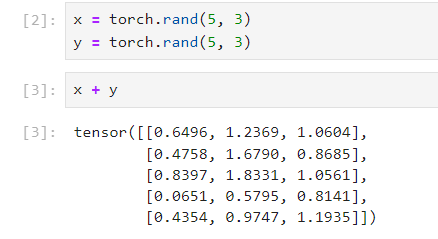
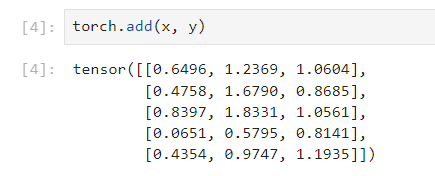
张量

Tensor(张量）类似于NumPy的ndarray，但还可以在GPU上使用来加速计算

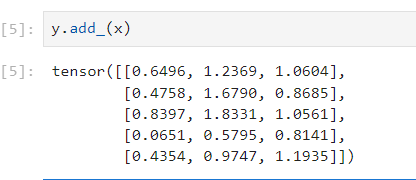
加法：形势一



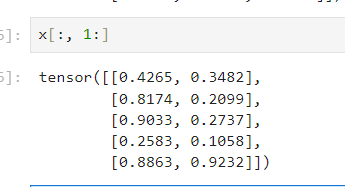
加法：形势二



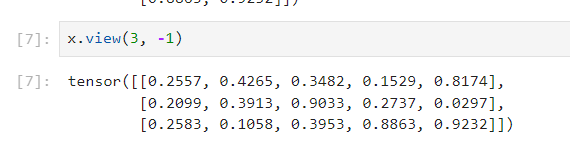
加法：原位/原地操作(in-place)



可以使用像标准的NumPy一样的各种索引操



使用torch.view改变形状



autograd 包为张量上的所有操作提供了自动求导机制。它是一个在运行时定义(define-by-run）的框架，这意味着反向传播是根据代码如何运行来决定的，并且每次迭代可以是不同的

torch.Tensor 是这个包的核心类。如果设置它的属性 .requires\_grad 为 True，那么autograd将会追踪对于该张量的所有操作。当完成计算后可以通过调用 .backward()，来自动计算所有的梯度。这个张量的所有梯度将会自动累加到.grad属性

一个神经网络的典型训练过程如下：

定义包含一些可学习参数(或者叫权重）的神经网络

在输入数据集上迭代

通过网络处理输入

计算loss(输出和正确答案的距离）

将梯度反向传播给网络的参数

更新网络的权重，一般使用一个简单的规则：weight = weight - learning\_rate \* gradient

只需要定义 forward 函数，backward函数会在使用autograd时自动定义，backward函数用来计算导数。

一个损失函数接受一对(output, target)作为输入，计算一个值来估计网络的输出和目标值相差多少。

nn包中有很多不同的损失函数。nn.MSELoss是比较简单的一种，它计算输出和目标的均方误差

当调用loss.backward()，整张图开始关于loss微分，图中所有设置了requires\_grad=True的张量的.grad属性累积着梯度张量。

只需要调用loss.backward()来反向传播误差。需要清零现有的梯度，否则梯度将会与已有的梯度累加。

最简单的更新规则是随机梯度下降法(SGD）

optimizer = optim.SGD(net.parameters(), lr=0.01)

# 在训练的迭代中：

optimizer.zero\_grad() # 清零梯度缓存

output = net(input)

loss = criterion(output, target)

loss.backward()

optimizer.step() # 更新参数

当必须处理图像、文本、音频或视频数据时，可以使用python标准库将数据加载到numpy数组里。然后将这个数组转化成torch.\*Tensor。

对于图片，有Pillow，OpenCV等包可以使用

对于音频，有scipy和librosa等包可以使用

对于文本，不管是原生python的或者是基于Cython的文本，可以使用NLTK和SpaCy

特别对于视觉方面，我们创建了一个包，名字叫torchvision，其中包含了针对Imagenet、CIFAR10、MNIST等常用数据集的数据加载器(data loaders），还有对图像数据转换的操作，即torchvision.datasets和torch.utils.data.DataLoader

torch.utils.data.Dataset是代表数据集的抽象类。 自定义数据集应继承Dataset并覆盖以下方法：

\_\_len\_\_，以便len(dataset)返回数据集的大小。

\_\_getitem\_\_支持索引，以便可以使用dataset[i]获取第个样本

Transforms 变换

从上面可以看到的一个问题是样本的大小不同。 大多数神经网络期望图像的大小固定。 因此，我们将需要编写一些预处理代码。 让我们创建三个转换：

Rescale：缩放图像

RandomCrop：从图像中随机裁剪。 这是数据增强。

ToTensor：将 numpy 图像转换为 torch 图像

在 Penn-Fudan 数据库中对行人检测和分割进行预训练的Mask R-CNN 模型进行微调

用于训练对象检测，实例细分和人员关键点检测的参考脚本可轻松支持添加新的自定义数据集。 数据集应继承自标准torch.utils.data.Dataset类，并实现\_\_len\_\_和\_\_getitem\_\_

两个主要的转移学习方案如下所示：

对卷积网络进行微调：代替随机初始化，我们使用经过预训练的网络初始化网络，例如在 imagenet 1000 数据集上进行训练的网络。 其余的训练照常进行。

ConvNet 作为固定特征提取器：在这里，我们将冻结除最终完全连接层以外的所有网络的权重。 最后一个完全连接的层将替换为具有随机权重的新层，并且仅训练该层

ConvNet 作为固定特征提取器

在这里，需要冻结除最后一层之外的所有网络。需要设置requires\_grad == False冻结参数，以便不在backward()中计算梯度。

空间变压器网络可归结为三个主要组成部分：

本地化网络是常规的 CNN，可以对转换参数进行回归。 永远不会从此数据集中显式学习变换，而是网络会自动学习增强全局精度的空间变换。

网格生成器在输入图像中生成与来自输出图像的每个像素相对应的坐标网格。

采样器使用转换的参数，并将其应用于输入图像。

训练模型

现在，使用 SGD 算法训练模型。 网络正在以监督方式学习分类任务。 同时，该模型以端到端的方式自动学习 STN。

定义了两个距离，一个为内容(），一个为样式(）。 测量两个图像之间的内容有多大不同，而测量两个图像之间的样式有多大不同。 然后，我们获取第三个图像(输入），并将其转换为最小化与内容图像的内容距离和与样式图像的样式距离

现在，将导入样式和内容图像。 原始的 PIL 图像的值在 0 到 255 之间，但是当转换为torch张量时，其值将转换为 0 到 1 之间。图像也需要调整大小以具有相同的尺寸。 需要注意的一个重要细节是，使用从 0 到 1 的张量值对torch库中的神经网络进行训练。如果尝试为网络提供 0 到 255 张量图像，则激活的特征图将无法感知预期的内容 和风格。 但是，使用 0 到 255 张量图像对 Caffe 库中的预训练网络进行训练

内容损失

内容损失是代表单个图层内容距离的加权版本的函数。 该功能获取网络处理输入中层的特征图，并返回图像和内容图像之间的加权内容距离。 为了计算内容距离，该功能必须知道内容图像的特征图(）。 我们将此功能实现为炬管模块，并使用以作为输入的构造函数。 距离是两组特征图之间的均方误差，可以使用nn.MSELoss进行计算。

将直接在用于计算内容距离的卷积层之后添加此内容丢失模块。 这样，每次向网络馈入输入图像时，都会在所需层上计算内容损失，并且由于自动渐变，将计算所有梯度。 现在，为了使内容丢失层透明，我们必须定义一种forward方法，该方法计算内容丢失，然后返回该层的输入。 计算出的损耗将保存为模块的参数

威胁模型

就上下文而言，有多种类型的对抗性攻击，每种攻击者的目标和假设都不同。 但是，总的来说，总体目标是向输入数据添加最少的扰动，以引起所需的错误分类。 攻击者的知识有几种假设，其中两种是：白盒和黑盒。 白盒\_攻击假定攻击者具有完整的知识并可以访问模型，包括体系结构，输入，输出和权重。 \_黑盒\_攻击假定攻击者仅有权访问模型的输入和输出，并且对底层体系结构或权重一无所知。 目标也有几种类型，包括错误分类和源/目标错误分类。 \_错误分类的目标是，这意味着对手只希望输出分类错误，而不在乎新分类是什么。 \_源/目标错误分类\_意味着对手想要更改最初属于特定源类别的图像，以便将其分类为特定目标类别

快速梯度符号攻击

迄今为止，最早的也是最流行的对抗性攻击之一称为\_快速梯度符号攻击(FGSM）\_，由 Goodfellow 等描述。 等 中的解释和利用对抗性示例。 攻击非常强大，而且直观。 它旨在利用神经网络的学习方式\_梯度\_来攻击神经网络。 这个想法很简单，不是通过基于反向传播的梯度来调整权重来使损失最小化，攻击\_会基于相同的反向传播的梯度来调整输入数据以使损失\_最大化。 换句话说，攻击使用损失了输入数据的梯度，然后调整输入数据以使损失最大化

GAN 是用于教授 DL 模型以捕获训练数据分布的框架，因此我们可以从同一分布中生成新数据。 GAN 由 Ian Goodfellow 于 2014 年发明，并首先在论文生成对抗网络中进行了描述。 它们由两个不同的模型组成：生成器\_和\_鉴别器。 生成器的工作是生成看起来像训练图像的“假”图像。 鉴别器的工作是查看图像并从生成器输出它是真实的训练图像还是伪图像。 在训练过程中，生成器不断尝试通过生成越来越好的伪造品而使鉴别器的性能超过智者，而鉴别器正在努力成为更好的侦探并正确地对真实和伪造图像进行分类。 博弈的平衡点是当生成器生成的伪造品看起来像直接来自训练数据时，而鉴别器则总是猜测生成器输出是真品还是伪造品的 50％置信度。

生成器

生成器旨在将潜在空间矢量映射到数据空间。 由于我们的数据是图像，因此将转换为数据空间意味着最终创建与训练图像大小相同的 RGB 图像(即 3x64x64）。 在实践中，这是通过一系列跨步的二维卷积转置层来完成的，每个层都与2d 批处理规范层和relu 激活配对。 生成器的输出通过 tanh 函数进行馈送，以使其返回到的输入数据范围。值得注意的是，在卷积转置层之后存在批处理规范函数，因为这是 DCGAN 论文的关键贡献。这些层有助于训练过程中的梯度流动。

鉴别器

如前所述，鉴别符是一个二进制分类网络，该二进制分类网络将图像作为输入并输出输入图像是真实的(与假的相对）的标量概率。 在这里，拍摄 3x64x64 的输入图像，通过一系列的 Conv2d，BatchNorm2d 和 LeakyReLU 层对其进行处理，然后通过 Sigmoid 激活函数输出最终概率。 如果需要解决此问题，可以用更多层扩展此体系结构，但是使用跨步卷积，BatchNorm 和 LeakyReLUs 具有重要意义。 DCGAN 论文提到，使用跨步卷积而不是合并以进行下采样是一个好习惯，因为它可以让网络学习自己的合并功能。 批处理规范和泄漏的 relu 函数还可以促进健康的梯度流，这对于和的学习过程都是至关重要的

训练

最后，既然我们已经定义了 GAN 框架的所有部分，我们就可以对其进行训练。 请注意，训练 GAN 某种程度上是一种艺术形式，因为不正确的超参数设置会导致模式崩溃，而对失败的原因几乎没有解释。 在这里，我们将严格遵循 Goodfellow 论文中的算法 1，同时遵守 ganhacks 中显示的一些最佳做法。 即，我们将“为真实和伪造构建不同的小批量”图像，并调整 G 的目标函数以最大化。 训练分为两个主要部分。 第 1 部分更新了鉴别器，第 2 部分更新了生成器。

将名称转换为张量

获取所有姓氏之后，需要将它们转换为张量。

将名称转换为张量

获取所有姓氏之后，需要将它们转换为张量。

为了表示单个字母，使用大小为<1 x n\_letters>的“ one-hot vector”。 一个“one hot”向量是当前字母的索引处为 1，其余部分为 0 的向量

建立网络

在 Torch 中创建一个循环神经网络需要在多个时间步上克隆神经层的参数。 现在不同的时间步骤下网络层所保存的隐藏状态和梯度均由计算图自身处理，编程者无需关心，因此您可以向构建常见的前馈网络一样在Torch中简便的构建循环神经网络。

通过序列到序列网络的简单但强大的构想，使这成为可能，在该网络中，两个循环神经网络协同工作，将一个序列转换为另一个序列。 编码器网络将输入序列压缩为一个向量，而解码器网络将该向量展开为一个新序列

Cartpole任务的设计为智能点输入代表环境状态(位置、速度等）的4个实际值。 但是，神经网络可以完全通过查看场景来解决任务，因此我们将以小车为中心的一部分屏幕作为输入。 因此，结果无法直接与官方排行榜上的结果进行比较-我们的任务更加艰巨

经验回放

将使用经验回放内存来训练我们的 DQN。 它存储智能体观察到的转换，使以后可以重用此数据。 通过从中随机抽样，可以构建成批的过渡。 实践表明，这极大稳定并改进了 DQN 训练过程。

步骤 1：将 PyTorch 模型转换为 Torch 脚本

PyTorch 模型从 Python 到 C ++的旅程由 Torch 脚本启用，它是 PyTorch 模型的表示形式，可以由 Torch 脚本编译器理解，编译和序列化。 如果您是从使用香草“渴望” API 编写的现有 PyTorch 模型开始的，则必须首先将模型转换为 Torch 脚本

步骤 2：将脚本模块序列化为文件

跟踪或注释 PyTorch 模型后，一旦您有了ScriptModule，就可以将其序列化为文件了。 稍后，您将可以使用 C ++从此文件加载模块并执行它，而无需依赖 Python。

步骤 3：在 C ++中加载脚本模块

要在 C ++中加载序列化的 PyTorch 模型，您的应用程序必须依赖于 PyTorch C ++ API –也称为 LibTorch 。 LibTorch 发行版包含共享库，头文件和 CMake 构建配置文件的集合。 虽然 CMake 不是依赖 LibTorch 的要求，但它是推荐的方法，将来会得到很好的支持

步骤 4：在 C ++中执行脚本模块

在用 C ++成功加载序列化的ResNet18之后，我们现在离执行它仅几行代码了！ 让我们将这些行添加到 C ++应用程序的main()函数中：

DistributedDataParallel (DDP）在模块级别实现数据并行性。 它使用 Torch.distributed 程序包中的通信集合来同步梯度，参数和缓冲区。 并行性在流程内和跨流程均可用。 在一个过程中，DDP 将输入模块复制到device\_ids中指定的设备，将输入沿批次维度分散，然后将输出收集到output\_device，这与 DataParallel 相似。 在整个过程中，DDP 在正向传递中插入必要的参数同步，在反向传递中插入梯度同步。 用户可以将进程映射到可用资源，只要进程不共享 GPU 设备即可。 推荐的方法(通常是最快的方法）是为每个模块副本创建一个过程，即在一个过程中不进行任何模块复制

PyTorch 中包含的分布式软件包(即torch.distributed）使研究人员和从业人员可以轻松地并行化他们在跨进程和机器集群的计算。 为此，它利用了传递消息的语义，从而允许每个进程将数据传递给其他任何进程。 与并行处理(HTG1）包相反，进程可以使用不同的通信后端，而不仅限于在同一台计算机上执行。

通过带有 Flask 的 REST API 在 Python 中部署 PyTorch

API 定义

我们将首先定义 API 端点，请求和响应类型。 我们的 API 端点将位于/predict，该端点通过包含图片的file参数接受 HTTP POST 请求。 响应将是包含预测的 JSON 响应

简单的 Web 服务器

以下是一个简单的网络服务器，摘自 Flask 的文档

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def hello():

return 'Hello World!'

准备图像

DenseNet 模型要求图像为尺寸为 224 x 224 的 3 通道 RGB 图像。我们还将使用所需的均值和标准偏差值对图像张量进行归一化

预测

现在将使用预训练的 DenseNet 121 模型来预测图像类别。 我们将使用torchvision库中的一个，加载模型并进行推断

将模型集成到我们的 API 服务器中

在最后一部分中，我们将模型添加到 Flask API 服务器中。 由于我们的 API 服务器应该获取图像文件，因此我们将更新predict方法以从请求中读取文件