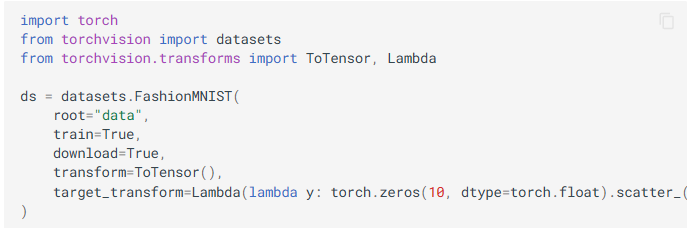
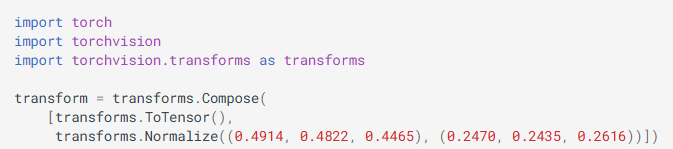
**1.PyTorch有两个[处理数据的基本操作](https://pytorch.org/docs/stable/data.html)：**torch.utils.data.DataLoader和torch.utils.data.Dataset。

我们将Dataset作为参数传递给DataLoader。这将在我们的数据集上包装一个迭代器，并支持自动批处理、采样、随机打乱和多进程数据加载。



每个Dataset都包含两个参数: transform 和 target\_transform，分别用于转换样本和标签。我们使用 ToTensor （将 PIL 图像或 NumPy 的 ndarray 转换为 FloatTensor）和 Lambda（把整数变成一个独热(one-hot)编码的tensor）。





transforms.ToTensor() 将 Pillow 加载的图像转换成 [PyTorch](https://pytorch.apachecn.org/2.0/tutorials/beginner/introyt/introyt1_tutorial/) tensor。

transforms.Normalize() 调整tensor值，使其平均值为零，标准差为 1.0。大多数激活函数在 x = 0 附近具有最强梯度，因此将数据集中在这里可以加快学习速度。传递给该变换的值是数据集中图像 rgb 值的均值(第一个元组)和标准差(第二个元组)。您可以通过运行以下几行代码来计算这些值：

在实例化数据集对象时，\_ \_init\_ \_ 函数会运行一次，用于初始化图像目录、标签文件和图像转换属性。

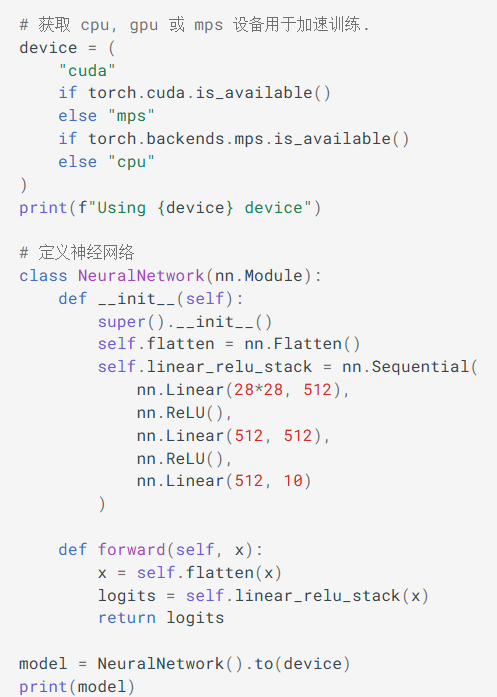
函数 \_ \_len\_ \_ 返回我们数据集中的样本数。

函数 \_ \_getitem\_ \_ 从数据集中给定的索引 idx 处加载并返回一个样本。根据索引可以确定图像在硬盘上的位置，用 read\_image 将其转换为tensor，从 self.img\_labels 的csv数据中获取相应的标签，再对它们调用 transform 函数(如果适用)，并返回tensor图像和相应的标签的元组。



1. **为了在 PyTorch 中定义一个神经网络，我们需要创建了一个继承自 [nn.Module](https://pytorch.org/docs/stable/generated/torch.nn.Module.html) 的自定义类。**

我们在 \_ \_init\_ \_ 方法中定义网络的层次结构，并在 forward 方法中指定数据将如何通过网络的各个层。为了加速神经网络中的操作，我们将其移动到 GPU 或 MPS (如果有的话)。



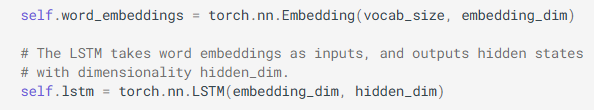
我们初始化 [nn.Flatten(展平层)](https://pytorch.org/docs/stable/generated/torch.nn.Flatten.html) 层来将每个 2 维的 28x28 图像转换成一个包含 784 像素值的连续数组。

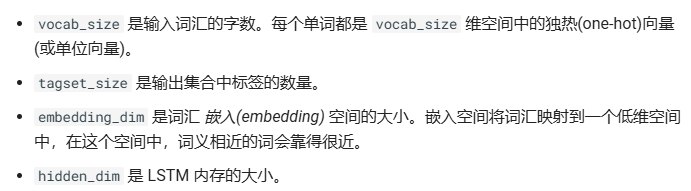
nn.Linear(线性层)是一个对输入值使用自己存储的权重 (w) 和偏差 (b) 来做线性转换的模块。

nn.Conv2d(卷积层)第一个参数是输入通道数，第二个参数是输出特征的数量，第三个参数是窗口或内核大小（可以设置为一个元组）



nn.RNN/nn.LSTM（循环层）





nn.ReLU非线性的激活函数在模型的输入和输出之间数创造了复杂的映射关系。

[nn.Sequential](https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html) 是一个模块的有序容器。数据会沿着模块定义的顺序流动。

nn.MaxPool2d(3)最大池化(和它兄弟，最小池化)通过合并单元来减少tensor，并将输入单元的最大值分配给输出单元

nn.Softmax模型的最后一层返回 logits(介于[负无穷,正无穷]之间的原始值)，然后被传递给 [nn.Softmax](https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html) 模块。这些 logits 值被缩放到 [0,1]，代表模型对每种类型的预测概率， dim 参数代表沿着该维度数值应该加总为 1。

通过你模型的 parameters() 或者 named\_parameters() 方法访问所有参数。

**3.Tensors 是一种特殊的数据结构，与数组和矩阵非常相似。**

Tensors 可以用不同的方式初始化:

* **直接从原生数据创建**



* **从 NumPy 数组创建**



* **从 tensor 变量创建，torch.\*\_like()方法创建一个shape相同的tensor**



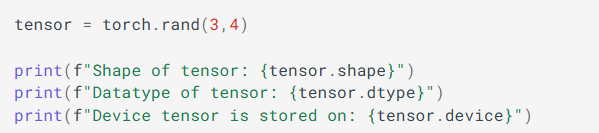
* **从随机数据或常量创建**

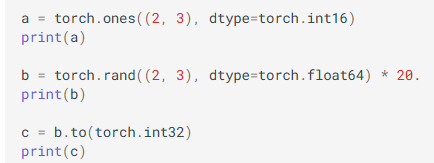
随机数种子



## **4.Tensor的属性**

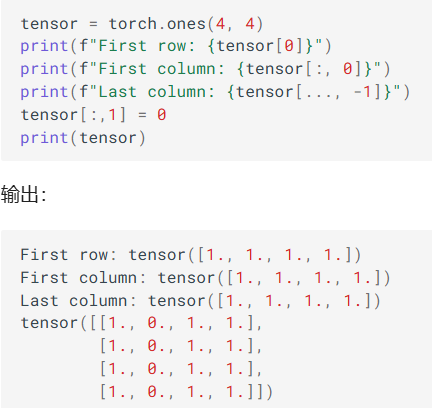
形状、数据类型和存储它们的设备





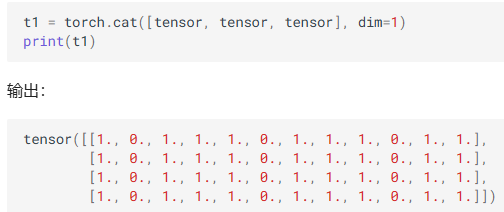
1. **Tensors的操作**

* **类似 numpy 索引和分片的标准操作**

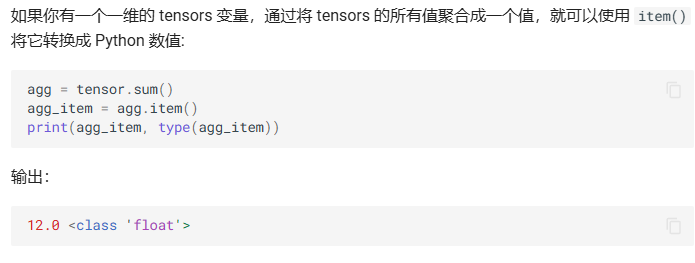


* ****连接 tensors****

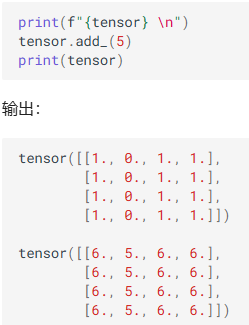
使用 torch.cat 将一系列tensor沿着给定的维数连接起来



* ****算术运算****
* ****单个元素的 tensors****



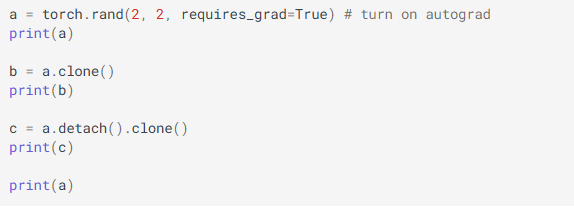
* ****就地操作：****将修改结果存储到操作数中的操作被称为就地操作，通常它们以后缀 \_ 来表示



* ****复制tensor：clone()****

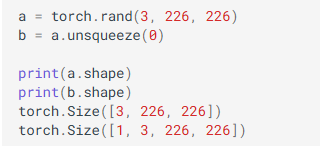


注：您在模型的 forward() 函数中执行计算，默认情况下梯度都是打开的，但您想在中途取出一些值来生成一些指标。在这种情况下，您不希望源tensor的克隆副本跟踪梯度——关闭自动微分的历史跟踪可以提高性能。为此，您可以在源tensor上使用 .detach() 方法：



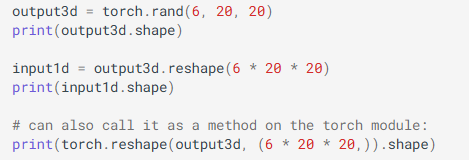
* ****更改维数****

unsqueeze() 方法添加了一个范围为 1 的维度，unsqueeze(0) 则将其添加为一个新的第零维度——现在您有了一批 1 的维度！



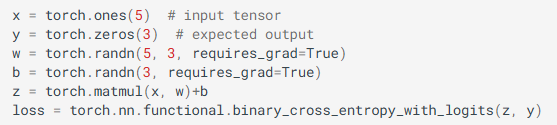
对 squeeze() 和 unsqueeze() 的调用只能作用于范围为 1 的维度，否则会改变tensor中元素的数量。（均有原地版）

彻底改变维数用reshape



1. **自动微分运算**

PyTorch 有一个内置的微分运算引擎叫 torch.autograd。它支持对任何计算图自动计算梯度。



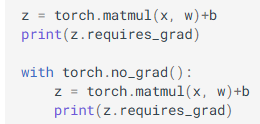
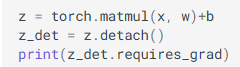
w 和 b 都是我们需要优化的参数，设置 requires\_grad 属性

我们需要在给定 x 和 y 下的 和 。要计算这些导数，我们调用loss.backward()，然后从 w.grad 和 b.grad 中获取值。

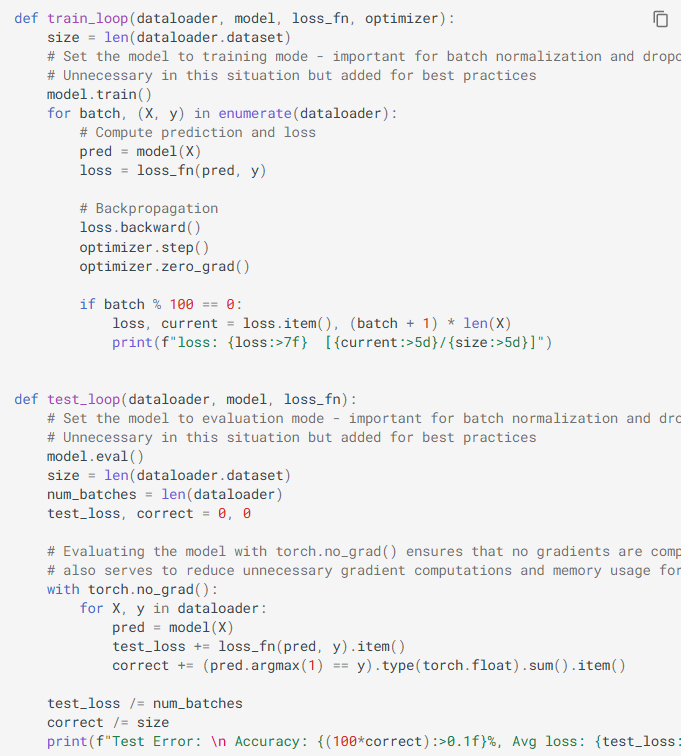
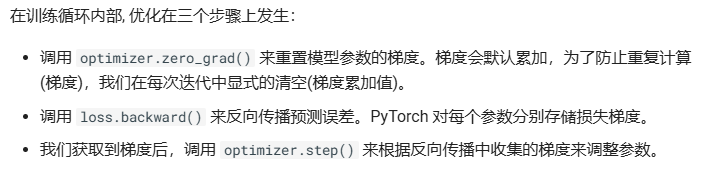


在给定的计算图中，出于性能原因我们只能用 backward 进行一次梯度计算。如果我们想要对同一张计算图做几次 backward 调用，我们需要在 backward 调用时传递 retain\_graph=True 参数。

torch.no\_grad/detach()来禁用梯度追踪

/

1. **优化模型参数**



1. **模型保存和加载**

PyTorch 将模型学习到的参数存储在一个内部状态字典中叫 state\_dict。它们可以通过 torch.save方法来持久化。



要加载模型权重，你需要先创建一个跟要加载权重的模型结构一样的模型，然后使用load\_state\_dict() 方法加载参数。

