Pytorch神经网络基础知识

之前首次介绍神经网络时，我们关注的是具有单一输出的线性模型。在这里，整个模型只有一个输出。注意，单个神经网络（1）接受一些输入；（2）生成相应的标量输出；（3）具有一组相关 参数（parameters），更新这些参数可以优化某目标函数。然后，当考虑具有多个输出的网络时，我们利用矢量化算法来描述整层神经元。像单个神经元一样，层（1）接受一组输入，（2）生成相应的输出，（3）由一组可调整参数描述。

当我们使用**softmax回归**时，一个单层本身就是模型。然而，即使我们随后引入了多层感知机，我们仍然可以认为该模型保留了上面所说的基本架构。

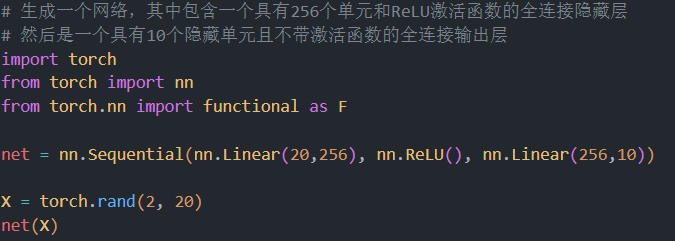
对于**多层感知机**而言，整个模型及其组成层都是这种架构。整个模型接受原始输入（特征），生成输出（预测），并包含一些参数（所有组成层的参数集合）。同样，每个单独的层接收输入（由前一层提供），生成输出（到下一层的输入），并且具有一组可调参数，这些参数根据从下一层反向传播的信号进行更新。

1. 层和块

**层是神经网络中的基本构建块**，负责特定的计算功能。常见的层包括：**全连接层、卷积层、激活层、归一化层**

**块通常指多个层的组合**，形成一个更复杂的结构。块可以是自定义的，也可以是已有模块的组合。常见的块包括：**残差块（Residual Block）、序列块（Sequential Block）（将多个层按顺序组合）**

块可以描述单个层、由多个层组成的组件或整个模型本身。



这里通过实例化nn.Sequential来构建模型，层的执行顺序是作为参数传递的。简而言之，nn.Sequential定义了一种特殊的Module，即在PyTorch中表示一个块的类，它维护了一个由Module组成的有序列表。（注：两个全连接层都是Linear类的实例，Linear类本身就是Module的子类）net(X)实际上是net.\_\_call\_\_(X)的简写。这个前向传播函数非常简单：它将列表中的每个块连接在一起，将每个块的输出作为下一个块的输入

1.1．自定义块



1. 将输入数据作为其前向传播函数的参数。

2. 通过前向传播函数来生成输出。（注：输出的形状可能与输入的形状不同）。例如，上面模型中的第一个全连接的层接收一个20维的输入，但是返回一个维度为256的输出。

3. 计算其输出关于输入的梯度，可通过其反向传播函数进行访问。通常这是自动发生的。

4. 存储和访问前向传播计算所需的参数。

5. 根据需要初始化模型参数。

上面代码定义了一个块。它包含一个多层感知机，其具有256个隐藏单元的隐藏层和一个10维输出层。（注：下面的MLP类继承了表示块的类）。我们实现只需要提供我们自己的构造函数（Python中的\_\_init\_\_函数）和前向传播函数。 这里前向传播函数以X作为输入，计算带有激活函数的隐藏表示，并输出其未规范化的输出值。在这个MLP实现中，两个层都是实例变量。接着我们实例化多层感知机的层，然后在每次调用前向传播函数时调用这些层。首先，我们定制的\_\_init\_\_函数通过super().\_\_init\_\_() 调用父类的\_\_init\_\_函数，省去了重复编写模版代码的痛苦。然后，我们实例化两个全连接层，分别为self.hidden和self.out。

**块的一个主要优点是它的多功能性。我们可以子类化块以创建层（如全连接层的类）、整个模型（如上面的MLP类）或具有中等复杂度的各种组件**

（任何一个层和神经网络都是Module的一个子类，可以继承一些好用的函数）

在深度学习中，"Module" 通常指的是一个可重用的组件或层，通常用于构建神经网络。在PyTorch中，torch.nn.Module 是所有神经网络模块的基类。用户可以通过继承这个类来定义自己的模型。

1.2．顺序块



Sequential的设计是为了把其他模块串起来。为了构建我们自己的简化的MySequential，我们只需要定义两个关键函数：

1. 一种将块逐个追加到列表中的函数；（\_init\_函数）

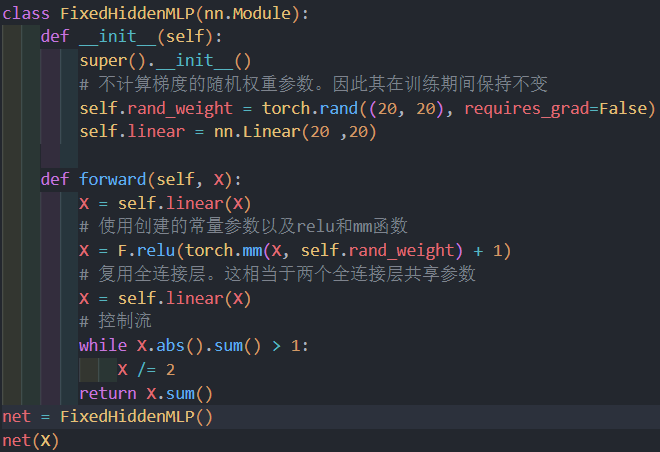
2. 一种前向传播函数，用于将输入按追加块的顺序传递给块组成的“链条”。

下面的MySequential类提供了与默认Sequential类相同的功能。

\_\_init\_\_函数将每个模块逐个添加到有序字典\_modules中。\_modules的主要优点是：在模块的参数初始化过程中，系统知道在\_modules字典中查找需要初始化参数的子块。当MySequential的前向传播函数被调用时，每个添加的块都按照它们被添加的顺序执行。

1.3. 在前向传播函数中执行代码

Sequential类使模型构造变得简单，允许我们组合新的架构，而不必定义自己的类。然而，并不是所有的架构都是简单的顺序架构。当需要更强的灵活性时，我们需要定义自己的块。有时我们可能希望合并既不是上一层的结果也不是可更新参数的项，我们称之为常数参数（constant parameter）。例如，我们需要

一个计算函数 f(**x**, **w**) = c · **w**⊤**x**的层，其中**x**是输入，**w**是参数，c是某个在优化过程中没有更新的指定常量。

在这个FixedHiddenMLP模型中，我们实现了一个隐藏层，其权重（self.rand\_weight）在实例化时被随机初始化，之后为常量。这个权重不是一个模型参数，因此它永远不会被反向传播更新。然后，神经网络将这个固定层的输出通过一个全连接层。

注意，在返回输出之前，模型做了一些不寻常的事情：它运行了一个while循环，在L1范数大于1的条件下，将输出向量除以2，直到它满足条件为止。最后，模型返回了X中所有项的和。

总结：

1．一个块可以由许多层组成；一个块可以由许多块组成。

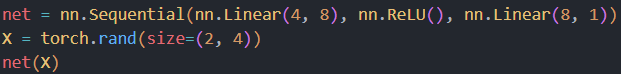
2．块可以包含代码。

3．块负责大量的内部处理，包括参数初始化和反向传播。

4．层和块的顺序连接由Sequential块处理。

2．参数管理

在选择了架构并设置了超参数后，我们就进入了训练阶段。此时，我们的目标是找到使损失函数最小化的模型参数值。经过训练后，我们将需要使用这些参数来做出未来的预测。此外，有时我们希望提取参数，以便在其他环境中复用它们，将模型保存下来，以便它可以在其他软件中执行，或者为了获得科学的理解而进行检查



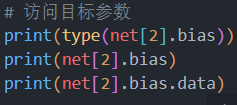
单隐藏层的多层感知机

2.1 参数访问



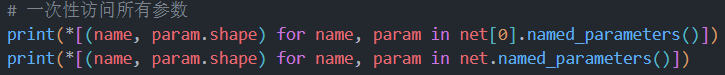


从已有模型中访问参数。当通过Sequential类定义模型时，我们可以通过索引来访问模型的任意层。这就像模型（net）是一个列表一样，每层的参数都在其属性中。这个全连接层包含两个参数，分别是该层的权重和偏置。两者都存储为单精度浮点数（float32）。（注：参数名称允许唯一标识每个参数，即使在包含数百个层的网络中也是如此）

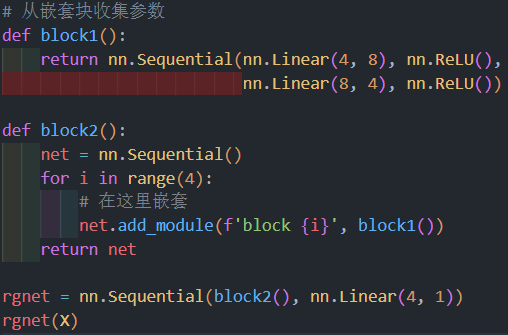


注：每个参数都表示为参数类的一个实例。要对参数执行任何操作，首先我们需要访问底层的数值

这里从第二个全连接层（即第三个神经网络层）提取偏置，提取后返回的是一个参数类实例，并进一步访问该参数的值。参数是复合的对象，包含值、梯度和额外信息。这就是我们需要显式参数值的原因。



当我们需要对所有参数执行操作时，逐个访问它们可能会很麻烦。当我们处理更复杂的块（例如，嵌套块）时，情况可能会变得特别复杂，因为我们需要递归整个树来提取每个子块的参数。

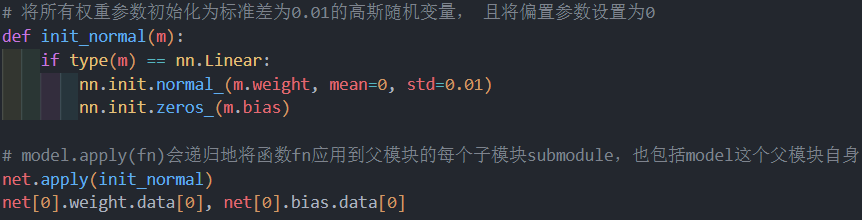


定义一个生成块的函数（可以说是“块工厂”），然后将这些块组合到更大的块中。将复杂网络模块化，有利于查找和访问。

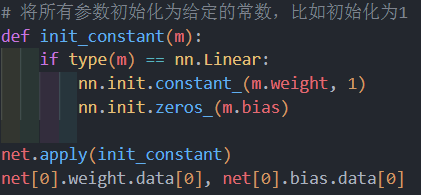
2.2 参数初始化

深度学习框架提供默认随机初始化，也允许我们创建自定义初始化方法，满足我们通过其他规则实现初始化权重。默认情况下，PyTorch会根据一个范围均匀地初始化权重和偏置矩阵，这个范围是根据输入和输出维度计算出的。PyTorch的nn.init模块提供了多种预置初始化方法。

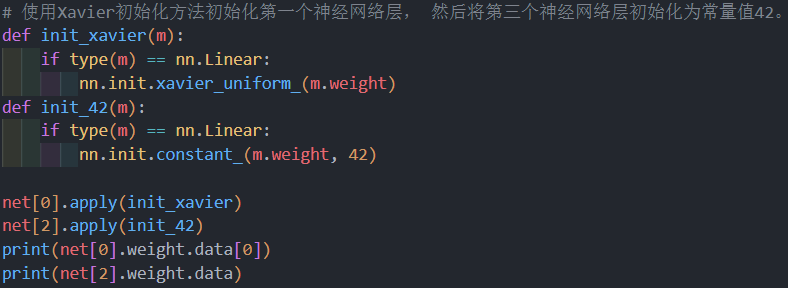
内置初始化



将所有权重参数初始化为标准差为0.01的高斯随机变量，且将偏置参数设置为0。



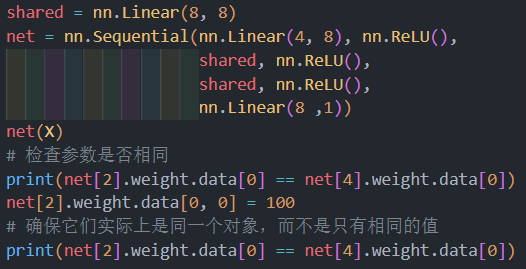
将所有参数初始化为给定的常数，比如初始化为1。



使用Xavier初始化（权重被初始化为一个均匀分布或正态分布，其方差与输入和输出的节点数量有关）方法初始化第一个神经网络层，然后将第三个神经网络层初始化为常量值42。 （**可以实现不同层不同的初始化**）

自定义初始化

2.3 参数绑定



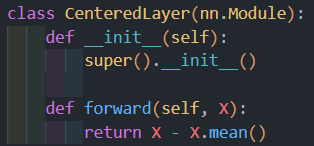
定义一个稠密层，然后使用它的参数来设置另一个层的参数。

第三个和第五个神经网络层的参数是绑定的。它们不仅值相等，而且由相同的张量表示。因此如果我们改变其中一个参数，另一个参数也会改变。由于模型参数包含梯度，因此在反向传播期间第二个隐藏层（即第三个神经网络层）和第三个隐藏层（即第五个神经网络层）的梯度会加在一起。

3. 自定义层

深度学习成功背后的一个因素是神经网络的灵活性：可以用创造性的方式组合不同的层，从而设计出适用于各种任务的架构。

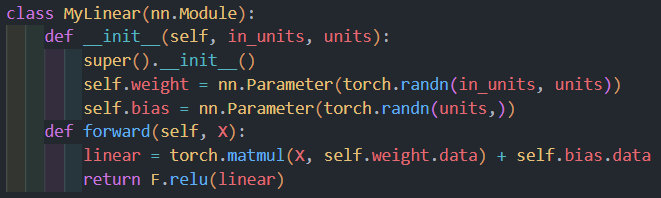
不带参数的层



构造一个没有任何参数的自定义层。上面的CenteredLayer类要从其输入中减去均值，使得均值变为0。

带参数的层

可以使用内置函数来创建参数，这些函数提供一些基本的管理功能。比如管理访问、初始化、共享、保存和加载模型参数。这样做的好处之一是：我们不需要为每个自定义层编写自定义的序列化程序。



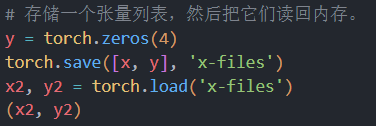
实现自定义版本的全连接层。该层需要两个参数，一个用于表示权重，另一个用于表示偏置项(用nn.Parameter定义)。在此实现中，我们使用修正线性单元作为激活函数。该层需要输入参数：in\_units和units，分别表示输入数和输出数。

4．读写文件

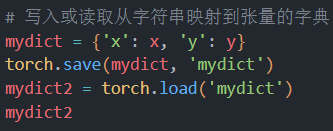
为了保存训练的模型，以备将来在各种环境中使用（比如在部署中进行预测）。此外，当运行一个耗时较长的训练过程时，最佳的做法是定期保存中间结果，以确保在服务器电源被不小心断掉时，我们不会损失几天的计算结果。

4.1．加载和保存张量

对于单个张量，我们可以直接调用load和save函数分别读写它们。这两个函数都要求我们提供一个名称，save要求将要保存的变量作为输入。

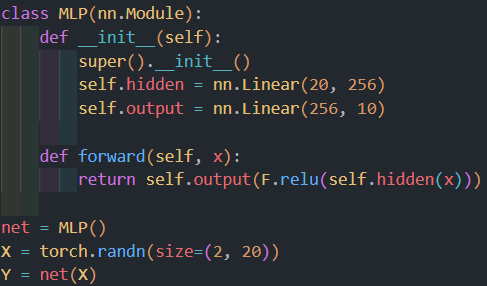


存储一个张量列表，然后把它们读回内存。



可以写入或读取从字符串映射到张量的字典。当我们要读取或写入模型中的所有权重时，这很方便。

4.2．加载和保存模型参数



深度学习框架提供了内置函数来保存和加载整个网络。需要注意的一个重要细节是，这将保存模型的参数而不是保存整个模型。例如，如果我们有一个3层多层感知机，我们需要单独指定架构。因为模型本身可以包含任意代码，所以模型本身难以序列化。因此，为了恢复模型，我们需要用代码生成架构，然后从磁盘加载参数。



为了恢复模型，我们实例化了原始多层感知机模型的一个备份。这里我们不需要随机初始化模型参数，而是直接读取文件中存储的参数。