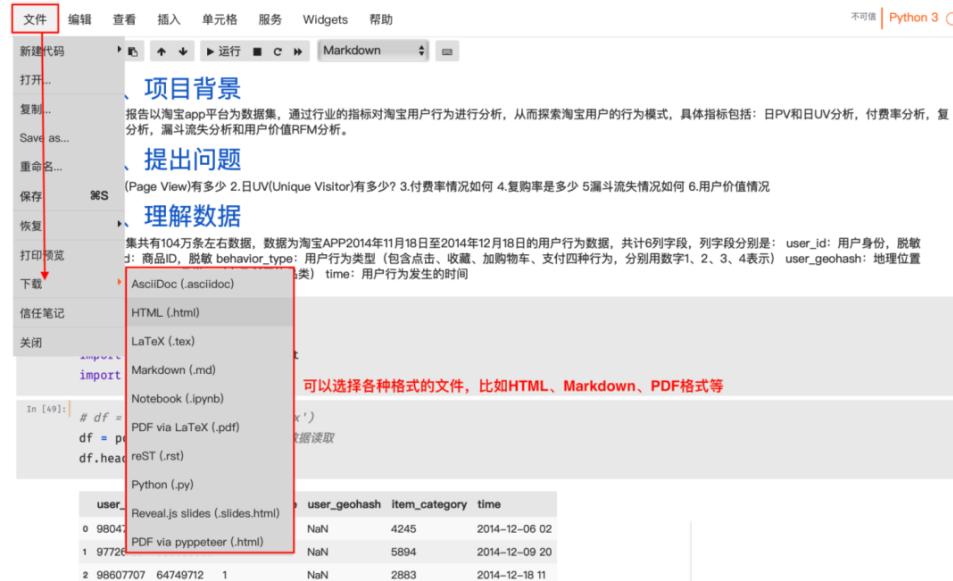


就会看到如下页面，输入简单的 Python 代码试一试。

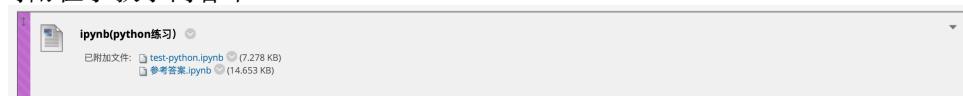


每一行的命令，都在一个可编辑的输入框。即使你的命令输入有误，你也不用从头写代码，直接编辑后重新执行即可。还有一点，命令的执行是可间断的，某个命令执行错误，不会导致整个程序中断，这将很方便我们调试代码，只要改完代码，再重新执行该行代码即可，而不用重新执行全部代码。

如果你的代码任务完成，还可以导出你的所有代码，具体操作如下：



在我们课程教学网“教学内容”中，已经附有了一个已经创建好的 ipynb 文档，请同学们自行尝试练习使用 Jupyter Notebook 打开 ipynb 文档并且尝试完成里边的习题。本次练习不计分，参考答案同时附在了教学内容中。



1.4 Jupyter Notebook: 参考资料与高级用法

我们将不在这个 Notes 中提到 Jupyter Notebook 的任何高级用法，这里有两个叙述的比较好的推文链接，有兴趣的同学可以用来参考：

1. 一些快捷命令:前置机器学习 (二): 30 分钟掌握常用 Jupyter Notebook 用法
2. 一些常用拓展:Jupyter Notebook 的 10 个常用扩展介绍

1.5 PyTorch: 功能简介与安装

PyTorch 是一个由 Facebook 的人工智能研究团队开发的开源深度学习框架。在 2016 年发布后，PyTorch 很快就因其易用性、灵活性和强大的功能而在科研社区中广受欢迎。下面我们将详细介绍 PyTorch 的发展历程。



在 2016 年，Facebook 的 AI 研究团队（FAIR）公开了 PyTorch，其旨在提供一个快速，灵活且动态的深度学习框架。PyTorch 的设计哲学与 Python 的设计哲学非常相似：易读性和简洁性优于隐式的复杂性。PyTorch 用 Python 语言编写，是 Python 的一种扩展，这使得其更易于学习和使用。

在发布后的几年里，PyTorch 迅速在科研社区中取得了广泛的认可。在 2019 年，PyTorch 发布了 1.0 版本，引入了一些重要的新功能，包括支持 ONNX、一个新的分布式包以及对 C++ 的前端支持等。这些功能使得 PyTorch 在工业界的应用更加广泛，同时也保持了其在科研领域的强劲势头。

到了近两年，PyTorch 已经成为全球最流行的深度学习框架之一。其在 GitHub 上的星标数量超过了 50k，被用在了各种各样的项目中，从最新的研究论文到大规模的工业应用。

接下来我们就来详细地探讨一下 PyTorch 的优点。

1. 动态计算图

PyTorch 最突出的优点之一就是它使用了动态计算图 (Dynamic Computation Graphs, DCGs)，与 TensorFlow 和其他框架使用的静态计算图不同。动态计算图允许你在运行时更改图的行为。这使得 PyTorch 非常灵活，在处理不确定性或复杂性时具有优势，因此非常适合研究和原型设计。

2. 易用性

PyTorch 被设计成易于理解和使用。其 API 设计的直观性使得学习和使用 PyTorch 成为一件非常愉快的事情。此外，由于 PyTorch 与 Python 的深度集成，它在 Python 程序员中非常流行。

3. 易于调试

由于 PyTorch 的动态性和 Python 性质，调试 PyTorch 程序变得相当直接。你可以使用 Python 的标准调试工具，如 PDB 或 PyCharm，直接查看每个操作的结果和中间变量的状态。

4. 强大的社区支持

PyTorch 的社区非常活跃和支持。官方论坛、GitHub、Stack Overflow 等平台上有大量的 PyTorch 用户和开发者，你可以从中找到大量的资源和帮助。

5. 广泛的预训练模型

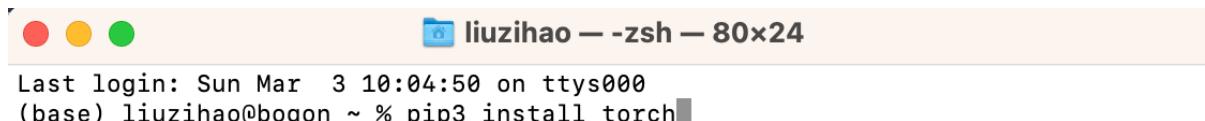
PyTorch 提供了大量的预训练模型，包括但不限于 ResNet，VGG，Inception，SqueezeNet，EfficientNet 等等。这些预训练模型可以帮助你快速开始新的项目。

6. 高效的 GPU 利用

PyTorch 可以非常高效地利用 NVIDIA 的 CUDA 库来进行 GPU 计算。同时，它还支持分布式计算，让你可以在多个 GPU 或服务器上训练模型。

综上所述，PyTorch 因其易用性、灵活性、丰富的功能以及强大的社区支持，在深度学习领域中备受欢迎。

Torch 模块的安装也是简单的，由于我们试图在 Jupyter Notebook 中使用 Torch，我们只需要在中短程应用 `pip/pip3 install torch` 技巧安装 torch 库即可。



```
Last login: Sun Mar  3 10:04:50 on ttys000
(base) liuzihao@bogon ~ % pip3 install torch
```

1.6 PyTorch: 基本代码实现

在我们开始深入使用 PyTorch 之前，让我们先了解一些基础概念和操作。这一部分将涵盖 PyTorch 的基础，包括 tensor 操作、GPU 加速以及自动求导机制。

1.6.1 Tensor 操作

Tensor 是 PyTorch 中最基本的数据结构，你可以将其视为多维数组或者矩阵。PyTorch tensor 和 NumPy array 非常相似，但是 tensor 还可以在 GPU 上运算，而 NumPy array 则只能在 CPU 上运算。下面，我们将介绍一些基本的 tensor 操作。

首先，我们需要导入 PyTorch 库: `import torch`

然后，我们可以创建一个新的 tensor。以下是一些创建 tensor 的方法:

```
In [3]: import torch
# 创建一个未初始化的5x3矩阵
x = torch.empty(5, 3)
print(x)

# 创建一个随机初始化的5x3矩阵
x = torch.rand(5, 3)
print(x)

# 创建一个5x3的零矩阵, 类型为long
x = torch.zeros(5, 3, dtype=torch.long)
print(x)

# 直接从数据创建tensor
x = torch.tensor([5.5, 3])
print(x)
```

我们还可以对已有的 tensor 进行操作。以下是一些基本操作:

```
In [4]: # 创建一个tensor, 并设置requires_grad=True以跟踪计算历史
x = torch.ones(2, 2, requires_grad=True)
print(x)

# 对tensor进行操作
y = x + 2
print(y)

# y是操作的结果, 所以它有grad_fn属性
print(y.grad_fn)

# 对y进行更多操作
z = y * y * 3
out = z.mean()

print(z, out)
```

在 PyTorch 中，我们可以使用`.backward()`方法来计算梯度。例如:

```
In [5]: # 因为out包含一个标量, out.backward() 等价于out.backward(torch.tensor(1.))
out.backward()

# 打印梯度 d(out)/dx
print(x.grad)
```

以上是 PyTorch tensor 的基本操作，我们可以看到 PyTorch tensor 操作非常简单和直观。在后续的学习中，我们将会使用到更多的 tensor 操作，例如索引、切片、数学运算、线性代数、随机数等等。

1.6.2 自动求导

在深度学习中，我们经常需要进行梯度下降优化。这就需要我们计算梯度，也就是函数的导数。在 PyTorch 中，我们可以使用自动求导机制（autograd）来自动计算梯度。

在 PyTorch 中，我们可以设置 `tensor.requires_grad=True` 来追踪其上的所有操作。完成计算后，我们可以调用 `.backward()` 方法，PyTorch 会自动计算和存储梯度。这个梯度可以通过 `.grad` 属性进行访问。

下面是一个简单的示例：

```
In [6]: import torch

# 创建一个tensor并设置requires_grad=True来追踪其计算历史
x = torch.ones(2, 2, requires_grad=True)

# 对这个tensor做一次运算:
y = x + 2

# y是计算的结果, 所以它有grad_fn属性
print(y.grad_fn)

# 对y进行更多的操作
z = y * y * 3
out = z.mean()

print(z, out)

# 使用.backward()来进行反向传播, 计算梯度
out.backward()

# 输出梯度d(out)/dx
print(x.grad)
```

以上示例中，`out.backward()` 等同于 `out.backward(torch.tensor(1.))`。如果 `out` 不是一个标量，因为 `tensor` 是矩阵，那么在调用 `.backward()` 时需要传入一个与 `out` 同形的权重向量进行相乘。

例如：

```
In [7]: x = torch.randn(3, requires_grad=True)

y = x * 2
while y.data.norm() < 1000:
    y = y * 2

print(y)

v = torch.tensor([0.1, 1.0, 0.0001], dtype=torch.float)
y.backward(v)

print(x.grad)
```

以上就是 PyTorch 中自动求导的基本使用方法。自动求导是 PyTorch 的重要特性之一，它为深度学习模型的训练提供了极大的便利。

1.7 PyTorch 高级用法：训练神经网络/数据集处理/全连接神经网络实战

1.7.1 训练神经网络原理简介

PyTorch 提供了 `torch.nn` 库，它是用于构建神经网络的工具库。`torch.nn` 库依赖于 `autograd` 库来定义和计算梯度。`nn.Module` 包含了神经网络的层以及返回输出的 `forward(input)` 方法。

以下是一个简单的神经网络的构建示例：

```
In [8]: import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

    # 输入图像channel: 1, 输出channel: 6, 5x5卷积核
    self.conv1 = nn.Conv2d(1, 6, 5)
    self.conv2 = nn.Conv2d(6, 16, 5)

    # 全连接层
    self.fc1 = nn.Linear(16 * 5 * 5, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # 使用2x2窗口进行最大池化
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # 如果窗口是方的, 只需要指定一个维度
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)

        x = x.view(-1, self.num_flat_features(x))

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x

    def num_flat_features(self, x):
        size = x.size()[1:] # 获取除了batch维度之外的其他维度
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

net = Net()
print(net)
```

以上就是一个简单的神经网络的构建方法。我们首先定义了一个 Net 类, 这个类继承自 nn.Module。然后在 `__init__` 方法中定义了网络的结构, 在 `forward` 方法中定义了数据的流向。在网络的构建过程中, 我们可以使用任何 tensor 操作。

需要注意的是, `backward` 函数 (用于计算梯度) 会被 autograd 自动创建和实现。你只需要在 nn.Module 的子类中定义 `forward` 函数。

在创建好神经网络后, 我们可以使用 `net.parameters()` 方法来返回网络的可学习参数。

1.7.2 数据集处理

除了模型设计之外, 数据的加载和处理也是非常重要的一部分。

PyTorch 提供了 `torch.utils.data.DataLoader` 类, 可以帮助我们方便地进行数据的加载和处理。

`DataLoader` 类提供了对数据集的并行加载, 可以有效地加载大量数据, 并提供了多种数据采样方式。常用的参数有:

`dataset`: 加载的数据集 (Dataset 对象)

`batch_size`: batch 大小

`shuffle`: 是否每个 epoch 时都打乱数据

`num_workers`: 使用多进程加载的进程数, 0 表示不使用多进程

以下是一个简单的使用示例:

```

)
[*]: from torch.utils.data import DataLoader
      from torchvision import datasets, transforms

# 数据转换
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# 下载并加载训练集
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=4, shuffle=True, num_workers=2)

# 下载并加载测试集
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = DataLoader(testset, batch_size=4, shuffle=False, num_workers=2)

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
  3%|██████████| 4489216/170498071 [00:15<09:22, 295259.94it/s]

```

除了使用内置的数据集，我们也可以自定义数据集。自定义数据集需要继承 Dataset 类，并实现 `__len__` 和 `__getitem__` 两个方法（具体略）。

在深度学习模型的训练过程中，我们经常需要保存模型的参数以便于将来重新加载。这对于中断的训练过程的恢复，或者用于模型的分享和部署都是非常有用的。

PyTorch 提供了简单的 API 来保存和加载模型。最常见的方法是使用 `torch.save` 来保存模型的参数，然后通过 `torch.load` 来加载模型的参数（具体略）。

1.7.3 全连接神经网络实战

人工神经网络 (ANN) 简称神经网络，可以对一组输入信号和一组输出信号之间的关系进行建模，是机器学习和认知科学领域中一种模仿生物神经网络的结构和功能的数学模型。用于对函数进行估计或近似，其灵感来源于动物的中枢神经系统，特别是大脑。神经网络由大量的人工神经元（或节点）联结进行计算，大多数情况下人工神经网络能在外界信息的基础上改变内部结构，是一种自适应系统。

在具有 n 个输入一个输出的单一的神经元模型中，神经元接收到来自 n 个其他神经元传递过来的输入信号，这些输入信号通过带权重的连接进行传递，神经元收到的总输入值将经过激活函数 f 处理后产生神经元的输出。

全连接神经网络 (Multi-Layer Perception, MLP) 或者叫多层感知机，是一种连接方式较为简单的人工神经网络，属于前馈神经网络的一种，主要由输入层、隐藏层和输出层构成，并且在每个隐藏层中可以有多个神经元。MLP 网络是可以应用于几乎所有任务的多功能学习方法，包括分类、回归，甚至是无监督学习。

神经网络的学习能力主要来源于网络结构，而且根据层的数量不同、每层神经元数量的多少，以及信息在层之间的传播方式，可以组合成多种神经网络模型。全连接神经网络主要由输入层、隐藏层和输出层构成。输入层仅接收外界的输入，不进行任何函数处理，所以输入层的神经元个数往往和输入的特征数量相同，隐藏层和输出层神经元对信号进行加工处理，最终结果由输出层神经元输出。根据隐藏层的数量可以分为单隐藏层 MLP 和多隐藏层 MLP。

针对单隐藏层 MLP 和多隐藏层 MLP，每个隐藏层的神经元数量是可以变化的，通常没有一个很好的标准用于确定每层神经元的数量和隐藏层的个数。根据经验，更多的神经元就会有更强的表达能力，同时更容易造成网络的过拟合，所以在使用全连接神经网络时，对模型泛化能力的测试很重要，最好的方式是在训练模型时使用验证集来验证模型的泛化能力，且尽可能地去尝试多种网络结构，以寻找更好的模型，但这往往需要耗费大量的时间。

以下是一个使用 PyTorch 构建的简单的全连接神经网络的示例代码，它可以用于分类任务。我将在代码之后详细解释每个部分的作用：

```
[*]: import torch
import torch.nn as nn
import torch.nn.functional as F

# 定义网络结构
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 全连接层1： 输入维度是784（比如MNIST数据集的图片为28x28=784）， 输出维度是512
        self.fc1 = nn.Linear(784, 512)
        # 全连接层2： 输入维度是512， 输出维度是256
        self.fc2 = nn.Linear(512, 256)
        # 全连接层3： 输入维度是256， 输出维度是10（假设我们有10个分类）
        self.fc3 = nn.Linear(256, 10)

    # 前向传播
    def forward(self, x):
        # 将图片展平为一维向量
        x = x.view(-1, 784)
        # 通过第一个全连接层后使用ReLU激活函数
        x = F.relu(self.fc1(x))
        # 通过第二个全连接层后使用ReLU激活函数
        x = F.relu(self.fc2(x))
        # 通过第三个全连接层，不使用激活函数，因为这是输出层
        x = self.fc3(x)
        return x

# 实例化网络
net = Net()

# 打印网络结构
print(net)
```

代码解读：

1. 首先，我们导入了 PyTorch 中的必要模块：torch, torch.nn, 和 torch.nn.functional。
2. 定义了一个名为 Net 的类，它继承自 nn.Module。这是创建任何 PyTorch 神经网络的基础。
3. 在 Net 类的构造函数 `__init__` 中，我们使用 `super()` 来调用父类 nn.Module 的构造函数，然后定义了三个全连接层（也称为线性层）。
4. `self.fc1` 是第一个全连接层，它接受一个 784 维的输入（例如来自 28x28 像素的 MNIST 手写数字图像），并输出一个 512 维的向量。
5. `self.fc2` 是第二个全连接层，它将 512 维的向量转换为 256 维的向量。
6. `self.fc3` 是第三个也是最后一个全连接层，它将 256 维的向量转换为 10 维的向量，这假设我们有 10 个类别进行分类。
7. 在 `forward` 方法中，我们定义了数据如何通过网络。`forward` 方法是必须要重写的，因为它指定了数据的前向传播路径。
8. 输入数据 `x` 首先被展平为一维向量。
9. 接下来，数据通过第一个全连接层，并应用 ReLU 激活函数。
10. 然后，经过第二个全连接层，并再次应用 ReLU 激活函数。
11. 最后，数据通过第三个全连接层。因为这是输出层，所以我们不在这里应用激活函数（在分类任务中，通常在计算损失时会应用 softmax 函数）。
12. 最后，我们实例化了 Net 类的一个对象，并打印出网络结构。

请注意，这只是创建网络的代码。要训练网络，往往需要定义损失函数和优化器，并编写训练循环代码来更新网络的权重。此外，对于分类任务，通常在计算损失之前会对网络的输出应用 softmax 函数，下边给一个完整任务的参考代码：

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

# 设置随机种子以便结果可复现
torch.manual_seed(0)

# 创建随机数据集
input_size = 10 # 输入特征的数量
hidden_size = 5 # 隐藏层神经元的数量
output_size = 2 # 输出类别的数量
batch_size = 16 # 批量大小
data_size = 1000 # 数据集大小

# 生成随机特征和标签
# 特征是在一个正态分布中随机抽取的
# 标签是随机整数0或1
features = torch.randn(data_size, input_size)
labels = torch.randint(0, output_size, (data_size,))

# 创建数据集和数据加载器
dataset = TensorDataset(features, labels)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# 定义全连接神经网络
class FullyConnectedNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(FullyConnectedNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# 创建模型实例
model = FullyConnectedNN(input_size, hidden_size, output_size)

# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 训练网络
epochs = 5
for epoch in range(epochs):
    for batch_features, batch_labels in dataloader:
        # 正向传播
        outputs = model(batch_features)
        loss = criterion(outputs, batch_labels)

        # 反向传播和优化
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item()}")

# 测试网络性能
correct = 0
total = 0
with torch.no_grad():
    for batch_features, batch_labels in dataloader:
        outputs = model(batch_features)
        _, predicted = torch.max(outputs.data, 1)
        total += batch_labels.size(0)
        correct += (predicted == batch_labels).sum().item()

print(f'Accuracy of the network on the {data_size} test inputs: {100 * correct / total}
```

这个代码片段是使用 Python 编程语言中的 PyTorch 库来实现的一个简单的全连接神经网络。我会逐步解释这段代码，以便新手也能理解其工作原理。

首先，代码引入了必要的库。torch 是 PyTorch 的核心库，提供了构建神经网络所需的数据结构和操作。torch.nn 模块包含了构建网络层的类和函数。torch.optim 包含了优化算法，如 Adam，用于更新网络中的权重。DataLoader 和 TensorDataset 用于创建和管理数据集。

然后，设置了一个随机种子，以确保实验的可重复性。随机种子是随机数生成算法的起始点，固定它可以保证每次运行代码时生成的随机数是相同的。

接着，我们定义了一些网络和数据的参数，包括输入层大小、隐藏层大小、输出层大小、批量大小和数据集的大小。然后生成了随机的特征 (features) 和标签 (labels)。特征是标准正态分布中的随机值，标签是 0 和 1 之间的随机整数。

下面，我们创建了一个 TensorDataset 对象，它将特征和标签封装到一个数据集中，然后创建了一个 DataLoader 对象，它会在训练时提供批量数据，并且每次提供数据时会打乱数据顺序，以便于训练过程的随机性。

然后，定义了一个名为 FullyConnectedNN 的神经网络类，它继承自 nn.Module。在初始化方法、`__init__` 中，定义了两个全连接层 (fc1 和 fc2) 以及一个激活函数 ReLU。forward 方法定义了数据通过网络的正向传播方式。

接着，创建了 FullyConnectedNN 的一个实例，定义了交叉熵损失函数 criterion，它通常用于分类问题，并定义了用于训练的优化器 optimizer，这里使用了 Adam 优化算法。

下面的循环代码是训练过程的核心。它首先迭代多个训练周期 (epochs)，在每个周期中，网络会在批量数据上进行训练。网络的输出 (outputs) 通过损失函数计算损失 (loss)。然后执行反向传播 (`loss.backward()`)，计算梯度，并通过优化器更新权重 (`optimizer.step()`)。

最后，代码测试了网络的性能，计算了在整个数据集上的准确率。使用 `torch.no_grad()` 来确保在测试过程中不会计算梯度。通过比较模型的预测和真实标签来计算正确预测的数量和总预测数量，从而得到准确率。

这段代码是一个很好的起点，但是请记住，实际应用中的神经网络通常要复杂得多，并且会使用实际的数据集进行训练和测试。

Pytorch 部分的代码也已附在了教学网上，请大家查收。(Last updated:2023.3.2)