PyTorch官方教程中文版

教程介绍

PyTorch是一个基于Torch的Python开源机器学习库，用于自然语言处理等应用程序。它主要由Facebookd的人工智能小组开发，不仅能够 实现强大的GPU加速，同时还支持动态神经网络，这一点是现在很多主流框架如TensorFlow都不支持的。 PyTorch提供了两个高级功能： 1.具有强大的GPU加速的张量计算（ 如Numpy ） 2.包含自动求导系统的深度神经网络 除了Facebook之外，Twitter、GMU和Salesforce等机构都采用了PyTorch。

官方教程包含了 PyTorch 介绍，安装教程；60分钟快速入门教程，可以迅速从小白阶段完成一个分类器模型；计算机视觉常用模型，方便基于自己的数据进行调整，不再需要从头开始写；自然语言处理模型，聊天机器人，文本生成等生动有趣的项目。

总而言之：

- 如果你想了解一下 PyTorch，可以看介绍部分。
- 如果你想快速入门 PyTorch，可以看60分钟快速入门。
- 如果你想解决计算机视觉问题，可以看计算机视觉部分。
- 如果你想解决自然语言处理问题，可以看NLP 部分。

作者：

磐创 News and PytorchChina

PyTorch 教程目录

PyTorch 之简介与下载

PyTorch 简介

PyTorch 环境搭建

PyTorch 之 60min 入门教程

PyTorch 入门

PyTorch 自动微分

PyTorch 神经网络

PyTorch 图像分类器

PyTorch 数据并行处理

PyTorch 之入门强化教程

数据加载和处理

PyTorch 小试牛刀

迁移学习

混合前端的 seq2seq 模型部署

混合前端

预备环境

保存和加载模型

PyTorch 之图像篇

微调基于 torchvision 0.3 的目标检测模型

微调 TorchVision 模型

空间变换器网络

使用 PyTorch 进行 Neural-Transfer

生成对抗示例

使用 ONNX 将模型转移至 Caffe2 和移动端

PyTorch 之文本篇

聊天机器人教程

使用字符级 RNN 生成名字

使用字符级 RNN 进行名字分类

在深度学习和 NLP 中使用 Pytorch

使用 Sequence2Sequence 网络和注意力进行翻译

PyTorch 之生成对抗网络

生成对抗网络

PyTorch 之强化学习

强化学习

生产环境部署 PyTorch 模型

使用Flask来部署PyTorch模型

TorchScript的简介

在C++中加载TorchScript模型

PyTorch123 配套录制视频教程

网易云课堂，第一节数据处理与加载免费观看

https://study.163.com/course/introduction/1209674804.htm

另外，提供官方教程的一个精简版配套学习资料，书名《Deep Learning with PyTorch》。

下载地址：http://pytorchchina.com/2019/12/02/deep-learning-with-pytorch-pdf/

PyTorch 快速访问地址

镜像网站，快速访问

PyTorch 微信群

PytorchChina

GitHub

https://github.com/fendouai/PyTorchDocs

磐创AI 微信公众号

在公众号后台回复 tensorflow,keras,pytorch 可以获得更多电子书。

PyTorch简介

要介绍PyTorch之前，不得不说一下Torch。Torch是一个有大量机器学习算法支持的科学计算框架，是一个与Numpy类似的张量（Tensor）操作库，其特点是特别灵活，但因其采用了小众的编程语言是Lua，所以流行度不高，这也就有了PyTorch的出现。所以其实Torch是 PyTorch的前身，它们的底层语言相同，只是使用了不同的上层包装语言。

PyTorch是一个基于Torch的Python开源机器学习库，用于自然语言处理等应用程序。它主要由Facebookd的人工智能小组开发，不仅能够 实现强大的GPU加速，同时还支持动态神经网络，这一点是现在很多主流框架如TensorFlow都不支持的。 PyTorch提供了两个高级功能：* 具有强大的GPU加速的张量计算（如Numpy）* 包含自动求导系统的深度神经网络

除了Facebook之外，Twitter、GMU和Salesforce等机构都采用了PyTorch。

TensorFlow和Caffe都是命令式的编程语言，而且是静态的，首先必须构建一个神经网络，然后一次又一次使用相同的结构，如果想要改 变网络的结构，就必须从头开始。但是对于PyTorch，通过反向求导技术，可以让你零延迟地任意改变神经网络的行为，而且其实现速度 快。正是这一灵活性是PyTorch对比TensorFlow的最大优势。

另外，PyTorch的代码对比TensorFlow而言，更加简洁直观，底层代码也更容易看懂，这对于使用它的人来说理解底层肯定是一件令人激 动的事。

所以，总结一下PyTorch的优点：* 支持GPU * 灵活，支持动态神经网络 * 底层代码易于理解 * 命令式体验 * 自定义扩展

当然，现今任何一个深度学习框架都有其缺点，PyTorch也不例外，对比TensorFlow，其全面性处于劣势，目前PyTorch还不支持快速傅里 叶、沿维翻转张量和检查无穷与非数值张量；针对移动端、嵌入式部署以及高性能服务器端的部署其性能表现有待提升；其次因为这个框 架较新，使得他的社区没有那么强大，在文档方面其C库大多数没有文档。

1. 安装Anaconda 3.5

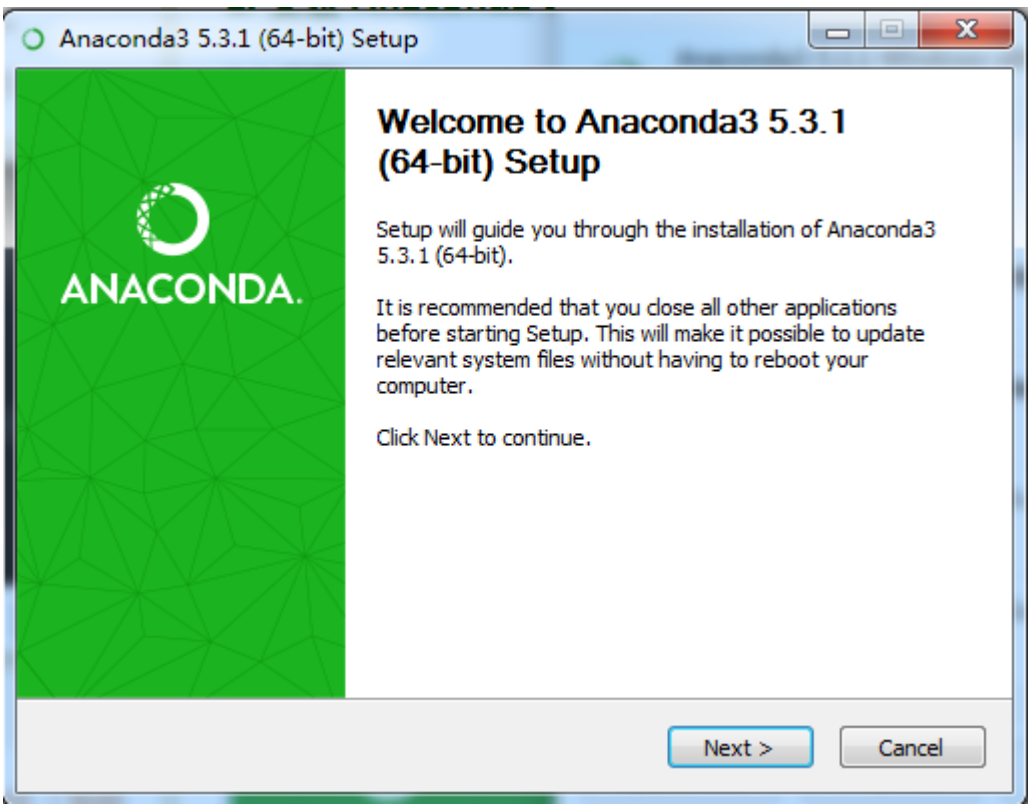
Anaconda是一个用于科学计算的Python发行版，支持Linux、Mac和Window系统，提供了包管理与环境管理的功能，可以很方便地解决Python并存、切换，以及各种第三方包安装的问题。

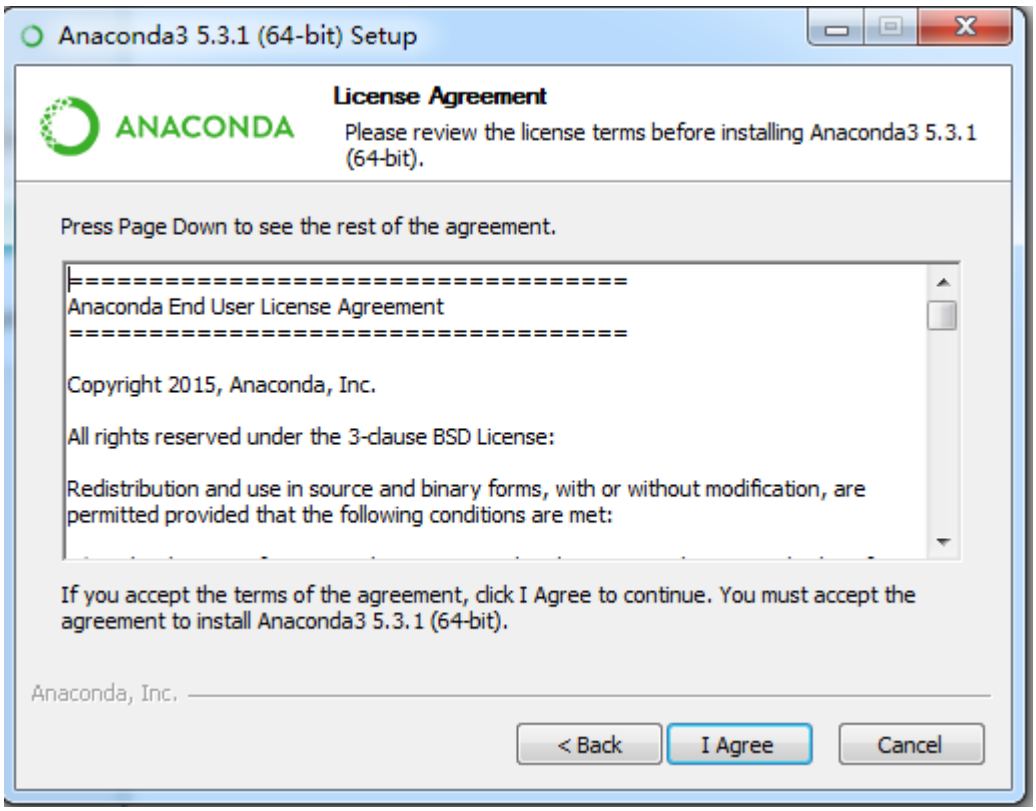
1.1 下载：

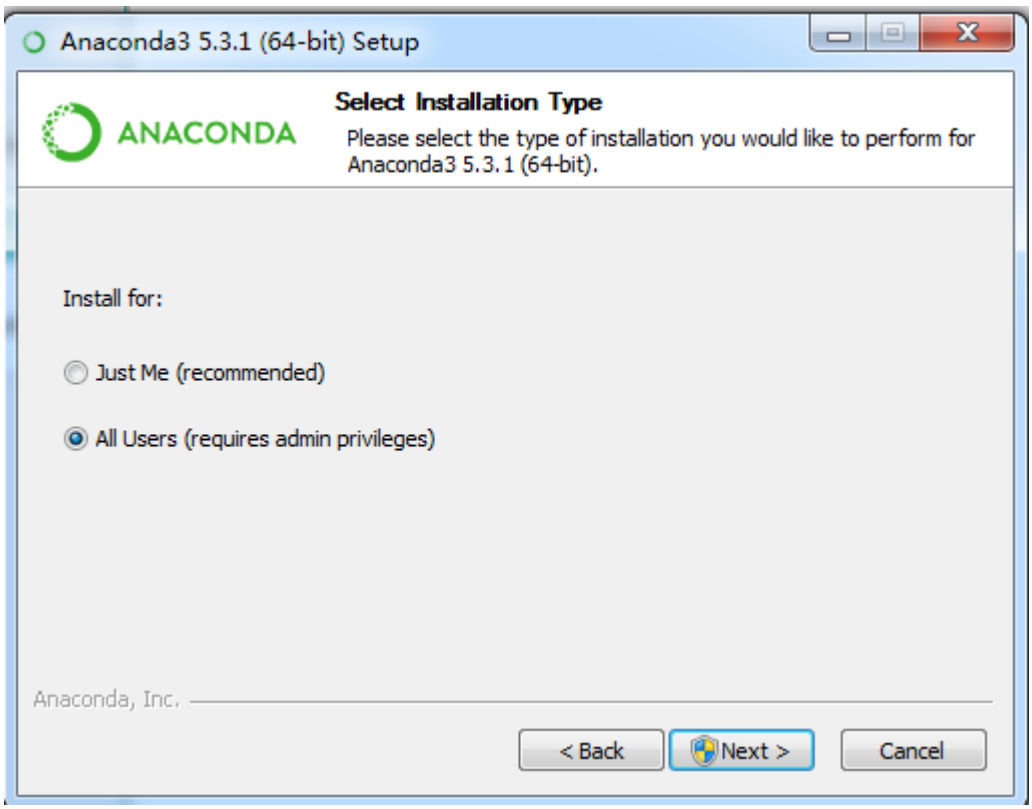
可以直接从 Anaconda官网下载，但因为Anaconda的服务器在国外，所以下载速度会很慢，这里推荐使用清华的镜像来下载。选择合适你的版本下载，我这里选择Anaconda3-5.1.0-Windows-x86_64.exe

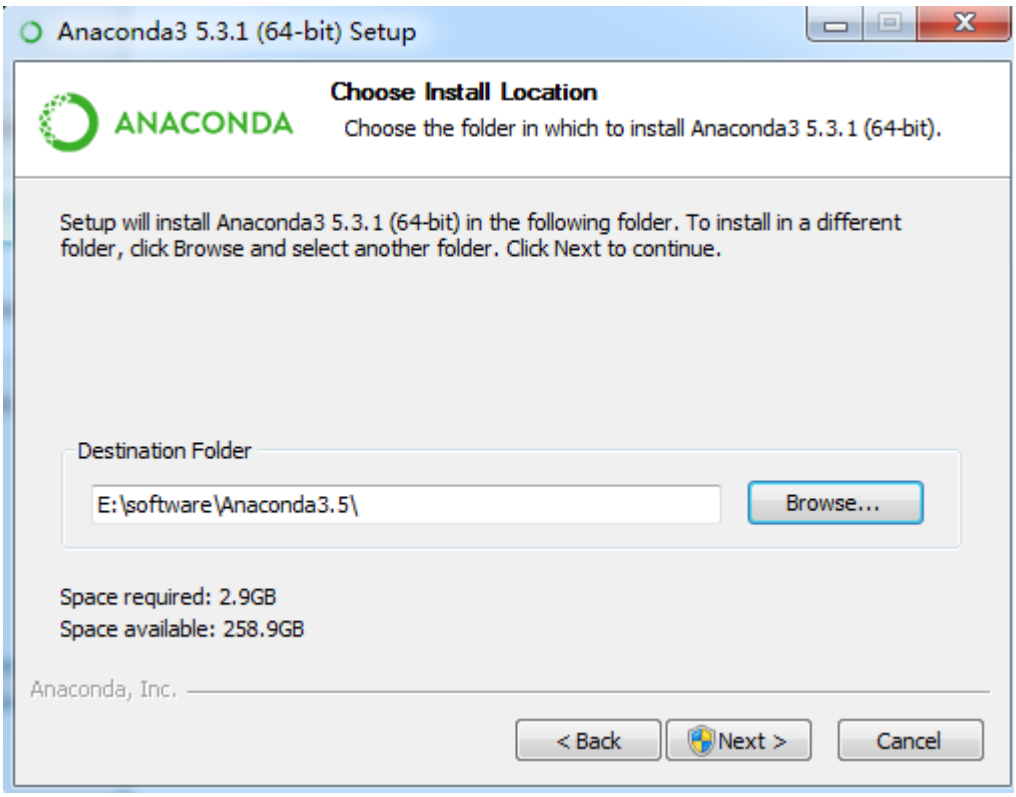
1.2 安装

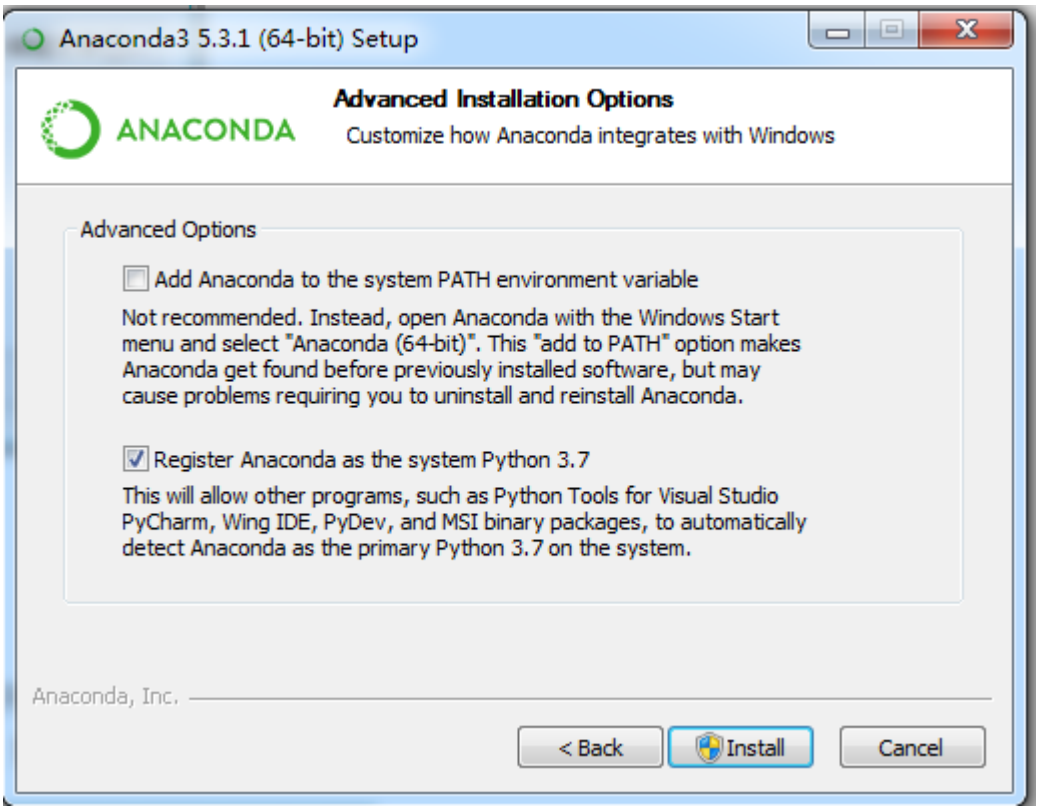
下载之后，点击安装即可，步骤依次如下：

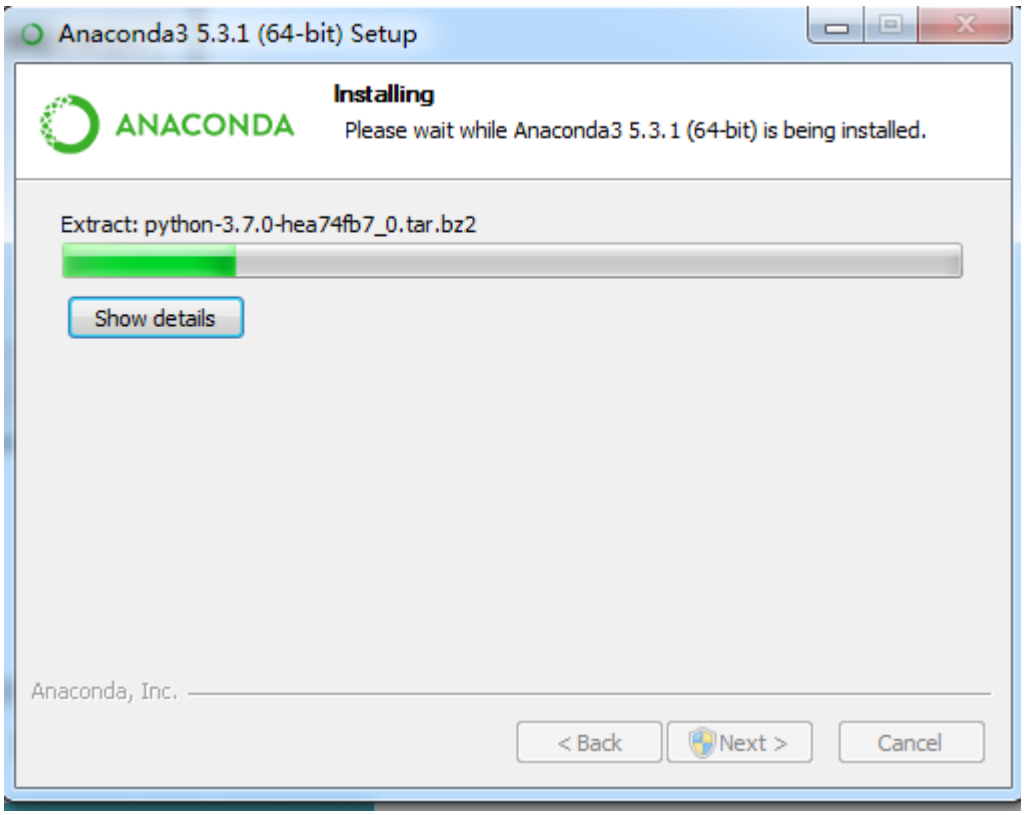


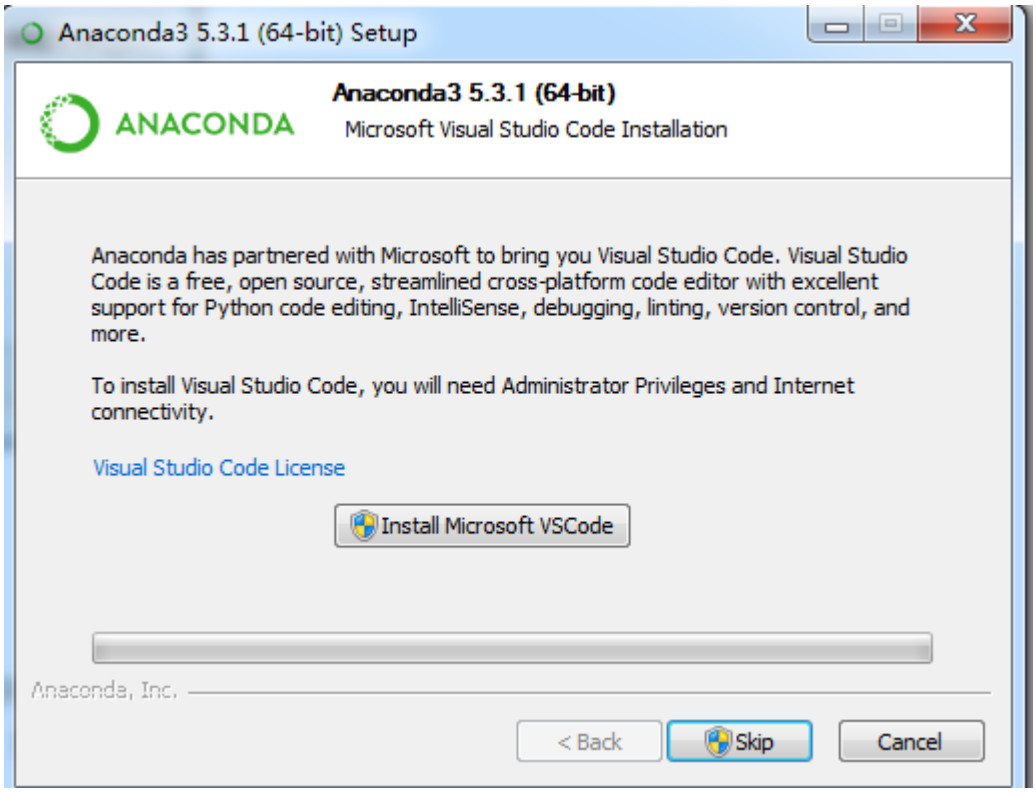

Anaconda3 5.3.1 (64-bit) Setup
ANACONDA
License Agreement
Please review the license terms before installing Anaconda3 5.3.1 (64-bit).
Press Page Down to see the rest of the agreement.
==========
Anaconda End User License Agreement
==========
Copyright 2015, Anaconda, Inc.
All rights reserved under the 3-clause BSD License:
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Anaconda3 5.3.1 (64-bit).
Anaconda, Inc.
< Back
I Agree
Cancel

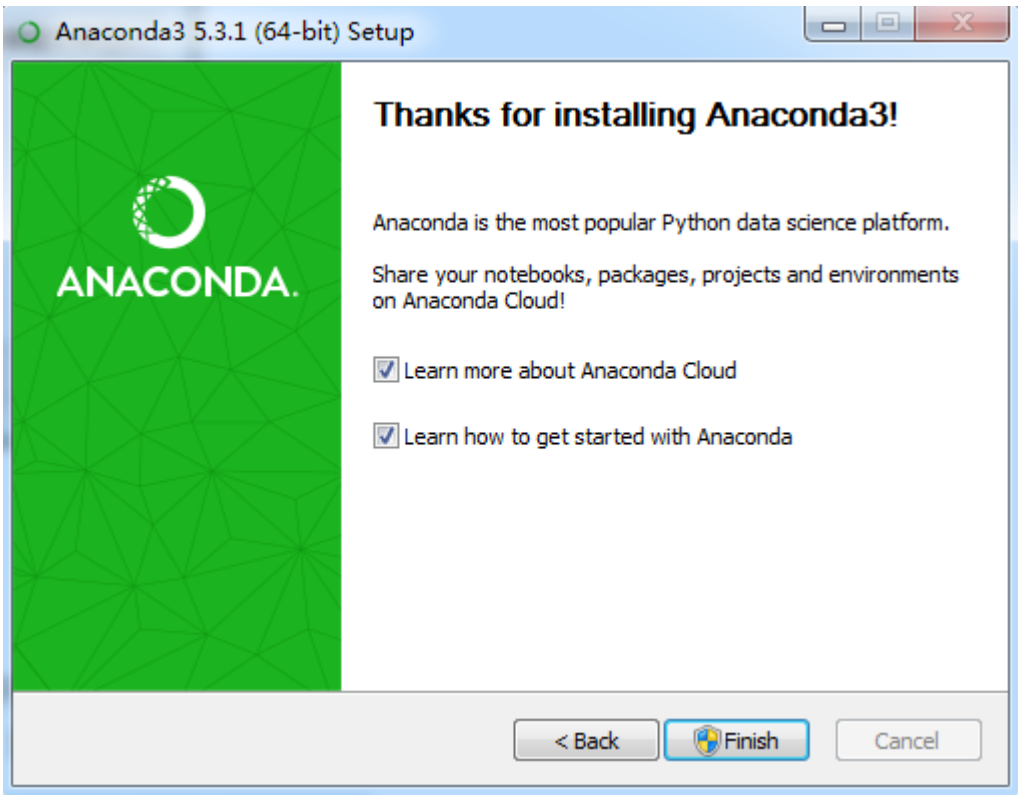

Anaconda3 5.3.1 (64-bit) Setup
ANACONDA
Select Installation Type
Please select the type of installation you would like to perform for Anaconda3 5.3.1 (64-bit).
Install for:
Just Me (recommended)
All Users (requires admin privileges)
Anaconda, Inc.
< Back
Next >
Cancel


Anaconda3 5.3.1 (64-bit) Setup
ANACONDA
Choose Install Location
Choose the folder in which to install Anaconda3 5.3.1 (64-bit).
Setup will install Anaconda3 5.3.1 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.
Destination Folder
E:\software\Anaconda3.5\
Browse...
Space required: 2.9GB
Space available: 258.9GB
Anaconda, Inc.
< Back
Next >
Cancel

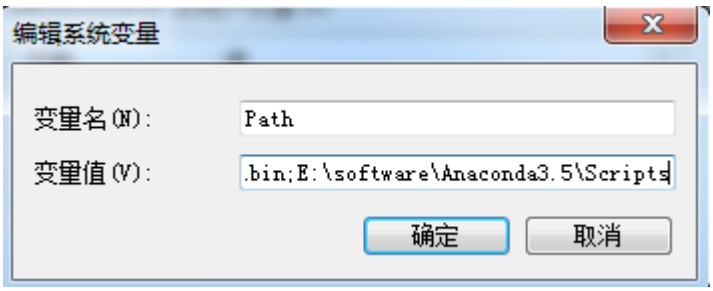

Anaconda3 5.3.1 (64-bit) Setup
ANACONDA
Advanced Installation Options
Customize how Anaconda integrates with Windows
Advanced Options
Add Anaconda to the system PATH environment variable
Not recommended. Instead, open Anaconda with the Windows Start menu and select "Anaconda (64-bit)". This "add to PATH" option makes Anaconda get found before previously installed software, but may cause problems requiring you to uninstall and reinstall Anaconda.
Register Anaconda as the system Python 3.7
This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.7 on the system.
Anaconda, Inc.
< Back
Install
Cancel


Anaconda3 5.3.1 (64-bit) Setup
ANACONDA
Installing
Please wait while Anaconda3 5.3.1 (64-bit) is being installed.
Extract: python-3.7.0-hea74fb7_0.tar.bz2
Show details
Anaconda, Inc.
< Back
Next >
Cancel

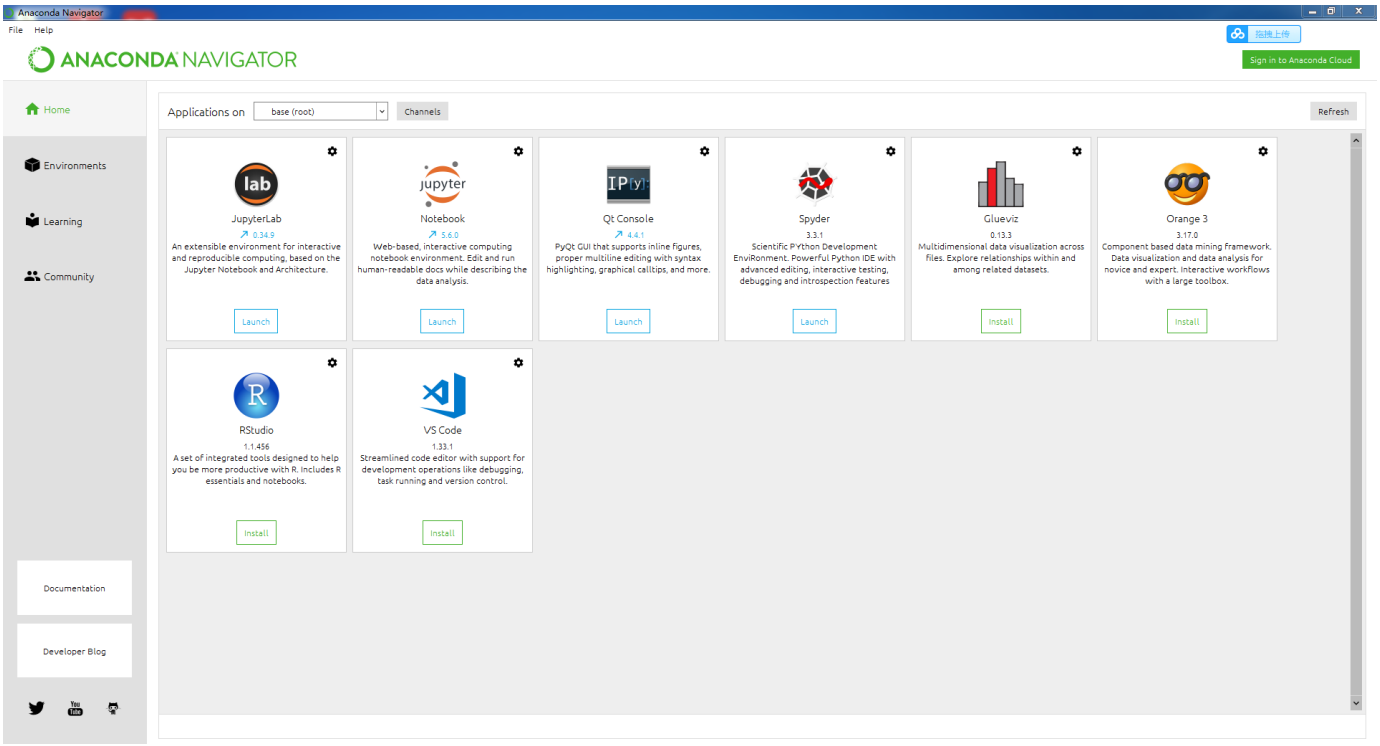

Anaconda3 5.3.1 (64-bit) Setup
ANACONDA
Anaconda3 5.3.1 (64-bit)
Microsoft Visual Studio Code Installation
Anaconda has partnered with Microsoft to bring you Visual Studio Code. Visual Studio Code is a free, open source, streamlined cross-platform code editor with excellent support for Python code editing, IntelliSense, debugging, linting, version control, and more.
To install Visual Studio Code, you will need Administrator Privileges and Internet connectivity.
Visual Studio Code License
Install Microsoft VSCode
Anaconda, Inc.
< Back
Skip
Cancel



安装完成后，进行Anaconda的环境变量配置，打开控制面板->高级系统设置->环境变量->系统变量找到Path，点击编辑，加入三个文件夹的存储路径（注意三个路径之间需用分号隔开），步骤如下：



至此，Anaconda 3.5 windows版就安装设置好了，打开程序找到Anaconda Navigator，启动后可以看到：

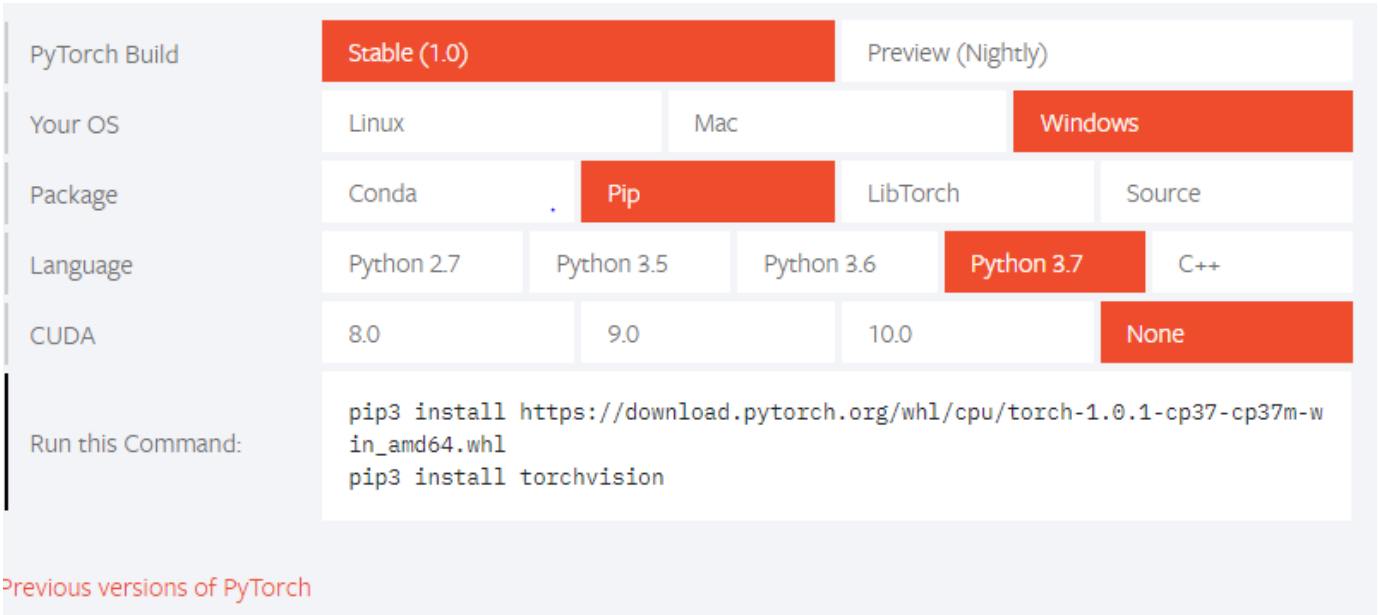


2. 安装PyTorch & torchvision

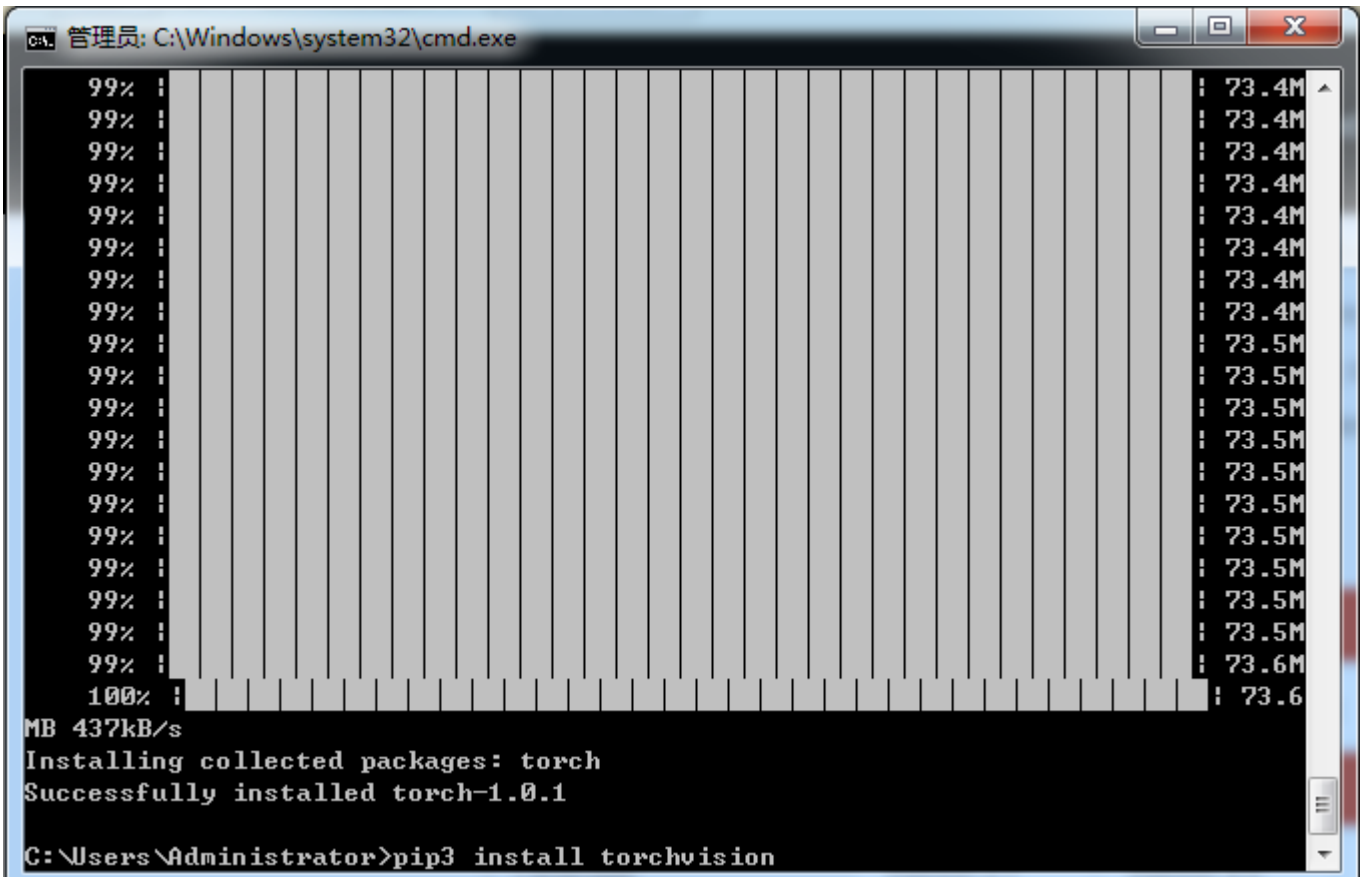
2.1 命令获取

进入 PyTorch官网，依次选择你电脑的配置（我这里已经下载了python3.7），这里提供使用pip和conda两种环境下安装的步骤截图

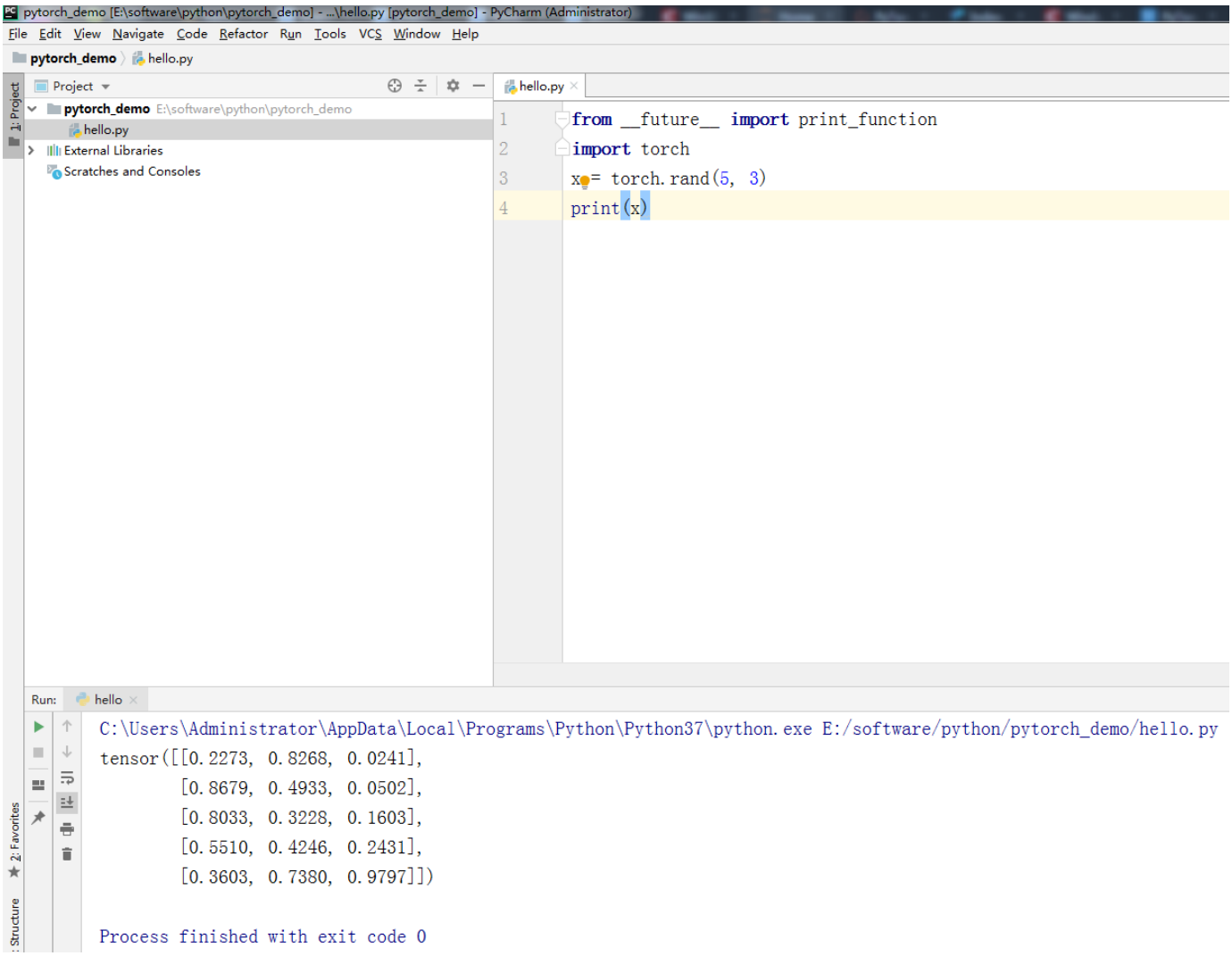
(1)使用pip：windows+pip+python3.7+None



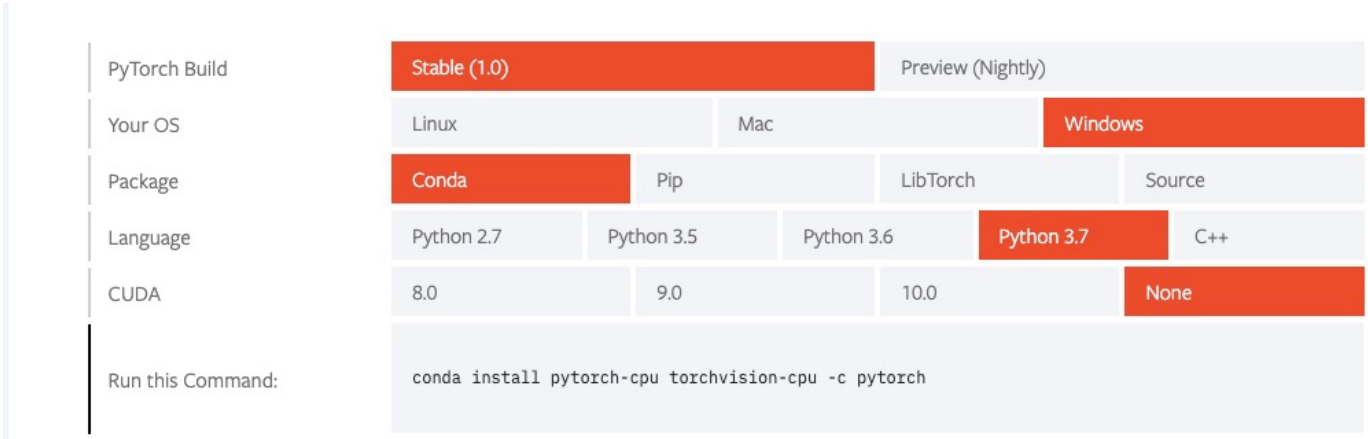
拷贝给出的命令在cmd下运行



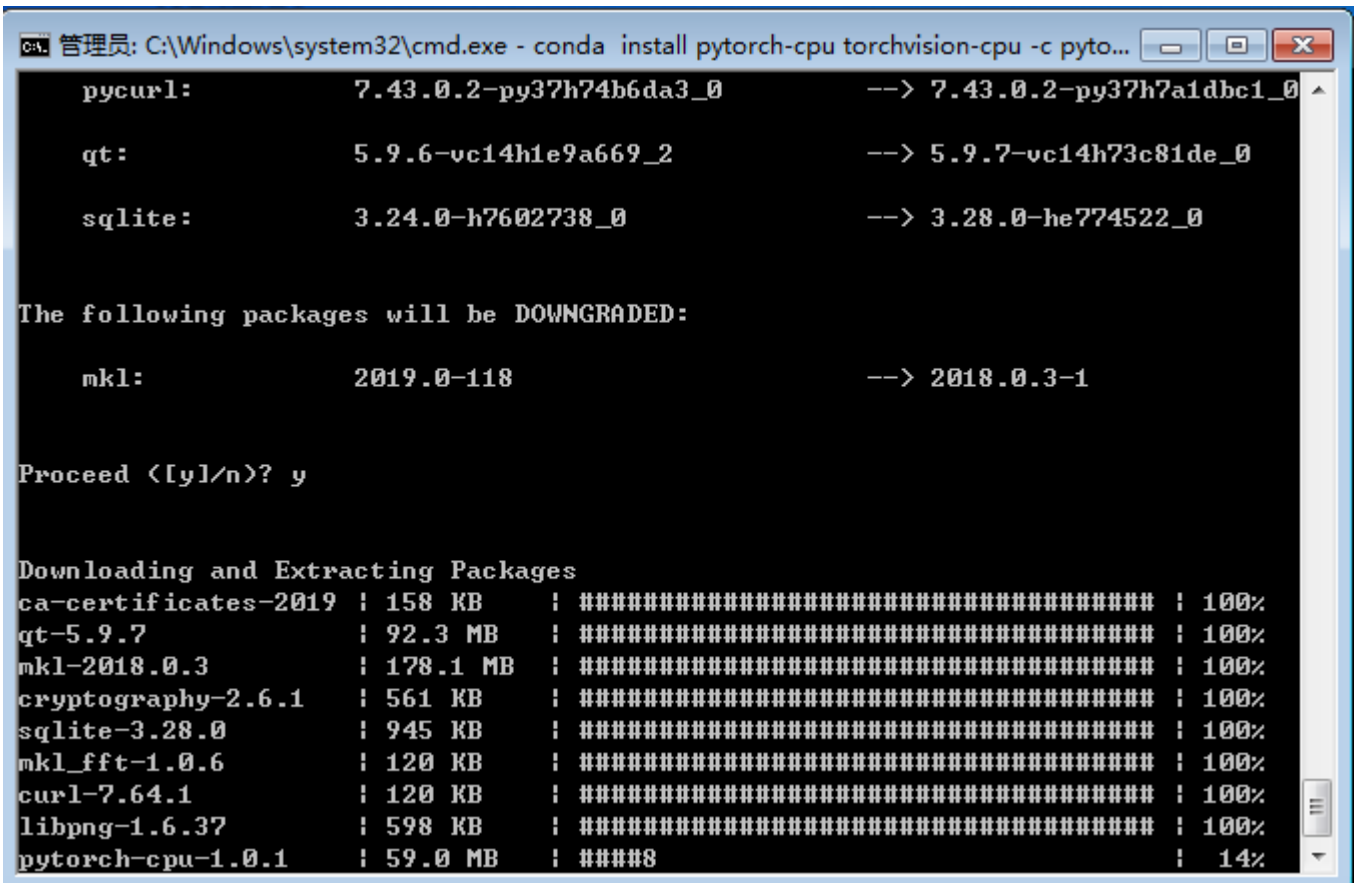
安装成功后检验是否安装成功，打开pycharm运行一个小demo：



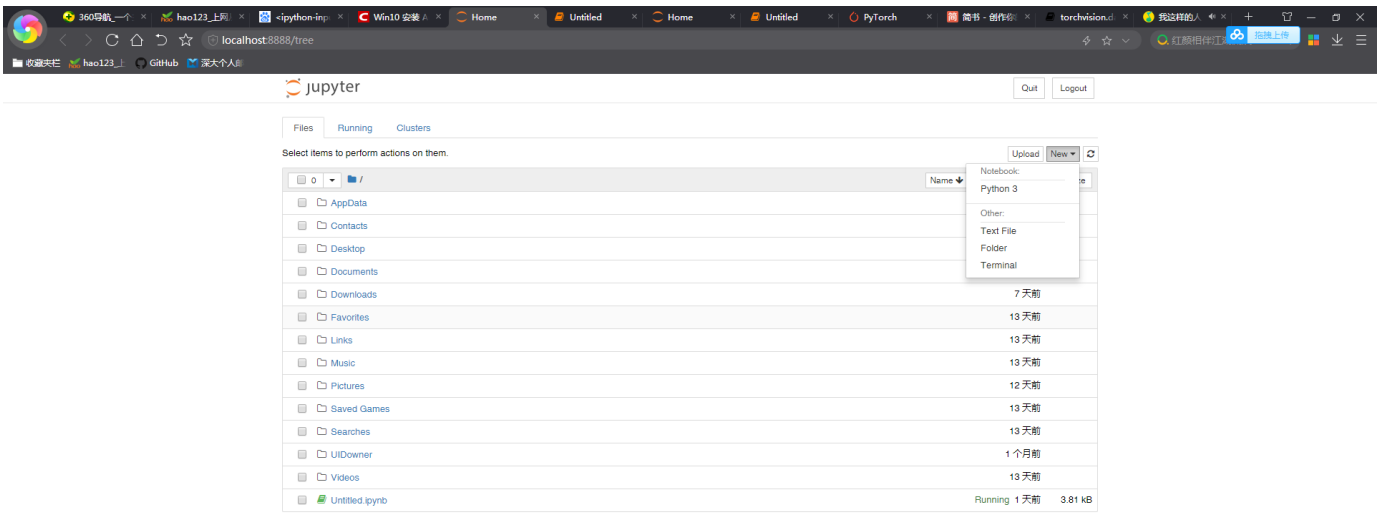
(2)使用conda：windows+conda+python3.7+None

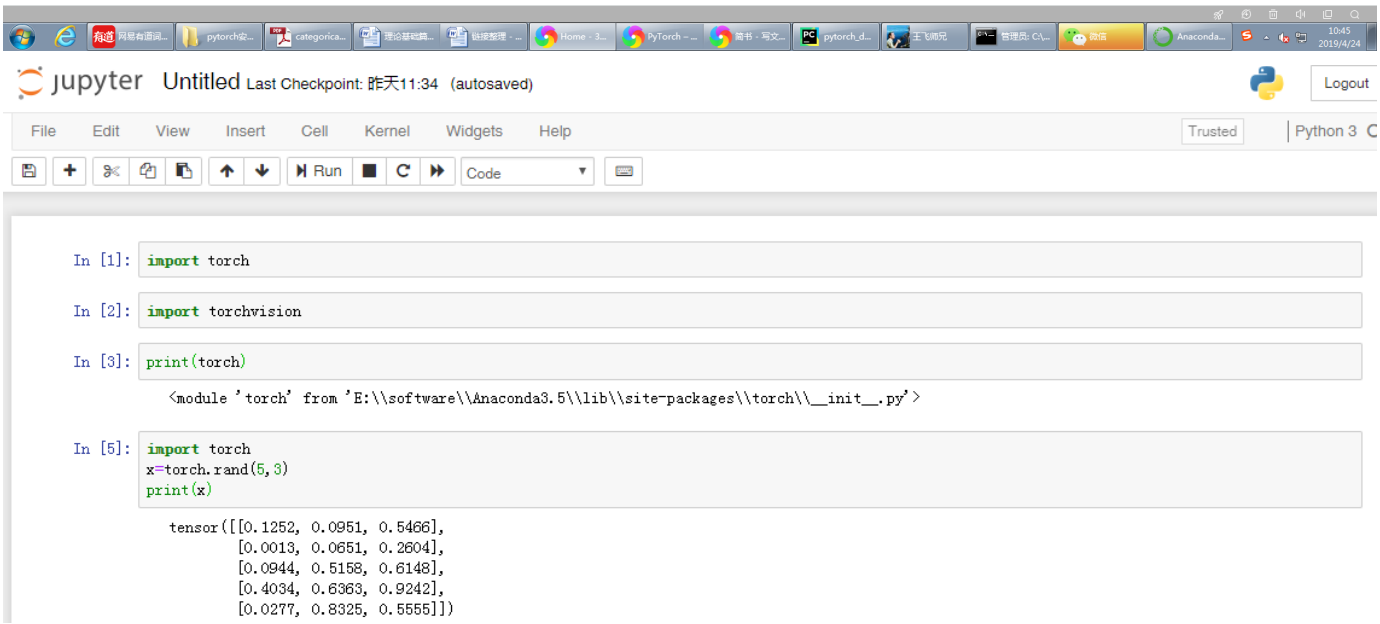


拷贝给出的命令在cmd下运行



安装完毕后，验证是否安装成功，打开Anaconda的Jupyter新建python文件，运行demo：





出现这个结果，那么恭喜你，至此PyTorch1.0 & Anaconda3.5已经安装成功。

什么是 PyTorch?

PyTorch 是一个基于 Python 的科学计算包，主要定位两类人群：

- NumPy 的替代品，可以利用 GPU 的性能进行计算。
- 深度学习研究平台拥有足够的灵活性和速度

开始学习

Tensors (张量)

Tensors 类似于 NumPy 的 ndarrays ，同时 Tensors 可以使用 GPU 进行计算。

```
from __future__ import print_function
import torch
```

构造一个5x3矩阵，不初始化。

```
x = torch.empty(5, 3)
print(x)
```

输出:

```
tensor(1.00000e-04 *
       [[-0.0000,  0.0000,  1.5135],
        [ 0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000]])
```

构造一个随机初始化的矩阵：

```
x = torch.rand(5, 3)
print(x)
```

输出:

```
tensor([[ 0.6291,  0.2581,  0.6414],
        [ 0.9739,  0.8243,  0.2276],
        [ 0.4184,  0.1815,  0.5131],
        [ 0.5533,  0.5440,  0.0718],
        [ 0.2908,  0.1850,  0.5297]])
```

构造一个矩阵全为 0，而且数据类型是 long.

```
x = torch.zeros(5, 3, dtype=torch.long)
print(x)
```

输出:

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

构造一个张量，直接使用数据：

```
x = torch.tensor([5.5, 3])
print(x)
```

输出:

```
tensor([ 5.5000,  3.0000])
```

创建一个 tensor 基于已经存在的 tensor。

```
x = x.new_ones(5, 3, dtype=torch.double)
# new_* methods take in sizes
print(x)

x = torch.randn_like(x, dtype=torch.float)
# override dtype!
print(x)
# result has the same size
```

输出:

```
```

```
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]], dtype=torch.float64)
tensor([[-0.2183,  0.4477, -0.4053],
        [ 1.7353, -0.0048,  1.2177],
        [-1.1111,  1.0878,  0.9722],
        [-0.7771, -0.2174,  0.0412],
        [-2.1750,  1.3609, -0.3322]])
```

获取它的维度信息:

```
print(x.size())
```

输出:

```
torch.Size([5, 3])
```

注意

torch.Size 是一个元组，所以它支持左右的元组操作。

操作

在接下来的例子中，我们将会看到加法操作。

加法: 方式 1

```
y = torch.rand(5, 3)
print(x + y)
```

输出：

```
tensor([[-0.1859,  1.3970,  0.5236],
        [ 2.3854,  0.0707,  2.1970],
        [-0.3587,  1.2359,  1.8951],
        [-0.1189, -0.1376,  0.4647],
        [-1.8968,  2.0164,  0.1092]])
```

加法: 方式2

```
```

```
print(torch.add(x, y))
```

输出：

```
tensor([[-0.1859,  1.3970,  0.5236],
        [ 2.3854,  0.0707,  2.1970],
        [-0.3587,  1.2359,  1.8951],
        [-0.1189, -0.1376,  0.4647],
        [-1.8968,  2.0164,  0.1092]])
```

加法: 提供一个输出 tensor 作为参数

```
result = torch.empty(5, 3)
torch.add(x, y, out=result)
print(result)
```

输出：

```
tensor([[-0.1859,  1.3970,  0.5236],
        [ 2.3854,  0.0707,  2.1970],
        [-0.3587,  1.2359,  1.8951],
        [-0.1189, -0.1376,  0.4647],
        [-1.8968,  2.0164,  0.1092]])
```

加法: in-place

```
# adds x to y
y.add_(x)
print(y)
```

输出：

```
tensor([[-0.1859,  1.3970,  0.5236],
        [ 2.3854,  0.0707,  2.1970],
        [-0.3587,  1.2359,  1.8951],
        [-0.1189, -0.1376,  0.4647],
        [-1.8968,  2.0164,  0.1092]])
```

注意

任何使张量会发生变化的操作都有一个前缀 ''。*例如：x.copy*(y), x.t_(), 将会改变 x.

你可以使用标准的 NumPy 类似的索引操作

```
print(x[:, 1])
```

输出

```
tensor([ 0.4477, -0.0048,  1.0878, -0.2174,  1.3609])
```

改变大小：如果你想改变一个 tensor 的大小或者形状，你可以使用 torch.view:

```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8)  # the size -1 is inferred from other dimensions
print(x.size(), y.size(), z.size())
```

输出

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

如果你有一个元素 tensor ，使用 .item() 来获得这个 value 。

```
x = torch.randn(1)
print(x)
print(x.item())
```

```
tensor([ 0.9422])
0.9422121644020081
```

PyTorch windows 安装教程：两行代码搞定 PyTorch 安装

http://pytorchchina.com/2018/12/11/pytorch-windows-install-1/

PyTorch Mac 安装教程

http://pytorchchina.com/2018/12/11/pytorch-mac-install/

PyTorch Linux 安装教程

http://pytorchchina.com/2018/12/11/pytorch-linux-install/

PyTorch QQ群

PyTorch,Torch,机器学习

扫一扫二维码，加入群聊。

PyTorch 自动微分

autograd 包是 PyTorch 中所有神经网络的核心。首先让我们简要地介绍它，然后我们将会去训练我们的第一个神经网络。该 autograd 软件包为 Tensors 上的所有操作提供自动微分。它是一个由运行定义的框架，这意味着以代码运行方式定义你的后向传播，并且每次迭代都可以不同。我们从 tensor 和 gradients 来举一些例子。

1、TENSOR

torch.Tensor 是包的核心类。如果将其属性 .requires_grad 设置为 True，则会开始跟踪针对 tensor 的所有操作。完成计算后，您可以调用 .backward() 来自动计算所有梯度。该张量的梯度将累积到 .grad 属性中。

要停止 tensor 历史记录的跟踪，您可以调用 .detach()，它将其与计算历史记录分离，并防止将来的计算被跟踪。

要停止跟踪历史记录（和使用内存），您还可以将代码块使用 with torch.no_grad(): 包装起来。在评估模型时，这是特别有用，因为模型在训练阶段具有 requires_grad = True 的可训练参数有利于调参，但在评估阶段我们不需要梯度。

还有一个类对于 autograd 实现非常重要那就是 Function。Tensor 和 Function 互相连接并构建一个非循环图，它保存整个完整的计算过程的历史信息。每个张量都有一个 .grad_fn 属性保存着创建了张量的 Function 的引用，（如果用户自己创建张量，则g rad_fn 是 None ）。

如果你想计算导数，你可以调用 Tensor.backward()。如果 Tensor 是标量（即它包含一个元素数据），则不需要指定任何参数backward()，但是如果它有更多元素，则需要指定一个gradient 参数来指定张量的形状。

```
import torch
```

创建一个张量，设置 requires_grad=True 来跟踪与它相关的计算

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
```

输出：

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

针对张量做一个操作

```
y = x + 2
print(y)
```

输出：

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

y 作为操作的结果被创建，所以它有 grad_fn

```
print(y.grad_fn)
```

输出：

```
<AddBackward0 object at 0x7fe1db427470>
```

针对 y 做更多的操作：

```
z = y * y * 3
out = z.mean()

print(z, out)
```

输出：

```
tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>)
tensor(27., grad_fn=<MeanBackward0>)
```

.requires_grad_(…) 会改变张量的 requires_grad 标记。输入的标记默认为 False ，如果没有提供相应的参数。

```
a = torch.randn(2, 2)
a = ((a * 3) / (a - 1))
print(a.requires_grad)
```

```
a.requires_grad_(True)
print(a.requires_grad)
b = (a * a).sum()
print(b.grad_fn)
```

输出：

```
False
True
<SumBackward0 object at 0x7fe1db427dd8>
```

梯度：

我们现在后向传播，因为输出包含了一个标量，out.backward() 等同于 out.backward(torch.tensor(1.))。

```
out.backward()
```

打印梯度 d(out)/dx

```
print(x.grad)
```

输出：

```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

原理解释：

You should have got a matrix of `4.5`. Let's call the `out` *Tensor* "o". We have that $o = \frac{1}{4}\sum_i z_i$, $z_i = 3(x_i + 2)^2$ and $z_i\big|_{x_i=1} = 27$. Therefore, $\frac{\partial o}{\partial x_i} = \frac{3}{2}(x_i + 2)$, hence $\frac{\partial o}{\partial x_i}\big|_{x_i=1} = \frac{9}{2} = 4.5$.

Mathematically, if you have a vector valued function $\vec{y} = f(\vec{x})$, then the gradient of $\vec{y}$ with respect to $\vec{x}$ is a Jacobian matrix:

$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

Generally speaking, `torch.autograd` is an engine for computing Jacobian-vector product. That is, given any vector $v = \begin{pmatrix} v_1 & v_2 & \cdots & v_m \end{pmatrix}^T$, compute the product $J \cdot v$. If v happens to be the gradient of a scalar function $l = g\left(\vec{y}\right)$, that is, $v = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m} \end{pmatrix}^T$, then by the chain rule, the Jacobian-vector product would be the gradient of l with respect to $\vec{x}$:

$$J \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}$$

This characteristic of Jacobian-vector product makes it very convenient to feed external gradients into a model that has non-scalar output.

现在让我们看一个雅可比向量积的例子：

```
x = torch.randn(3, requires_grad=True)

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

print(y)
```

输出：

```
tensor([ -444.6791,   762.9810, -1690.0941], grad_fn=<MulBackward0>)
```

现在在这种情况下，y 不再是一个标量。torch.autograd 不能够直接计算整个雅可比，但是如果我们只想要雅可比向量积，只需要简单的传递向量给 backward 作为参数。

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

print(x.grad)
```

输出：

```
tensor([1.0240e+02, 1.0240e+03, 1.0240e-01])
```

你可以通过将代码包裹在 with torch.no_grad()，来停止对从跟踪历史中 的 .requires_grad=True 的张量自动求导。

```
print(x.requires_grad)
print((x ** 2).requires_grad)

with torch.no_grad():
    print((x ** 2).requires_grad)
```

输出：

```
True
True
False
```

稍后可以阅读：

autograd 和 Function 的文档在： https://pytorch.org/docs/autograd

下载 Python 源代码：

autograd_tutorial.py

下载 Jupyter 源代码：

autograd_tutorial.ipynb

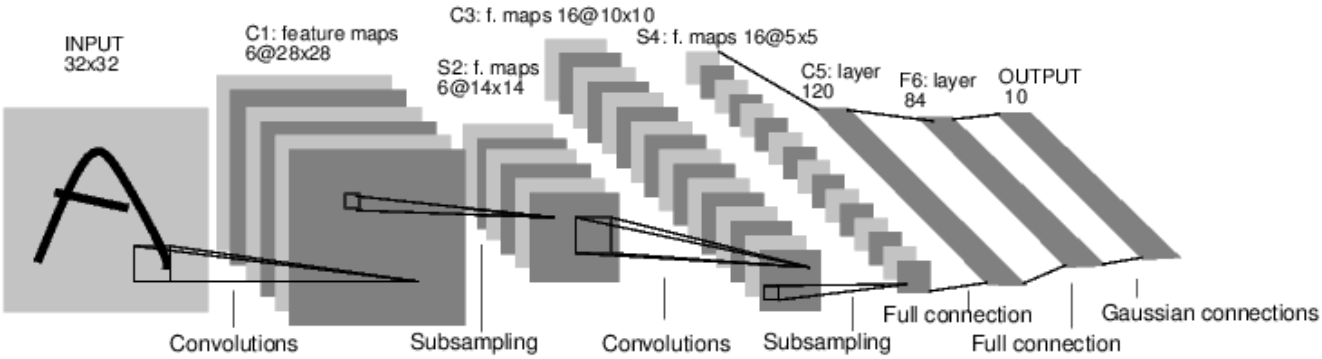
PyTorch 神经网络

神经网络

神经网络可以通过 torch.nn 包来构建。

现在对于自动梯度(autograd)有一些了解，神经网络是基于自动梯度 (autograd)来定义一些模型。一个 nn.Module 包括层和一个方法 forward(input) 它会返回输出(output)。

例如，看一下数字图片识别的网络：



这是一个简单的前馈神经网络，它接收输入，让输入一个接着一个的通过一些层，最后给出输出。

一个典型的神经网络训练过程包括以下几点：

1.定义一个包含可训练参数的神经网络

2.迭代整个输入

3.通过神经网络处理输入

4.计算损失(loss)

5.反向传播梯度到神经网络的参数

6.更新网络的参数，典型的用一个简单的更新方法：weight = weight - learning_rate *gradient

定义神经网络

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


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        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:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


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

输出：

```
Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
```

```
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

你刚定义了一个前馈函数，然后反向传播函数被自动通过 autograd 定义了。你可以使用任何张量操作在前馈函数上。

一个模型可训练的参数可以通过调用 net.parameters() 返回：

```
params = list(net.parameters())
print(len(params))
print(params[0].size())  # conv1's .weight
```

输出：

```
10
torch.Size([6, 1, 5, 5])
```

让我们尝试随机生成一个 32x32 的输入。注意：期望的输入维度是 32x32 。为了使用这个网络在 MNIST 数据及上，你需要把数据集中的图片维度修改为 32x32。

```
input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)
```

输出：

```
tensor([[-0.0233,  0.0159, -0.0249,  0.1413,  0.0663,  0.0297, -0.0940, -0.0135,
          0.1003, -0.0559]], grad_fn=<AddmmBackward>)
```

把所有参数梯度缓存器置零，用随机的梯度来反向传播

```
net.zero_grad()
out.backward(torch.randn(1, 10))
```

在继续之前，让我们复习一下所有见过的类。

torch.Tensor - A multi-dimensional array with support for autograd operations like backward(). Also holds the gradient w.r.t. the tensor.

nn.Module - Neural network module. Convenient way of encapsulating parameters, with helpers for moving them to GPU, exporting, loading, etc.

nn.Parameter - A kind of Tensor, that is automatically registered as a parameter when assigned as an attribute to a Module.

autograd.Function - Implements forward and backward definitions of an autograd operation. Every Tensor operation, creates at least a single Function node, that connects to functions that created a Tensor and encodes its history.

在此，我们完成了：

1.定义一个神经网络

2.处理输入以及调用反向传播

还剩下：

1.计算损失值

2.更新网络中的权重

损失函数

一个损失函数需要一对输入：模型输出和目标，然后计算一个值来评估输出距离目标有多远。

有一些不同的损失函数在 nn 包中。一个简单的损失函数就是 nn.MSELoss ，这计算了均方误差。

例如：

```
output = net(input)
target = torch.randn(10)  # a dummy target, for example
target = target.view(1, -1)  # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
```

输出：

```
tensor(1.3389, grad_fn=<MseLossBackward>)
```

现在，如果你跟随损失到反向传播路径，可以使用它的 .grad_fn 属性，你将会看到一个这样的计算图：

```
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d
      -> view -> linear -> relu -> linear -> relu -> linear
      -> MSELoss
      -> loss
```

所以，当我们调用 loss.backward()，整个图都会微分，而且所有的在图中的requires_grad=True 的张量将会让他们的 grad 张量累计梯度。

为了演示，我们将跟随以下步骤来反向传播。

```
print(loss.grad_fn)  # MSELoss
print(loss.grad_fn.next_functions[0][0])  # Linear
print(loss.grad_fn.next_functions[0][0].next_functions[0][0])  # ReLU
```

输出：

```
<MseLossBackward object at 0x7fab77615278>
<AddmmBackward object at 0x7fab77615940>
<AccumulateGrad object at 0x7fab77615940>
```

反向传播

为了实现反向传播损失，我们所有需要做的事情仅仅是使用 loss.backward()。你需要清空现存的梯度，要不然帝都将会和现存的梯度累计到一起。

现在我们调用 loss.backward() ，然后看一下 con1 的偏置项在反向传播之前和之后的变化。

```
net.zero_grad()     # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

输出：

```
conv1.bias.grad before backward
tensor([0., 0., 0., 0., 0., 0.])
```

```
conv1.bias.grad after backward
tensor([-0.0054,  0.0011,  0.0012,  0.0148, -0.0186,  0.0087])
```

现在我们看到了，如何使用损失函数。

唯一剩下的事情就是更新神经网络的参数。

更新神经网络参数：

最简单的更新规则就是随机梯度下降。

```
weight = weight - learning_rate * gradient
```

我们可以使用 python 来实现这个规则：

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

尽管如此，如果你是用神经网络，你想使用不同的更新规则，类似于 SGD, Nesterov-SGD, Adam, RMSProp, 等。为了让这可行，我们建立了一个小包：torch.optim 实现了所有的方法。使用它非常的简单。

```
import torch.optim as optim

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

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

下载 Python 源代码：

neural_networks_tutorial.py

下载 Jupyter 源代码：

neural_networks_tutorial.ipynb

PyTorch 图像分类器

你已经了解了如何定义神经网络，计算损失值和网络里权重的更新。

现在你也许会想应该怎么处理数据？

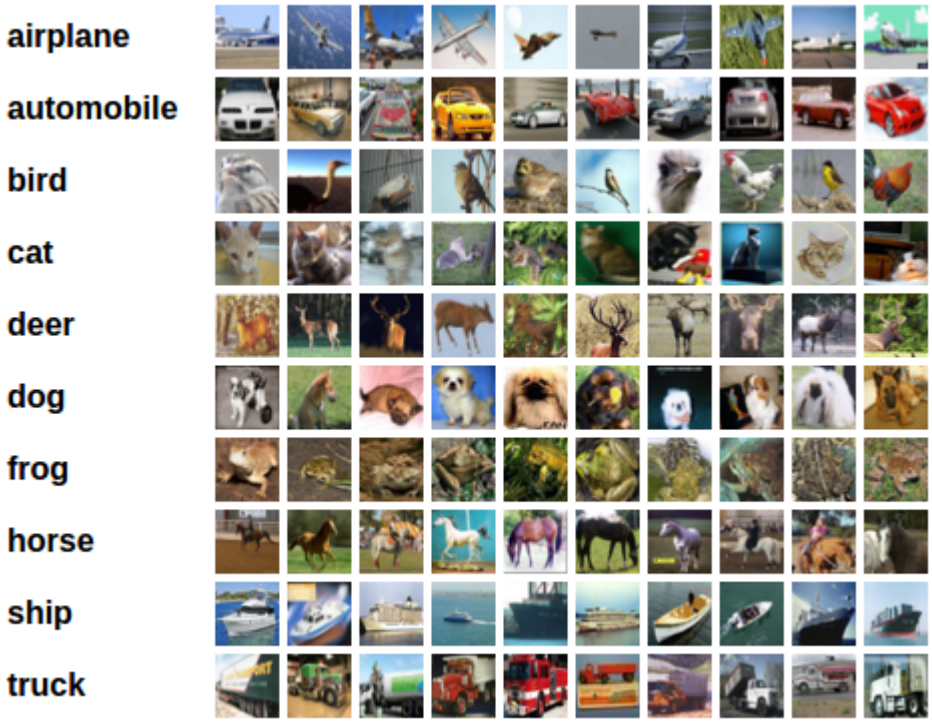
通常来说，当你处理图像，文本，语音或者视频数据时，你可以使用标准 python 包将数据加载成 numpy 数组格式，然后将这个数组转换成 torch.*Tensor

- 对于图像，可以用 Pillow，OpenCV
- 对于语音，可以用 scipy，librosa
- 对于文本，可以直接用 Python 或 Cython 基础数据加载模块，或者用 NLTK 和 SpaCy

特别是对于视觉，我们已经创建了一个叫做 totchvision 的包，该包含有支持加载类似Imagenet，CIFAR10，MNIST 等公共数据集的数据加载模块 torchvision.datasets 和支持加载图像数据数据转换模块 torch.utils.data.DataLoader。

这提供了极大的便利，并且避免了编写“样板代码”。

对于本教程，我们将使用CIFAR10数据集，它包含十个类别：‘airplane’, ‘automobile’, ‘bird’, ‘cat’, ‘deer’, ‘dog’, ‘frog’, ‘horse’, ‘ship’, ‘truck’。CIFAR-10 中的图像尺寸为3*32*32，也就是RGB的3层颜色通道，每层通道内的尺寸为32*32。



训练一个图像分类器

我们将按次序的做如下几步：

1. 使用torchvision加载并且归一化CIFAR10的训练和测试数据集
2. 定义一个卷积神经网络
3. 定义一个损失函数
4. 在训练样本数据上训练网络
5. 在测试样本数据上测试网络

加载并归一化 CIFAR10 使用 torchvision ,用它来加载 CIFAR10 数据非常简单。

```
import torch
import torchvision
import torchvision.transforms as transforms
```

torchvision 数据集的输出是范围在[0,1]之间的 PILImage，我们将他们转换成归一化范围为[-1,1]之间的张量 Tensors。

```
transform = transforms.Compose(
    [transforms.ToTensor(),
```

```
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

输出：

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/
cifar-10-python.tar.gz
Files already downloaded and verified
```

让我们来展示其中的一些训练图片。

```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

输出：

```
cat plane  ship  frog
```

定义一个卷积神经网络 在这之前先 从神经网络章节 复制神经网络，并修改它为3通道的图片(在此之前它被定义为1通道)

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


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        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 = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

定义一个损失函数和优化器 让我们使用分类交叉熵Cross-Entropy 作损失函数，动量SGD做优化器。

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

训练网络 这里事情开始变得有趣，我们只需要在数据迭代器上循环传给网络和优化器 输入就可以。

```
for epoch in range(2):  # loop over the dataset multiple times
```

```
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

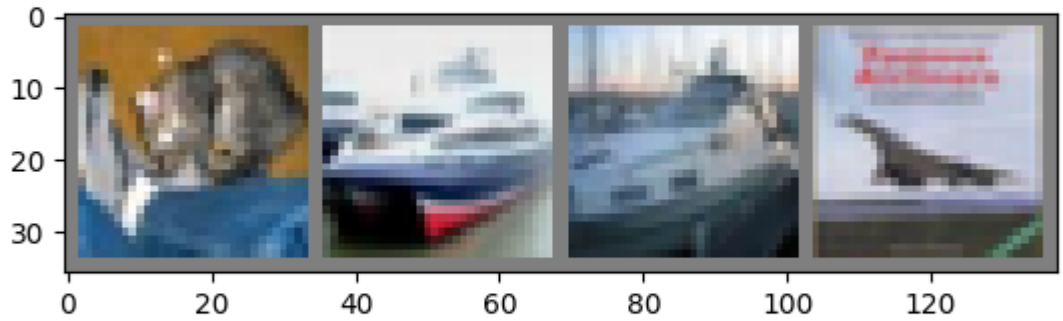
输出：

```
[1,  2000] loss: 2.187
[1,  4000] loss: 1.852
[1,  6000] loss: 1.672
[1,  8000] loss: 1.566
[1, 10000] loss: 1.490
[1, 12000] loss: 1.461
[2,  2000] loss: 1.389
[2,  4000] loss: 1.364
[2,  6000] loss: 1.343
[2,  8000] loss: 1.318
[2, 10000] loss: 1.282
[2, 12000] loss: 1.286
Finished Training
```

在测试集上测试网络 我们已经通过训练数据集对网络进行了2次训练，但是我们需要检查网络是否已经学到了东西。

我们将用神经网络的输出作为预测的类标来检查网络的预测性能，用样本的真实类标来校对。如果预测是正确的，我们将样本添加到正确预测的列表里。

好的，第一步，让我们从测试集中显示一张图像来熟悉它。



输出：

```
GroundTruth:    cat  ship  ship plane
```

现在让我们看看 神经网络认为这些样本应该预测成什么：

```
outputs = net(images)
```

输出是预测与十个类的近似程度，与某一个类的近似程度越高，网络就越认为图像是属于这一类别。所以让我们打印其中最相似类别类标：

```
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```

输出：

```
Predicted:    cat  ship   car  ship
```

结果看起开非常好，让我们看看网络在整个数据集上的表现。

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

输出：

```
Accuracy of the network on the 10000 test images: 54 %
```

这看起来比随机预测要好，随机预测的准确率为10%（随机预测出为10类中的哪一类）。看来网络学到了东西。

```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

输出：

```
Accuracy of plane : 57 %
Accuracy of   car : 73 %
Accuracy of  bird : 49 %
Accuracy of   cat : 54 %
Accuracy of  deer : 18 %
Accuracy of   dog : 20 %
Accuracy of  frog : 58 %
Accuracy of horse : 74 %
Accuracy of  ship : 70 %
Accuracy of truck : 66 %
```

所以接下来呢？

我们怎么在GPU上跑这些神经网络？

在GPU上训练 就像你怎么把一个张量转移到GPU上一样，你要将神经网络转到GPU上。 如果CUDA可以用，让我们首先定义下我们的设备为第一个可见的cuda设备。

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Assume that we are on a CUDA machine, then this should print a CUDA device:

print(device)
```

输出：

```
cuda:0
```

本节剩余部分都会假定设备就是台CUDA设备。

接着这些方法会递归地遍历所有模块，并将它们的参数和缓冲器转换为CUDA张量。

```
net.to(device)
```

记住你也必须在每一个步骤向GPU发送输入和目标：

```
inputs, labels = inputs.to(device), labels.to(device)
```

为什么没有注意到与CPU相比巨大的加速？因为你的网络非常小。

练习：尝试增加你的网络宽度（首个 nn.Conv2d 参数设定为 2，第二个nn.Conv2d参数设定为1–它们需要有相同的个数），看看会得到怎么的速度提升。

目标：

- 深度理解了PyTorch的张量和神经网络
- 训练了一个小的神经网络来分类图像

在多个GPU上训练

如果你想要来看到大规模加速，使用你的所有GPU，请查看：数据并行性（https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html）。PyTorch 60 分钟入门教程：数据并行处理

http://pytorchchina.com/2018/12/11/optional-data-parallelism/

下载 Python 源代码：

cifar10_tutorial.py

下载 Jupyter 源代码：

cifar10_tutorial.ipynb

可选择：数据并行处理（文末有完整代码下载） 作者：Sung Kim 和 Jenny Kang

在这个教程中，我们将学习如何用 DataParallel 来使用多 GPU。 通过 PyTorch 使用多个 GPU 非常简单。你可以将模型放在一个 GPU：

```
device = torch.device("cuda:0")
model.to(device)
```

然后，你可以复制所有的张量到 GPU：

```
mytensor = my_tensor.to(device)
```

请注意，只是调用 my_tensor.to(device) 返回一个 my_tensor 新的复制在GPU上，而不是重写 my_tensor。你需要分配给他一个新的张量并且在 GPU 上使用这个张量。

在多 GPU 中执行前馈，后馈操作是非常自然的。尽管如此，PyTorch 默认只会使用一个 GPU。通过使用 DataParallel 让你的模型并行运行，你可以很容易的在多 GPU 上运行你的操作。

```
model = nn.DataParallel(model)
```

这是整个教程的核心，我们接下来将会详细讲解。 引用和参数

引入 PyTorch 模块和定义参数

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
```

参数

```
input_size = 5
output_size = 2

batch_size = 30
data_size = 100
```

设备

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

实验（玩具）数据

生成一个玩具数据。你只需要实现 getitem.

```
class RandomDataset(Dataset):

    def __init__(self, size, length):
        self.len = length
        self.data = torch.randn(length, size)

    def __getitem__(self, index):
        return self.data[index]

    def __len__(self):
        return self.len

rand_loader = DataLoader(dataset=RandomDataset(input_size,
data_size),batch_size=batch_size, shuffle=True)
```

简单模型

为了做一个小 demo，我们的模型只是获得一个输入，执行一个线性操作，然后给一个输出。尽管如此，你可以使用 DataParallel 在任何模型(CNN, RNN, Capsule Net 等等.)

我们放置了一个输出声明在模型中来检测输出和输入张量的大小。请注意在 batch rank 0 中的输出。

```
class Model(nn.Module):
    # Our model

    def __init__(self, input_size, output_size):
        super(Model, self).__init__()
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, input):
        output = self.fc(input)
        print("\tIn Model: input size", input.size(),
              "output size", output.size())

        return output
```

创建模型并且数据并行处理

这是整个教程的核心。首先我们需要一个模型的实例，然后验证我们是否有多个 GPU。如果我们有多个 GPU，我们可以用 nn.DataParallel 来 包裹 我们的模型。然后我们使用 model.to(device) 把模型放到多 GPU 中。

```
model = Model(input_size, output_size)
if torch.cuda.device_count() > 1:
  print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
  model = nn.DataParallel(model)

model.to(device)
```

输出：

```
Let's use 2 GPUs!
```

运行模型： 现在我们可以看到输入和输出张量的大小了。

```
for data in rand_loader:
    input = data.to(device)
    output = model(input)
    print("Outside: input size", input.size(),
          "output_size", output.size())
```

输出：

```
In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
        In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
        In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
        In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
        In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
        In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
        In Model: input size torch.Size([5, 5]) output size torch.Size([5, 2])
        In Model: input size torch.Size([5, 5]) output size torch.Size([5, 2])
Outside: input size torch.Size([10, 5]) output_size torch.Size([10, 2])
```

结果：

如果你没有 GPU 或者只有一个 GPU，当我们获取 30 个输入和 30 个输出，模型将期望获得 30 个输入和 30 个输出。但是如果你有多个 GPU ，你会获得这样的结果。

多 GPU

如果你有 2 个GPU，你会看到：

```
# on 2 GPUs
Let‘s use 2 GPUs!
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([5, 5]) output size torch.Size([5, 2])
    In Model: input size torch.Size([5, 5]) output size torch.Size([5, 2])
Outside: input size torch.Size([10, 5]) output_size torch.Size([10, 2])
```

如果你有 3个GPU，你会看到：

```
Let‘s use 3 GPUs!
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
Outside: input size torch.Size([10, 5]) output_size torch.Size([10, 2])
```

如果你有 8个GPU，你会看到：

```
Let‘s use 8 GPUs!
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
```

```
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
Outside: input size torch.Size([10, 5]) output_size torch.Size([10, 2])
```

总结

数据并行自动拆分了你的数据并且将任务单发送到多个 GPU 上。当每一个模型都完成自己的任务之后，DataParallel 收集并且合并这些结果，然后再返回给你。

更多信息，请访问： https://pytorch.org/tutorials/beginner/former_torchies/parallelism_tutorial.html

下载 Python 版本完整代码：

data_parallel_tutorial.py

下载 jupyter notebook 版本完整代码：

data_parallel_tutorial.ipynb

加入 PyTorch 交流 QQ 群：

PyTorch之数据加载和处理

PyTorch提供了许多工具来简化和希望数据加载，使代码更具可读性。

1.下载安装包

- scikit-image：用于图像的IO和变换
- pandas：用于更容易地进行csv解析

```
from __future__ import print_function, division
import os
import torch
import pandas as pd              #用于更容易地进行csv解析
from skimage import io, transform    #用于图像的IO和变换
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, utils

# 忽略警告
import warnings
warnings.filterwarnings("ignore")

plt.ion()   # interactive mode
```

2.下载数据集

从此处下载数据集，数据存于“data / faces /”的目录中。这个数据集实际上是imagenet数据集标注为face的图片当中在 dlib 面部检测 (dlib’s pose estimation) 表现良好的图片。我们要处理的是一

个面部姿态的数据集。也就是按如下方式标注的人脸:

2.1 数据集注释

数据集是按如下规则打包成的csv文件:

```
image_name,part_0_x,part_0_y,part_1_x,part_1_y,part_2_x, ... ,part_67_x,part_67_y
0805personali01.jpg,27,83,27,98, ... 84,134
1084239450_e76e00b7e7.jpg,70,236,71,257, ... ,128,312
```

3.读取数据集

将csv中的标注点数据读入（N，2）数组中，其中N是特征点的数量。读取数据代码如下：

```
```

```
landmarks_frame = pd.read_csv('data/faces/face_landmarks.csv')

n = 65
img_name = landmarks_frame.iloc[n, 0]
landmarks = landmarks_frame.iloc[n, 1:].as_matrix()
landmarks = landmarks.astype('float').reshape(-1, 2)

print('Image name: {}'.format(img_name))
print('Landmarks shape: {}'.format(landmarks.shape))
print('First 4 Landmarks: {}'.format(landmarks[:4]))
```

3.1 数据结果

输出：

```
Image name: person-7.jpg
Landmarks shape: (68, 2)
First 4 Landmarks: [[32. 65.]
 [33. 76.]
 [34. 86.]
 [34. 97.]]
```

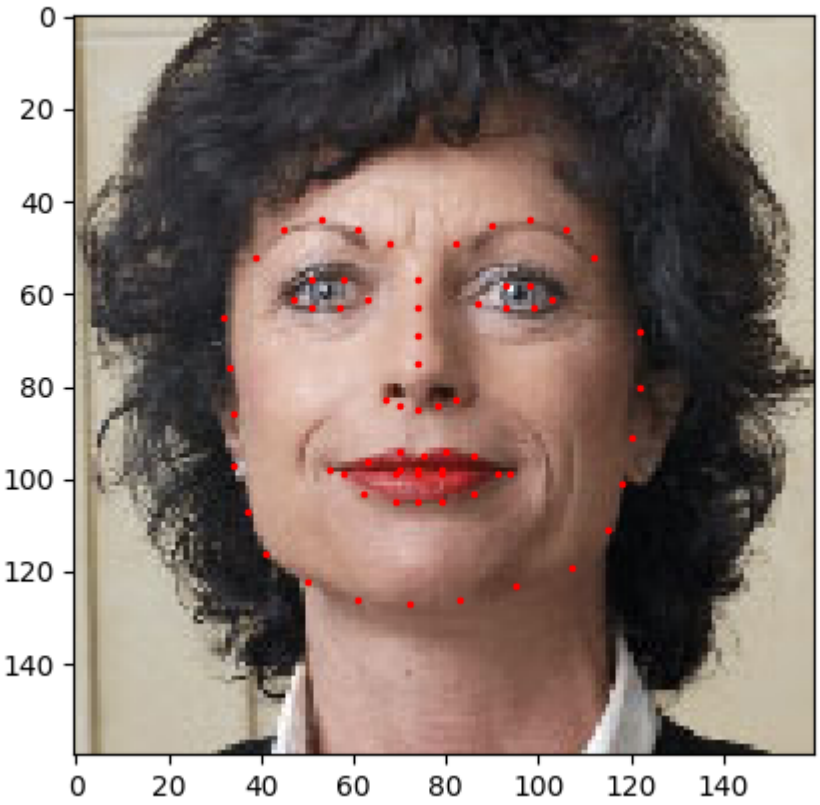
4 编写函数

写一个简单的函数来展示一张图片和它对应的标注点作为例子。

```
def show_landmarks(image, landmarks):
    """显示带有地标的图片"""
    plt.imshow(image)
    plt.scatter(landmarks[:, 0], landmarks[:, 1], s=10, marker='.', c='r')
    plt.pause(0.001)  # pause a bit so that plots are updated

plt.figure()
show_landmarks(io.imread(os.path.join('data/faces/', img_name)),
               landmarks)
plt.show()
```

函数展示结果如下图所示:



5.数据集类

`torch.utils.data.Dataset` 是表示数据集的抽象类，因此自定义数据集应继承Dataset并覆盖以下方法 * `__len__` 实现 `len(dataset)` 返还数据集的尺寸。 * `__getitem__` 用来获取一些索引数据，例如 dataset[i] 中的(i)。

5.1 建立数据集类

为面部数据集创建一个数据集类。我们将在 `__init__` 中读取csv的文件内容，在 `__getitem__` 中读取图片。这么做是为了节省内存 空间。只有在需要用到图片的时候才读取它而不是一开始就把图片全部存进内存里。

我们的数据样本将按这样一个字典 `{'image': image, 'landmarks': landmarks}` 组织。 我们的数据集类将添加一个可选参数 `transform` 以方便对样本进行预处理。下一节我们会看到什么时候需要用到 `transform` 参数。 `__init__` 方法如下图所示：

```
class FaceLandmarksDataset(Dataset):
    """面部标记数据集."""

    def __init__(self, csv_file, root_dir, transform=None):
        """
        csv_file (string) ：带注释的csv文件的路径。
        root_dir (string) ：包含所有图像的目录。
        transform (callable, optional) ：一个样本上的可用的可选变换
        """
        self.landmarks_frame = pd.read_csv(csv_file)
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.landmarks_frame)

    def __getitem__(self, idx):
        img_name = os.path.join(self.root_dir,
                                self.landmarks_frame.iloc[idx, 0])
        image = io.imread(img_name)
        landmarks = self.landmarks_frame.iloc[idx, 1:]
        landmarks = np.array([landmarks])
        landmarks = landmarks.astype('float').reshape(-1, 2)
        sample = {'image': image, 'landmarks': landmarks}

        if self.transform:
            sample = self.transform(sample)

        return sample
```

6.数据可视化

实例化这个类并遍历数据样本。我们将会打印出前四个例子的尺寸并展示标注的特征点。 代码如下图所示：

```
face_dataset = FaceLandmarksDataset(csv_file='data/faces/face_landmarks.csv',
                                    root_dir='data/faces/')

fig = plt.figure()

for i in range(len(face_dataset)):
    sample = face_dataset[i]

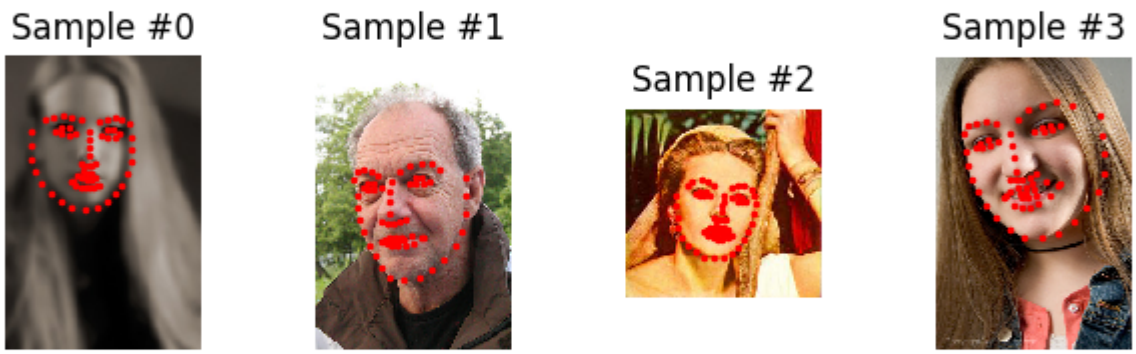
    print(i, sample['image'].shape, sample['landmarks'].shape)
```

```
    ax = plt.subplot(1, 4, i + 1)
    plt.tight_layout()
    ax.set_title('Sample #{}'.format(i))
    ax.axis('off')
    show_landmarks(**sample)

    if i == 3:
        plt.show()
        break
```

数据结果：

6.1 图形展示结果



6.2 控制台输出结果:

```
0 (324, 215, 3) (68, 2)
1 (500, 333, 3) (68, 2)
2 (250, 258, 3) (68, 2)
3 (434, 290, 3) (68, 2)
```

7.数据变换

通过上面的例子我们会发现图片并不是同样的尺寸。绝大多数神经网络都假定图片的尺寸相同。因此我们需要做一些预处理。让我们创建三个转换: * `Rescale` ：缩放图片 * `RandomCrop` ：对图片进行随机裁剪。这是一种数据增强操作 * `ToTensor` ：把numpy格式图片转为torch格式图片 (我们需要交换坐标轴).

我们会把它们写成可调用的类的形式而不是简单的函数，这样就不需要每次调用时传递一遍参数。我们只需要实现 `__call__` 方法，必 要的时候实现 `__init__` 方法。我们可以这样调用这些转换:

```
tsfm = Transform(params)
transformed_sample = tsfm(sample)
```

观察下面这些转换是如何应用在图像和标签上的。

```
class Rescale(object):
    """将样本中的图像重新缩放到给定大小。.

    Args:
        output_size (tuple或int) ：所需的输出大小。 如果是元组，则输出为
         与output_size匹配。 如果是int，则匹配较小的图像边缘到output_size保持纵横比相同。
    """

    def __init__(self, output_size):
        assert isinstance(output_size, (int, tuple))
        self.output_size = output_size

    def __call__(self, sample):
        image, landmarks = sample['image'], sample['landmarks']

        h, w = image.shape[:2]
        if isinstance(self.output_size, int):
            if h > w:
                new_h, new_w = self.output_size * h / w, self.output_size
            else:
                new_h, new_w = self.output_size, self.output_size * w / h
        else:
            new_h, new_w = self.output_size

        new_h, new_w = int(new_h), int(new_w)

        img = transform.resize(image, (new_h, new_w))
```

```
        # h and w are swapped for landmarks because for images,
        # x and y axes are axis 1 and 0 respectively
        landmarks = landmarks * [new_w / w, new_h / h]

        return {'image': img, 'landmarks': landmarks}


class RandomCrop(object):
    """随机裁剪样本中的图像.

    Args:
       output_size (tuple或int) ：所需的输出大小。 如果是int，方形裁剪是。
    """

    def __init__(self, output_size):
        assert isinstance(output_size, (int, tuple))
        if isinstance(output_size, int):
            self.output_size = (output_size, output_size)
        else:
            assert len(output_size) == 2
            self.output_size = output_size

    def __call__(self, sample):
        image, landmarks = sample['image'], sample['landmarks']

        h, w = image.shape[:2]
        new_h, new_w = self.output_size

        top = np.random.randint(0, h - new_h)
        left = np.random.randint(0, w - new_w)

        image = image[top: top + new_h,
                      left: left + new_w]

        landmarks = landmarks - [left, top]

        return {'image': image, 'landmarks': landmarks}


class ToTensor(object):
    """将样本中的ndarrays转换为Tensors."""

    def __call__(self, sample):
        image, landmarks = sample['image'], sample['landmarks']

        # 交换颜色轴因为
        # numpy包的图片是: H * W * C
        # torch包的图片是: C * H * W
```

```
        image = image.transpose((2, 0, 1))
        return {'image': torch.from_numpy(image),
                'landmarks': torch.from_numpy(landmarks)}
```

8.组合转换

接下来我们把这些转换应用到一个例子上。

我们想要把图像的短边调整为256，然后随机裁剪（randomcrop）为224大小的正方形。也就是说，我们打算组合一个 `Rescale` 和 `RandomCrop` 的变换。 我们可以调用一个简单的类 `torchvision.transforms.Compose` 来实现这一操作。具体实现如下图：

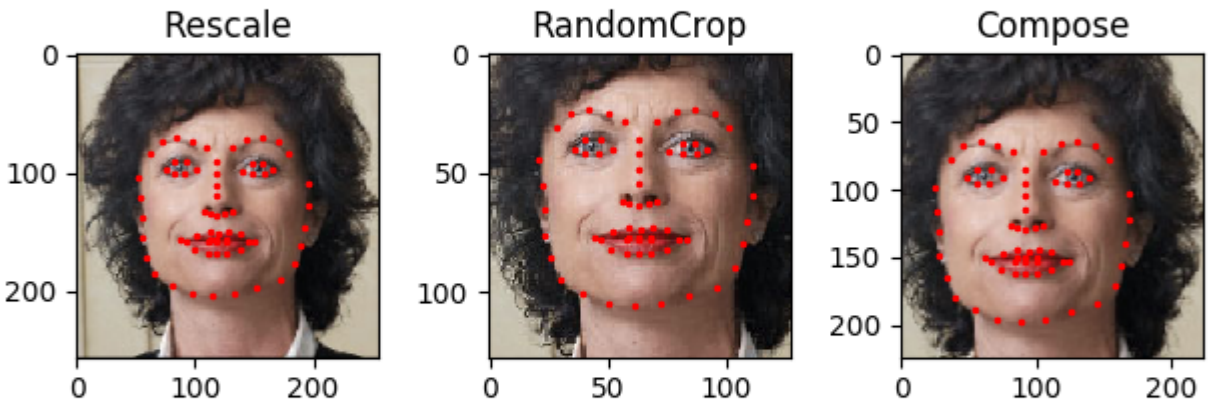
```
scale = Rescale(256)
crop = RandomCrop(128)
composed = transforms.Compose([Rescale(256),
                               RandomCrop(224)])

# 在样本上应用上述的每个变换。
fig = plt.figure()
sample = face_dataset[65]
for i, tsfrm in enumerate([scale, crop, composed]):
    transformed_sample = tsfrm(sample)

    ax = plt.subplot(1, 3, i + 1)
    plt.tight_layout()
    ax.set_title(type(tsfrm).__name__)
    show_landmarks(**transformed_sample)

plt.show()
```

* 输出效果：



9.迭代数据集

让我们把这些整合起来以创建一个带组合转换的数据集。总结一下，每次这个数据集被采样时: * 及时地从文件中读取图片 * 对读取的图片应用转换 * 由于其中一步操作是随机的 (randomcrop)，数据被增强了

我们可以像之前那样使用 `for i in range` 循环来对所有创建的数据集执行同样的操作。

```
transformed_dataset = FaceLandmarksDataset(csv_file='data/faces/
face_landmarks.csv',
                                           root_dir='data/faces/',
                                           transform=transforms.Compose([
                                               Rescale(256),
                                               RandomCrop(224),
                                               ToTensor()
                                           ]))

for i in range(len(transformed_dataset)):
    sample = transformed_dataset[i]
```

```
    print(i, sample['image'].size(), sample['landmarks'].size())

    if i == 3:
        break
```

* 输出结果：

```
0 torch.Size([3, 224, 224]) torch.Size([68, 2])
1 torch.Size([3, 224, 224]) torch.Size([68, 2])
2 torch.Size([3, 224, 224]) torch.Size([68, 2])
3 torch.Size([3, 224, 224]) torch.Size([68, 2])
```

但是，对所有数据集简单的使用 `for` 循环牺牲了许多功能，尤其是: * 批量处理数据 * 打乱数据 * 使用多线程 `multiprocessingworker` 并行加载数据。

`torch.utils.data.DataLoader` 是一个提供上述所有这些功能的迭代器。下面使用的参数必须是清楚的。一个值得关注的参数是 `collate_fn`，可以通过它来决定如何对数据进行批处理。但是绝大多数情况下默认值就能运行良好。

```
dataloader = DataLoader(transformed_dataset, batch_size=4,
                        shuffle=True, num_workers=4)


# 辅助功能：显示批次
def show_landmarks_batch(sample_batched):
    """Show image with landmarks for a batch of samples."""
    images_batch, landmarks_batch = \
            sample_batched['image'], sample_batched['landmarks']
    batch_size = len(images_batch)
    im_size = images_batch.size(2)
    grid_border_size = 2

    grid = utils.make_grid(images_batch)
    plt.imshow(grid.numpy().transpose((1, 2, 0)))

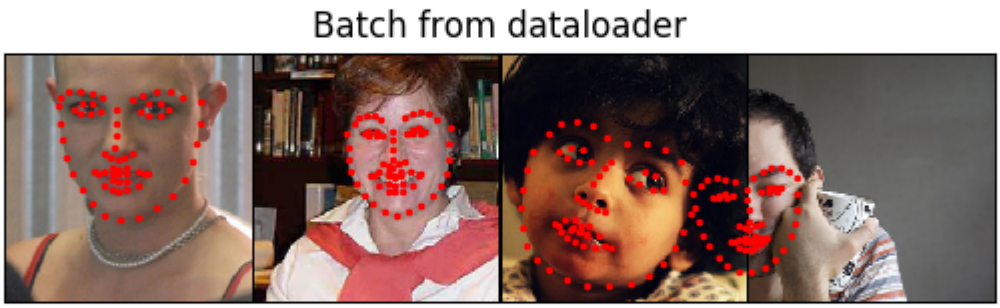
    for i in range(batch_size):
        plt.scatter(landmarks_batch[i, :, 0].numpy() + i * im_size + (i + 1) *
grid_border_size,
                    landmarks_batch[i, :, 1].numpy() + grid_border_size,
                    s=10, marker='.', c='r')

        plt.title('Batch from dataloader')

for i_batch, sample_batched in enumerate(dataloader):
```

```
    print(i_batch, sample_batched['image'].size(),
          sample_batched['landmarks'].size())

    # 观察第4批次并停止。
    if i_batch == 3:
        plt.figure()
        show_landmarks_batch(sample_batched)
        plt.axis('off')
        plt.ioff()
        plt.show()
        break
```



- 输出

```
0 torch.Size([4, 3, 224, 224]) torch.Size([4, 68, 2])
1 torch.Size([4, 3, 224, 224]) torch.Size([4, 68, 2])
2 torch.Size([4, 3, 224, 224]) torch.Size([4, 68, 2])
3 torch.Size([4, 3, 224, 224]) torch.Size([4, 68, 2])
```

10.后记：torchvision

在这篇教程中我们学习了如何构造和使用数据集类(datasets),转换(transforms)和数据加载器(dataloader)。 `torchvision` 包提供了 常用的数据集类(datasets)和转换(transforms)。你可能不需要自己构造这些类。 `torchvision` 中还有一个更常用的数据集类 `ImageFolder` 。 它假定了数据集是以如下方式构造的:

```
root/ants/xxx.png
root/ants/xxy.jpeg
root/ants/xxz.png
.
.
.
root/bees/123.jpg
root/bees/nsdf3.png
root/bees/asd932_.png
```

其中'ants',bees'等是分类标签。在 `PIL.Image` 中你也可以使用类似的转换(transforms)例如 `RandomHorizontalFlip` , `Scale` 。利 用这些你可以按如下的方式创建一个数据加载器(dataloader):

```
import torch
from torchvision import transforms, datasets

data_transform = transforms.Compose([
        transforms.RandomSizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225])
    ])
hymenoptera_dataset = datasets.ImageFolder(root='hymenoptera_data/train',
                                           transform=data_transform)
dataset_loader = torch.utils.data.DataLoader(hymenoptera_dataset,
                                             batch_size=4, shuffle=True,
                                             num_workers=4)
```

PyTorch之小试牛刀

1 PyTorch的核心是两个主要特征：

- 一个n维张量，类似于numpy，但可以在GPU上运行
- 搭建和训练神经网络时的自动微分/求导机制

本章节我们将使用全连接的ReLU网络作为运行示例。该网络将有一个单一的隐藏层，并将使用梯度下降训练，通过最小化网络输出和真正 结果的欧几里得距离，来拟合随机生成的数据。

2.张量

2.1 热身: Numpy

在介绍PyTorch之前，本章节将首先使用numpy实现网络。 Numpy提供了一个n维数组对象，以及许多用于操作这些数组的 函数。Numpy是用于科学计算的通用框架;它对计算图、深度学习和梯度一无所知。然而，我们可以很容易地使用NumPy，手动实现网络的 前向和反向传播，来拟合随机数据：

```
# -*- coding: utf-8 -*-
import numpy as np

# N是批量大小; D_in是输入维度;
# 49/5000 H是隐藏的维度; D_out是输出维度。
N, D_in, H, D_out = 64, 1000, 100, 10

# 创建随机输入和输出数据
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# 随机初始化权重
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for t in range(500):
    # 前向传递：计算预测值y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
```

```
    y_pred = h_relu.dot(w2)

    # 计算和打印损失loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # 反向传播，计算w1和w2对loss的梯度
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # 更新权重
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

2.2 PyTorch：张量

Numpy是一个很棒的框架，但它不能利用GPU来加速其数值计算。 对于现代深度神经网络，GPU通常提供50倍或更高的加速，所以，numpy不能满足当代深度学习的需求。

在这里，先介绍最基本的PyTorch概念：

张量（Tensor）：PyTorch的tensor在概念上与numpy的array相同： tensor是一个n维数组，PyTorch提供了许多函数用于操作这些张量。任何希望使用NumPy执行的计算也可以使用PyTorch的tensor来完成，可以认为它们是科学计算的通用工具。

与Numpy不同，PyTorch可以利用GPU加速其数值计算。要在GPU上运行Tensor,在构造张量使用`device`参数把tensor建立在GPU上。

在这里，本章使用tensors将随机数据上训练一个两层的网络。和前面NumPy的例子类似，我们使用PyTorch的tensor，手动在网络中实现前向传播和反向传播：

```
# -*- coding: utf-8 -*-

import torch


dtype = torch.float
device = torch.device("cpu")
# device = torch.device ("cuda：0") #取消注释以在GPU上运行
```

```
# N是批量大小；D_in是输入维度；
# H是隐藏的维度；D_out是输出维度。
N, D_in, H, D_out = 64, 1000, 100, 10

#创建随机输入和输出数据
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# 随机初始化权重
w1 = torch.randn(D_in, H, device=device, dtype=dtype)
w2 = torch.randn(H, D_out, device=device, dtype=dtype)

learning_rate = 1e-6
for t in range(500):
    # 前向传递：计算预测y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # 计算和打印损失
    loss = (y_pred - y).pow(2).sum().item()
    print(t, loss)

    # Backprop计算w1和w2相对于损耗的梯度
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # 使用梯度下降更新权重
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

3.自动求导

3.1 PyTorch：张量和自动求导

在上面的例子中，需要手动实现神经网络的前向和后向 传递。手动实现反向传递对于小型双层网络来说并不是什么大问 题，但对于大型复杂网络来说很快就会变得非常繁琐。

但是可以使用自动微分来自动计算神经网络中的后向传递。 PyTorch中的 `autograd` 包提供了这个功能。当使用autograd时，网络前向传播将定义一个计算图；图中的节点是tensor，边是函数，

这些函数是输出tensor到输入tensor的映射。这张计算图使得在网络中反向传播时梯度的计算十分简单。

这听起来很复杂，在实践中使用起来非常简单。 如果我们想计算某些的tensor的梯度，我们只需要在建立这个tensor时加入这么一句： `requires_grad=True` 。这个tensor上的任何PyTorch的操作都将构造一个计算图，从而允许我们稍后在图中执行反向传播。如果这个 `tensor x` 的 `requires_grad=True` ，那么反向传播之后 `x.grad` 将会是另一个张量，其为x关于某个标量值的梯度。

有时可能希望防止PyTorch在 `requires_grad=True` 的张量执行某些操作时构建计算图；例如，在训练神经网络时，我们通常不希望通过权重更新步骤进行反向传播。在这种情况下，我们可以使用 `torch.no_grad()` 上下文管器来防止构造计算图。

下面我们使用PyTorch的Tensors和autograd来实现我们的两层的神经网络；我们不再需要手动执行网络的反向传播：

```
# -*- coding: utf-8 -*-
import torch

dtype = torch.float
device = torch.device("cpu")
# device = torch.device (“cuda：0”) #取消注释以在GPU上运行

# N是批量大小； D_in是输入维度；
# H是隐藏的维度； D_out是输出维度。
N, D_in, H, D_out = 64, 1000, 100, 10

# 创建随机Tensors以保持输入和输出。
# 设置requires_grad = False表示我们不需要计算渐变
# 在向后传球期间对于这些Tensors。
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# 为权重创建随机Tensors。
# 设置requires_grad = True表示我们想要计算渐变
# 在向后传球期间尊重这些张贴。
w1 = torch.randn(D_in, H, device=device, dtype=dtype, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, dtype=dtype, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # 前向传播：使用tensors上的操作计算预测值y；
      # 由于w1和w2有requires_grad=True，涉及这些张量的操作将让PyTorch构建计算图，
    # 从而允许自动计算梯度。由于我们不再手工实现反向传播，所以不需要保留中间值的引用。
```

```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # 使用Tensors上的操作计算和打印丢失。
    # loss是一个形状为()的张量
    # loss.item() 得到这个张量对应的python数值
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.item())

    # 使用autograd计算反向传播。这个调用将计算loss对所有requires_grad=True的tensor的梯
度。
    # 这次调用后，w1.grad和w2.grad将分别是loss对w1和w2的梯度张量。
    loss.backward()

    # 使用梯度下降更新权重。对于这一步，我们只想对w1和w2的值进行原地改变；不想为更新阶段构建
计算图，
    # 所以我们使用torch.no_grad()上下文管理器防止PyTorch为更新构建计算图
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad

        # 反向传播后手动将梯度设置为零
        w1.grad.zero_()
        w2.grad.zero_()
```

3.2 PyTorch：定义新的自动求导函数

在底层，每一个原始的自动求导运算实际上是两个在Tensor上运行的函数。其中，`forward`函数计算从输入Tensors获得的输出Tensors。而`backward`函数接收输出Tensors对于某个标量值的梯度，并且计算输入Tensors相对于该相同标量值的梯度。

在PyTorch中，我们可以很容易地通过定义`torch.autograd.Function`的子类并实现`forward`和backward函数，来定义自己的自动求导运算。之后我们就可以使用这个新的自动梯度运算符了。然后，我们可以通过构造一个实例并像调用函数一样，传入包含输入数据的tensor调用它，这样来使用新的自动求导运算。

这个例子中，我们自定义一个自动求导函数来展示ReLU的非线性。并用它实现我们的两层网络：

```
import torch

class MyReLU(torch.autograd.Function):
    """
    我们可以通过建立torch.autograd的子类来实现我们自定义的autograd函数，
    并完成张量的正向和反向传播。
    """
```

```
    @staticmethod
    def forward(ctx, x):
        """
        在正向传播中，我们接收到一个上下文对象和一个包含输入的张量；
        我们必须返回一个包含输出的张量，
        并且我们可以使用上下文对象来缓存对象，以便在反向传播中使用。
        """
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        """
        在反向传播中，我们接收到上下文对象和一个张量，
        其包含了相对于正向传播过程中产生的输出的损失的梯度。
        我们可以从上下文对象中检索缓存的数据，
        并且必须计算并返回与正向传播的输入相关的损失的梯度。
        """
        x, = ctx.saved_tensors
        grad_x = grad_output.clone()
        grad_x[x < 0] = 0
        return grad_x


device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# N是批大小； D_in 是输入维度；
# H 是隐藏层维度； D_out 是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 产生输入和输出的随机张量
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)

# 产生随机权重的张量
w1 = torch.randn(D_in, H, device=device, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # 正向传播：使用张量上的操作来计算输出值y；
    # 我们通过调用 MyReLU.apply 函数来使用自定义的ReLU
    y_pred = MyReLU.apply(x.mm(w1)).mm(w2)

    # 计算并输出loss
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.item())
```

```
    # 使用autograd计算反向传播过程。
    loss.backward()

    with torch.no_grad():
        # 用梯度下降更新权重
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad

        # 在反向传播之后手动清零梯度
        w1.grad.zero_()
        w2.grad.zero_()
```

3.3 TensorFlow：静态图

PyTorch自动求导看起来非常像TensorFlow：这两个框架中，我们都定义计算图，使用自动微分来计算梯度。两者最大的不同就是TensorFlow的计算图是静态的，而PyTorch使用动态的计算图。

在TensorFlow中，我们定义计算图一次，然后重复执行这个相同的图，可能会提供不同的输入数据。而在PyTorch中，每一个前向通道定义一个新的计算图。

静态图的好处在于你可以预先对图进行优化。例如，一个框架可能要融合一些图的运算来提升效率，或者产生一个策略来将图分布到多个GPU或机器上。如果重复使用相同的图，那么在重复运行同一个图时，，前期潜在的代价高昂的预先优化的消耗就会被分摊开。

静态图和动态图的一个区别是控制流。对于一些模型，我们希望对每个数据点执行不同的计算。例如，一个递归神经网络可能对于每个数据点执行不同的时间步数，这个展开（unrolling）可以作为一个循环来实现。对于一个静态图，循环结构要作为图的一部分。因此，TensorFlow提供了运算符（例如`tf.scan`）来把循环嵌入到图当中。对于动态图来说，情况更加简单：既然我们为每个例子即时创建图，我们可以使用普通的命令式控制流来为每个输入执行不同的计算。

为了与上面的PyTorch自动梯度实例做对比，我们使用TensorFlow来拟合一个简单的2层网络：

```
import tensorflow as tf
import numpy as np

# 首先我们建立计算图 (computational graph)

# N是批大小；D是输入维度；
# H是隐藏层维度；D_out是输出维度。
N, D_in, H, D_out = 64, 1000, 100, 10

# 为输入和目标数据创建placeholder；
# 当执行计算图时，他们将会被真实的数据填充
```

```
x = tf.placeholder(tf.float32, shape=(None, D_in))
y = tf.placeholder(tf.float32, shape=(None, D_out))

# 为权重创建Variable并用随机数据初始化
# TensorFlow的Variable在执行计算图时不会改变
w1 = tf.Variable(tf.random_normal((D_in, H)))
w2 = tf.Variable(tf.random_normal((H, D_out)))

# 前向传播：使用TensorFlow的张量运算计算预测值y。
# 注意这段代码实际上不执行任何数值运算；
# 它只是建立了我们稍后将执行的计算图。
h = tf.matmul(x, w1)
h_relu = tf.maximum(h, tf.zeros(1))
y_pred = tf.matmul(h_relu, w2)

# 使用TensorFlow的张量运算损失（loss）
loss = tf.reduce_sum((y - y_pred) ** 2.0)

# 计算loss对于w1和w2的导数
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

# 使用梯度下降更新权重。为了实际更新权重，我们需要在执行计算图时计算new_w1和new_w2。
# 注意，在TensorFlow中，更新权重值的行为是计算图的一部分；
# 但在PyTorch中，这发生在计算图形之外。
learning_rate = 1e-6
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

# 现在我们搭建好了计算图，所以我们开始一个TensorFlow的会话（session）来实际执行计算图。
with tf.Session() as sess:

    # 运行一次计算图来初始化Variable w1和w2
    sess.run(tf.global_variables_initializer())

    # 创建numpy数组来存储输入x和目标y的实际数据
    x_value = np.random.randn(N, D_in)
    y_value = np.random.randn(N, D_out)

    for _ in range(500):
        # 多次运行计算图。每次执行时，我们都用feed_dict参数，
        # 将x_value绑定到x，将y_value绑定到y，
        # 每次执行图形时我们都要计算损失、new_w1和new_w2；
        # 这些张量的值以numpy数组的形式返回。
        loss_value, _, _ = sess.run([loss, new_w1, new_w2],
                                    feed_dict={x: x_value, y: y_value})
        print(loss_value)
```

4.nn模块

4.1 PyTorch： nn

计算图和autograd是十分强大的工具，可以定义复杂的操作并自动求导；然而对于大规模的网络，autograd太过于底层。 在构建神经网络时，我们经常考虑将计算安排成层，其中一些具有可学习的参数，它们将在学习过程中进行优化。

TensorFlow里，有类似Keras，TensorFlow-Slim和TFLearn这种封装了底层计算图的高度抽象的接口，这使得构建网络十分方便。

在PyTorch中，包 `nn` 完成了同样的功能。nn包中定义一组大致等价于层的模块。一个模块接受输入的tesnor，计算输出的tensor，而且 还保存了一些内部状态比如需要学习的tensor的参数等。nn包中也定义了一组损失函数（loss functions），用来训练神经网络。

这个例子中，我们用nn包实现两层的网络：

```
# -*- coding: utf-8 -*-
import torch

# N是批大小；D是输入维度
# H是隐藏层维度；D_out是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

#创建输入和输出随机张量
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# 使用nn包将我们的模型定义为一系列的层。
# nn.Sequential是包含其他模块的模块，并按顺序应用这些模块来产生其输出。
# 每个线性模块使用线性函数从输入计算输出，并保存其内部的权重和偏差张量。
# 在构造模型之后，我们使用.to()方法将其移动到所需的设备。
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

# nn包还包含常用的损失函数的定义；
# 在这种情况下，我们将使用平均平方误差(MSE)作为我们的损失函数。
# 设置reduction='sum'，表示我们计算的是平方误差的“和”，而不是平均值；
# 这是为了与前面我们手工计算损失的例子保持一致，
# 但是在实践中，通过设置reduction='elementwise_mean'来使用均方误差作为损失更为常见。
loss_fn = torch.nn.MSELoss(reduction='sum')
```

```
learning_rate = 1e-4
for t in range(500):
    # 前向传播：通过向模型传入x计算预测的y。
    # 模块对象重载了__call__运算符，所以可以像函数那样调用它们。
    # 这么做相当于向模块传入了一个张量，然后它返回了一个输出张量。
    y_pred = model(x)

     # 计算并打印损失。
     # 传递包含y的预测值和真实值的张量，损失函数返回包含损失的张量。
    loss = loss_fn(y_pred, y)
    print(t, loss.item())

    # 反向传播之前清零梯度
    model.zero_grad()

    # 反向传播：计算模型的损失对所有可学习参数的导数（梯度）。
    # 在内部，每个模块的参数存储在requires_grad=True的张量中，
    # 因此这个调用将计算模型中所有可学习参数的梯度。
    loss.backward()

    # 使用梯度下降更新权重。
    # 每个参数都是张量，所以我们可以像我们以前那样可以得到它的数值和梯度
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

4.2 PyTorch： `optim`

到目前为止，我们已经通过手动改变包含可学习参数的张量来更新模型的权重。对于随机梯度下降(SGD/stochastic gradient descent)等简单的优化算法来说，这不是一个很大的负担，但在实践中，我们经常使用AdaGrad、RMSProp、Adam等更复杂的优化器来训练神经网络。

```
import torch

# N是批大小；D是输入维度
# H是隐藏层维度；D_out是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 产生随机输入和输出张量
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# 使用nn包定义模型和损失函数
model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
```

```
            torch.nn.ReLU(),
            torch.nn.Linear(H, D_out),
        )
loss_fn = torch.nn.MSELoss(reduction='sum')

# 使用optim包定义优化器（Optimizer）。Optimizer将会为我们更新模型的权重。
# 这里我们使用Adam优化方法；optim包还包含了许多别的优化算法。
# Adam构造函数的第一个参数告诉优化器应该更新哪些张量。
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

for t in range(500):

    # 前向传播：通过像模型输入x计算预测的y
    y_pred = model(x)

    # 计算并打印loss
    loss = loss_fn(y_pred, y)
    print(t, loss.item())

    # 在反向传播之前，使用optimizer将它要更新的所有张量的梯度清零(这些张量是模型可学习的权
重)
    optimizer.zero_grad()

    # 反向传播：根据模型的参数计算loss的梯度
    loss.backward()

    # 调用Optimizer的step函数使它所有参数更新
    optimizer.step()
```

4.3 PyTorch：自定义 nn 模块

有时候需要指定比现有模块序列更复杂的模型；对于这些情况，可以通过继承 `nn.Module` 并定义 `forward` 函数，这个 `forward` 函数可以 使用其他模块或者其他的自动求导运算来接收输入tensor，产生输出tensor。

在这个例子中，我们用自定义Module的子类构建两层网络：

```
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        在构造函数中，我们实例化了两个nn.Linear模块，并将它们作为成员变量。
        """
        super(TwoLayerNet, self).__init__()
```

```
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        在前向传播的函数中，我们接收一个输入的张量，也必须返回一个输出张量。
        我们可以使用构造函数中定义的模块以及张量上的任意的（可微分的）操作。
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

# N是批大小； D_in 是输入维度；
# H 是隐藏层维度； D_out 是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 产生输入和输出的随机张量
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# 通过实例化上面定义的类来构建我们的模型。
model = TwoLayerNet(D_in, H, D_out)

# 构造损失函数和优化器。
# SGD构造函数中对model.parameters()的调用，
# 将包含模型的一部分，即两个nn.Linear模块的可学习参数。
loss_fn = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    # 前向传播：通过向模型传递x计算预测值y
    y_pred = model(x)

    #计算并输出loss
    loss = loss_fn(y_pred, y)
    print(t, loss.item())

    # 清零梯度，反向传播，更新权重
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

4.4 PyTorch：控制流和权重共享

作为动态图和权重共享的一个例子，我们实现了一个非常奇怪的模型：一个全连接的ReLU网络，在每一次前向传播时，它的隐藏层的层数为随机1到4之间的数，这样可以多次重用相同的权重来计算。

因为这个模型可以使用普通的Python流控制来实现循环，并且我们可以通过在定义转发时多次重用同一个模块来实现最内层之间的权重共享。

我们利用Mudule的子类很容易实现这个模型：

```
import random
import torch

class DynamicNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        在构造函数中，我们构造了三个nn.Linear实例，它们将在前向传播时被使用。
        """
        super(DynamicNet, self).__init__()
        self.input_linear = torch.nn.Linear(D_in, H)
        self.middle_linear = torch.nn.Linear(H, H)
        self.output_linear = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        对于模型的前向传播，我们随机选择0、1、2、3，
        并重用了多次计算隐藏层的middle_linear模块。
        由于每个前向传播构建一个动态计算图，
        我们可以在定义模型的前向传播时使用常规Python控制流运算符，如循环或条件语句。
        在这里，我们还看到，在定义计算图形时多次重用同一个模块是完全安全的。
        这是Lua Torch的一大改进，因为Lua Torch中每个模块只能使用一次。
        """
        h_relu = self.input_linear(x).clamp(min=0)
        for _ in range(random.randint(0, 3)):
            h_relu = self.middle_linear(h_relu).clamp(min=0)
        y_pred = self.output_linear(h_relu)
        return y_pred


# N是批大小；D是输入维度
# H是隐藏层维度；D_out是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 产生输入和输出随机张量
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# 实例化上面定义的类来构造我们的模型
model = DynamicNet(D_in, H, D_out)

# 构造我们的损失函数（loss function）和优化器（Optimizer）。
# 用平凡的随机梯度下降训练这个奇怪的模型是困难的，所以我们使用了momentum方法。
```

```
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
for t in range(500):

    # 前向传播：通过向模型传入x计算预测的y。
    y_pred = model(x)

    # 计算并打印损失
    loss = criterion(y_pred, y)
    print(t, loss.item())

    # 清零梯度，反向传播，更新权重
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

PyTorch之迁移学习

实际中，基本没有人会从零开始（随机初始化）训练一个完整的卷积网络，因为相对于网络，很难得到一个足够大的数据集[网络很深, 需要足够大数据集]。通常的做法是在一个很大的数据集上进行预训练得到卷积网络ConvNet, 然后将这个ConvNet的参数作为目标任务的初始化参数或者固定这些参数。

转移学习的两个主要场景：

- 微调**Convnet**：使用预训练的网络(如在 `imagenet 1000` 上训练而来的网络)来初始化自己的网络，而不是随机初始化。其他的训练步骤不变。
- 将**Convnet**看成固定的特征提取器:首先固定ConvNet除了最后的全连接层外的其他所有层。最后的全连接层被替换成一个新的随机 初始化的层，只有这个新的层会被训练[只有这层参数会在反向传播时更新]

下面是利用PyTorch进行迁移学习步骤，要解决的问题是训练一个模型来对蚂蚁和蜜蜂进行分类。

1.导入相关的包

```
# License: BSD
# Author: Sasank Chilamkurthy

from __future__ import print_function, division

import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy

plt.ion()   # interactive mode
```

2.加载数据

今天要解决的问题是训练一个模型来分类蚂蚁ants和蜜蜂bees。ants和bees各有约120张训练图片。每个类有75张验证图片。从零开始在 如此小的数据集上进行训练通常是很难泛化的。由于我们使用迁移学习，模型的泛化能力会相当好。 该数据集是imagenet的一个非常小的子集。从此处下载数据，并将其解压缩到当前目录。

```
#训练集数据扩充和归一化
#在验证集上仅需要归一化
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224), #随机裁剪一个area然后再resize
        transforms.RandomHorizontalFlip(), #随机水平翻转
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

data_dir = 'data/hymenoptera_data'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=4,
                                              shuffle=True, num_workers=4)
               for x in ['train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
class_names = image_datasets['train'].classes

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

3.可视化部分图像数据

可视化部分训练图像，以便了解数据扩充。

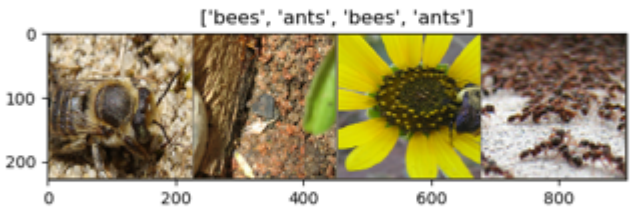
```
def imshow(inp, title=None):
    """Imshow for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
```

```
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)  # pause a bit so that plots are updated


# 获取一批训练数据
inputs, classes = next(iter(dataloaders['train']))

# 批量制作网格
out = torchvision.utils.make_grid(inputs)

imshow(out, title=[class_names[x] for x in classes])
```



4.训练模型

编写一个通用函数来训练模型。下面将说明：* 调整学习速率 * 保存最好的模型

下面的参数scheduler是一个来自 `torch.optim.lr_scheduler` 的学习速率调整类的对象(LR scheduler object)。

```
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
```

```
for epoch in range(num_epochs):
    print('Epoch {}/{}'.format(epoch, num_epochs - 1))
    print('-' * 10)

    # 每个epoch都有一个训练和验证阶段
    for phase in ['train', 'val']:
        if phase == 'train':
            scheduler.step()
            model.train()  # Set model to training mode
        else:
            model.eval()   # Set model to evaluate mode

        running_loss = 0.0
        running_corrects = 0

        # 迭代数据.
        for inputs, labels in dataloaders[phase]:
            inputs = inputs.to(device)
            labels = labels.to(device)

            # 零参数梯度
            optimizer.zero_grad()

            # 前向
            # track history if only in train
            with torch.set_grad_enabled(phase == 'train'):
                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)
                loss = criterion(outputs, labels)

                # 后向+仅在训练阶段进行优化
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

            # 统计
            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_loss = running_loss / dataset_sizes[phase]
        epoch_acc = running_corrects.double() / dataset_sizes[phase]

        print('{} Loss: {:.4f} Acc: {:.4f}'.format(
            phase, epoch_loss, epoch_acc))

        # 深度复制mo
        if phase == 'val' and epoch_acc > best_acc:
```

```
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

        print()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    # 加载最佳模型权重
    model.load_state_dict(best_model_wts)
    return model
```

5.可视化模型的预测结果

```
#一个通用的展示少量预测图片的函数
def visualize_model(model, num_images=6):
    was_training = model.training
    model.eval()
    images_so_far = 0
    fig = plt.figure()

    with torch.no_grad():
        for i, (inputs, labels) in enumerate(dataloaders['val']):
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)

            for j in range(inputs.size()[0]):
                images_so_far += 1
                ax = plt.subplot(num_images//2, 2, images_so_far)
                ax.axis('off')
                ax.set_title('predicted: {}'.format(class_names[preds[j]]))
                imshow(inputs.cpu().data[j])

                if images_so_far == num_images:
                    model.train(mode=was_training)
                    return
        model.train(mode=was_training)
```

6.场景1：微调ConvNet

加载预训练模型并重置最终完全连接的图层。

```
model_ft = models.resnet18(pretrained=True)
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, 2)

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# 观察所有参数都正在优化
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)

# 每7个epochs衰减LR通过设置gamma=0.1
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)
```

训练和评估模型

（1）训练模型 该过程在CPU上需要大约15-25分钟，但是在GPU上，它只需不到一分钟。

```
model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                       num_epochs=25)
```

* 输出

```
Epoch 0/24
----------
train Loss: 0.7032 Acc: 0.6025
val Loss: 0.1698 Acc: 0.9412

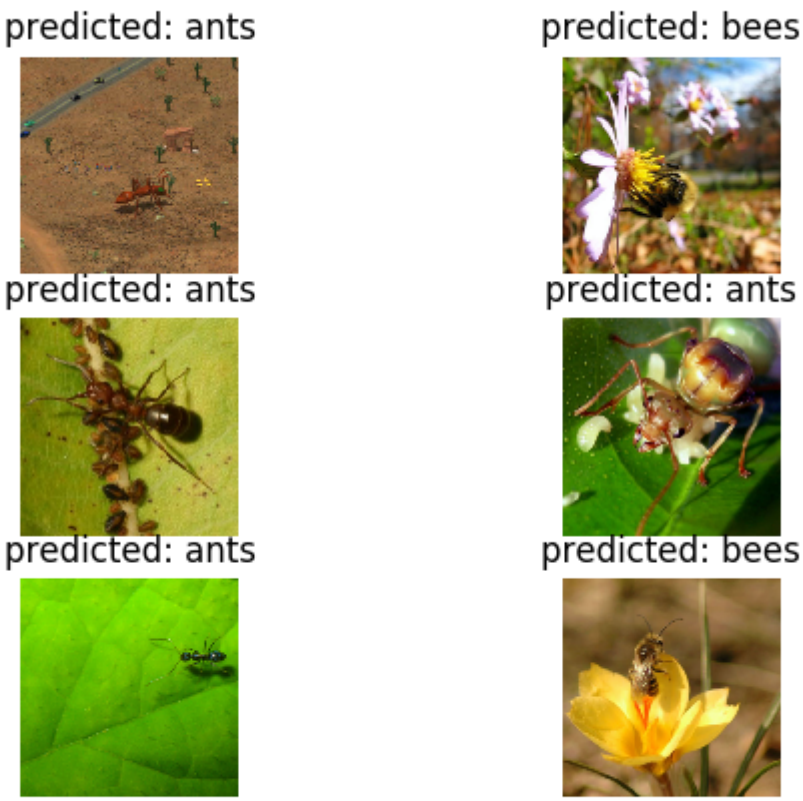
Epoch 1/24
----------
train Loss: 0.6411 Acc: 0.7787
val Loss: 0.1981 Acc: 0.9281
.
.
.
Epoch 24/24
----------
train Loss: 0.2812 Acc: 0.8730
val Loss: 0.2647 Acc: 0.9150
```

```
Training complete in 1m 7s
Best val Acc: 0.941176
```

（2）模型评估效果可视化

```
visualize_model(model_ft)
```

* 输出



7.场景2：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。通过设置 `requires_grad == Falsebackward()` 来冻结参数，这样在反向传播backward()的时候他们的梯度就不会被计算。

```
model_conv = torchvision.models.resnet18(pretrained=True)
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
```

```
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)
```

训练和评估

（1）训练模型 在CPU上，与前一个场景相比，这将花费大约一半的时间，因为不需要为大多数网络计算梯度。但需要计算转发。

```
model_conv = train_model(model_conv, criterion, optimizer_conv,
                         exp_lr_scheduler, num_epochs=25)
```

* 输出

```
Epoch 0/24
----------
train Loss: 0.6400 Acc: 0.6434
val Loss: 0.2539 Acc: 0.9085
.
.
.
Epoch 23/24
----------
train Loss: 0.2988 Acc: 0.8607
val Loss: 0.2151 Acc: 0.9412

Epoch 24/24
----------
train Loss: 0.3519 Acc: 0.8484
val Loss: 0.2045 Acc: 0.9412

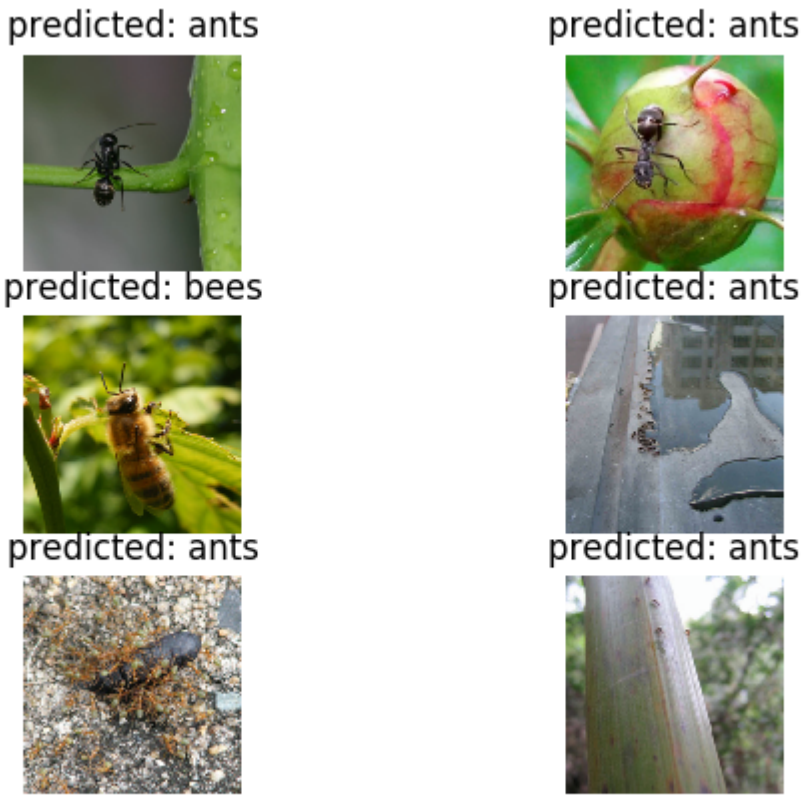
Training complete in 0m 35s
Best val Acc: 0.954248
```

（2）模型评估效果可视化

```
visualize_model(model_conv)

plt.ioff()
plt.show()
```

* 输出



8.文件下载

- py文件
- jupyter文件

混合前端的seq2seq模型部署

本教程将介绍如何是seq2seq模型转换为PyTorch可用的前端混合Torch脚本。我们要转换的模型来自于聊天机器人教程Chatbot tutorial。

1.混合前端

在一个基于深度学习项目的研发阶段, 使用像PyTorch这样**即时** `eager` 、命令式的界面进行交互能带来很大便利。 这使用户能够在使用Python数据结构、控制流操作、打印语句和调试实用程序时通过熟悉的、惯用的Python脚本编写。

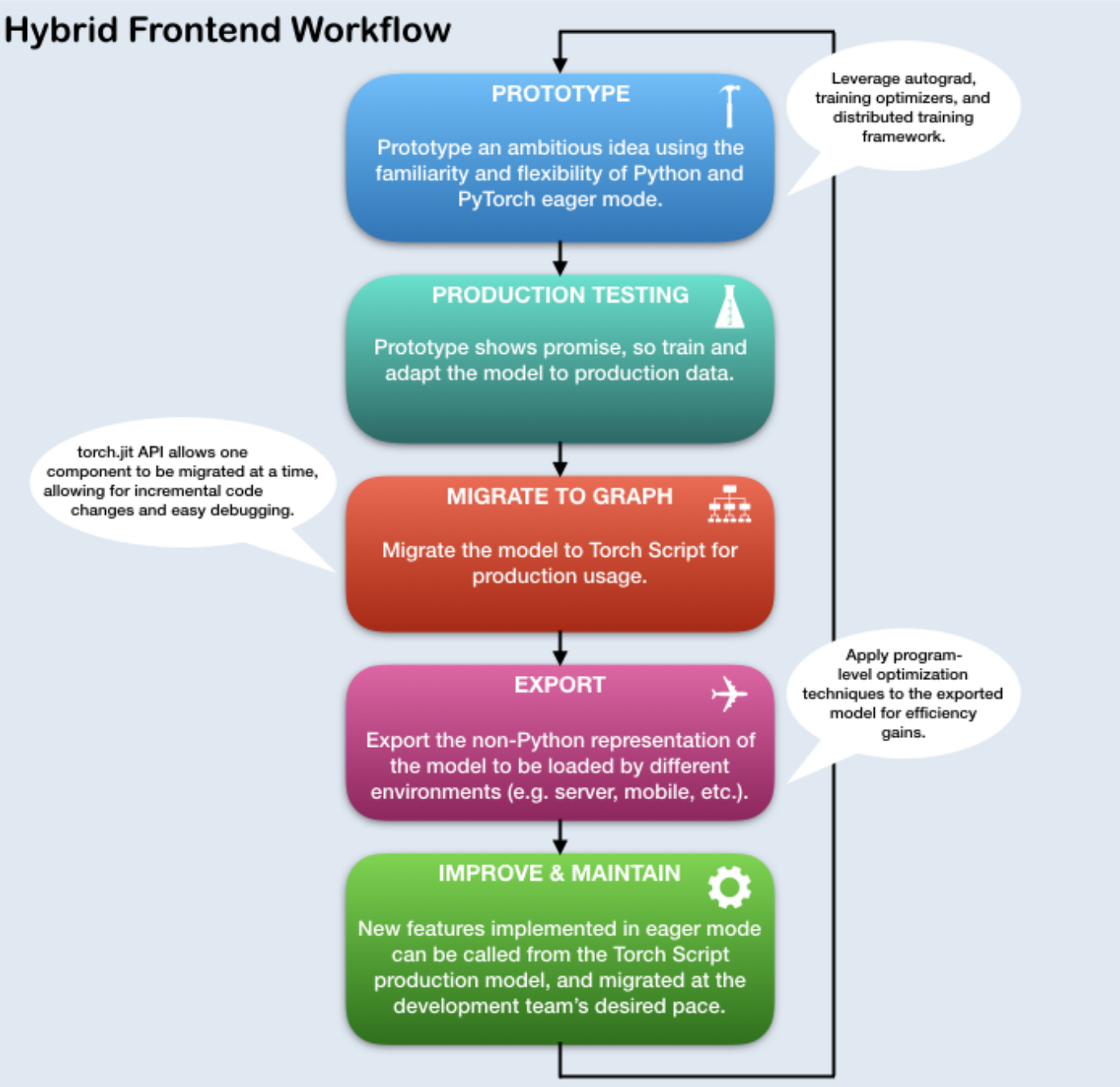
尽管即时性界面对于研究和试验应用程序是一个有用的工具，但是对于生产环境中部署模型时，使用基于图形 `graph-based` 的模型表示将更加适用的。 一个延迟的图型展示意味着可以优化，比如无序执行操作，以及针对高度优化的硬件架构的能力。 此外，基于图形的表示支持框架无关的模型导出。PyTorch提供了将即时模式的代码增量转换为Torch脚本的机制，Torch脚本是一个在Python中的静态可分析和可优化的子集，Torch使用它来在Python运行时独立进行深度学习。

在Torch中的 `torch.jit` 模块可以找到将即时模式的PyTorch程序转换为Torch脚本的API。 这个模块有两个核心模式用于将即时模式模型转换为Torch脚本图形表示: **跟踪** `tracing` 以及 **脚本化** `scripting` 。 `torch.jit.trace` 函数接受一个模块或者一个函数和一组示例的输入，然后通过函数或模块运行输入示例，同时跟跟踪遇到的计算步骤，然后输出一个可以展示跟踪流程的基于图形的函数。**跟踪** `Tracing` 对于不涉及依赖于数据的控制流的直接的模块和函数非常有用，就比如标准的卷积神经网络。

然而，如果一个有数据依赖的if语句和循环的函数被跟踪，则只记录示例输入沿执行路径调用的操作。换句话说，控制流本身并没有被捕获。要将带有数据依赖控制流的模块和函数进行转化，已提供了一个脚本化机制。脚本显式地将模块或函数代码转换为Torch脚本，包括所有可能的控制流路径。 如需使用脚本模式 `script mode` ，要确定继承了 `torch.jit.ScriptModule` 基本类 (取代 `torch.nn.Module`) 并且增加 `torch.jit.script` 装饰器到你的Python函数或者 `torch.jit.script_method` 装饰器到你的模块方法。

使用脚本化的一个警告是，它只支持Python的一个受限子集。要获取与支持的特性相关的所有详细信息，请参考 Torch Script language reference。为了达到最大的灵活性，可以组合Torch脚本的

模式来表示整个程序，并且可以增量地应用这些技术。



2.预备环境

首先，导入所需的模块以及设置一些常量。如果想使用自己的模型，需要保证 `MAX_LENGTH` 常量设置正确。 提醒：这个常量定义了在训练过程中允许的最大句子长度以及模型能够产生的最大句子长度输出。

```
source-python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import torch
import torch.nn as nn
import torch.nn.functional as F
import re
import os
import unicodedata
import numpy as np

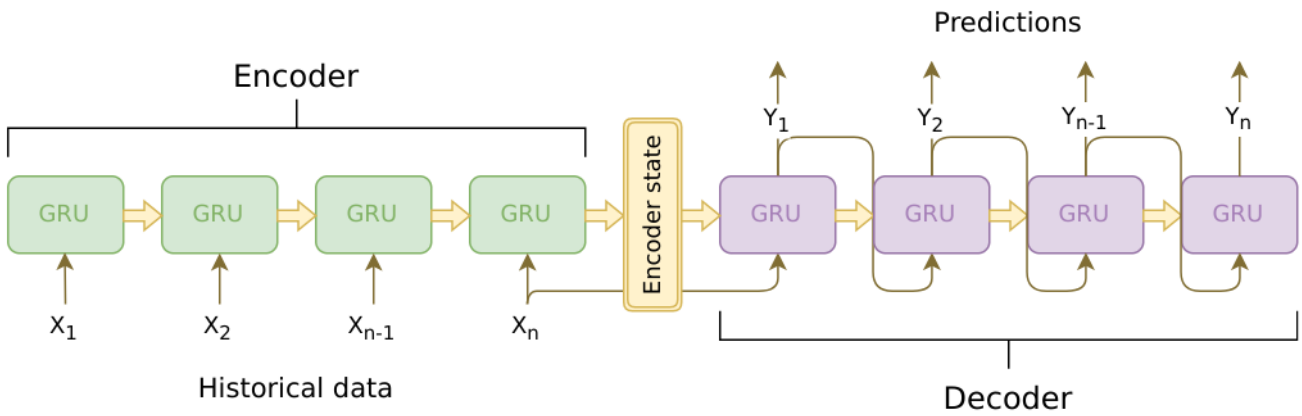
device = torch.device("cpu")

MAX_LENGTH = 10 # Maximum sentence length

# 默认的词向量
PAD_token = 0 # Used for padding short sentences
SOS_token = 1 # Start-of-sentence token
EOS_token = 2 # End-of-sentence token
```

3.模型概述

正如前文所言，我们使用的sequence-to-sequence (seq2seq) 模型。这种类型的模型用于输入是可变长度序列的情况，我们的输出也是一个可变长度序列它不一定是一对一输入映射。`seq2seq` 模型由两个递归神经网络(RNNs)组成：编码器 **encoder**和解码器**decoder**.



（1）编码器(Encoder)

编码器RNN在输入语句中每次迭代一个标记(例如单词)，每次步骤输出一个“输出”向量和一个“隐藏状态”向量。”隐藏状态“向量在之后则传递到下一个步骤，同时记录输出向量。编码器将序列中每个坐标代表的文本转换为高维空间中的一组坐标，解码器将使用这些坐标为给定的任务生成有意义的输出。

（2）解码器(Decoder)

解码器RNN以逐个令牌的方式生成响应语句。它使用来自于编码器的文本向量和内部隐藏状态来生成序列中的下一个单词。它继续生成单词，直到输出表示句子结束的EOS语句。我们在解码器中使用专注机制attention mechanism来帮助它在输入的某些部分生成输出时”保持专注”。对于我们的模型，我们实现了 Luong et al等人的“全局关注 `Global attention` ”模块，并将其作为解码模型中的子模块。

4.数据处理

尽管我们的模型在概念上处理标记序列，但在现实中，它们与所有机器学习模型一样处理数字。在这种情况下，在训练之前建立的模型词汇表中的每个单词都映射到一个整数索引。我们使用 `Voc` 对象来包含从单词到索引的映射，以及词汇表中的单词总数。我们将在运行模型之前加载对象。

此外，为了能够进行评估，我们必须提供一个处理字符串输入的工具。 `normalizeString` 函数将字符串中的所有字符转换为小写，并删除所有非字母字符。 `indexesFromSentence` 函数接受一个单词的句子并返回相应的单词索引序列。

```
class Voc:
    def __init__(self, name):
        self.name = name
        self.trimmed = False
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
        self.num_words = 3 # 统计SOS, EOS, PAD

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
```

```
        if word not in self.word2index:
            self.word2index[word] = self.num_words
            self.word2count[word] = 1
            self.index2word[self.num_words] = word
            self.num_words += 1
        else:
            self.word2count[word] += 1

    # Remove words below a certain count threshold
    def trim(self, min_count):
        if self.trimmed:
            return
        self.trimmed = True
        keep_words = []
        for k, v in self.word2count.items():
            if v >= min_count:
                keep_words.append(k)

        print('keep_words {} / {} = {:.4f}'.format(
            len(keep_words), len(self.word2index), len(keep_words) /
len(self.word2index)
        ))
        # Reinitialize dictionaries
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
        self.num_words = 3 # 统计默认的令牌
        for word in keep_words:
            self.addWord(word)

# 小写并删除非字母字符
def normalizeString(s):
    s = s.lower()
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    return s

# 使用字符串句子，返回单词索引的句子
def indexesFromSentence(voc, sentence):
    return [voc.word2index[word] for word in sentence.split(' ')] + [EOS_token]
```

5.定义编码器

通过 `torch.nn.GRU` 模块实现编码器的RNN。本模块接受一批语句(嵌入单词的向量)的输入，它在内部遍历这些句子，每次一个标记， 计算隐藏状态。我们将这个模块初始化为双向的，这意味着我们有两个独立的GRUs:一个按时间顺序遍历序列，另一个按相反顺序遍历序列。 我们最终返回

这两个GRUs输出的和。由于我们的模型是使用批处理进行训练的，所以我们的EncoderRNN模型的 `forward` 函数需要一个填充 的输入批处理。为了批量处理可变长度的句子，我们通过 `MAX_LENGTH` 令牌允许一个句子中支持的最大长度，并且批处理中所有小于 `MAX_LENGTH` 令牌的句子都使用我们专用的 `PAD_token` 令牌填充在最后。要使用带有PyTorch RNN模块的批量填充，我们必须把转发 `forward` 密令在 调用 `torch.nn.utils.rnn.pack_padded_sequence` 和 `torch.nn.utils.rnn.pad_packed_sequence` 数据转换时进行打包。注意， `forward` 函数还接受一个 `input_length` 列表，其中包含批处理中每个句子的长度。该输入在填充时通过 `torch.nn.utils.rnn.pack_padded_sequence` 使用。

- 混合前端笔记 由于编码器的转发函数 `forward` 不包含任何依赖于数据的控制流，因此我们将使用跟踪 `tracing` 将其转换为脚本模式 `script mode` 。在跟踪模块时， 我们可以保持模块定义不变。在运行评估之前，我们将在本文末尾初始化所有模型。

```
class EncoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding, n_layers=1, dropout=0):
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size
        self.embedding = embedding

        # 初始化GRU;input_size和hidden_size参数都设置为'hidden_size'
        # 因为我们输入的大小是一个有多个特征的词向量== hidden_size
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers,
                          dropout=(0 if n_layers == 1 else dropout),
bidirectional=True)

    def forward(self, input_seq, input_lengths, hidden=None):
        # 将单词索引转换为向量
        embedded = self.embedding(input_seq)
        # 为RNN模块填充批次序列
        packed = torch.nn.utils.rnn.pack_padded_sequence(embedded,
input_lengths)
        # 正向通过GRU
        outputs, hidden = self.gru(packed, hidden)
        # 打开填充
        outputs, _ = torch.nn.utils.rnn.pad_packed_sequence(outputs)
        # 将双向GRU的输出结果总和
        outputs = outputs[:, :, :self.hidden_size] +
outputs[:, : ,self.hidden_size:]
        # 返回输出以及最终的隐藏状态
        return outputs, hidden
```

6.定义解码器的注意力模块

接下来，将定义注意力模块(Attn)。请注意，此模块将用作解码器模型中的子模块。Luong等人考虑了各种“分数函数” `score functions` ，它们取当前解码器RNN输出和整个编码器输出，并返回关注点“能值” `engergies` 。这个关注能值张量 `attension energies tensor` 与 编码器输出的大小相同，两者最终相乘，得到一个加权张量，其最大值表示在特定时间步长解码的查询语句最重要的部分。

```
# Luong的注意力层
class Attn(torch.nn.Module):
    def __init__(self, method, hidden_size):
        super(Attn, self).__init__()
        self.method = method
        if self.method not in ['dot', 'general', 'concat']:
            raise ValueError(self.method, "is not an appropriate attention
method.")
        self.hidden_size = hidden_size
        if self.method == 'general':
            self.attn = torch.nn.Linear(self.hidden_size, hidden_size)
        elif self.method == 'concat':
            self.attn = torch.nn.Linear(self.hidden_size * 2, hidden_size)
            self.v = torch.nn.Parameter(torch.FloatTensor(hidden_size))

    def dot_score(self, hidden, encoder_output):
        return torch.sum(hidden * encoder_output, dim=2)

    def general_score(self, hidden, encoder_output):
        energy = self.attn(encoder_output)
        return torch.sum(hidden * energy, dim=2)

    def concat_score(self, hidden, encoder_output):
        energy = self.attn(torch.cat((hidden.expand(encoder_output.size(0), -1,
-1), encoder_output), 2)).tanh()
        return torch.sum(self.v * energy, dim=2)

    def forward(self, hidden, encoder_outputs):
        # 根据给定的方法计算注意力权重（能量）
        if self.method == 'general':
            attn_energies = self.general_score(hidden, encoder_outputs)
        elif self.method == 'concat':
            attn_energies = self.concat_score(hidden, encoder_outputs)
        elif self.method == 'dot':
            attn_energies = self.dot_score(hidden, encoder_outputs)

        # 转置max_length和batch_size维度
```

```
        attn_energies = attn_energies.t()

        # 返回softmax归一化概率分数（增加维度）
        return F.softmax(attn_energies, dim=1).unsqueeze(1)
```

7.定义解码器

类似于 `EncoderRNN` ，我们使用 `torch.nn.GRU` 模块作为我们的解码器RNN。然而，这一次我们使用单向GRU。需要注意的是，与编码器不同，我们将向解码器RNN每次提供一个单词。我们首先得到当前单词的嵌入并应用抛出功能 `dropout` 。接下来，我们将嵌入和最后的隐藏 状态转发给GRU，得到当前的GRU输出和隐藏状态。然后，我们使用Attn模块作为一个层来获得专注权重，我们将其乘以编码器的输出来获得我们的参与编码器输出。我们使用这个参与编码器输出作为文本 `context` 张量，它表示一个加权和，表示编码器输出的哪些部分需要注意。 在这里，我们使用线性层 `linear layer` 和 `softmax normalization` 归一化来选择输出序列中的下一个单词。

- 混合前端笔记 与 `EncoderRNN` 类似，此模块不包含任何依赖于数据的控制流。因此，在初始化该模型并加载其参数之后，我们可以再次使用跟踪 `tracing` 将其转换为Torch脚本。

```
class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, embedding, hidden_size, output_size,
n_layers=1, dropout=0.1):
        super(LuongAttnDecoderRNN, self).__init__()

        # 保持参考
        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout = dropout

        # 定义层
        self.embedding = embedding
        self.embedding_dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers, dropout=(0 if
n_layers == 1 else dropout))
        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

        self.attn = Attn(attn_model, hidden_size)

    def forward(self, input_step, last_hidden, encoder_outputs):
        # 注意：我们这步只运行一次
        # 获取当前输入字对应的向量映射
```

```
        embedded = self.embedding(input_step)
        embedded = self.embedding_dropout(embedded)
        # 通过单向GRU转发
        rnn_output, hidden = self.gru(embedded, last_hidden)
        # 通过当前GRU的输出计算注意力权重
        attn_weights = self.attn(rnn_output, encoder_outputs)
        # 注意力权重乘以编码器输出以获得新的“加权和”上下文向量
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1))
        # 使用Luong的公式5来连接加权上下文向量和GRU输出
        rnn_output = rnn_output.squeeze(0)
        context = context.squeeze(1)
        concat_input = torch.cat((rnn_output, context), 1)
        concat_output = torch.tanh(self.concat(concat_input))
        # 使用Luong的公式6来预测下一个单词
        output = self.out(concat_output)
        output = F.softmax(output, dim=1)
        # 返回输出和最终的隐藏状态
        return output, hidden
```

8.定义评估

8.1 贪婪搜索解码器

在聊天机器人教程中，我们使用 `GreedySearchDecoder` 模块来简化实际的解码过程。该模块将训练好的编码器和解码器模型作为属性， 驱动输入语句(词索引向量)的编码过程，并一次一个词(词索引)迭代地解码输出响应序列。

对输入序列进行编码很简单:只需将整个序列张量及其对应的长度向量转发给编码器。需要注意的是，这个模块一次只处理一个输入序列， 而不是成批的序列。因此，当常数1用于声明张量大小时，它对应于批处理大小为1。要解码给定的解码器输出，我们必须通过解码器模型 迭代地向前运行，该解码器模型输出 `softmax` 分数，该分数对应于每个单词在解码序列中是正确的下一个单词的概率。我们将 `decoder_input` 初始化为一个包含 `SOS_token` 的张量。在每次通过解码器之后，我们贪婪地将 `softmax` 概率最高的单词追加到 `decoded_words` 列表中。 我们还使用这个单词作为下一个迭代的 `decoder_input` 。如果 `decoded_words` 列表的长度达到 `MAX_LENGTH` ，或者预测的单词是 `EOS_token` ， 那么解码过程将终止。

- 混合前端笔记

该模块的 `forward` 方法涉及到在每次解码一个单词的输出序列时，遍历/([0,max/_length]/)的范围。因此，我们应该使用脚本将这个模 块转换为Torch脚本。与我们可以跟踪的编码器和解码器模型不同，我们必须对 `GreedySearchDecoder` 模块进行一些必要的更改，以便在 不出错的情况下初

始化对象。换句话说，我们必须确保我们的模块遵守脚本机制的规则，并且不使用Torch脚本包含的Python子集之外的任何语言特性。

为了了解可能需要的一些操作，我们将回顾聊天机器人教程中的GreedySearchDecoder实现与下面单元中使用的实现之间的区别。请注意， 用红色突出显示的行是从原始实现中删除的行，而用绿色突出显示的行是新的。

```
- class GreedySearchDecoder(nn.Module):
-     def __init__(self, encoder, decoder):
+ class GreedySearchDecoder(torch.jit.ScriptModule):
+     def __init__(self, encoder, decoder, decoder_n_layers):
          super(GreedySearchDecoder, self).__init__()
          self.encoder = encoder
          self.decoder = decoder
+         self._device = device
+         self._SOS_token = SOS_token
+         self._decoder_n_layers = decoder_n_layers

-     def forward(self, input_seq, input_length, max_length):
+     __constants__ = ['_device', '_SOS_token', '_decoder_n_layers']
+
+     @torch.jit.script_method
+     def forward(self, input_seq : torch.Tensor, input_length : torch.Tensor, max_length : int):
          # Forward input through encoder model
          encoder_outputs, encoder_hidden = self.encoder(input_seq, input_length)
          # Prepare encoder's final hidden layer to be first hidden input to the decoder
-         decoder_hidden = encoder_hidden[:decoder.n_layers]
+         decoder_hidden = encoder_hidden[:self._decoder_n_layers]
          # Initialize decoder input with SOS_token
-         decoder_input = torch.ones(1, 1, device=device, dtype=torch.long) * SOS_token
+         decoder_input = torch.ones(1, 1, device=self._device, dtype=torch.long) * self._SOS_token
          # Initialize tensors to append decoded words to
-         all_tokens = torch.zeros([0], device=device, dtype=torch.long)
-         all_scores = torch.zeros([0], device=device)
+         all_tokens = torch.zeros([0], device=self._device, dtype=torch.long)
+         all_scores = torch.zeros([0], device=self._device)
          # Iteratively decode one word token at a time
          for _ in range(max_length):
              # Forward pass through decoder
```

变更事项：

- `nn.Module` -> `torch.jit.ScriptModule` 为了在模块上使用PyTorch的脚本化机制, 模型需要从 `torch.jit.ScriptModule` 继承。
- 将 `decoder_n_layers` 追加到结构参数 这种变化源于这样一个事实，即我们传递给这个模块的编码器和解码器模型将是 `TracedModule` (非模块)的子模块。因此，我们无法使用 `decoder.n_layers` 访问解码器的层数。相反，我们对此进行计划，并在模块构建过程中传入此值。
- 将新属性作为常量保存 在最初的实现中，我们可以在 `GreedySearchDecoder` 的 `forward` 方法中自由地使用来自周围(全局)范围的变量. 然而，现在我们正 在使用脚本，我们没有这种自

由，因为脚本处理的设想4是我们不一定要保留Python对象，尤其是在导出时。 一个简单的解决方案是 将全局作用域中的这些值作为属性存储到构造函数中的模块中， 并将它们添加到一个名为 `__constants__` 的特殊列表中，以便在 `forward` 方法中构造图形时将它们用作文本值。这种用法的一个例子在第19行，取代使用 `device` 和 `SOS_token` 全局值，我们使用常量属性 `self._device` 和 `self._SOS_token` 。

- 将 `torch.jit.script_method` 装饰器添加到 `forward` 方法 添加这个装饰器可以让JIT编译器知道它所装饰的函数应该是脚本化的。
- 强制 `forward` 方法的参数类型 默认情况下，Torch脚本函数的所有参数都假定为张量。如果需要传递不同类型的参数，可以使用PEP 3107中引入的函数类型注释。 此外，还可以使用MyPy-style类型的注释声明不同类型的参数。
- 变更 `decoder_input` 的初始化 在原有实现中，我们用 `torch.LongTensor([[SOS_token]])` 初始化了 `decoder_input` 的张量。当脚本编写时,我们不允许像这样以 一种文字方式初始化张量。取而代之的是，我们可以用一个显式的torch函数，比如 `torch.ones` 来初始化我们的张量。这种情况下， 我们可以很方便的复制标量 `decoder_input` 和通过将1乘以我们存在常量中的 `SOS_token` 的值 `self._SOS_token` 得到的张量。 ```buildoutcfg class GreedySearchDecoder(torch.jit.ScriptModule): def **init**(self, encoder, decoder, decoder_n_layers): super(GreedySearchDecoder, self).**init**() self.encoder = encoder self.decoder = decoder self._device = device self._SOS_token = SOS_token self._decoder_n_layers = decoder_n_layers

constants = ['_device', '_SOS_token', '_decoder_n_layers']

@torch.jit.script_method def forward(self, input_seq : torch.Tensor, input_length : torch.Tensor, max_length : int): # 通过编码器模型转发输入 encoder_outputs, encoder_hidden = self.encoder(input_seq, input_length) # 准备编码器的最终隐藏层作为解码器的第一个隐藏输入 decoder_hidden = encoder_hidden[:self._decoder_n_layers] # 使用SOS_token初始化解码器输入 decoder_input = torch.ones(1, 1, device=self._device, dtype=torch.long) * self._SOS_token # 初始化张量以将解码后的单词附加到 all_tokens = torch.zeros([0], device=self._device, dtype=torch.long) all_scores = torch.zeros([0], device=self._device) # 一次迭代地解码一个词令牌 for _ in range(max_length): # 正向通过解码器 decoder_output, decoder_hidden = self.decoder(decoder_input, decoder_hidden, encoder_outputs) # 获得最可能的单词标记及其softmax分数 decoder_scores, decoder_input = torch.max(decoder_output, dim=1) # 记录令牌和分数 all_tokens = torch.cat((all_tokens, decoder_input), dim=0) all_scores = torch.cat((all_scores, decoder_scores), dim=0) # 准备当前令牌作为下一个解码器

输入（添加维度） decoder_input = torch.unsqueeze(decoder_input, 0) # 返回词令牌和分数的集合 return all_tokens, all_scores

```
#### 8.2 输入评估
接下来，我们定义一些函数来计算输入。求值函数`evaluate`接受一个规范化字符串语句，将其处理为其对应的单词索引张量(批处理大小
为1)，并将该张量传递给一个名为`searcher`的`GreedySearchDecoder`实例，以处理编码/解码过程。检索器返回输出的单词索引向量和
一个分数张量，该张量对应于每个解码的单词标记的`softmax`分数。最后一步是使用
`voc.index2word`将每个单词索引转换回其字符串表示形式。

我们还定义了两个函数来计算输入语句。`evaluateInput`函数提示用户输入，并计算输入。它持续请求另一次输入，直到用户输入“q”或“quit”。

`evaluateExample`函数只接受一个字符串输入语句作为参数，对其进行规范化、计算并输出响应。
```buildoutcfg
def evaluate(encoder, decoder, searcher, voc, sentence, max_length=MAX_LENGTH):
    # 格式化输入句子作为批处理
    # words -> indexes
    indexes_batch = [indexesFromSentence(voc, sentence)]
    # 创建长度张量
    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])
    # 转置批量的维度以匹配模型的期望
    input_batch = torch.LongTensor(indexes_batch).transpose(0, 1)
    # 使用适当的设备
    input_batch = input_batch.to(device)
    lengths = lengths.to(device)
    # 用earcher解码句子s
    tokens, scores = searcher(input_batch, lengths, max_length)
    # indexes -> words
    decoded_words = [voc.index2word[token.item()] for token in tokens]
    return decoded_words


# 评估来自用户输入的输入(stdin)
def evaluateInput(encoder, decoder, searcher, voc):
    input_sentence = ''
    while(1):
        try:
            # 获取输入的句子
            input_sentence = input('> ')
            # Check if it is quit case
            if input_sentence == 'q' or input_sentence == 'quit': break
            # 规范化句子
            input_sentence = normalizeString(input_sentence)
            # 评估句子
            output_words = evaluate(encoder, decoder, searcher, voc,
```
```

```
input_sentence)
            # 格式化和打印回复句
            output_words[:] = [x for x in output_words if not (x == 'EOS' or x
== 'PAD')]
            print('Bot:', ' '.join(output_words))

        except KeyError:
            print("Error: Encountered unknown word.")

# 规范化输入句子并调用evaluate()
def evaluateExample(sentence, encoder, decoder, searcher, voc):
    print("> " + sentence)
    # 规范化句子
    input_sentence = normalizeString(sentence)
    # 评估句子
    output_words = evaluate(encoder, decoder, searcher, voc, input_sentence)
    output_words[:] = [x for x in output_words if not (x == 'EOS' or x ==
'PAD')]
    print('Bot:', ' '.join(output_words))
```

9.加载预训练参数

9.1 使用托管模型

托管模型使用步骤:

1.下载模型这里. 2.设置 `loadFilename` 变量作为下载的检查点文件的路径 3.将 `checkpoint = torch.load(loadFilename)` 行取消注释，表示托管模型在CPU上训练。

9.2 使用自己的模型

加载自己的预训练模型设计步骤:

1.将 `loadFilename` 变量设置为希望加载的检查点文件的路径。注意，如果您遵循从chatbot tutorial中保存模型的协议，这会涉及更改 `model_name` 、 `encoder_n_layers` 、 `decoder_n_layers` 、 `hidden_size` 和 `checkpoint_iter` (因为这些值在模型路径中使用到)。 2.如果你在CPU上训练，确保你在 `checkpoint = torch.load(loadFilename)` 行打开了检查点。如果你在GPU上训练，并且在CPU运行这篇 教程，解除 `checkpoint = torch.load(loadFilename, map_location=torch.device('cpu'))` 的注释。

- 混合前端笔记

请注意，我们像往常一样初始化并将参数加载到编码器和解码器模型中。另外，在跟踪模型之前，我们必须调用 `.to(device)` 来设置模型 的设备选项，调用 `.eval()` 来设置抛出层 `dropout layer` 为test mode。 `TracedModule` 对象不继承 `to` 或 `eval` 方法。

```
save_dir = os.path.join("data", "save")
corpus_name = "cornell movie-dialogs corpus"

# 配置模型
model_name = 'cb_model'
attn_model = 'dot'
#attn_model = 'general'
#attn_model = 'concat'
hidden_size = 500
encoder_n_layers = 2
decoder_n_layers = 2
dropout = 0.1
batch_size = 64

# 如果你加载的是自己的模型
# 设置要加载的检查点
checkpoint_iter = 4000
# loadFilename = os.path.join(save_dir, model_name, corpus_name,
#                             '{}-{}_{}'.format(encoder_n_layers,
decoder_n_layers, hidden_size),
#                             '{}_checkpoint.tar'.format(checkpoint_iter))

# 如果你加载的是托管模型
loadFilename = 'data/4000_checkpoint.tar'

# 加载模型
# 强制CPU设备选项 (与本教程中的张量匹配)
checkpoint = torch.load(loadFilename, map_location=torch.device('cpu'))
encoder_sd = checkpoint['en']
decoder_sd = checkpoint['de']
encoder_optimizer_sd = checkpoint['en_opt']
decoder_optimizer_sd = checkpoint['de_opt']
embedding_sd = checkpoint['embedding']
voc = Voc(corpus_name)
voc.__dict__ = checkpoint['voc_dict']


print('Building encoder and decoder ...')
# 初始化词向量
embedding = nn.Embedding(voc.num_words, hidden_size)
embedding.load_state_dict(embedding_sd)
# 初始化编码器和解码器模型
encoder = EncoderRNN(hidden_size, embedding, encoder_n_layers, dropout)
```

```
decoder = LuongAttnDecoderRNN(attn_model, embedding, hidden_size, voc.num_words,
decoder_n_layers, dropout)
# 加载训练模型参数
encoder.load_state_dict(encoder_sd)
decoder.load_state_dict(decoder_sd)
# 使用适当的设备
encoder = encoder.to(device)
decoder = decoder.to(device)
# 将dropout层设置为eval模式
encoder.eval()
decoder.eval()
print('Models built and ready to go!')
```

* 输出

```
Building encoder and decoder ...
Models built and ready to go!
```

10.将模型转换为 Torch 脚本

10.1 编码器

正如前文所述，要将编码器模型转换为Torch脚本，我们需要使用跟踪 `Tracing` 。跟踪任何需要通过模型的 `forward` 方法运行一个示例 输入，以及跟踪数据相遇时的图形计算。编码器模型接收一个输入序列和一个长度相关的张量。因此，我们创建一个输入序列 `test_seq` ，配置合适的大小 `(MAX_LENGTH,1)` 包含适当范围内的数值[0,voc.num_words]以及搭配的类型(int64)。我们还创建了 `test_seq_length` 标量，该标量实际包含与test_seq中单词数量对应的值。下一步是使用 `torch.jit.trace` 函数来跟踪模型。注意，我们传递的第一个参数 是要跟踪的模块，第二个参数是模块forward方法的参数元组。

10.2 解码器

我们对解码器的跟踪过程与对编码器的跟踪过程相同。请注意，我们对 `traced_encoder` 的一组随机输入调用 `forward` ，以获得解码器所 需的输出。这不是必需的，因为我们也可以简单地生成一个形状、类型和值范围正确的张量。这种方法是可行的，因为在我们的例子中， 我们对张量的值没有任何约束，因为我们没有任何操作可能导致超出范围的输入出错。

10.3 贪婪搜索解码器

回想一下，由于存在依赖于数据的控制流，我们为搜索器模块编写了脚本。在脚本化的情况下，我们通过添加修饰符并确保实现符合脚本 规则来预先完成转换工作。我们初始化脚本搜索器的方式与初始化未脚本化变量的方式相同。

```
### 转换编码器模型
# 创建人工输入
test_seq = torch.LongTensor(MAX_LENGTH, 1).random_(0, voc.num_words).to(device)
test_seq_length = torch.LongTensor([test_seq.size()[0]]).to(device)
# 跟踪模型
traced_encoder = torch.jit.trace(encoder, (test_seq, test_seq_length))

### 转换解码器模型
# 创建并生成人工输入
test_encoder_outputs, test_encoder_hidden = traced_encoder(test_seq,
test_seq_length)
test_decoder_hidden = test_encoder_hidden[:decoder.n_layers]
test_decoder_input = torch.LongTensor(1, 1).random_(0, voc.num_words)
# 跟踪模型
traced_decoder = torch.jit.trace(decoder, (test_decoder_input,
test_decoder_hidden, test_encoder_outputs))

### 初始化searcher模块
scripted_searcher = GreedySearchDecoder(traced_encoder, traced_decoder,
decoder.n_layers)
```

11.图形打印

现在我们的模型是Torch脚本形式的，我们可以打印每个模型的图形，以确保适当地捕获计算图形。因为 `scripted_searcher` 包含 `traced_encoder` 和 `traced_decoder` ，所以这些图将以内联方式打印。

- 输出

```
scripted_searcher graph:
 graph(%input_seq : Tensor,
      %input_length : Tensor,
      %max_length : int,
      %126 : Tensor,
      %127 : Tensor,
      %128 : Tensor,
      %129 : Tensor,
```

```
      %130 : Tensor,
      %131 : Tensor,
      %132 : Tensor,
      %133 : Tensor,
      %134 : Tensor,
      %135 : Tensor,
      %136 : Tensor,
      %137 : Tensor,
      %138 : Tensor,
      %139 : Tensor,
      %140 : Tensor,
      %141 : Tensor,
      %142 : Tensor,
      %143 : Tensor,
      %144 : Tensor,
      %145 : Tensor,
      %146 : Tensor,
      %147 : Tensor,
      %148 : Tensor,
      %149 : Tensor,
      %150 : Tensor,
      %151 : Tensor,
      %152 : Tensor,
      %153 : Tensor,
      %154 : Tensor,
      %155 : Tensor):
  %4 : bool? = prim::Constant()
  %5 : int? = prim::Constant()
  %6 : int = prim::Constant[value=9223372036854775807](), scope: EncoderRNN
  %7 : float = prim::Constant[value=0](), scope: EncoderRNN
  %8 : float = prim::Constant[value=0.1](), scope: EncoderRNN/GRU[gru]
  %9 : int = prim::Constant[value=2](), scope: EncoderRNN/GRU[gru]
  %10 : bool = prim::Constant[value=1](), scope: EncoderRNN/GRU[gru]
  %11 : int = prim::Constant[value=6](), scope: EncoderRNN/GRU[gru]
  %12 : int = prim::Constant[value=500](), scope: EncoderRNN/GRU[gru]
  %13 : int = prim::Constant[value=4](), scope: EncoderRNN
  %14 : Device = prim::Constant[value="cpu"](), scope: EncoderRNN
  %15 : bool = prim::Constant[value=0](), scope: EncoderRNN/
Embedding[embedding]
  %16 : int = prim::Constant[value=-1](), scope: EncoderRNN/
Embedding[embedding]
  %17 : int = prim::Constant[value=0]()
  %18 : int = prim::Constant[value=1]()
  %input.7 : Float(10, 1, 500) = aten::embedding(%155, %input_seq, %16, %15,
%15), scope: EncoderRNN/Embedding[embedding]
  %lengths : Long(1) = aten::to(%input_length, %14, %13, %15, %15), scope:
EncoderRNN
  %input.1 : Float(10, 500), %batch_sizes : Long(10) =
aten::_pack_padded_sequence(%input.7, %lengths, %15), scope: EncoderRNN
```

```
  %43 : int[] = prim::ListConstruct(%13, %18, %12), scope: EncoderRNN/GRU[gru]
  %hx : Float(4, 1, 500) = aten::zeros(%43, %11, %17, %14, %15), scope:
EncoderRNN/GRU[gru]
  %45 : Tensor[] = prim::ListConstruct(%154, %153, %152, %151, %150, %149,
%148, %147, %146, %145, %144, %143, %142, %141, %140, %139), scope: EncoderRNN/
GRU[gru]
  %46 : Float(10, 1000), %encoder_hidden : Float(4, 1, 500) = aten::gru(%input.
1, %batch_sizes, %hx, %45, %10, %9, %8, %15, %10), scope: EncoderRNN/GRU[gru]
  %48 : int = aten::size(%batch_sizes, %17), scope: EncoderRNN
  %max_seq_length : Long() = prim::NumToTensor(%48), scope: EncoderRNN
  %50 : int = prim::Int(%max_seq_length), scope: EncoderRNN
  %outputs : Float(10, 1, 1000), %52 : Long(1) =
aten::_pad_packed_sequence(%46, %batch_sizes, %15, %7, %50), scope: EncoderRNN
  %53 : Float(10, 1, 1000) = aten::slice(%outputs, %17, %17, %6, %18), scope:
EncoderRNN
  %54 : Float(10, 1, 1000) = aten::slice(%53, %18, %17, %6, %18), scope:
EncoderRNN
  %55 : Float(10, 1!, 500) = aten::slice(%54, %9, %17, %12, %18), scope:
EncoderRNN
  %56 : Float(10, 1, 1000) = aten::slice(%outputs, %17, %17, %6, %18), scope:
EncoderRNN
  %57 : Float(10, 1, 1000) = aten::slice(%56, %18, %17, %6, %18), scope:
EncoderRNN
  %58 : Float(10, 1!, 500) = aten::slice(%57, %9, %12, %6, %18), scope:
EncoderRNN
  %encoder_outputs : Float(10, 1, 500) = aten::add(%55, %58, %18), scope:
EncoderRNN
  %decoder_hidden.1 : Tensor = aten::slice(%encoder_hidden, %17, %17, %9, %18)
  %61 : int[] = prim::ListConstruct(%18, %18)
  %62 : Tensor = aten::ones(%61, %13, %5, %14, %4)
  %decoder_input.1 : Tensor = aten::mul(%62, %18)
  %64 : int[] = prim::ListConstruct(%17)
  %all_tokens.1 : Tensor = aten::zeros(%64, %13, %5, %14, %4)
  %66 : int[] = prim::ListConstruct(%17)
  %all_scores.1 : Tensor = aten::zeros(%66, %5, %5, %14, %4)
  %all_scores : Tensor, %all_tokens : Tensor, %decoder_hidden : Tensor,
%decoder_input : Tensor = prim::Loop(%max_length, %10, %all_scores.1,
%all_tokens.1, %decoder_hidden.1, %decoder_input.1)
    block0(%72 : int, %73 : Tensor, %74 : Tensor, %75 : Tensor, %76 : Tensor):
      %input.2 : Float(1, 1, 500) = aten::embedding(%138, %76, %16, %15, %15),
scope: LuongAttnDecoderRNN/Embedding[embedding]
      %input.3 : Float(1, 1, 500) = aten::dropout(%input.2, %8, %15), scope:
LuongAttnDecoderRNN/Dropout[embedding_dropout]
      %97 : Tensor[] = prim::ListConstruct(%137, %136, %135, %134, %133, %132,
%131, %130), scope: LuongAttnDecoderRNN/GRU[gru]
      %hidden : Float(1, 1, 500), %decoder_hidden.2 : Float(2, 1, 500) =
aten::gru(%input.3, %75, %97, %10, %9, %8, %15, %15, %15), scope:
LuongAttnDecoderRNN/GRU[gru]
      %100 : Float(10, 1, 500) = aten::mul(%hidden, %encoder_outputs), scope:
```

```
LuongAttnDecoderRNN/Attn[attn]
      %101 : int[] = prim::ListConstruct(%9), scope: LuongAttnDecoderRNN/
Attn[attn]
      %attn_energies : Float(10, 1) = aten::sum(%100, %101, %15), scope:
LuongAttnDecoderRNN/Attn[attn]
      %input.4 : Float(1!, 10) = aten::t(%attn_energies), scope:
LuongAttnDecoderRNN/Attn[attn]
      %104 : Float(1, 10) = aten::softmax(%input.4, %18), scope:
LuongAttnDecoderRNN/Attn[attn]
      %attn_weights : Float(1, 1, 10) = aten::unsqueeze(%104, %18), scope:
LuongAttnDecoderRNN/Attn[attn]
      %106 : Float(1!, 10, 500) = aten::transpose(%encoder_outputs, %17, %18),
scope: LuongAttnDecoderRNN
      %context.1 : Float(1, 1, 500) = aten::bmm(%attn_weights, %106), scope:
LuongAttnDecoderRNN
      %rnn_output : Float(1, 500) = aten::squeeze(%hidden, %17), scope:
LuongAttnDecoderRNN
      %context : Float(1, 500) = aten::squeeze(%context.1, %18), scope:
LuongAttnDecoderRNN
      %110 : Tensor[] = prim::ListConstruct(%rnn_output, %context), scope:
LuongAttnDecoderRNN
      %input.5 : Float(1, 1000) = aten::cat(%110, %18), scope:
LuongAttnDecoderRNN
      %112 : Float(1000!, 500!) = aten::t(%129), scope: LuongAttnDecoderRNN/
Linear[concat]
      %113 : Float(1, 500) = aten::addmm(%128, %input.5, %112, %18, %18),
scope: LuongAttnDecoderRNN/Linear[concat]
      %input.6 : Float(1, 500) = aten::tanh(%113), scope: LuongAttnDecoderRNN
      %115 : Float(500!, 7826!) = aten::t(%127), scope: LuongAttnDecoderRNN/
Linear[out]
      %input : Float(1, 7826) = aten::addmm(%126, %input.6, %115, %18, %18),
scope: LuongAttnDecoderRNN/Linear[out]
      %decoder_output : Float(1, 7826) = aten::softmax(%input, %18), scope:
LuongAttnDecoderRNN
      %decoder_scores : Tensor, %decoder_input.2 : Tensor =
aten::max(%decoder_output, %18, %15)
      %120 : Tensor[] = prim::ListConstruct(%74, %decoder_input.2)
      %all_tokens.2 : Tensor = aten::cat(%120, %17)
      %122 : Tensor[] = prim::ListConstruct(%73, %decoder_scores)
      %all_scores.2 : Tensor = aten::cat(%122, %17)
      %decoder_input.3 : Tensor = aten::unsqueeze(%decoder_input.2, %17)
      -> (%10, %all_scores.2, %all_tokens.2, %decoder_hidden.2, %decoder_input.
3)
  %125 : (Tensor, Tensor) = prim::TupleConstruct(%all_tokens, %all_scores)
  return (%125)
```

12.运行结果评估

最后，我们将使用Torch脚本模型对聊天机器人模型进行评估。如果转换正确，模型的行为将与它们在即时模式表示中的行为完全相同。

默认情况下，我们计算一些常见的查询语句。如果您想自己与机器人聊天，取消对 `evaluateInput` 行的注释并让它旋转。

```
# 评估例子
sentences = ["hello", "what's up?", "who are you?", "where am I?", "where are you
from?"]
for s in sentences:
    evaluateExample(s, traced_encoder, traced_decoder, scripted_searcher, voc)

# 评估你的输入
#evaluateInput(traced_encoder, traced_decoder, scripted_searcher, voc)
```

* 输出

```
> hello
Bot: hello .
> what's up?
Bot: i m going to get my car .
> who are you?
Bot: i m the owner .
> where am I?
Bot: in the house .
> where are you from?
Bot: south america .
```

13.保存模型

现在我们已经成功地将模型转换为Torch脚本，接下来将对其进行序列化，以便在非python部署环境中使用。为此，我们只需保存 `scripted_searcher` 模块，因为这是用于对聊天机器人模型运行推理的面向用户的接口。保存脚本模块时，使用 `script_module.save(PATH)` 代替 `torch.save(model, PATH)` 。

保存和加载模型

当保存和加载模型时，需要熟悉三个核心功能：

1. `torch.save`：将序列化对象保存到磁盘。此函数使用Python的 `pickle` 模块进行序列化。使用此函数可以保存如模型、tensor、字典等各种对象。
2. `torch.load`：使用pickle的 `unpickling` 功能将pickle对象文件反序列化到内存。此功能还可以有助于设备加载数据。
3. `torch.nn.Module.load_state_dict`：使用反序列化函数 state_dict 来加载模型的参数字典。

1.什么是状态字典：state_dict?

在PyTorch中，`torch.nn.Module` 模型的可学习参数（即权重和偏差）包含在模型的参数中，（使用 `model.parameters()` 可以进行访问）。 `state_dict` 是Python字典对象，它将每一层映射到其参数张量。注意，只有具有可学习参数的层（如卷积层，线性层等）的模型 才具有 `state_dict` 这一项。目标优化 `torch.optim` 也有 `state_dict` 属性，它包含有关优化器的状态信息，以及使用的超参数。

因为state_dict的对象是Python字典，所以它们可以很容易的保存、更新、修改和恢复，为PyTorch模型和优化器添加了大量模块。

下面通过从简单模型训练一个分类器中来了解一下 `state_dict` 的使用。

```
# 定义模型
class TheModelClass(nn.Module):
    def __init__(self):
        super(TheModelClass, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        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 = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
```

```
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# 初始化模型
model = TheModelClass()

# 初始化优化器
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

# 打印模型的状态字典
print("Model's state_dict:")
for param_tensor in model.state_dict():
    print(param_tensor, "\t", model.state_dict()[param_tensor].size())

# 打印优化器的状态字典
print("Optimizer's state_dict:")
for var_name in optimizer.state_dict():
    print(var_name, "\t", optimizer.state_dict()[var_name])
```

* 输出

```
Model's state_dict:
conv1.weight     torch.Size([6, 3, 5, 5])
conv1.bias   torch.Size([6])
conv2.weight     torch.Size([16, 6, 5, 5])
conv2.bias   torch.Size([16])
fc1.weight   torch.Size([120, 400])
fc1.bias     torch.Size([120])
fc2.weight   torch.Size([84, 120])
fc2.bias     torch.Size([84])
fc3.weight   torch.Size([10, 84])
fc3.bias     torch.Size([10])

Optimizer's state_dict:
state    {}
param_groups     [{'lr': 0.001, 'momentum': 0.9, 'dampening': 0, 'weight_decay':
0, 'nesterov': False, 'params': [4675713712, 4675713784, 4675714000, 4675714072,
4675714216, 4675714288, 4675714432, 4675714504, 4675714648, 4675714720]}]
```

2.保存和加载推理模型

2.1 保存/加载 `state_dict`（推荐使用）

- 保存

```
torch.save(model.state_dict(), PATH)
```

- 加载

```
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
model.eval()
```

当保存好模型用来推断的时候，只需要保存模型学习到的参数，使用 `torch.save()` 函数来保存模型 `state_dict`,它会给模型恢复提供 最大的灵活性，这就是为什么要推荐它来保存的原因。

在 PyTorch 中最常见的模型保存使‘.pt’或者是‘.pth’作为模型文件扩展名。

请记住，在运行推理之前，务必调用 `model.eval()` 去设置 dropout 和 batch normalization 层为评估模式。如果不这么做，可能导致 模型推断结果不一致。

- 注意

`load_state_dict()` 函数只接受字典对象，而不是保存对象的路径。这就意味着在你传给 `load_state_dict()` 函数之前，你必须反序列化 你保存的 `state_dict` 。例如，你无法通过 `model.load_state_dict(PATH)` 来加载模型。

2.2 保存/加载完整模型

- 保存

```
torch.save(model, PATH)
```

- 加载

```

```

```
# 模型类必须在此之前被定义
model = torch.load(PATH)
model.eval()
```

此部分保存/加载过程使用最直观的语法并涉及最少量的代码。以 Python `pickle 模块的方式来保存模型。这种方法的缺点是序列化数据受 限于某种特殊的类而且需要确切的字典结构。这是因为pickle无法保存模型类本身。相反，它保存包含类的文件的路径，该文件在加载时使用。 因此，当在其他项目使用或者重构之后，您的代码可能会以各种方式中断。

在 PyTorch 中最常见的模型保存使用‘.pt’或者是‘.pth’作为模型文件扩展名。

请记住，在运行推理之前，务必调用 `model.eval()` 设置 dropout 和 batch normalization 层为评估模式。如果不这么做，可能导致模型推断结果不一致。

3. 保存和加载 Checkpoint 用于推理/继续训练

- 保存

```
torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss,
            ...
            }, PATH)
```

- 加载

```
model = TheModelClass(*args, **kwargs)
optimizer = TheOptimizerClass(*args, **kwargs)

checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']

model.eval()
# - or -
model.train()
```

> 当保存成 Checkpoint 的时候，可用于推理或者是继续训练，保存的不仅仅是模型的 state_dict 。保存优化器的 state_dict 也很重要, 因为它包含作为模型训练更新的缓冲区和参数。你也许想保存其他项目，比如最新记录的训练损失，外部的 `torch.nn.Embedding` 层等等。

要保存多个组件，请在字典中组织它们并使用 `torch.save()` 来序列化字典。PyTorch 中常见的保存checkpoint 是使用 .tar 文件扩展名。

要加载项目，首先需要初始化模型和优化器，然后使用 `torch.load()` 来加载本地字典。这里,你可以非常容易的通过简单查询字典来访问你所保存的项目。

请记住在运行推理之前，务必调用 `model.eval()` 去设置 dropout 和 batch normalization 为评估。如果不这样做，有可能得到不一致的推断结果。 如果你想要恢复训练，请调用 `model.train()` 以确保这些层处于训练模式。

4. 在一个文件中保存多个模型

- 保存

```
torch.save({
            'modelA_state_dict': modelA.state_dict(),
            'modelB_state_dict': modelB.state_dict(),
            'optimizerA_state_dict': optimizerA.state_dict(),
            'optimizerB_state_dict': optimizerB.state_dict(),
            ...
            }, PATH)
```

- 加载

```
modelA = TheModelAClass(*args, **kwargs)
modelB = TheModelBClass(*args, **kwargs)
optimizerA = TheOptimizerAClass(*args, **kwargs)
optimizerB = TheOptimizerBClass(*args, **kwargs)

checkpoint = torch.load(PATH)
modelA.load_state_dict(checkpoint['modelA_state_dict'])
modelB.load_state_dict(checkpoint['modelB_state_dict'])
optimizerA.load_state_dict(checkpoint['optimizerA_state_dict'])
optimizerB.load_state_dict(checkpoint['optimizerB_state_dict'])

modelA.eval()
modelB.eval()
# - or -
```

```
modelA.train()
modelB.train()
```

当保存一个模型由多个 `torch.nn.Modules` 组成时，例如GAN(对抗生成网络)、sequence-to-sequence (序列到序列模型), 或者是多个模 型融合, 可以采用与保存常规检查点相同的方法。换句话说，保存每个模型的 state_dict 的字典和相对应的优化器。如前所述，可以通 过简单地将它们附加到字典的方式来保存任何其他项目，这样有助于恢复训练。

PyTorch 中常见的保存 checkpoint 是使用 .tar 文件扩展名。

要加载项目，首先需要初始化模型和优化器，然后使用 `torch.load()` 来加载本地字典。这里，你可以非常容易的通过简单查询字典来访问你所保存的项目。

请记住在运行推理之前，务必调用 `model.eval()` 去设置 dropout 和 batch normalization 为评估。如果不这样做，有可能得到不一致的推断结果。 如果你想要恢复训练，请调用 `model.train()` 以确保这些层处于训练模式。

5. 使用在不同模型参数下的热启动模式

- 保存

```
torch.save(modelA.state_dict(), PATH)
```

- 加载

```
modelB = TheModelBClass(*args, **kwargs)
modelB.load_state_dict(torch.load(PATH), strict=False)
```

在迁移学习或训练新的复杂模型时，部分加载模型或加载部分模型是常见的情况。利用训练好的参数，有助于热启动训练过程，并希望帮助你的模型比从头开始训练能够更快地收敛。

无论是从缺少某些键的 state_dict 加载还是从键的数目多于加载模型的 state_dict , 都可以通过在 `load_state_dict()` 函数中将 `strict` 参数设置为 False 来忽略非匹配键的函数。

如果要将参数从一个层加载到另一个层，但是某些键不匹配，主要修改正在加载的 state_dict 中的参数键的名称以匹配要在加载到模型中的键即可。

6. 通过设备保存/加载模型

6.1 保存到 CPU、加载到 CPU

- 保存

```
torch.save(model.state_dict(), PATH)
```

- 加载

```
device = torch.device('cpu')
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH, map_location=device))
```

当从CPU上加载模型在GPU上训练时, 将 `torch.device('cpu')` 传递给 `torch.load()` 函数中的 `map_location` 参数.在这种情况下，使用 `map_location` 参数将张量下的存储器动态的重新映射到CPU设备。

6.2 保存到 GPU、加载到 GPU

- 保存

```
torch.save(model.state_dict(), PATH)
```

- 加载

```
device = torch.device("cuda")
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
model.to(device)
# 确保在你提供给模型的任何输入张量上调用input = input.to(device)
```

当在GPU上训练并把模型保存在GPU，只需要使用 `model.to(torch.device('cuda'))` ，将初始化的 model 转换为 CUDA 优化模型。另外，请 务必在所有模型输入上使用 `.to(torch.device('cuda'))` 函数来为模型准备数据。请注意，调用 `my_tensor.to(device)` 会在GPU上返回 `my_tensor` 的副本。 因此，请记住手动覆盖张量： `my_tensor= my_tensor.to(torch.device('cuda'))` 。

6.3 保存到 CPU，加载到 GPU

- 保存

```
torch.save(model.state_dict(), PATH)
```

- 加载

```
device = torch.device("cuda")
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH, map_location="cuda:0"))  # Choose
whatever GPU device number you want
model.to(device)
# 确保在你提供给模型的任何输入张量上调用input = input.to(device)
```

在CPU上训练好并保存的模型加载到GPU时，将 `torch.load()` 函数中的 `map_location` 参数设置为 `cuda:device_id` 。这会将模型加载到 指定的GPU设备。接下来，请务必调用 `model.to(torch.device('cuda'))` 将模型的参数张量转换为 CUDA 张量。最后，确保在所有模型输入上使用 `.to(torch.device('cuda'))` 函数来为CUDA优化模型。请注意，调用 `my_tensor.to(device)` 会在GPU上返回 `my_tensor` 的新副本。它不会覆盖 `my_tensor` 。 因此， 请手动覆盖张量 `my_tensor = my_tensor.to(torch.device('cuda'))` 。

6.4 保存 `torch.nn.DataParallel` 模型

- 保存

```
torch.save(model.module.state_dict(), PATH)
```

- 加载

```
# 加载任何你想要的设备
```

`torch.nn.DataParallel` 是一个模型封装，支持并行GPU使用。要普通保存 DataParallel 模型，请保存 `model.module.state_dict()` 。 这样，你就可以非常灵活地以任何方式加载模型到你想要的设备中。

微调基于 torchvision 0.3的目标检测模型

在本教程中，我们将微调在 Penn-Fudan 数据库中对行人检测和分割的已预先训练的 Mask R-CNN 模型。它包含170个图像和345个行人实例，我们 将用它来说明如何在 torchvision 中使用新功能，以便在自定义数据集上训练实例分割模型。

1.定义数据集

对于训练对象检测的引用脚本，实例分割和人员关键点检测要求能够轻松支持添加新的自定义数据。数据集应该从标准的类 `torch.utils.data.Dataset` 继承而来，并实现 `_len` 和 `_getitem_`

我们要求的唯一特性是数据集的 `__getitem__` 应该返回： * 图像：PIL图像大小(H,W) * 目标：包含以下字段的字典
<1> `boxes(FloatTensor[N,4])` ：N边框（bounding boxes）坐标的格式[x0,x1,y0,y1]，取值范围是0到W,0到H。
<2> `labels(Int64Tensor[N])` ：每个边框的标签。
<3> `image_id(Int64Tensor[1])` ：图像识别器，它应该在数据集中的所有图像中是唯一的，并在评估期间使用。
<4> `area(Tensor[N])` ：边框的面积，在使用COCO指标进行评估时使用此项来分隔小、中和大框之间的度量标准得分。
<5> `iscrowed(UInt8Tensor[N,H,W])` ：在评估期间属性设置为 `iscrowed=True` 的实例会被忽略。
<6> (可选) `masks(UInt8Tesor[N,H,W])` ：每个对象的分段掩码。
<7> (可选) `keypoints (FloatTensor[N, K, 3]` ：对于N个对象中的每一个，它包含[x，y，visibility]格式的K个关键点，用 于定义对象。 `visibility = 0` 表示关键点不可见。请注意，对于数据扩充，翻转关键点的概念取决于数据表示，您应该调整 reference/detection/transforms.py 以用于新的关键点表示。

如果你的模型返回上述方法，它们将使其适用于培训和评估，并将使用 pycocotools 的评估脚本。

此外，如果要在训练期间使用宽高比分组（以便每个批次仅包含具有相似宽高比的图像），则建议还实现 `get_height_and_width` 方法， 该方法返回图像的高度和宽度。如果未提供此方法，我们将通过 `__getitem__` 查询数据集的所有元素，这会将图像加载到内存中，但比提供自定义方法时要慢。

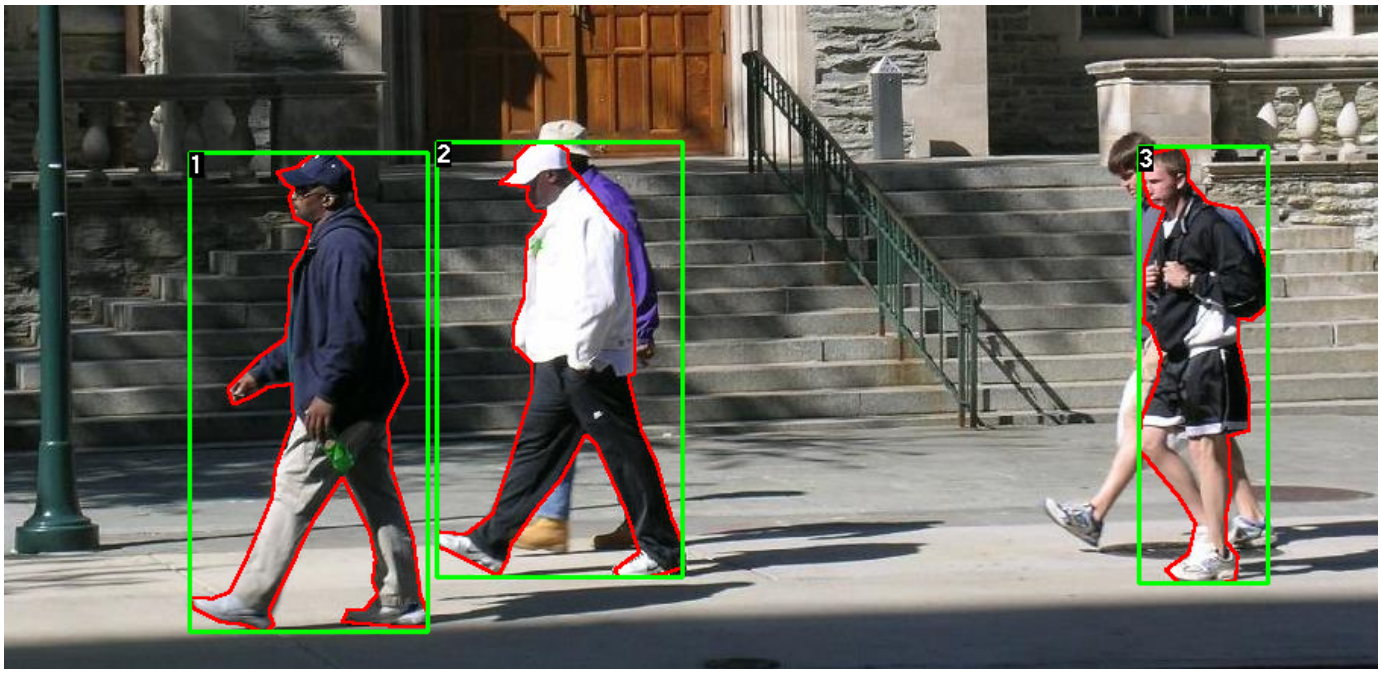
2.为 PennFudan 编写自定义数据集

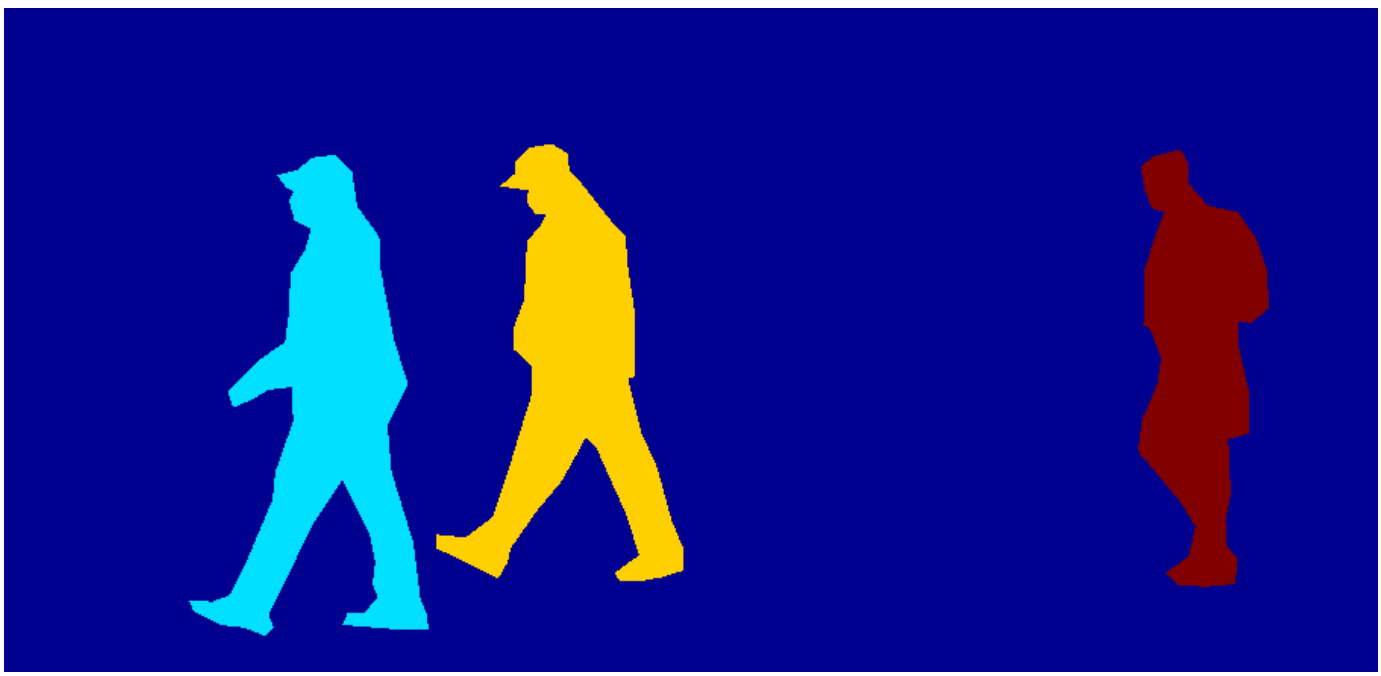
2.1 下载数据集

下载并解压缩zip文件后，我们有以下文件夹结构：

```
PennFudanPed/
  PedMasks/
    FudanPed00001_mask.png
    FudanPed00002_mask.png
    FudanPed00003_mask.png
    FudanPed00004_mask.png
    ...
  PNGImages/
    FudanPed00001.png
    FudanPed00002.png
    FudanPed00003.png
    FudanPed00004.png
```

下面是一个图像以及其分割掩膜的例子：





因此每个图像具有相应的分割掩膜，其中每个颜色对应于不同的实例。让我们为这个数据集写一个 `torch.utils.data.Dataset` 类。

2.2 为数据集编写类

```
import os
import numpy as np
import torch
from PIL import Image
```

```
class PennFudanDataset(object):
    def __init__(self, root, transforms):
        self.root = root
        self.transforms = transforms
        # 下载所有图像文件，为其排序
        # 确保它们对齐
        self.imgs = list(sorted(os.listdir(os.path.join(root, "PNGImages"))))
        self.masks = list(sorted(os.listdir(os.path.join(root, "PedMasks"))))

    def __getitem__(self, idx):
        # load images ad masks
        img_path = os.path.join(self.root, "PNGImages", self.imgs[idx])
        mask_path = os.path.join(self.root, "PedMasks", self.masks[idx])
        img = Image.open(img_path).convert("RGB")
        # 请注意我们还没有将mask转换为RGB，
        # 因为每种颜色对应一个不同的实例
        # 0是背景
        mask = Image.open(mask_path)
        # 将PIL图像转换为numpy数组
        mask = np.array(mask)
        # 实例被编码为不同的颜色
        obj_ids = np.unique(mask)
        # 第一个id是背景，所以删除它
        obj_ids = obj_ids[1:]

        # 将颜色编码的mask分成一组
        # 二进制格式
        masks = mask == obj_ids[:, None, None]

        # 获取每个mask的边界框坐标
        num_objs = len(obj_ids)
        boxes = []
        for i in range(num_objs):
            pos = np.where(masks[i])
            xmin = np.min(pos[1])
            xmax = np.max(pos[1])
            ymin = np.min(pos[0])
            ymax = np.max(pos[0])
            boxes.append([xmin, ymin, xmax, ymax])

        # 将所有转换为torch.Tensor
        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        # 这里仅有一个类
        labels = torch.ones((num_objs,), dtype=torch.int64)
        masks = torch.as_tensor(masks, dtype=torch.uint8)
```

```
        image_id = torch.tensor([idx])
        area = (boxes[:, 3] - boxes[:, 1]) * (boxes[:, 2] - boxes[:, 0])
        # 假设所有实例都不是人群
        iscrowd = torch.zeros((num_objs,), dtype=torch.int64)

        target = {}
        target["boxes"] = boxes
        target["labels"] = labels
        target["masks"] = masks
        target["image_id"] = image_id
        target["area"] = area
        target["iscrowd"] = iscrowd

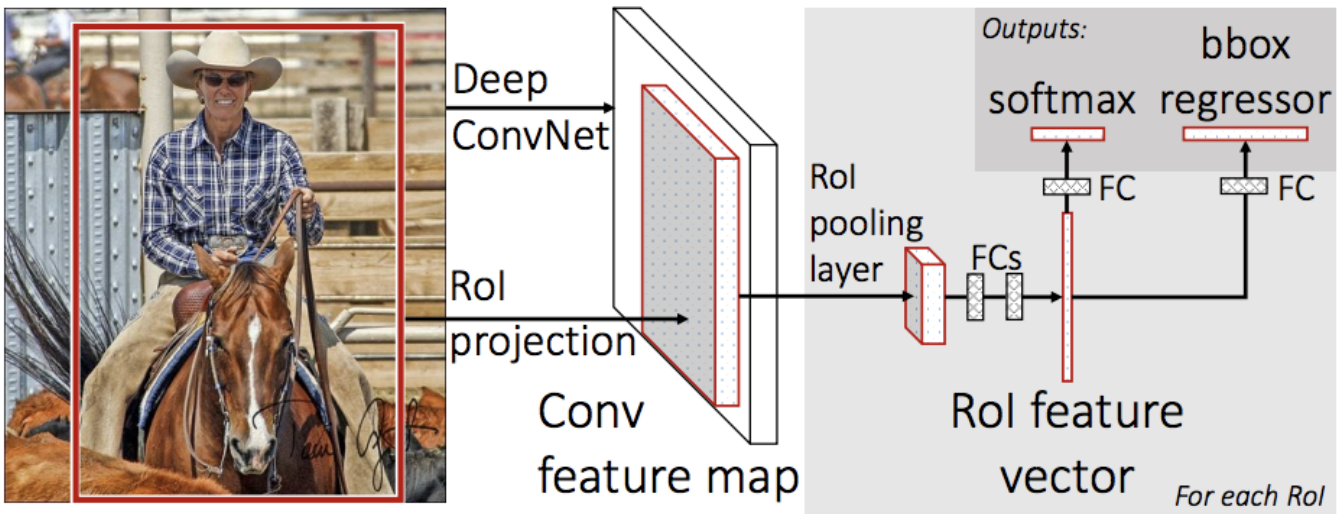
        if self.transforms is not None:
            img, target = self.transforms(img, target)

        return img, target

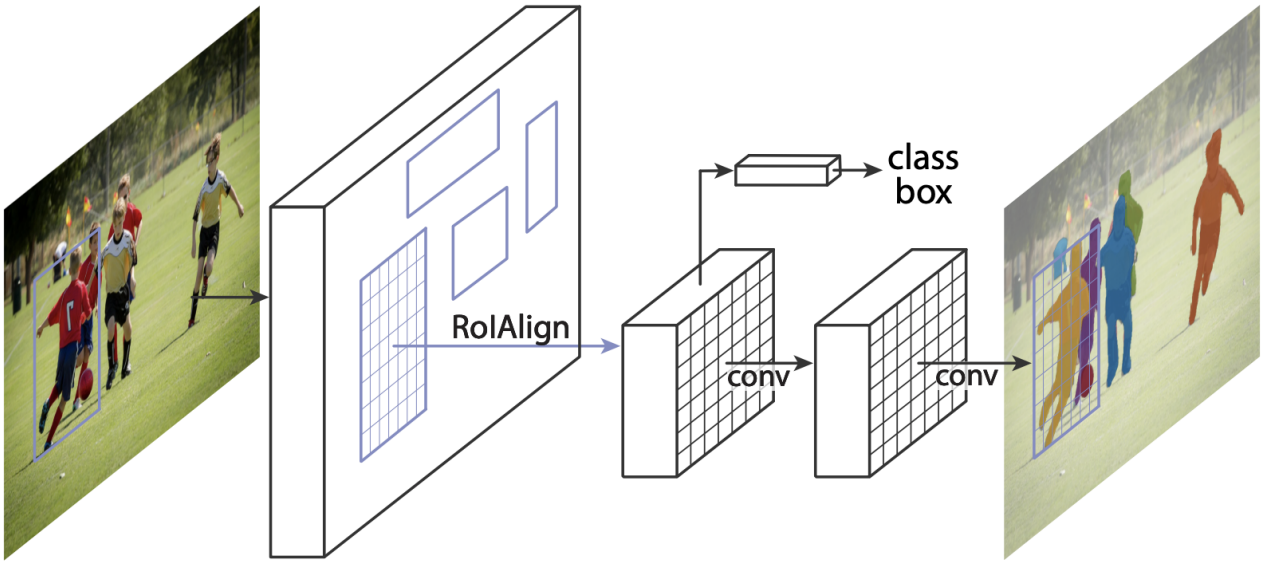
    def __len__(self):
        return len(self.imgs)
```

3.定义模型

现在我们需要定义一个可以上述数据集执行预测的模型。在本教程中，我们将使用 Mask R-CNN，它基于 Faster R-CNN。Faster R-CNN 是一种模型，可以预测图像中潜在对象的边界框和类别得分。



Mask R-CNN 在 Faster R-CNN 中添加了一个额外的分支，它还预测每个实例的分割蒙版。



有两种常见情况可能需要修改 `torchvision modelzoo` 中的一个可用模型。第一个是我们想要从预先训练的模型开始，然后微调最后一层。 另一种是当我们想要用不同的模型替换模型的主干时（例如，用于更快的预测）。

下面是对这两种情况的处理。 * 1 微调已经预训练的模型 让我们假设你想从一个在COCO上已预先训练过的模型开始，并希望为你的特定类进行微调。这是一种可行的方法：

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor

# 在COCO上加载经过预训练的预训练模型
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)

# replace the classifier with a new one, that has
# 将分类器替换为具有用户定义的 num_classes的新分类器
num_classes = 2  # 1 class (person) + background
# 获取分类器的输入参数的数量
in_features = model.roi_heads.box_predictor.cls_score.in_features
# 用新的头部替换预先训练好的头部
model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
```

- 2 修改模型以添加不同的主干

```
import torchvision
from torchvision.models.detection import FasterRCNN
from torchvision.models.detection.rpn import AnchorGenerator
```

```
# 加载预先训练的模型进行分类和返回
# 只有功能
backbone = torchvision.models.mobilenet_v2(pretrained=True).features
# FasterRCNN需要知道骨干网中的输出通道数量。对于mobilenet_v2，它是1280，所以我们需要在这里添加它
backbone.out_channels = 1280

# 我们让RPN在每个空间位置生成5 x 3个锚点
# 具有5种不同的大小和3种不同的宽高比。
# 我们有一个元组[元组[int]]
# 因为每个特征映射可能具有不同的大小和宽高比
anchor_generator = AnchorGenerator(sizes=((32, 64, 128, 256, 512),),
                                   aspect_ratios=((0.5, 1.0, 2.0),))

# 定义一下我们将用于执行感兴趣区域裁剪的特征映射，以及重新缩放后裁剪的大小。
# 如果您的主干返回Tensor，则featmap_names应为[0]。
# 更一般地，主干应该返回OrderedDict [Tensor]
# 并且在featmap_names中，您可以选择要使用的功能映射。
roi_pooler = torchvision.ops.MultiScaleRoIAlign(featmap_names=[0],
                                                output_size=7,
                                                sampling_ratio=2)

# 将这些pieces放在FasterRCNN模型中
model = FasterRCNN(backbone,
                   num_classes=2,
                   rpn_anchor_generator=anchor_generator,
                   box_roi_pool=roi_pooler)
```

3.1 PennFudan 数据集的实例分割模型

在我们的例子中，我们希望从预先训练的模型中进行微调，因为我们的数据集非常小，所以我们将遵循上述第一种情况。

这里我们还要计算实例分割掩膜，因此我们将使用 Mask R-CNN：

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import MaskRCNNPredictor


def get_model_instance_segmentation(num_classes):
    # 加载在COCO上预训练的预训练的实例分割模型
    model = torchvision.models.detection.maskrcnn_resnet50_fpn(pretrained=True)

    # 获取分类器的输入特征数
    in_features = model.roi_heads.box_predictor.cls_score.in_features
```

```
    # 用新的头部替换预先训练好的头部
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    # 现在获取掩膜分类器的输入特征数
    in_features_mask = model.roi_heads.mask_predictor.conv5_mask.in_channels
    hidden_layer = 256
    # 并用新的掩膜预测器替换掩膜预测器
    model.roi_heads.mask_predictor = MaskRCNNPredictor(in_features_mask,
                                                       hidden_layer,
                                                       num_classes)

    return model
```

就是这样，这将使模型准备好在您的自定义数据集上进行训练和评估。

4.整合

在 `references/detection/` 中，我们有许多辅助函数来简化训练和评估检测模型。在这里，我们将使用 `references/detection/engine.py` ， `references/detection/utils.py` 和 `references/detection/transforms.py` 。 只需将它们复制到您的文件夹并在此处使用它们。

4.1 为数据扩充/转换编写辅助函数：

```
import transforms as T

def get_transform(train):
    transforms = []
    transforms.append(T.ToTensor())
    if train:
        transforms.append(T.RandomHorizontalFlip(0.5))
    return T.Compose(transforms)
```

4.2 编写执行训练和验证的主要功能

```
from engine import train_one_epoch, evaluate
import utils


def main():
    # 在GPU上训练，若无GPU，可选择在CPU上训练
    device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')

    # 我们的数据集只有两个类 - 背景和人
```

```
    num_classes = 2
    # 使用我们的数据集和定义的转换
    dataset = PennFudanDataset('PennFudanPed', get_transform(train=True))
    dataset_test = PennFudanDataset('PennFudanPed', get_transform(train=False))

    # 在训练和测试集中拆分数据集
    indices = torch.randperm(len(dataset)).tolist()
    dataset = torch.utils.data.Subset(dataset, indices[:-50])
    dataset_test = torch.utils.data.Subset(dataset_test, indices[-50:])

    # 定义训练和验证数据加载器
    data_loader = torch.utils.data.DataLoader(
        dataset, batch_size=2, shuffle=True, num_workers=4,
        collate_fn=utils.collate_fn)

    data_loader_test = torch.utils.data.DataLoader(
        dataset_test, batch_size=1, shuffle=False, num_workers=4,
        collate_fn=utils.collate_fn)

    # 使用我们的辅助函数获取模型
    model = get_model_instance_segmentation(num_classes)

    # 将我们的模型迁移到合适的设备
    model.to(device)

    # 构造一个优化器
    params = [p for p in model.parameters() if p.requires_grad]
    optimizer = torch.optim.SGD(params, lr=0.005,
                                momentum=0.9, weight_decay=0.0005)
    # 和学习率调度程序
    lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                                   step_size=3,
                                                   gamma=0.1)

    # 训练10个epochs
    num_epochs = 10

    for epoch in range(num_epochs):
        # 训练一个epoch，每10次迭代打印一次
        train_one_epoch(model, optimizer, data_loader, device, epoch,
print_freq=10)
        # 更新学习速率
        lr_scheduler.step()
        # 在测试集上评价
        evaluate(model, data_loader_test, device=device)

    print("That's it!")
```

在第一个epoch训练后可以得到下面的结果：

```
Epoch: [0]  [ 0/60]  eta: 0:01:18  lr: 0.000090  loss: 2.5213 (2.5213)
loss_classifier: 0.8025 (0.8025)  loss_box_reg: 0.2634 (0.2634)  loss_mask: 1.4265
(1.4265)  loss_objectness: 0.0190 (0.0190)  loss_rpn_box_reg: 0.0099 (0.0099)
time: 1.3121  data: 0.3024  max mem: 3485
Epoch: [0]  [10/60]  eta: 0:00:20  lr: 0.000936  loss: 1.3007 (1.5313)
loss_classifier: 0.3979 (0.4719)  loss_box_reg: 0.2454 (0.2272)  loss_mask: 0.6089
(0.7953)  loss_objectness: 0.0197 (0.0228)  loss_rpn_box_reg: 0.0121 (0.0141)
time: 0.4198  data: 0.0298  max mem: 5081
Epoch: [0]  [20/60]  eta: 0:00:15  lr: 0.001783  loss: 0.7567 (1.1056)
loss_classifier: 0.2221 (0.3319)  loss_box_reg: 0.2002 (0.2106)  loss_mask: 0.2904
(0.5332)  loss_objectness: 0.0146 (0.0176)  loss_rpn_box_reg: 0.0094 (0.0123)
time: 0.3293  data: 0.0035  max mem: 5081
Epoch: [0]  [30/60]  eta: 0:00:11  lr: 0.002629  loss: 0.4705 (0.8935)
loss_classifier: 0.0991 (0.2517)  loss_box_reg: 0.1578 (0.1957)  loss_mask: 0.1970
(0.4204)  loss_objectness: 0.0061 (0.0140)  loss_rpn_box_reg: 0.0075 (0.0118)
time: 0.3403  data: 0.0044  max mem: 5081
Epoch: [0]  [40/60]  eta: 0:00:07  lr: 0.003476  loss: 0.3901 (0.7568)
loss_classifier: 0.0648 (0.2022)  loss_box_reg: 0.1207 (0.1736)  loss_mask: 0.1705
(0.3585)  loss_objectness: 0.0018 (0.0113)  loss_rpn_box_reg: 0.0075 (0.0112)
time: 0.3407  data: 0.0044  max mem: 5081
Epoch: [0]  [50/60]  eta: 0:00:03  lr: 0.004323  loss: 0.3237 (0.6703)
loss_classifier: 0.0474 (0.1731)  loss_box_reg: 0.1109 (0.1561)  loss_mask: 0.1658
(0.3201)  loss_objectness: 0.0015 (0.0093)  loss_rpn_box_reg: 0.0093 (0.0116)
time: 0.3379  data: 0.0043  max mem: 5081
Epoch: [0]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.2540 (0.6082)
loss_classifier: 0.0309 (0.1526)  loss_box_reg: 0.0463 (0.1405)  loss_mask: 0.1568
(0.2945)  loss_objectness: 0.0012 (0.0083)  loss_rpn_box_reg: 0.0093 (0.0123)
time: 0.3489  data: 0.0042  max mem: 5081
Epoch: [0] Total time: 0:00:21 (0.3570 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:19  model_time: 0.2152 (0.2152)  evaluator_time: 0.0133
(0.0133)  time: 0.4000  data: 0.1701  max mem: 5081
Test:  [49/50]  eta: 0:00:00  model_time: 0.0628 (0.0687)  evaluator_time: 0.0039
(0.0064)  time: 0.0735  data: 0.0022  max mem: 5081
Test: Total time: 0:00:04 (0.0828 s / it)
Averaged stats: model_time: 0.0628 (0.0687)  evaluator_time: 0.0039 (0.0064)
Accumulating evaluation results...
DONE (t=0.01s).
Accumulating evaluation results...
DONE (t=0.01s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.606
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.984
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.780
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.313
```

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.582
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.612
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.270
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.672
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.672
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.650
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.755
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.664
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.704
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.979
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.871
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.325
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.488
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.727
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.316
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.748
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.749
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.650
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.673
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.758
```

因此，在一个epoch训练之后，我们获得了COCO-style mAP为60.6，并且mask mAP为70.4。

经过训练10个epoch后，我得到了以下指标：

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.799
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.969
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.935
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.349
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.592
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.831
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.324
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.844
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.844
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.400
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.777
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.870
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.761
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.969
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.919
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.341
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.464
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.788
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.303
```

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.799
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.799
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.400
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.769
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.818
```

但预测结果如何呢？让我们在数据集中拍摄一张图像并进行验证。

训练的模型预测了此图像中的9个人物，让我们看看其中的几个，由下图可以看到预测效果很好。

5.总结

在本教程中，您学习了如何在自定义数据集上为实例分段模型创建自己的训练管道。为此，您编写了一个 `torch.utils.data.Dataset` 类，它返回图像以及地面实况框和分割掩码。您还利用了在COCO train2017上预训练的Mask R-CNN模型，以便对此新数据集执行传输学习。

有关包含multi-machine / multi-gpu training的更完整示例，请检查 torchvision 存储库中的 `references/detection/train.py` 。

您可以在此处下载本教程的完整源文件。

微调 Torchvision 模型

在本教程中，我们将深入探讨如何对 torchvision 模型进行微调和特征提取，所有这些模型都已经预先在1000类的magenet数据集上训练完成。 本教程将深入介绍如何使用几个现代的CNN架构，并将直观展示如何微调任意的PyTorch模型。由于每个模型架构是有差异的，因此没有 可以在所有场景中使用的微调代码样板。然而，研究人员必须查看现有架构并对每个模型进行自定义调整。

在本文档中，我们将执行两种类型的转移学习：**微调和特征提取**。
在*微调*中，我们从预训练模型开始，更新我们新任务的所有模型参数，实质上是重新训练整个模型。
在*特征提取*中，我们从预训练模型开始，仅更新从中导出预测的最终图层权重。它被称为特征提取，因为我们使用预训练的CNN作为固定 的特征提取器，并且仅改变输出层。
有关迁移学习的更多技术信息，请参阅此处和这里。

通常，这两种迁移学习方法都遵循以下几个步骤：

- 初始化预训练模型
- 重组最后一层，使其具有与新数据集类别数相同的输出数
- 为优化算法定义我们想要在训练期间更新的参数
- 运行训练步骤

1.导入相关包并打印版本号

```
```

```
from __future__ import print_function
from __future__ import division
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy
print("PyTorch Version: ",torch.__version__)
print("Torchvision Version: ",torchvision.__version__)
```

输出结果：

```
PyTorch Version:  1.1.0
Torchvision Version:  0.3.0
```

2.输入

以下为运行时需要更改的所有参数。我们将使用的数据集 hymenoptera_data 可在此处 下载。该数据集包含两类：*蜜蜂*和*蚂蚁*，其结构使得我们可以使用 ImageFolder 数据集，不需要编写我们自己的自定义数据集。 下载数据并设置 `data_dir` 为数据集的根目录。 `model_name` 是您要使用的模型名称，必须从此列表中选择：

```
[resnet, alexnet, vgg, squeezenet, densenet, inception]
```

其他输入如下： `num_classes` 为数据集的类别数， `batch_size` 是训练的 batch 大小，可以根据您机器的计算能力进行调整， `num_epochsis` 是 我们想要运行的训练 epoch 数， `feature_extractis` 是定义我们选择微调还是特征提取的布尔值。如果 `feature_extract = False` ，将微调模型，并更新所有模型参数。如果 `feature_extract = True` ，则仅更新最后一层的参数，其他参数保持不变。

```
# 顶级数据目录。 这里我们假设目录的格式符合ImageFolder结构
data_dir = "./data/hymenoptera_data"

# 从[resnet, alexnet, vgg, squeezenet, densenet, inception]中选择模型
model_name = "squeezenet"
```

```
# 数据集中类别数量
num_classes = 2

# 训练的批量大小（根据您的内存量而变化）
batch_size = 8

# 你要训练的epoch数
num_epochs = 15

# 用于特征提取的标志。 当为False时，我们微调整个模型，
# 当True时我们只更新重新形成的图层参数
feature_extract = True
```

3.辅助函数 在编写调整模型的代码之前，我们先定义一些辅助函数。 #### 3.1 模型训练和验证代码 `train_model` 函数处理给定模型的训练和验证。作为输入，它需要PyTorch模型、数据加载器字典、损失函数、优化器、用于训练和验 证epoch数，以及当模型是初始模型时的布尔标志。 `is_inception` 标志用于容纳 Inception v3 模型，因为该体系结构使用辅助输出， 并且整体模型损失涉及辅助输出和最终输出，如此处所述。 这个函数训练指定数量的epoch,并且在每个epoch之后运行完整的验证步骤。它还跟踪最佳性能的模型（从验证准确率方面），并在训练 结束时返回性能最好的模型。在每个epoch之后，打印训练和验证正确率。 ```buildoutcfg def train_model(model, dataloaders, criterion, optimizer, num_epochs=25, is_inception=False): since = time.time()

```
val_acc_history = []

best_model_wts = copy.deepcopy(model.state_dict())
best_acc = 0.0

for epoch in range(num_epochs):
    print('Epoch {}/{}'.format(epoch, num_epochs - 1))
    print('-' * 10)

    # 每个epoch都有一个训练和验证阶段
    for phase in ['train', 'val']:
        if phase == 'train':
            model.train()  # Set model to training mode
        else:
            model.eval()   # Set model to evaluate mode

        running_loss = 0.0
        running_corrects = 0

        # 迭代数据
        for inputs, labels in dataloaders[phase]:
```

```
            inputs = inputs.to(device)
            labels = labels.to(device)

            # 零参数梯度
            optimizer.zero_grad()

            # 前向
            # 如果只在训练时则跟踪轨迹
            with torch.set_grad_enabled(phase == 'train'):
                # 获取模型输出并计算损失
                # 开始的特殊情况，因为在训练中它有一个辅助输出。
                # 在训练模式下，我们通过将最终输出和辅助输出相加来计算损耗
                # 但在测试中我们只考虑最终输出。
                if is_inception and phase == 'train':
                    # From https://discuss.pytorch.org/t/how-to-optimize-
inception-model-with-auxiliary-classifiers/7958
                    outputs, aux_outputs = model(inputs)
                    loss1 = criterion(outputs, labels)
                    loss2 = criterion(aux_outputs, labels)
                    loss = loss1 + 0.4*loss2
                else:
                    outputs = model(inputs)
                    loss = criterion(outputs, labels)

                _, preds = torch.max(outputs, 1)

                # backward + optimize only if in training phase
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

            # 统计
            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_loss = running_loss / len(dataloaders[phase].dataset)
        epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)

        print('{} Loss: {:.4f} Acc: {:.4f}'.format(phase, epoch_loss, epoch_acc))

        # deep copy the model
        if phase == 'val' and epoch_acc > best_acc:
            best_acc = epoch_acc
            best_model_wts = copy.deepcopy(model.state_dict())
        if phase == 'val':
            val_acc_history.append(epoch_acc)

    print()
```

```
time_elapsed = time.time() - since
print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60,
time_elapsed % 60))
print('Best val Acc: {:4f}'.format(best_acc))

# load best model weights
model.load_state_dict(best_model_wts)
return model, val_acc_history
```

```

### 3.2 设置模型参数的 `.requires_grad` 属性

当我们进行特征提取时，此辅助函数将模型中参数的 .requires_grad 属性设置为False。 默认情况下，当我们加载一个预训练模型时，所有参数都是 `.requires_grad = True` ，如果我们从头开始训练或微调，这种设置就没问题。 但是，如果我们要运行特征提取并且只想为新初始化的层计算梯度，那么我们希望所有其他参数不需要梯度变化。这将在稍后更能理解。 `buildoutcfgdef set_parameter_requires_grad(model, feature_extracting):if feature_extracting:for param in model.parameters():param.requires_grad = False`

## 4.初始化和重塑网络

现在来到最有趣的部分。在这里我们对每个网络进行重塑。请注意，这不是一个自动过程，并且对每个模型都是唯一的。 回想一下，CNN模型的最后一层（通常是FC层）与数据集中的输出类的数量具有相同的节点数。由于所有模型都已在 Imagenet 上预先训练， 因此它们都具有大小为1000的输出层，每个类一个节点。这里的目标是将最后一层重塑为与之前具有相同数量的输入，并且具有与数据集 中的类别数相同的输出数。在以下部分中，我们将讨论如何更改每个模型的体系结构。但首先，有一个关于微调和特征提取之间差异的重要细节。

当进行特征提取时，我们只想更新最后一层的参数，换句话说，我们只想更新我们正在重塑层的参数。因此，我们不需要计算不需要改变 的参数的梯度，因此为了提高效率，我们将其它层的 `.requires_grad` 属性设置为False。这很重要，因为默认情况下，此属性设置为True。 然后，当我们初始化新层时，默认情况下新参数 `.requires_grad = True` ，因此只更新新层的参数。当我们进行微调时，我们可以将所有 `.required_grad` 设置为默认值True。

最后，请注意 `inception_v3` 的输入大小为（299,299），而所有其他模型都输入为（224,224）。
```

4.1 Resnet

论文Deep Residual Learning for Image Recognition介绍了Resnet模型。有几种不同尺寸的变体，包括Resnet18、Resnet34、Resnet50、Resnet101和Resnet152，所有这些模型都可以从torchvision 模型中获得。因为我们的数据集很小，只有两个类，所以我们使用Resnet18。 当我们打印这个模型时，我们看到最后一层是全连接层，如下所示：

```
(fc): Linear(in_features=512, out_features=1000, bias=True)
```

因此，我们必须将 `model.fc` 重新初始化为具有512个输入特征和2个输出特征的线性层：

```
model.fc = nn.Linear(512, num_classes)
```

4.2 Alexnet

Alexnet在论文ImageNet Classification with Deep Convolutional Neural Networks 中被介绍，是ImageNet数据集上第一个非常成功的CNN。当我们打印模型架构时，我们看到模型输出为分类器的第6层:

```
(classifier): Sequential(
    ...
    (6): Linear(in_features=4096, out_features=1000, bias=True)
 )
```

要在我们的数据集中使用这个模型，我们将此图层重新初始化为:

```
model.classifier[6] = nn.Linear(4096,num_classes)
```

4.3 VGG

VGG在论文Very Deep Convolutional Networks for Large-Scale Image Recognition 中被引入。Torchvision 提供了8种不同长度的VGG版本，其中一些版本具有批标准化层。这里我们使用VGG-11进行批标准化。 输出层与Alexnet类似，即

```
(classifier): Sequential(
    ...
    (6): Linear(in_features=4096, out_features=1000, bias=True)
 )
```

因此，我们使用相同的方法来修改输出层

```
model.classifier[6] = nn.Linear(4096,num_classes)
```

4.4 Squeezenet

论文SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size 描述了 Squeeznet 架构，使用了与此处显示的任何其他模型不同的输出结构。Torchvision 的 Squeezenet 有两个版本，我们使用1.0版本。 输出来自1x1卷积层，它是分类器的第一层：

```
(classifier): Sequential(
    (0): Dropout(p=0.5)
    (1): Conv2d(512, 1000, kernel_size=(1, 1), stride=(1, 1))
    (2): ReLU(inplace)
    (3): AvgPool2d(kernel_size=13, stride=1, padding=0)
 )
```

为了修改网络，我们重新初始化Conv2d层，使输出特征图深度为2

```
model.classifier[1] = nn.Conv2d(512, num_classes, kernel_size=(1,1), stride=(1,1))
```

4.5 Densenet

论文Densely Connected Convolutional Networks引入了Densenet模型。Torchvision 有四种 Densenet 变型，但在这里我们只使用 Densenet-121。输出层是一个具有1024个输入特征的线性层：

```
(classifier): Linear(in_features=1024, out_features=1000, bias=True)
```

为了重塑这个网络，我们将分类器的线性层重新初始化为

```
model.classifier = nn.Linear(1024, num_classes)
```

4.6 Inception v3

Inception v3首先在论文Rethinking the Inception Architecture for Computer Vision 中描述。该网络的独特之处在于它在训练时有两个输出层。第二个输出称为辅助输出，包含在网络的 AuxLogits 部分中。主输出是网络末端的线性层。 注意，测试时我们只考虑主输出。加载模型的辅助输出和主输出打印为：

```
(AuxLogits): InceptionAux(
    ...
    (fc): Linear(in_features=768, out_features=1000, bias=True)
 )
 ...
(fc): Linear(in_features=2048, out_features=1000, bias=True)
```

要微调这个模型，我们必须重塑这两个层。可以通过以下方式完成

```
model.AuxLogits.fc = nn.Linear(768, num_classes)
model.fc = nn.Linear(2048, num_classes)
```

请注意，许多模型具有相似的输出结构，但每个模型的处理方式略有不同。另外，请查看重塑网络的模型体系结构，并确保输出特征数与 数据集中的类别数相同。

4.7 重塑代码

```
def initialize_model(model_name, num_classes, feature_extract,
use_pretrained=True):
    # 初始化将在此if语句中设置的这些变量。
    # 每个变量都是模型特定的。
    model_ft = None
    input_size = 0

    if model_name == "resnet":
        """ Resnet18
 """
        model_ft = models.resnet18(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.fc.in_features
        model_ft.fc = nn.Linear(num_ftrs, num_classes)
        input_size = 224

    elif model_name == "alexnet":
        """ Alexnet
 """
        model_ft = models.alexnet(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.classifier[6].in_features
        model_ft.classifier[6] = nn.Linear(num_ftrs,num_classes)
        input_size = 224

    elif model_name == "vgg":
        """ VGG11_bn
 """
```

```
        model_ft = models.vgg11_bn(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.classifier[6].in_features
        model_ft.classifier[6] = nn.Linear(num_ftrs,num_classes)
        input_size = 224

    elif model_name == "squeezenet":
        """ Squeezenet
 """
        model_ft = models.squeezenet1_0(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        model_ft.classifier[1] = nn.Conv2d(512, num_classes, kernel_size=(1,1),
stride=(1,1))
        model_ft.num_classes = num_classes
        input_size = 224

    elif model_name == "densenet":
        """ Densenet
 """
        model_ft = models.densenet121(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.classifier.in_features
        model_ft.classifier = nn.Linear(num_ftrs, num_classes)
        input_size = 224

    elif model_name == "inception":
        """ Inception v3
 Be careful, expects (299,299) sized images and has auxiliary output
 """
        model_ft = models.inception_v3(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        # 处理辅助网络
        num_ftrs = model_ft.AuxLogits.fc.in_features
        model_ft.AuxLogits.fc = nn.Linear(num_ftrs, num_classes)
        # 处理主要网络
        num_ftrs = model_ft.fc.in_features
        model_ft.fc = nn.Linear(num_ftrs,num_classes)
        input_size = 299

    else:
        print("Invalid model name, exiting...")
        exit()

    return model_ft, input_size

# 在这步中初始化模型
model_ft, input_size = initialize_model(model_name, num_classes, feature_extract,
use_pretrained=True)
```

```
# 打印我们刚刚实例化的模型
print(model_ft)
```

- 输出结果

```
SqueezeNet(
  (features): Sequential(
    (0): Conv2d(3, 96, kernel_size=(7, 7), stride=(2, 2))
    (1): ReLU(inplace)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=True)
    (3): Fire(
      (squeeze): Conv2d(96, 16, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace)
      (expand1x1): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace)
      (expand3x3): Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace)
    )
    (4): Fire(
      (squeeze): Conv2d(128, 16, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace)
      (expand1x1): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace)
      (expand3x3): Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace)
    )
    (5): Fire(
      (squeeze): Conv2d(128, 32, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace)
      (expand1x1): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace)
      (expand3x3): Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace)
    )
    (6): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=True)
    (7): Fire(
      (squeeze): Conv2d(256, 32, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace)
      (expand1x1): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace)
      (expand3x3): Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1),
```

```
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace)
    )
    (8): Fire(
      (squeeze): Conv2d(256, 48, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace)
      (expand1x1): Conv2d(48, 192, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace)
      (expand3x3): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace)
    )
    (9): Fire(
      (squeeze): Conv2d(384, 48, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace)
      (expand1x1): Conv2d(48, 192, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace)
      (expand3x3): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace)
    )
    (10): Fire(
      (squeeze): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace)
      (expand1x1): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace)
      (expand3x3): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace)
    )
    (11): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=True)
    (12): Fire(
      (squeeze): Conv2d(512, 64, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace)
      (expand1x1): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace)
      (expand3x3): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace)
    )
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5)
    (1): Conv2d(512, 2, kernel_size=(1, 1), stride=(1, 1))
    (2): ReLU(inplace)
    (3): AdaptiveAvgPool2d(output_size=(1, 1))
  )
)
```

5.加载数据

现在我们知道输入尺寸大小必须是什么，我们可以初始化数据转换，图像数据集和数据加载器。请注意，模型是使用硬编码标准化值进行 预先训练的，如这里所述。

```
# 数据扩充和训练规范化
# 只需验证标准化
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(input_size),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(input_size),
        transforms.CenterCrop(input_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

print("Initializing Datasets and Dataloaders...")

# 创建训练和验证数据集
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
data_transforms[x]) for x in ['train', 'val']}
# 创建训练和验证数据加载器
dataloaders_dict = {x: torch.utils.data.DataLoader(image_datasets[x],
batch_size=batch_size, shuffle=True, num_workers=4) for x in ['train', 'val']}

# 检测我们是否有可用的GPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

- 输出结果

```
Initializing Datasets and Dataloaders...
```

6.创建优化器

现在模型结构是正确的，微调和特征提取的最后一步是创建一个只更新所需参数的优化器。回想一下，在加载预训练模型之后，但在重 塑之前，如果 `feature_extract = True` ，我们手动将所有

参数的 `.requires_grad` 属性设置为False。然后重新初始化默认为 `.requires_grad = True` 的网络层参数。所以现在我们知道应该优化所有具有 `.requires_grad = True` 的参数。接下来，我们列出这些参数并将此列表输入到 SGD 算法构造器。

要验证这一点，可以查看要学习的参数。微调时，此列表应该很长并包含所有模型参数。但是，当进行特征提取时，此列表应该很短并且仅包括重塑层的权重和偏差。

```
# 将模型发送到GPU
model_ft = model_ft.to(device)

# 在此运行中收集要优化/更新的参数。
# 如果我们正在进行微调，我们将更新所有参数。
# 但如果我们正在进行特征提取方法，我们只会更新刚刚初始化的参数，即`requires_grad`的参数为True。
params_to_update = model_ft.parameters()
print("Params to learn:")
if feature_extract:
    params_to_update = []
    for name,param in model_ft.named_parameters():
        if param.requires_grad == True:
            params_to_update.append(param)
            print("\t",name)
else:
    for name,param in model_ft.named_parameters():
        if param.requires_grad == True:
            print("\t",name)

# 观察所有参数都在优化
optimizer_ft = optim.SGD(params_to_update, lr=0.001, momentum=0.9)
```

*输出结果

```
Params to learn:
         classifier.1.weight
         classifier.1.bias
```

7.运行训练和验证

最后一步是为模型设置损失，然后对设定的epoch数运行训练和验证函数。请注意，取决于epoch的数量，此步骤在CPU上可能需要执行一 段时间。此外，默认的学习率对所有模型都不是最佳的，因此为了获得最大精度，有必要分别调整每个模型。

```
# 设置损失函数
criterion = nn.CrossEntropyLoss()

# Train and evaluate
model_ft, hist = train_model(model_ft, dataloaders_dict, criterion, optimizer_ft,
num_epochs=num_epochs, is_inception=(model_name=="inception"))
```

- 输出结果

```
Epoch 0/14
----------
train Loss: 0.5066 Acc: 0.7336
val Loss: 0.3781 Acc: 0.8693

Epoch 1/14
----------
train Loss: 0.3227 Acc: 0.8893
val Loss: 0.3254 Acc: 0.8889

Epoch 2/14
----------
train Loss: 0.2080 Acc: 0.9057
val Loss: 0.3137 Acc: 0.9216

Epoch 3/14
----------
train Loss: 0.2211 Acc: 0.9262
val Loss: 0.3126 Acc: 0.9020

Epoch 4/14
----------
train Loss: 0.1523 Acc: 0.9426
val Loss: 0.3000 Acc: 0.9085

Epoch 5/14
----------
train Loss: 0.1480 Acc: 0.9262
val Loss: 0.3167 Acc: 0.9150

Epoch 6/14
----------
train Loss: 0.1943 Acc: 0.9221
val Loss: 0.3129 Acc: 0.9216

Epoch 7/14
----------
train Loss: 0.1247 Acc: 0.9549
```

```
val Loss: 0.3139 Acc: 0.9150

Epoch 8/14
----------
train Loss: 0.1825 Acc: 0.9098
val Loss: 0.3336 Acc: 0.9150

Epoch 9/14
----------
train Loss: 0.1436 Acc: 0.9303
val Loss: 0.3295 Acc: 0.9281

Epoch 10/14
----------
train Loss: 0.1419 Acc: 0.9303
val Loss: 0.3548 Acc: 0.8889

Epoch 11/14
----------
train Loss: 0.1407 Acc: 0.9549
val Loss: 0.2953 Acc: 0.9216

Epoch 12/14
----------
train Loss: 0.0900 Acc: 0.9713
val Loss: 0.3457 Acc: 0.9216

Epoch 13/14
----------
train Loss: 0.1283 Acc: 0.9467
val Loss: 0.3451 Acc: 0.9281

Epoch 14/14
----------
train Loss: 0.0975 Acc: 0.9508
val Loss: 0.3381 Acc: 0.9281

Training complete in 0m 20s
Best val Acc: 0.928105
```

8.对比从头开始模型

这部分内容出于好奇心理，看看如果我们不使用迁移学习，模型将如何学习。微调与特征提取的性能在很大程度上取决于数据集，但一般而言，两种迁移学习方法相对于从头开始训练模型，在训练时间和总体准确性方面产生了良好的结果。

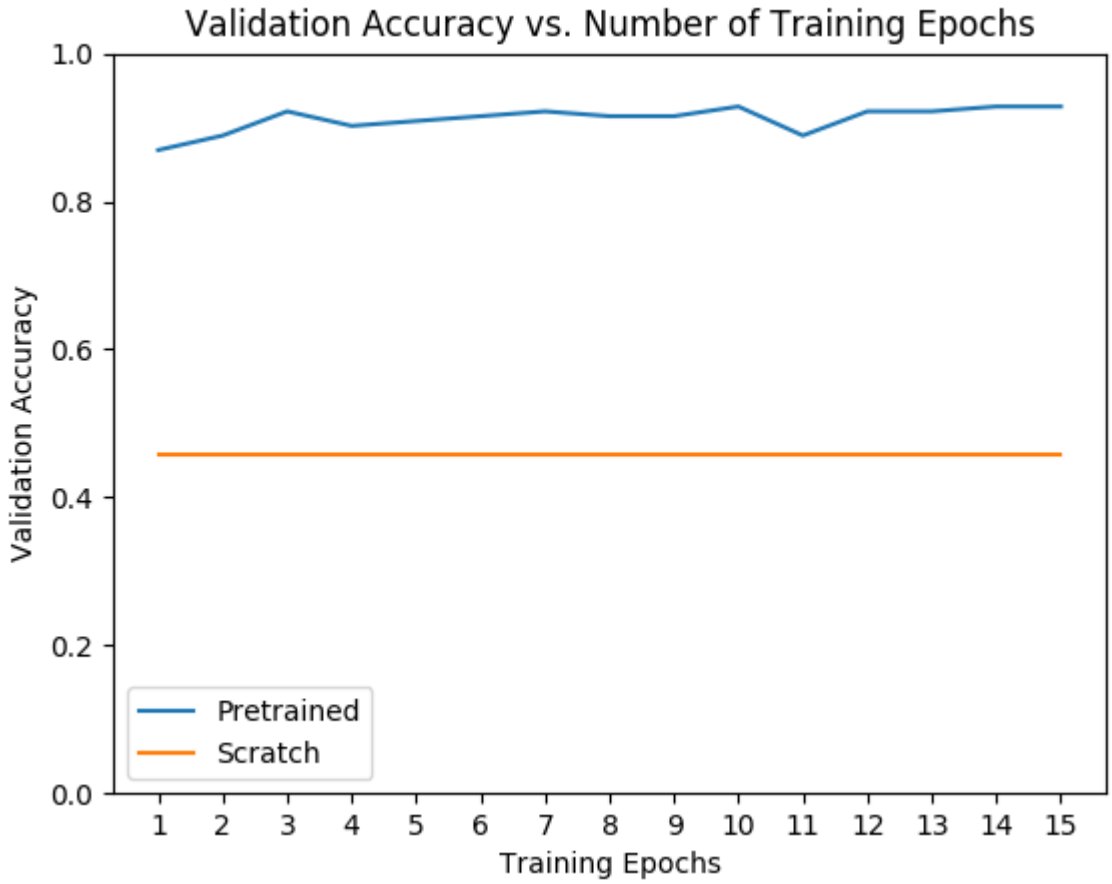
```
# 初始化用于此运行的模型的非预训练版本
scratch_model,_ = initialize_model(model_name, num_classes,
feature_extract=False, use_pretrained=False)
scratch_model = scratch_model.to(device)
scratch_optimizer = optim.SGD(scratch_model.parameters(), lr=0.001, momentum=0.9)
scratch_criterion = nn.CrossEntropyLoss()
_,scratch_hist = train_model(scratch_model, dataloaders_dict, scratch_criterion,
scratch_optimizer, num_epochs=num_epochs, is_inception=(model_name=="inception"))

# 绘制验证精度的训练曲线与转移学习方法
# 和从头开始训练的模型的训练epochs的数量
ohist = []
shist = []

ohist = [h.cpu().numpy() for h in hist]
shist = [h.cpu().numpy() for h in scratch_hist]

plt.title("Validation Accuracy vs. Number of Training Epochs")
plt.xlabel("Training Epochs")
plt.ylabel("Validation Accuracy")
plt.plot(range(1,num_epochs+1),ohist,label="Pretrained")
plt.plot(range(1,num_epochs+1),shist,label="Scratch")
plt.ylim((0,1.))
plt.xticks(np.arange(1, num_epochs+1, 1.0))
plt.legend()
plt.show()
```

- 输出结果




```
Epoch 0/14
----------
train Loss: 0.7131 Acc: 0.4959
val Loss: 0.6931 Acc: 0.4575

Epoch 1/14
----------
train Loss: 0.6930 Acc: 0.5041
val Loss: 0.6931 Acc: 0.4575

Epoch 2/14
----------
train Loss: 0.6932 Acc: 0.5041
val Loss: 0.6931 Acc: 0.4575

Epoch 3/14
----------
train Loss: 0.6932 Acc: 0.5041
val Loss: 0.6931 Acc: 0.4575
```

```
Epoch 4/14
----------
train Loss: 0.6931 Acc: 0.5041
val Loss: 0.6931 Acc: 0.4575

Epoch 5/14
----------
train Loss: 0.6929 Acc: 0.5041
val Loss: 0.6931 Acc: 0.4575

Epoch 6/14
----------
train Loss: 0.6931 Acc: 0.5041
val Loss: 0.6931 Acc: 0.4575

Epoch 7/14
----------
train Loss: 0.6918 Acc: 0.5041
val Loss: 0.6934 Acc: 0.4575

Epoch 8/14
----------
train Loss: 0.6907 Acc: 0.5041
val Loss: 0.6932 Acc: 0.4575

Epoch 9/14
----------
train Loss: 0.6914 Acc: 0.5041
val Loss: 0.6927 Acc: 0.4575

Epoch 10/14
----------
train Loss: 0.6851 Acc: 0.5041
val Loss: 0.6946 Acc: 0.4575

Epoch 11/14
----------
train Loss: 0.6841 Acc: 0.5041
val Loss: 0.6942 Acc: 0.4575

Epoch 12/14
----------
train Loss: 0.6778 Acc: 0.5041
val Loss: 0.7228 Acc: 0.4575

Epoch 13/14
----------
train Loss: 0.6874 Acc: 0.5041
```

```
val Loss: 0.6931 Acc: 0.4575

Epoch 14/14
----------
train Loss: 0.6931 Acc: 0.5041
val Loss: 0.6931 Acc: 0.4575

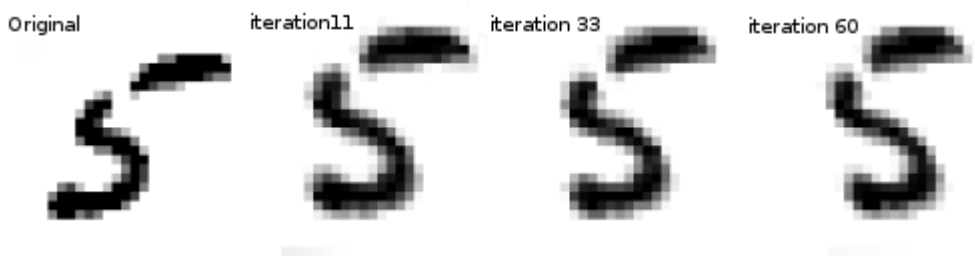
Training complete in 0m 30s
Best val Acc: 0.457516
```

9.总结展望

尝试运行其他模型，看看可以得到多好的正确率。另外，请注意特征提取花费的时间较少，因为在后向传播中我们不需要计算大部分的梯度。 还有很多地方可以尝试。例如：

- 在更难的数据集上运行此代码，查看迁移学习的更多好处。
- 在新的领域（比如NLP，音频等）中，使用此处描述的方法，使用迁移学习更新不同的模型。
- 一旦您对一个模型感到满意，可以将其导出为 ONNX 模型，或使用混合前端跟踪它以获得更快的速度和优化的机会。

空间变换器网络



在本教程中，您将学习如何使用称为空间变换器网络的视觉注意机制来扩充您的网络。你可以在 DeepMind paper 阅读更多有关空间变换器网络的内容。

空间变换器网络是对任何空间变换的差异化关注的概括。空间变换器网络（简称STN）允许神经网络学习如何在输入图像上执行空间变换，以增强模型的几何不变性。例如，它可以裁剪感兴趣的区域，缩放并校正图像的方向。而这可能是一种有用的机制，因为CNN对于旋转和 缩放以及更一般的仿射变换并不是不变的。

关于STN的最棒的事情之一是能够简单地将其插入任何现有的CNN，而且只需很少的修改。

```
from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np

plt.ion()   # 交互模式
```

1.加载数据

在这篇文章中，我们尝试了经典的 MNIST 数据集。使用标准卷积网络增强空间变换器网络。

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 训练数据集
train_loader = torch.utils.data.DataLoader(
```

```
    datasets.MNIST(root='.', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])), batch_size=64, shuffle=True, num_workers=4)
# 测试数据集
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(root='.', train=False, transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])), batch_size=64, shuffle=True, num_workers=4)
```

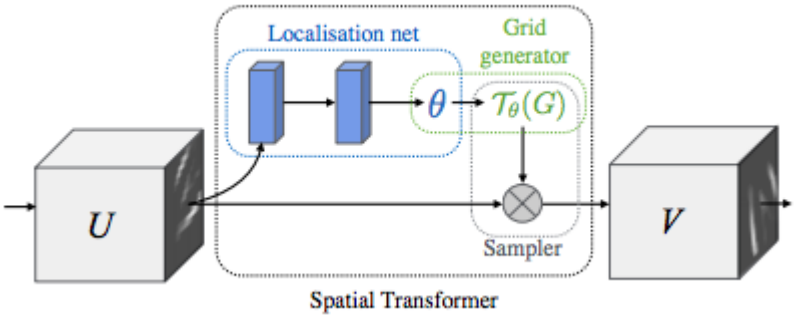
- 输出结果

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./
MNIST/raw/train-images-idx3-ubyte.gz
Extracting ./MNIST/raw/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./
MNIST/raw/train-labels-idx1-ubyte.gz
Extracting ./MNIST/raw/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./
MNIST/raw/t10k-images-idx3-ubyte.gz
Extracting ./MNIST/raw/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./
MNIST/raw/t10k-labels-idx1-ubyte.gz
Extracting ./MNIST/raw/t10k-labels-idx1-ubyte.gz
Processing...
Done!
```

2.什么是空间变换器网络？

空间变换器网络归结为三个主要组成部分：

- 本地网络（Localisation Network）是常规CNN，其对变换参数进行回归。不会从该数据集中明确地学习转换，而是网络自动学习增强 全局准确性的空间变换。
- 网格生成器(Grid Genator)在输入图像中生成与输出图像中的每个像素相对应的坐标网格。
- 采样器（Sampler）使用变换的参数并将其应用于输入图像。



注意：

我们使用最新版本的Pytorch，它应该包含affine_grid和grid_sample模块。

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

        # 空间变换器定位 - 网络
        self.localization = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=7),
            nn.MaxPool2d(2, stride=2),
            nn.ReLU(True),
            nn.Conv2d(8, 10, kernel_size=5),
            nn.MaxPool2d(2, stride=2),
            nn.ReLU(True)
        )

        # 3 * 2 affine矩阵的回归量
        self.fc_loc = nn.Sequential(
            nn.Linear(10 * 3 * 3, 32),
            nn.ReLU(True),
            nn.Linear(32, 3 * 2)
        )

        # 使用身份转换初始化权重/偏差
        self.fc_loc[2].weight.data.zero_()
        self.fc_loc[2].bias.data.copy_(torch.tensor([1, 0, 0, 0, 1, 0],
dtype=torch.float))

    # 空间变换器网络转发功能
    def stn(self, x):
        xs = self.localization(x)
```

```
        xs = xs.view(-1, 10 * 3 * 3)
        theta = self.fc_loc(xs)
        theta = theta.view(-1, 2, 3)

        grid = F.affine_grid(theta, x.size())
        x = F.grid_sample(x, grid)

        return x

    def forward(self, x):
        # transform the input
        x = self.stn(x)

        # 执行一般的前进传递
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)


model = Net().to(device)
```

3.训练模型

训练模型 现在我们使用 SGD（随机梯度下降）算法来训练模型。网络正在以有监督的方式学习分类任务。同时，该模型以端到端的方式自动学习STN。

```
optimizer = optim.SGD(model.parameters(), lr=0.01)

def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 500 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

```
#
# 一种简单的测试程序，用于测量STN在MNIST上的性能。.
#

def test():
    with torch.no_grad():
        model.eval()
        test_loss = 0
        correct = 0
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)

            # 累加批量损失
            test_loss += F.nll_loss(output, target, size_average=False).item()
            # 获取最大对数概率的索引
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()

        test_loss /= len(test_loader.dataset)
        print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'
              .format(test_loss, correct, len(test_loader.dataset),
                      100. * correct / len(test_loader.dataset)))
```

4.可视化 STN 结果

现在，我们将检查我们学习的视觉注意机制的结果。

我们定义了一个小辅助函数，以便在训练时可视化变换。

```
def convert_image_np(inp):
    """Convert a Tensor to numpy image."""
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    return inp

# 我们想要在训练之后可视化空间变换器层的输出
# 我们使用STN可视化一批输入图像和相应的变换批次。
def visualize_stn():
    with torch.no_grad():
        # Get a batch of training data
        data = next(iter(test_loader))[0].to(device)
```

```
        input_tensor = data.cpu()
        transformed_input_tensor = model.stn(data).cpu()

        in_grid = convert_image_np(
            torchvision.utils.make_grid(input_tensor))

        out_grid = convert_image_np(
            torchvision.utils.make_grid(transformed_input_tensor))

        # Plot the results side-by-side
        f, axarr = plt.subplots(1, 2)
        axarr[0].imshow(in_grid)
        axarr[0].set_title('Dataset Images')

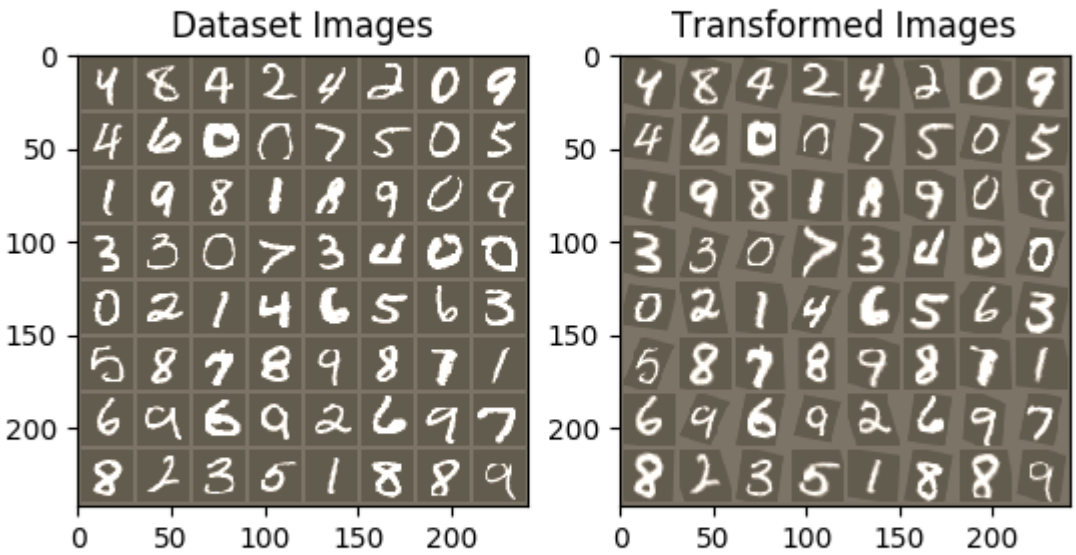
        axarr[1].imshow(out_grid)
        axarr[1].set_title('Transformed Images')

for epoch in range(1, 20 + 1):
    train(epoch)
    test()

# 在某些输入批处理上可视化STN转换
visualize_stn()

plt.ioff()
plt.show()
```

- 输出结果




```
Train Epoch: 1 [0/60000 (0%)]   Loss: 2.336866
Train Epoch: 1 [32000/60000 (53%)]      Loss: 0.841600

Test set: Average loss: 0.2624, Accuracy: 9212/10000 (92%)

Train Epoch: 2 [0/60000 (0%)]   Loss: 0.527656
Train Epoch: 2 [32000/60000 (53%)]      Loss: 0.428908

Test set: Average loss: 0.1176, Accuracy: 9632/10000 (96%)

Train Epoch: 3 [0/60000 (0%)]   Loss: 0.305364
Train Epoch: 3 [32000/60000 (53%)]      Loss: 0.263615

Test set: Average loss: 0.1099, Accuracy: 9677/10000 (97%)

Train Epoch: 4 [0/60000 (0%)]   Loss: 0.169776
Train Epoch: 4 [32000/60000 (53%)]      Loss: 0.408683

Test set: Average loss: 0.0861, Accuracy: 9734/10000 (97%)

Train Epoch: 5 [0/60000 (0%)]   Loss: 0.286635
Train Epoch: 5 [32000/60000 (53%)]      Loss: 0.122162
```

```
Test set: Average loss: 0.0817, Accuracy: 9743/10000 (97%)

Train Epoch: 6 [0/60000 (0%)]   Loss: 0.331074
Train Epoch: 6 [32000/60000 (53%)]      Loss: 0.126413

Test set: Average loss: 0.0589, Accuracy: 9822/10000 (98%)

Train Epoch: 7 [0/60000 (0%)]   Loss: 0.109780
Train Epoch: 7 [32000/60000 (53%)]      Loss: 0.172199

Test set: Average loss: 0.0629, Accuracy: 9814/10000 (98%)

Train Epoch: 8 [0/60000 (0%)]   Loss: 0.078934
Train Epoch: 8 [32000/60000 (53%)]      Loss: 0.156452

Test set: Average loss: 0.0563, Accuracy: 9839/10000 (98%)

Train Epoch: 9 [0/60000 (0%)]   Loss: 0.063500
Train Epoch: 9 [32000/60000 (53%)]      Loss: 0.186023

Test set: Average loss: 0.0713, Accuracy: 9799/10000 (98%)

Train Epoch: 10 [0/60000 (0%)]  Loss: 0.199808
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.083502

Test set: Average loss: 0.0528, Accuracy: 9850/10000 (98%)

Train Epoch: 11 [0/60000 (0%)]  Loss: 0.092909
Train Epoch: 11 [32000/60000 (53%)]     Loss: 0.204410

Test set: Average loss: 0.0471, Accuracy: 9857/10000 (99%)

Train Epoch: 12 [0/60000 (0%)]  Loss: 0.078322
Train Epoch: 12 [32000/60000 (53%)]     Loss: 0.041391

Test set: Average loss: 0.0634, Accuracy: 9796/10000 (98%)

Train Epoch: 13 [0/60000 (0%)]  Loss: 0.061228
Train Epoch: 13 [32000/60000 (53%)]     Loss: 0.137952

Test set: Average loss: 0.0654, Accuracy: 9802/10000 (98%)

Train Epoch: 14 [0/60000 (0%)]  Loss: 0.068635
Train Epoch: 14 [32000/60000 (53%)]     Loss: 0.084583

Test set: Average loss: 0.0515, Accuracy: 9853/10000 (99%)

Train Epoch: 15 [0/60000 (0%)]  Loss: 0.263158
```

```
Train Epoch: 15 [32000/60000 (53%)]     Loss: 0.127036

Test set: Average loss: 0.0493, Accuracy: 9851/10000 (99%)

Train Epoch: 16 [0/60000 (0%)]  Loss: 0.083642
Train Epoch: 16 [32000/60000 (53%)]     Loss: 0.028274

Test set: Average loss: 0.0461, Accuracy: 9867/10000 (99%)

Train Epoch: 17 [0/60000 (0%)]  Loss: 0.076734
Train Epoch: 17 [32000/60000 (53%)]     Loss: 0.034796

Test set: Average loss: 0.0409, Accuracy: 9864/10000 (99%)

Train Epoch: 18 [0/60000 (0%)]  Loss: 0.122501
Train Epoch: 18 [32000/60000 (53%)]     Loss: 0.152187

Test set: Average loss: 0.0474, Accuracy: 9860/10000 (99%)

Train Epoch: 19 [0/60000 (0%)]  Loss: 0.050512
Train Epoch: 19 [32000/60000 (53%)]     Loss: 0.270055

Test set: Average loss: 0.0416, Accuracy: 9878/10000 (99%)

Train Epoch: 20 [0/60000 (0%)]  Loss: 0.073357
Train Epoch: 20 [32000/60000 (53%)]     Loss: 0.017542

Test set: Average loss: 0.0713, Accuracy: 9816/10000 (98%)
```

脚本的总运行时间：1分48.736秒

使用 PyTorch 进行 Neural-Transfer

1.简介

本教程主要讲解如何实现由 Leon A. Gatys，Alexander S. Ecker和Matthias Bethge提出的Neural-Style 算法。Neural-Style 或者叫 Neural-Transfer，可以让你使用一种新的风格将指定的图片进行重构。 这个算法使用三张图片，一张输入图片，一张内容图片和一张风格图片，并将输入的图片变得与内容图片相似，且拥有风格图片的优美风格。

2.基本原理

我们定义两个间距，一个用于内容 `D_C` ，另一个用于风格 `D_S` 。 `D_C` 测量两张图片内容的不同，而 `D_S` 用来测量两张图片风格的不同。 然后，我们输入第三张图片，并改变这张图片，使其与内容图片的内容间距和风格图片的风格间距最小化。 现在，我们可以导入必要的包，开始图像风格转换。

3.导包并选择设备

下面是一张实现图像风格转换所需包的汇总。 * `torch`, `torch.nn`, `numpy` ：使用PyTorch进行风格转换必不可少的包 * `torch.optim` ：高效的梯度下降 * `PIL`, `PIL.Image`, `matplotlib.pyplot` ：加载和展示图片 * `torchvision.transforms` ：将PIL图片转换成张量 * `torchvision.models` ：训练或加载预训练模型 * `copy` ：对模型进行深度拷贝；系统包

```
from __future__ import print_function

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

from PIL import Image
import matplotlib.pyplot as plt

import torchvision.transforms as transforms
import torchvision.models as models

import copy
```

下一步，我们选择用哪一个设备来运行神经网络，导入内容和风格图片。在大量图片上运行图像风格算法需要很长时间，在GPU上运行 可以加速。我们可以使用 `torch.cuda.is_available()` 来判断是否有可用的GPU。下一步，我们在整个教程中使用 `torch.device` ，同时 `torch.device .to(device)` 方法也被用来将张量或者模型移动到指定设备。

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

4.加载图片

现在我们将导入风格和内容图片。原始的PIL图片的值介于0到255之间，但是当转换成torch张量时，它们的值被转换成0到1之间。图片也 需要被重设成相同的维度。一个重要的细节是，注意torch库中的神经网络用来训练的张量的值为0到1之间。如果你尝试将0到255的张量图 片加载到神经网络，然后激活的特征映射将不能侦测到目标内容和风格。然而，Caffe库中的预训练网络用来训练的张量值为0到255之间的图片。

> *注意：*
> 这是一个下载本教程需要用到的图片的链接： `picasso.jpg` 和 `dancing.jpg` 。下载这两张图片并且将它们添加到你当前工作目录中的 `images` 文件夹。

```
# 所需的输出图像大小
imsize = 512 if torch.cuda.is_available() else 128  # use small size if no gpu

loader = transforms.Compose([
    transforms.Resize(imsize),  # scale imported image
    transforms.ToTensor()])  # transform it into a torch tensor

def image_loader(image_name):
    image = Image.open(image_name)
    # fake batch dimension required to fit network's input dimensions
    image = loader(image).unsqueeze(0)
    return image.to(device, torch.float)

style_img = image_loader("./data/images/neural-style/picasso.jpg")
content_img = image_loader("./data/images/neural-style/dancing.jpg")

assert style_img.size() == content_img.size(), \
    "we need to import style and content images of the same size"
现在，让我们创建一个方法，通过重新将图片转换成PIL格式来展示，并使用plt.imshow展示它的拷贝。
我们将尝试展示内容和风格图片来确保它们被正确的导入。

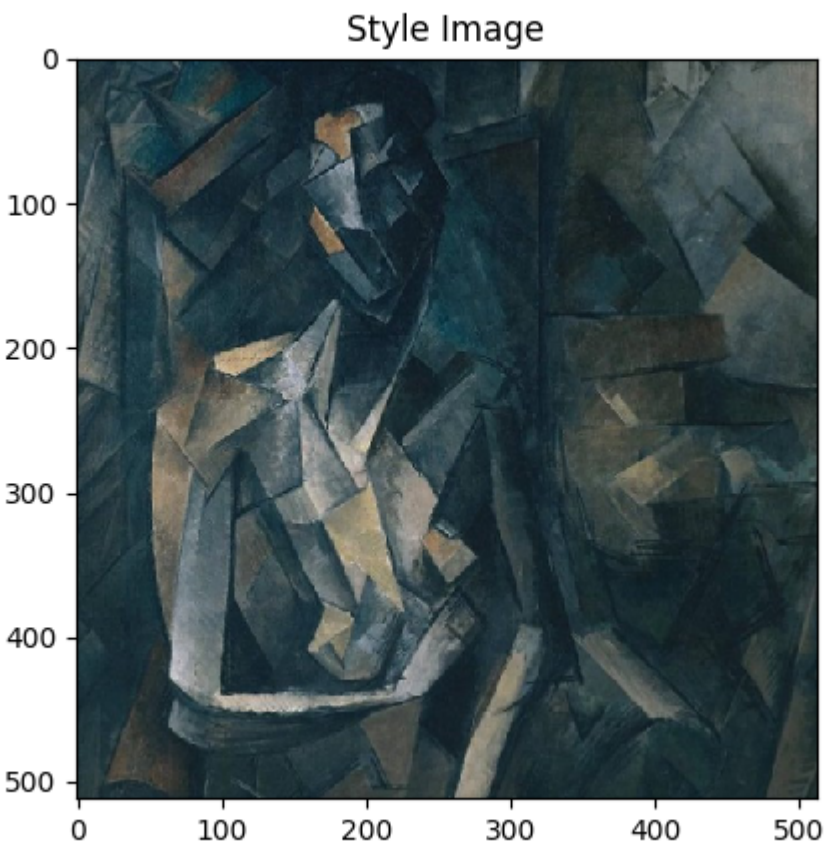
unloader = transforms.ToPILImage()  # reconvert into PIL image
```

```
plt.ion()

def imshow(tensor, title=None):
    image = tensor.cpu().clone()  # we clone the tensor to not do changes on it
    image = image.squeeze(0)      # remove the fake batch dimension
    image = unloader(image)
    plt.imshow(image)
    if title is not None:
        plt.title(title)
    plt.pause(0.001) # pause a bit so that plots are updated

plt.figure()
imshow(style_img, title='Style Image')

plt.figure()
imshow(content_img, title='Content Image')
```



5.损失函数

5.1 内容损失

内容损失是一个表示一层内容间距的加权版本。这个方法使用网络中的L层的特征映射 `F_XL` ，该网络处理输入X并返回在图片X和内容图片 C之间的加权内容间距 `W_CL*D_C^L(X,C)` 。该方法必须知道内容图片 `(F_CL)` 的特征映射来计算内容间距。我们使用一个以 `F_CL` 作为 构造参数输入的 torch 模型来实现这个方法。间距 `||F_XL-F_CL||^2` 是两个特征映射集合之间的平均方差，可以使用 `nn.MSELoss` 来计算。

我们将直接添加这个内容损失模型到被用来计算内容间距的卷积层之后。这样每一次输入图片到网络中时，内容损失都会在目标层被计算。 而且因为自动求导，所有的梯度都会被计算。现在，为了使内容损失层透明化，我们必须定义一个 `forward` 方法来计算内容损失，同时 返回该层的输入。计算的损失作为模型的参数被保存。

```
class ContentLoss(nn.Module):
```

```
    def __init__(self, target,):
        super(ContentLoss, self).__init__()
        # 我们从用于动态计算梯度的树中“分离”目标内容：
        # 这是一个声明的值，而不是变量。
        # 否则标准的正向方法将引发错误。
        self.target = target.detach()

    def forward(self, input):
        self.loss = F.mse_loss(input, self.target)
        return input
```

> *注意：*
> *重要细节*：尽管这个模型的名称被命名为 ContentLoss, 它不是一个真实的PyTorch损失方法。如果你想要定义你的内容损失为PyTorch Loss方法，你必须创建一个PyTorch自动求导方法来手动的在backward方法中重计算/实现梯度.

5.2 风格损失

风格损失模型与内容损失模型的实现方法类似。它要作为一个网络中的透明层，来计算相应层的风格损失。为了计算风格损失，我们需要 计算 Gram 矩阵 `G_XL` 。Gram 矩阵是将给定矩阵和它的转置矩阵的乘积。在这个应用中，给定的矩阵是L层特征映射 `F_XL` 的重塑版本。 `F_XL` 被重塑成 `F○ˆ_XL` ，一个 `KxN` 的矩阵，其中K是L层特征映射的数量，N是任何向量化特征映射F_XL^K的长度。例如，第一行的 `F○ˆ_XL` 与第一个向量化的 `F_XL^1` 。

最后，Gram 矩阵必须通过将每一个元素除以矩阵中所有元素的数量进行标准化。标准化是为了消除拥有很大的N维度 `F○ˆ_XL` 在Gram矩阵 中产生的很大的值。这些很大的值将在梯度下降的时候，对第一层（在池化层之前）产生很大的影响。风格特征往往在网络中更深的层， 所以标准化步骤是很重要的。

```
def gram_matrix(input):
    a, b, c, d = input.size()  # a=batch size(=1)
    # 特征映射 b=number
    # (c,d)=dimensions of a f. map (N=c*d)

    features = input.view(a * b, c * d)  # resise F_XL into \hat F_XL

    G = torch.mm(features, features.t())  # compute the gram product

    # 我们通过除以每个特征映射中的元素数来“标准化”gram矩阵的值.
    return G.div(a * b * c * d)
```

现在风格损失模型看起来和内容损失模型很像。风格间距也用 `G_XL` 和 `G_SL` 之间的均方差来计算。

```
class StyleLoss(nn.Module):

    def __init__(self, target_feature):
        super(StyleLoss, self).__init__()
        self.target = gram_matrix(target_feature).detach()

    def forward(self, input):
        G = gram_matrix(input)
        self.loss = F.mse_loss(G, self.target)
        return input
```

6.导入模型

现在我们需要导入预训练的神经网络。我们将使用19层的 VGG 网络，就像论文中使用的一样。

PyTorch 的 VGG 模型实现被分为了两个字 Sequential 模型：`features`（包含卷积层和池化层）和 `classifier`（包含全连接层）。 我们将使用 `features` 模型，因为我们需要每一层卷积层的输出来计算内容和风格损失。在训练的时候有些层会有和评估不一样的行为，所以我们必须用 `.eval()` 将网络设置成评估模式。

```
cnn = models.vgg19(pretrained=True).features.to(device).eval()
```

此外，VGG网络通过使用mean=[0.485, 0.456, 0.406]和std=[0.229, 0.224, 0.225]参数来标准化图片的每一个通道，并在图片上进行训练。 因此，我们将在把图片输入神经网络之前，先使用这些参数对图片进行标准化。

```
cnn_normalization_mean = torch.tensor([0.485, 0.456, 0.406]).to(device)
cnn_normalization_std = torch.tensor([0.229, 0.224, 0.225]).to(device)

# 创建一个模块来规范化输入图像
# 这样我们就可以轻松地将它放入nn.Sequential中
class Normalization(nn.Module):
    def __init__(self, mean, std):
        super(Normalization, self).__init__()
        # .view the mean and std to make them [C x 1 x 1] so that they can
        # directly work with image Tensor of shape [B x C x H x W].
        # B is batch size. C is number of channels. H is height and W is width.
        self.mean = torch.tensor(mean).view(-1, 1, 1)
        self.std = torch.tensor(std).view(-1, 1, 1)
```

```
    def forward(self, img):
        # normalize img
        return (img - self.mean) / self.std
```

一个 Sequential 模型包含一个顺序排列的子模型序列。例如，`vff19.features` 包含一个以正确的深度顺序排列的序列 `(Conv2d, ReLU,MaxPool2d, Conv2d, ReLU…)` 。我们需要将我们自己的内容损失和风格损失层在感知到卷积层之后立即添加进去。因此，我们必须创建 一个新的Sequential模型，并正确的插入内容损失和风格损失模型。

```
# 期望的深度层来计算样式/内容损失：
content_layers_default = ['conv_4']
style_layers_default = ['conv_1', 'conv_2', 'conv_3', 'conv_4', 'conv_5']

def get_style_model_and_losses(cnn, normalization_mean, normalization_std,
                               style_img, content_img,
                               content_layers=content_layers_default,
                               style_layers=style_layers_default):
    cnn = copy.deepcopy(cnn)

    # 规范化模块
    normalization = Normalization(normalization_mean,
normalization_std).to(device)

    # 只是为了拥有可迭代的访问权限或列出内容/系统损失
    content_losses = []
    style_losses = []

    # 假设cnn是一个`nn.Sequential`，
    # 所以我们创建一个新的`nn.Sequential`来放入应该按顺序激活的模块
    model = nn.Sequential(normalization)

    i = 0  # increment every time we see a conv
    for layer in cnn.children():
        if isinstance(layer, nn.Conv2d):
            i += 1
            name = 'conv_{}'.format(i)
        elif isinstance(layer, nn.ReLU):
            name = 'relu_{}'.format(i)
            # 对于我们在下面插入的`ContentLoss`和`StyleLoss`，
            # 本地版本不能很好地发挥作用。所以我们在这里替换不合适的
            layer = nn.ReLU(inplace=False)
        elif isinstance(layer, nn.MaxPool2d):
            name = 'pool_{}'.format(i)
        elif isinstance(layer, nn.BatchNorm2d):
            name = 'bn_{}'.format(i)
```

```
        else:
            raise RuntimeError('Unrecognized layer:
{}'.format(layer.__class__.__name__))

        model.add_module(name, layer)

        if name in content_layers:
            # 加入内容损失：
            target = model(content_img).detach()
            content_loss = ContentLoss(target)
            model.add_module("content_loss_{}".format(i), content_loss)
            content_losses.append(content_loss)

        if name in style_layers:
            # 加入风格损失：
            target_feature = model(style_img).detach()
            style_loss = StyleLoss(target_feature)
            model.add_module("style_loss_{}".format(i), style_loss)
            style_losses.append(style_loss)

    # 现在我们在最后的内容和风格损失之后剪掉了图层
    for i in range(len(model) - 1, -1, -1):
        if isinstance(model[i], ContentLoss) or isinstance(model[i], StyleLoss):
            break

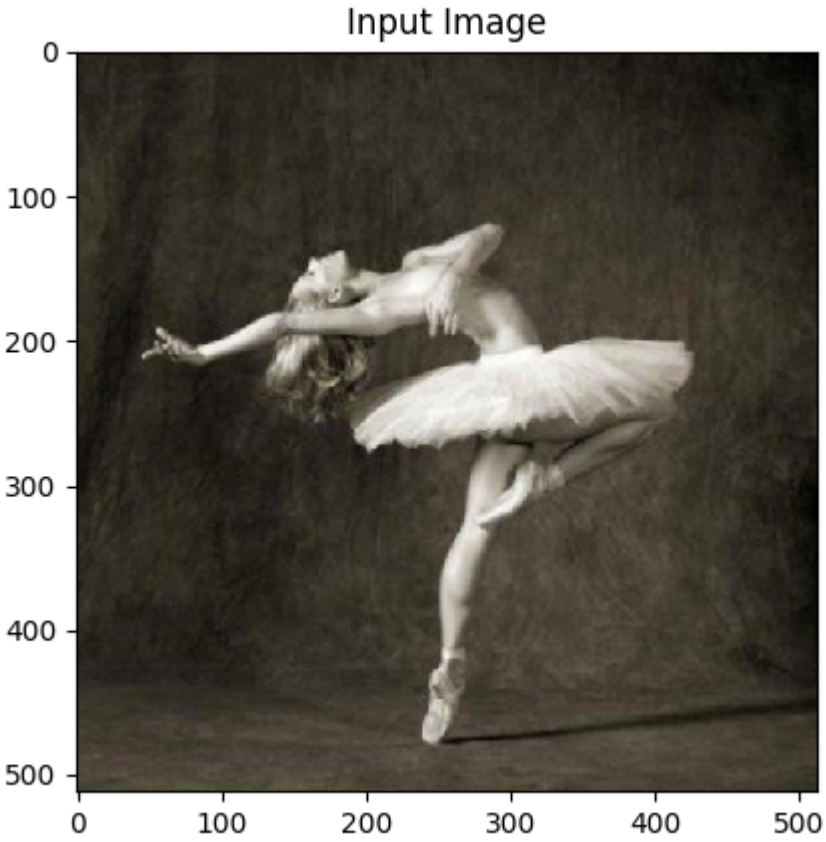
    model = model[:(i + 1)]

    return model, style_losses, content_losses
```

下一步，我们选择输入图片。你可以使用内容图片的副本或者白噪声。

```
input_img = content_img.clone()
# 如果您想使用白噪声而取消注释以下行：
# input_img = torch.randn(content_img.data.size(), device=device)

# 将原始输入图像添加到图中：
plt.figure()
imshow(input_img, title='Input Image')
```



7.梯度下降

和算法的作者 Leon Gatys 的在这里 建议的一样，我们将使用 L-BFGS 算法来进行我们的梯度下降。与训练一般网络不同，我们训练输入图片是为了最小化内容/风格损失。 我们要创建一个 PyTorch 的 L-BFGS 优化器 `optim.LBFGS` ，并传入我们的图片到其中，作为张量去优化。

```
def get_input_optimizer(input_img):
    # 此行显示输入是需要渐变的参数
    optimizer = optim.LBFGS([input_img.requires_grad_()])
    return optimizer
```

最后，我们必须定义一个方法来展示图像风格转换。对于每一次的网络迭代，都将更新过的输入传入其中并计算损失。我们要运行每一个 损失模型的 `backward` 方法来计算它们的梯度。优化器需要一个“关闭”方法，它重新估计模型并且返回损失。

我们还有最后一个问题要解决。神经网络可能会尝试使张量图片的值超过0到1之间来优化输入。我们可以通过在每次网络运行的时候将输 入的值矫正到0到1之间来解决这个问题。

```
def run_style_transfer(cnn, normalization_mean, normalization_std,
                       content_img, style_img, input_img, num_steps=300,
                       style_weight=1000000, content_weight=1):
    """Run the style transfer."""
    print('Building the style transfer model..')
    model, style_losses, content_losses = get_style_model_and_losses(cnn,
        normalization_mean, normalization_std, style_img, content_img)
    optimizer = get_input_optimizer(input_img)

    print('Optimizing..')
    run = [0]
    while run[0] <= num_steps:

        def closure():
            # 更正更新的输入图像的值
            input_img.data.clamp_(0, 1)

            optimizer.zero_grad()
            model(input_img)
            style_score = 0
            content_score = 0

            for sl in style_losses:
                style_score += sl.loss
            for cl in content_losses:
                content_score += cl.loss

            style_score *= style_weight
            content_score *= content_weight

            loss = style_score + content_score
            loss.backward()

            run[0] += 1
            if run[0] % 50 == 0:
                print("run {}:".format(run))
                print('Style Loss : {:4f} Content Loss: {:4f}'.format(
                    style_score.item(), content_score.item()))
                print()

            return style_score + content_score

        optimizer.step(closure)

    # 最后的修正......
    input_img.data.clamp_(0, 1)

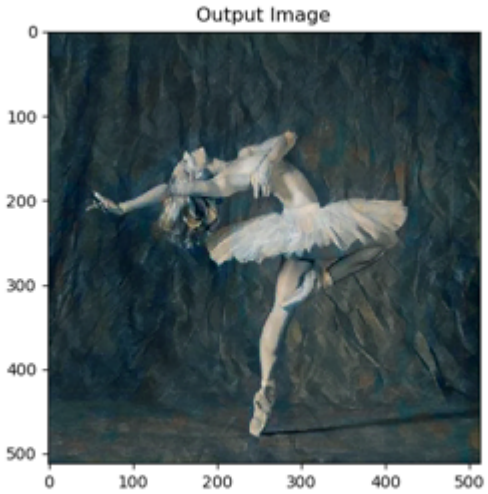
    return input_img
```

最后，我们可以运行这个算法。

```
output = run_style_transfer(cnn, cnn_normalization_mean, cnn_normalization_std,
                            content_img, style_img, input_img)

plt.figure()
imshow(output, title='Output Image')

# sphinx_gallery_thumbnail_number = 4
plt.ioff()
plt.show()
```



- 输出结果

```
Building the style transfer model..
Optimizing..
run [50]:
Style Loss : 4.169304 Content Loss: 4.235329

run [100]:
Style Loss : 1.145476 Content Loss: 3.039176

run [150]:
Style Loss : 0.716769 Content Loss: 2.663749

run [200]:
Style Loss : 0.476047 Content Loss: 2.500893

run [250]:
Style Loss : 0.347092 Content Loss: 2.410895
```

```
run [300]:
Style Loss : 0.263698 Content Loss: 2.358449
```

生成对抗示例

本教程将提高您对ML（机器学习）模型的安全漏洞的认识，并将深入了解对抗性机器学习的热门话题。您可能会惊讶地发现，为图像添加难以察觉的扰动会导致模型性能大不相同。鉴于这是一个教程，我们将通过图像分类器上的示例探讨该主题。具体来说，我们将 使用第一种也是最流行的攻击方法之一，即快速梯度符号攻击算法（FGSM）来迷惑 MNIST 分类器。

1.威胁模型

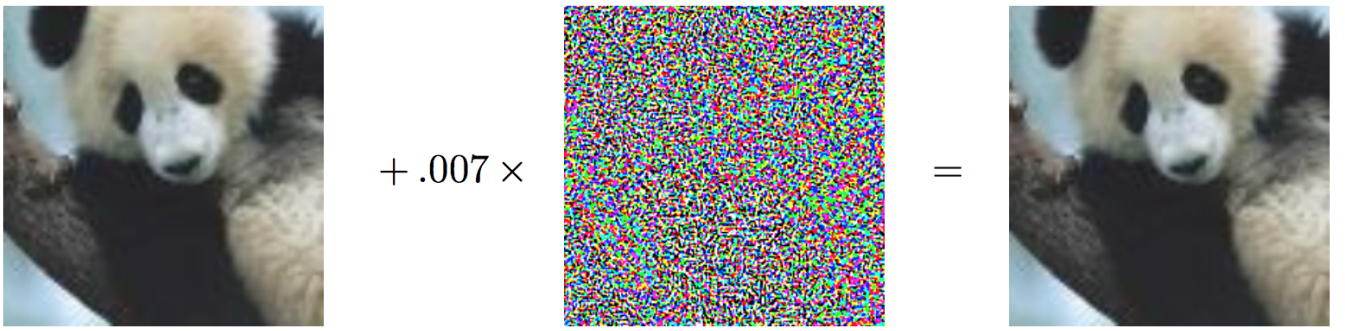
对于上下文，有许多类别的对抗性攻击，每种攻击具有不同的目标和对攻击者知识的假设。然而，通常，总体目标是向输入数据添加最少 量的扰动以引起期望的错误分类。对攻击者的知识有几种假设，其中两种是：白盒子和黑盒子。白盒攻击假定攻击者具有对模型的全部 知识和访问权限，包括体系结构、输入、输出和权重。黑盒攻击假设攻击者只能访问模型的输入和输出，并且对底层架构或权重一无所知。 还有几种类型的目标，包括错误分类和源/目标错误分类。错误分类的目标意味着攻击者只希望输出分类错误，但不关心新分类是什么。 源/目标错误分类意味着攻击者想要更改最初属于特定源类的图像，以便将其归类为特定目标类。

FGSM 攻击是一种白盒攻击，其目标是错误分类。有了这些背景信息，我们现在可以详细讨论攻击。

2.FGSM（Fast Gradient Sign Attack）

快速梯度标志攻击（FGSM），是迄今为止最早和最受欢迎的对抗性攻击之一，它由 Goodfellow 等人在[Explaining and Harnessing Adversarial Examples] (https://arxiv.org/abs/1412.6572)中提出，是一种简单但是有效的对抗样本生成算法。它旨在通过利用模型学习的方式和渐变来攻击神经 网络。这个想法很简单，攻击调整输入数据以基于相同的反向传播梯度来最大化损失，而不是通过基于反向传播的梯度调整权重来最小化损失。 换句话说，攻击是利用损失函数的梯度，然后调整输入数据以最大化损失。

在进入代码之前，先讲一下著名的 FGSM 熊猫示例并提取一些符号。



从图中可以看出，x是正确分类为“熊猫”的原始输入图像，y是x的基本事实标签，θ代表模型参数，$J(\theta, x, y)$是用于训练网络的损失。攻击是反向将梯度传播回输入数据以计算$\nabla_x J(\theta, x, y)$。然后，它在一个方向上（即$sign(\nabla_x J(\theta, x, y))$）调整输入数据（图中的$\epsilon$或0.007），这将使损失最大化。然后，当目标网络仍然明显是“熊猫”时，由此产生的扰动图像被错误地分类为“长臂猿”。

3.实现

在本节中，我们将讨论教程的输入参数，定义被攻击的模型，然后编写攻击代码并运行一些测试。

3.1 引入相关包

```
from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
import numpy as np
import matplotlib.pyplot as plt
```

3.2 输入

本教程只有三个输入，定义如下：* epsilons：用于运行的epsilon值列表。在列表中保留0非常重要，因为它表示原始测试集上的模型性能。而且，我们期望epsilon越大，扰 动就越明显，但就降低模型精度方面而言攻击越有效。由于此处的数据范围为[0,1]，因此epsilon值不应超过1。 * pretrained_model：pytorch/examples/mnist训练的预训练 MNIST 模型的路径。为简单起见，请

在此处 下载预训练模型。 * use_cuda：如果需要和可使用CUDA的布尔标志。注意，带有CUDA的GPU对本教程并不重要，因为本教程使用CPU不会花费太多时间。

```
epsilons = [0, .05, .1, .15, .2, .25, .3]
pretrained_model = "data/lenet_mnist_model.pth"
use_cuda=True
```

3.2 被攻击的模型

如上所述，受攻击的模型与pytorch/examples/mnist中的 MNIST 模型 相同。您可以训练并保存自己的 MNIST 模型，也可以下载并使用提供的模型。此处的 Net 定义和测试数据加载器已从 MNIST 示例中复制。 本小节的目的是定义模型和数据加载器，然后初始化模型并加载预训练的权重。

```
# 定义LeNet模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)

#声明 MNIST 测试数据集何数据加载
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False, download=True,
transform=transforms.Compose([
            transforms.ToTensor(),
            ])),
        batch_size=1, shuffle=True)

# 定义我们正在使用的设备
print("CUDA Available: ",torch.cuda.is_available())
device = torch.device("cuda" if (use_cuda and torch.cuda.is_available()) else
"cpu")
```

```
# 初始化网络
model = Net().to(device)

# 加载已经预训练的模型
model.load_state_dict(torch.load(pretrained_model, map_location='cpu'))

# 在评估模式下设置模型。在这种情况下，这适用于Dropout图层
model.eval()
```

- 输出结果：

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ../
data/MNIST/raw/train-images-idx3-ubyte.gz
Extracting ../data/MNIST/raw/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ../
data/MNIST/raw/train-labels-idx1-ubyte.gz
Extracting ../data/MNIST/raw/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ../
data/MNIST/raw/t10k-images-idx3-ubyte.gz
Extracting ../data/MNIST/raw/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ../
data/MNIST/raw/t10k-labels-idx1-ubyte.gz
Extracting ../data/MNIST/raw/t10k-labels-idx1-ubyte.gz
Processing...
Done!
CUDA Available:  True
```

3.3 FGSM算法攻击

现在，我们可以通过扰乱原始输入来定义创建对抗性示例的函数。`fgsm_attack` 函数有三个输入，$image$是原始的勿扰乱 图像(x)，$epsilon$是像素方式的扰动量(ϵ)，$data_grad$是 输入图像的损失梯度$\nabla_x J(\theta, x, y)$。然后该功能将扰动图像创建为:

$$perturbed_image = image + epsilon * sign(data_grad) = x + \epsilon * sign(\nabla_x J(\theta, x, y))$$

最后，为了保持数据的原始范围，将扰动的图像剪切到范围[0,1]。

```
# FGSM算法攻击代码
def fgsm_attack(image, epsilon, data_grad):
    # 收集数据梯度的元素符号
    sign_data_grad = data_grad.sign()
    # 通过调整输入图像的每个像素来创建扰动图像
    perturbed_image = image + epsilon*sign_data_grad
    # 添加剪切以维持[0,1]范围
    perturbed_image = torch.clamp(perturbed_image, 0, 1)
```

```
    # 返回被扰动的图像
    return perturbed_image
```

3.4 测试函数

最后，本教程的核心结果来自测试功能。每次调用此测试函数都会对 MNIST 测试集执行完整的测试步骤，并报告最终的准确性。但是，请注意，此函数也需要输入$epsilon$。这是因为 `test` 函数展示受到强度为ϵ的攻击下被攻击模型的准确性。 更具体地说，对于测试集中的每个样本，该函数计算输入数据$data_grad$的损失梯度，用 `fgsm_attack (perturbed_data)` 创建扰乱图像，然后检查扰动的例子是否是对抗性的。除了测试模型的准确性之外，该函数还保存并返回一些成功的对抗性示例，以便稍后可视化。

```
def test( model, device, test_loader, epsilon ):

    # 精度计数器
    correct = 0
    adv_examples = []

    # 循环遍历测试集中的所有示例
    for data, target in test_loader:

        # 把数据和标签发送到设备
        data, target = data.to(device), target.to(device)

        # 设置张量的requires_grad属性，这对于攻击很关键
        data.requires_grad = True

        # 通过模型前向传递数据
        output = model(data)
        init_pred = output.max(1, keepdim=True)[1] # get the index of the max log-
probability

        # 如果初始预测是错误的，不打断攻击，继续
        if init_pred.item() != target.item():
            continue

        # 计算损失
        loss = F.nll_loss(output, target)

        # 将所有现有的渐变归零
        model.zero_grad()

        # 计算后向传递模型的梯度
        loss.backward()
```

```
        # 收集datagrad
        data_grad = data.grad.data

        # 唤醒FGSM进行攻击
        perturbed_data = fgsm_attack(data, epsilon, data_grad)

        # 重新分类受扰乱的图像
        output = model(perturbed_data)

        # 检查是否成功
        final_pred = output.max(1, keepdim=True)[1] # get the index of the max
log-probability
        if final_pred.item() == target.item():
            correct += 1
            # 保存0 epsilon示例的特例
            if (epsilon == 0) and (len(adv_examples) < 5):
                adv_ex = perturbed_data.squeeze().detach().cpu().numpy()
                adv_examples.append( (init_pred.item(), final_pred.item(),
adv_ex) )
        else:
            # 稍后保存一些用于可视化的示例
            if len(adv_examples) < 5:
                adv_ex = perturbed_data.squeeze().detach().cpu().numpy()
                adv_examples.append( (init_pred.item(), final_pred.item(),
adv_ex) )

    # 计算这个epsilon的最终准确度
    final_acc = correct/float(len(test_loader))
    print("Epsilon: {}\tTest Accuracy = {} / {} = {}".format(epsilon, correct,
len(test_loader), final_acc))

    # 返回准确性和对抗性示例
    return final_acc, adv_examples
```

3.5 运行攻击

实现的最后一部分是实际运行攻击。在这里，我们为 epsilons 输入中的每个 epsilon 值运行一个完整的测试步骤。 对于每个epsilon，我们还 保存最终的准确性，并在接下来的部分中绘制一些成功的对抗性示例。注意随着 epsilon 值的增加，打印精度会如何降低。另外，请注意 `ε= 0` 的情况表示原始测试精度，没有攻击。

```
accuracies = []
examples = []

# 对每个epsilon运行测试
for eps in epsilons:
```

```
    acc, ex = test(model, device, test_loader, eps)
    accuracies.append(acc)
    examples.append(ex)
```

* 输出结果：

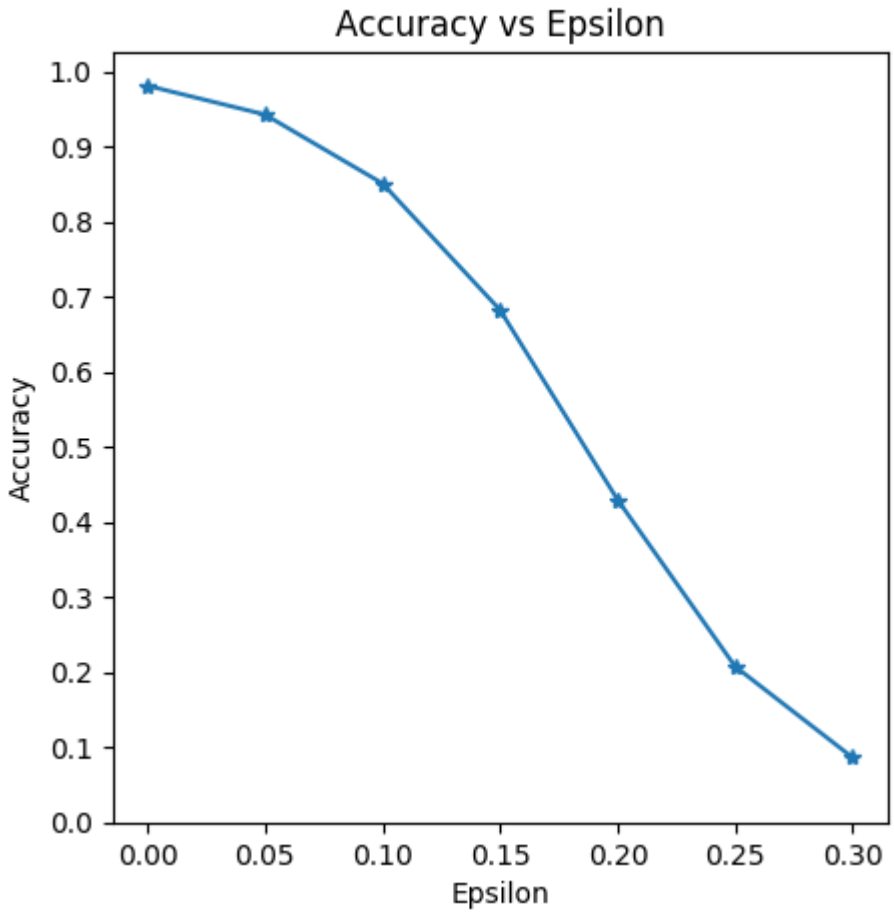
```
Epsilon: 0      Test Accuracy = 9810 / 10000 = 0.981
Epsilon: 0.05   Test Accuracy = 9426 / 10000 = 0.9426
Epsilon: 0.1    Test Accuracy = 8510 / 10000 = 0.851
Epsilon: 0.15   Test Accuracy = 6826 / 10000 = 0.6826
Epsilon: 0.2    Test Accuracy = 4301 / 10000 = 0.4301
Epsilon: 0.25   Test Accuracy = 2082 / 10000 = 0.2082
Epsilon: 0.3    Test Accuracy = 869 / 10000 = 0.0869
```

4.结果

4.1 准确度 vs Epsilon

第一个结果是精度与 epsilon 图。如前所述，随着 epsilon 的增加，我们期望测试精度降低。这是因为较大的 epsilons 意味着我们朝着最 大化损失的方向迈出更大的一步。请注意，即使 epsilon 值线性分布，曲线中的趋势也不是线性的。例如，ε= 0.05 时的精度仅比 ε= 0 低 约 4%，但ε= 0.2 时的精度比 ε= 0.15 低 25%。另外，请注意在 ε= 0.25 和 ε= 0.3 之间模型的准确性达到10级分类器的随机精度。

```
plt.figure(figsize=(5,5))
plt.plot(epsilons, accuracies, "*-")
plt.yticks(np.arange(0, 1.1, step=0.1))
plt.xticks(np.arange(0, .35, step=0.05))
plt.title("Accuracy vs Epsilon")
plt.xlabel("Epsilon")
plt.ylabel("Accuracy")
plt.show()
```

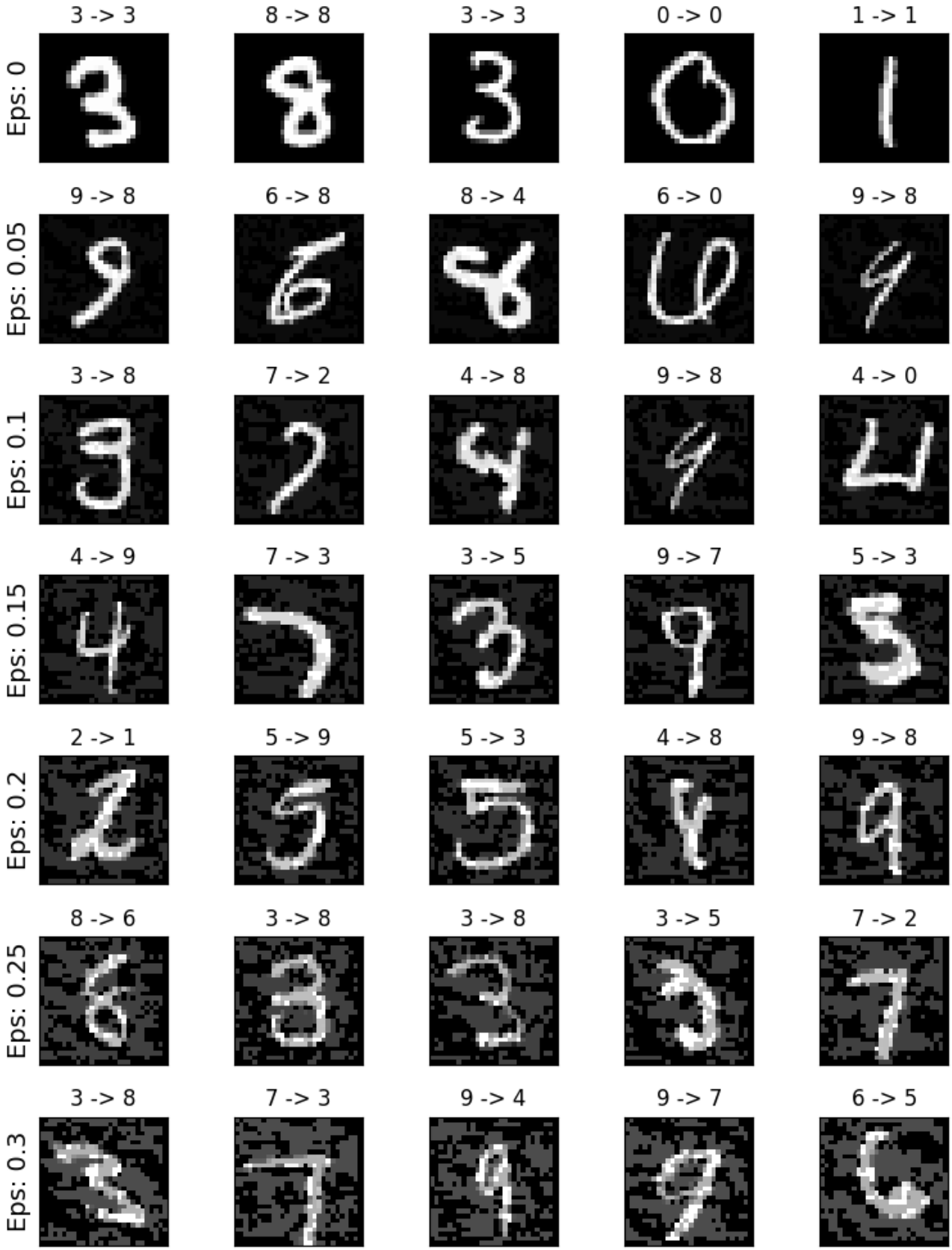


4.2 样本对抗性示例

正如天底下没有免费午餐。在这种情况下，随着 epsilon 增加，测试精度降低，但同时扰动也在变得更容易察觉。实际上，在攻击者必须考虑权衡 准确度降级和可感知性。在这里，我们展示了每个 epsilon 值的成功对抗性示例的一些例子。图的每一行显示不同的 epsilon 值。第一行是 ε= 0 的例子，它们代表没有扰动的原始“干净”图像。每个图像的标题显示“原始分类 - >对抗性分类。”注意，扰动在 ε= 0.15 时开始 变得明显，并且在 ε= 0.3 时非常明显。然而，在所有情况下，尽管增加了噪音，人类仍然能够识别正确的类别。

```
# 在每个epsilon上绘制几个对抗样本的例子
cnt = 0
plt.figure(figsize=(8,10))
for i in range(len(epsilons)):
    for j in range(len(examples[i])):
        cnt += 1
        plt.subplot(len(epsilons),len(examples[0]),cnt)
        plt.xticks([], [])
        plt.yticks([], [])
```

```
        if j == 0:
            plt.ylabel("Eps: {}".format(epsilons[i]), fontsize=14)
        orig,adv,ex = examples[i][j]
        plt.title("{} -> {}".format(orig, adv))
        plt.imshow(ex, cmap="gray")
plt.tight_layout()
plt.show()
```


Eps: 0
3 -> 3
8 -> 8
3 -> 3
0 -> 0
1 -> 1
Eps: 0.05
9 -> 8
6 -> 8
8 -> 4
6 -> 0
9 -> 8
Eps: 0.1
3 -> 8
7 -> 2
4 -> 8
9 -> 8
4 -> 0
Eps: 0.15
4 -> 9
7 -> 3
3 -> 5
9 -> 7
5 -> 3
Eps: 0.2
2 -> 1
5 -> 9
5 -> 3
4 -> 8
9 -> 8
Eps: 0.25
8 -> 6
3 -> 8
3 -> 8
3 -> 5
7 -> 2
Eps: 0.3
3 -> 8
7 -> 3
9 -> 4
9 -> 7
6 -> 5

5.展望

希望通过本教程能够深入了解对抗机器学习。在这里有很多潜在的方向。这次攻击代表了对抗性攻击研究的开始，因为后来有很多关于如何从 对手攻击和防御 ML 模型的想法。事实上，在NIPS 2017上有一场对抗性攻击和防守比赛，文章：[Adversarial Attacks and Defences Competition](https://arxiv.org/pdf/1804.00097.pdf)描述了竞赛中使用的许多方法。防御方面的工作也让我们萌发了使机器学习模型在一般情况下更加 健壮的想法，包括自然扰动和对抗性的输入。

另一个方向是不同领域的对抗性攻击和防御。对抗性研究不仅限于图像领域，对语音到文本模型的攻击可以查看这里。 但也许了解更多关于对抗性机器学习的最好方法就是动起手来。尝试从NIPS 2017竞赛中实施不同的攻击，并了解它与 FGSM 的区别。然后， 尝试从您自己的攻击中保护模型。

使用ONNX将模型转移至Caffe2和移动端

在本教程中，我们将介绍如何使用 ONNX 将 PyTorch 中定义的模型转换为 ONNX 格式，然后将其加载到 Caffe2 中。一旦进入 Caffe2，我们 就可以运行模型来仔细检查它是否正确导出，然后我们展示了如何使用 Caffe2 功能（如移动导出器）在移动设备上执行模型。

在本教程中，您需要安装onnx和Caffe2。 您可以使用 `pip install onnx` 来获取 onnx。

> 注意：本教程需要 PyTorch master 分支，可以按照这里说明进行安装。

1.引入模型

```
# 一些包的导入
import io
import numpy as np

from torch import nn
import torch.utils.model_zoo as model_zoo
import torch.onnx
```

1.1 SuperResolution 模型

超分辨率是一种提高图像、视频分辨率的方法，广泛用于图像处理或视频剪辑。在本教程中，我们将首先使用带有虚拟输入的小型超分辨率模型。

首先，让我们在 PyTorch 中创建一个 `SuperResolution` 模型。这个模型 直接来自 PyTorch 的例子，没有修改：

```
# PyTorch中定义的Super Resolution模型
import torch.nn as nn
import torch.nn.init as init

class SuperResolutionNet(nn.Module):
    def __init__(self, upscale_factor, inplace=False):
        super(SuperResolutionNet, self).__init__()

        self.relu = nn.ReLU(inplace=inplace)
        self.conv1 = nn.Conv2d(1, 64, (5, 5), (1, 1), (2, 2))
        self.conv2 = nn.Conv2d(64, 64, (3, 3), (1, 1), (1, 1))
```

```
        self.conv3 = nn.Conv2d(64, 32, (3, 3), (1, 1), (1, 1))
        self.conv4 = nn.Conv2d(32, upscale_factor ** 2, (3, 3), (1, 1), (1, 1))
        self.pixel_shuffle = nn.PixelShuffle(upscale_factor)

        self._initialize_weights()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.pixel_shuffle(self.conv4(x))
        return x

    def _initialize_weights(self):
        init.orthogonal_(self.conv1.weight, init.calculate_gain('relu'))
        init.orthogonal_(self.conv2.weight, init.calculate_gain('relu'))
        init.orthogonal_(self.conv3.weight, init.calculate_gain('relu'))
        init.orthogonal_(self.conv4.weight)

# 使用上面模型定义，创建super-resolution模型
torch_model = SuperResolutionNet(upscale_factor=3)
```

1.2 训练模型

通常，你现在会训练这个模型; 但是，对于本教程我们将下载一些预先训练的权重。请注意，此模型未经过充分训练来获得良好的准确性，此处 仅用于演示目的。

```
# 加载预先训练好的模型权重
del_url = 'https://s3.amazonaws.com/pytorch/test_data/export/
superres_epoch100-44c6958e.pth'
batch_size = 1    # just a random number

# 使用预训练的权重初始化模型
map_location = lambda storage, loc: storage
if torch.cuda.is_available():
    map_location = None
torch_model.load_state_dict(model_zoo.load_url(model_url,
map_location=map_location))

# 将训练模式设置为falsesince we will only run the forward pass.
torch_model.train(False)
```

1.3 导出模型

在 PyTorch 中通过跟踪工作导出模型。要导出模型，请调用 `torch.onnx._export()` 函数。这将执行模型，记录运算符用于计算输出的轨迹。 因为 `_export` 运行模型，我们需要提供输入张量 `x` 。这个张量的值并不重要; 它可以是图像或随机张量，只要它大小是正确的。

要了解有关PyTorch导出界面的更多详细信息，请查看torch.onnx documentation文档。

```
# 输入模型
x = torch.randn(batch_size, 1, 224, 224, requires_grad=True)

# 导出模型
torch_out = torch.onnx._export(torch_model,             # model being run
                               x,                       # model input (or a tuple
for multiple inputs)
                               "super_resolution.onnx", # where to save the model
(can be a file or file-like object)
                               export_params=True)      # store the trained
parameter weights inside the model file
```

`torch_out` 是执行模型后的输出。通常您可以忽略此输出，但在这里我们将使用它来验证我们导出的模型在Caffe2中运行时是否计算出相同的值。

1.4 采用ONNX表示模型并在Caffe2中使用

现在让我们采用 ONNX 表示并在 Caffe2 中使用它。这部分通常可以在一个单独的进程中或在另一台机器上完成，但我们将在同一个进程中继续， 以便我们可以验证 Caffe2 和 PyTorch 是否为网络计算出相同的值：

```
import onnx
import caffe2.python.onnx.backend as onnx_caffe2_backend

#加载ONNX ModelProto对象。模型是一个标准的Python protobuf对象
model = onnx.load("super_resolution.onnx")

# 为执行模型准备caffe2后端，将ONNX模型转换为可以执行它的Caffe2 NetDef。
# 其他ONNX后端，如CNTK的后端即将推出。
prepared_backend = onnx_caffe2_backend.prepare(model)

# 在Caffe2中运行模型

# 构造从输入名称到Tensor数据的映射。
# 模型图形本身包含输入图像之后所有权重参数的输入。由于权重已经嵌入，我们只需要传递输入图像。
```

```
# 设置第一个输入。
W = {model.graph.input[0].name: x.data.numpy()}

# 运行Caffe2 net:
c2_out = prepared_backend.run(W)[0]

# 验证数字正确性，最多3位小数
np.testing.assert_almost_equal(torch_out.data.cpu().numpy(), c2_out, decimal=3)

print("Exported model has been executed on Caffe2 backend, and the result looks good!")
```

我们应该看到 PyTorch 和 Caffe2 的输出在数字上匹配最多3位小数。作为旁注，如果它们不匹配则存在 Caffe2 和 PyTorch 中的运算符以 不同方式实现的问题，请在这种情况下与我们联系。

2.使用ONNX转换SRResNET

使用与上述相同的过程，我们参考文章中提出的超分辨率转移了一个有趣的新模型“SRResNet”（感谢Twitter上的作者为本教程的目的提供了代码和预训练参数）。可在此处 找到模型定义和预训练模型。下面是 SRResNet 模型的输入、输出。

Original low-resolution image

The same original image zoomed 4 times

Output of SRResNet model

3.在移动设备上运行模型

到目前为止，我们已经从 PyTorch 导出了一个模型，并展示了如何加载它并在 Caffe2 中运行它。现在模型已加载到 Caffe2 中，我们可以 将其转换为适合在移动设备上运行的格式。

我们将使用 Caffe2 的mobile_exporter 生成可在移动设备上运行的两个模型 `protobufs` 。第一个用于使用正确的权重初始化网络，第二个实际运行执行模型。在本教程的其余部分，我们将继续使用小型超分辨率模型。

```
# 从内部表示中提取工作空间和模型原型
c2_workspace = prepared_backend.workspace
c2_model = prepared_backend.predict_net

# 现在导入caffe2的`mobile_exporter`
from caffe2.python.predictor import mobile_exporter

# 调用Export来获取predict_net, init_net。 在移动设备上运行时需要这些网络
init_net, predict_net = mobile_exporter.Export(c2_workspace, c2_model,
c2_model.external_input)

# 我们还将init_net和predict_net保存到我们稍后将用于在移动设备上运行它们的文件中
with open('init_net.pb', "wb") as fopen:
    fopen.write(init_net.SerializeToString())
with open('predict_net.pb', "wb") as fopen:
    fopen.write(predict_net.SerializeToString())
```

`init_net` 具有模型参数和嵌入在其中的模型输入， `predict_net` 将用于指导运行时的 `init_net` 执行。在本教程中，我们将使用上面生成 的 `init_net` 和 `predict_net` ，并在正常的 Caffe2 后端和移动设备中运行它们，并验证两次运行中生成的输出高分辨率猫咪图像是否相同。

在本教程中，我们将使用广泛使用的著名猫咪图像，如下所示：

```
# 一些必备的导入包
from caffe2.proto import caffe2_pb2
from caffe2.python import core, net_drawer, net_printer, visualize, workspace,
utils
```

```
import numpy as np
import os
import subprocess
from PIL import Image
from matplotlib import pyplot
from skimage import io, transform
```

3.1 加载图像并预处理

首先，让我们加载图像，使用标准的 `skimage python` 库对其进行预处理。请注意，此预处理是处理用于训练/测试神经网络的数据的标准做法。

```
# 加载图像
img_in = io.imread("./_static/img/cat.jpg")

# 设置图片分辨率为 224x224
img = transform.resize(img_in, [224, 224])

# 保存好设置的图片作为模型的输入
io.imsave("./_static/img/cat_224x224.jpg", img)
```

3.2 在Caffe2运行并输出

现在，作为下一步，让我们拍摄调整大小的猫图像并在 Caffe2 后端运行超分辨率模型并保存输出图像。这里的图像处理步骤已经从 PyTorch 实 现的超分辨率模型中采用。

```
# 加载设置好的图片并更改为YCbCr的格式
img = Image.open("./_static/img/cat_224x224.jpg")
img_ycbcr = img.convert('YCbCr')
img_y, img_cb, img_cr = img_ycbcr.split()

# 让我们运行上面生成的移动网络，以便正确初始化caffe2工作区
workspace.RunNetOnce(init_net)
workspace.RunNetOnce(predict_net)

# Caffe2有一个很好的net_printer能够检查网络的外观
# 并确定我们的输入和输出blob名称是什么。
print(net_printer.to_string(predict_net))
```

从上面的输出中，我们可以看到输入名为“9”，输出名为“27”（我们将数字作为blob名称有点奇怪，但这是因为跟踪 `JIT` 为模型生成了编 号条目）。

```

```

```
# 现在，让我们传递调整大小的猫图像以供模型处理。
workspace.FeedBlob("9", np.array(img_y)[np.newaxis,
np.newaxis, :, :].astype(np.float32))

# 运行predict_net以获取模型输出
workspace.RunNetOnce(predict_net)

# 现在让我们得到模型输出blob
img_out = workspace.FetchBlob("27")
```

现在，我们将在这里回顾PyTorch实现超分辨率模型的后处理步骤，以构建最终输出图像并保存图像。

```
img_out_y = Image.fromarray(np.uint8((img_out[0, 0]).clip(0, 255)), mode='L')

# 获取输出图像遵循PyTorch实现的后处理步骤
final_img = Image.merge(
    "YCbCr", [
        img_out_y,
        img_cb.resize(img_out_y.size, Image.BICUBIC),
        img_cr.resize(img_out_y.size, Image.BICUBIC),
    ]).convert("RGB")

# 保存图像，我们将其与移动设备的输出图像进行比较
final_img.save("./_static/img/cat_superres.jpg")
```

3.3 在移动端上执行模型

我们已经完成了在纯Caffe2后端运行我们的移动网络，现在，让我们在Android设备上执行该模型并获取模型输出。

> 注意：对于 Android 开发，需要 `adb shell` ，否则教程的以下部分将无法运行。

在我们在移动设备上运行模型的第一步中，我们把基于移动设备的本机速度测试基准二进制文件推送到 adb 。这个二进制文件可以在移动设备 上执行模型，也可以导出我们稍后可以检索的模型输出。二进制文件可在此处 获得。要构建二进制文件，请按照此处的说明执行 `build_android.sh` 脚本。

> 注意：你需要已经安装了 `ANDROID_NDK` ,并且设置环境变量 `ANDROID_NDK=path to ndk root` 。
>
> ```
> # 让我们先把一堆东西推到adb，指定二进制的路径
> CAFFE2_MOBILE_BINARY = ('caffe2/binaries/speed_benchmark')
> ```

```
# 我们已经在上面的步骤中保存了`init_net`和`proto_net`，我们现在使用它们。
# 推送二进制文件和模型protos
os.system('adb push ' + CAFFE2_MOBILE_BINARY + ' /data/local/tmp/')
os.system('adb push init_net.pb /data/local/tmp')
os.system('adb push predict_net.pb /data/local/tmp')

# 让我们将输入图像blob序列化为blob proto，然后将其发送到移动设备以供执行。
with open("input.blobproto", "wb") as fid:
    fid.write(workspace.SerializeBlob("9"))

# 将输入图像blob推送到adb
os.system('adb push input.blobproto /data/local/tmp/')

# 现在我们在移动设备上运行网络，查看`speed_benchmark --help`，了解各种选项的含义
os.system(
    'adb shell /data/local/tmp/speed_benchmark '                    # binary to
execute
    '--init_net=/data/local/tmp/super_resolution_mobile_init.pb '    # mobile
init_net
    '--net=/data/local/tmp/super_resolution_mobile_predict.pb '      # mobile
predict_net
    '--input=9 '                                                    # name of
our input image blob
    '--input_file=/data/local/tmp/input.blobproto '                 #
serialized input image
    '--output_folder=/data/local/tmp '                              #
destination folder for saving mobile output
    '--output=27,9 '                                                # output
blobs we are interested in
    '--iter=1 '                                                     # number of
net iterations to execute
    '--caffe2_log_level=0 '
)

# 从adb获取模型输出并保存到文件
os.system('adb pull /data/local/tmp/27 ./output.blobproto')


# 我们可以使用与之前相同的步骤恢复输出内容并对模型进行后处理
blob_proto = caffe2_pb2.BlobProto()
blob_proto.ParseFromString(open('./output.blobproto').read())
img_out = utils.Caffe2TensorToNumpyArray(blob_proto.tensor)
img_out_y = Image.fromarray(np.uint8((img_out[0,0]).clip(0, 255)), mode='L')
final_img = Image.merge(
    "YCbCr", [
        img_out_y,
        img_cb.resize(img_out_y.size, Image.BICUBIC),
        img_cr.resize(img_out_y.size, Image.BICUBIC),
```

```
    ]).convert("RGB")
final_img.save("./_static/img/cat_superres_mobile.jpg")
```

现在，您可以比较图像 cat_superres.jpg（来自纯caffe2后端执行的模型输出）和 cat_superres_mobile.jpg（来自移动执行的模型输出），并看到两个图像看起来相同。如果它们看起来不一样，那么在移动设备上执行会出现问题，在这种情况下，请联系Caffe2社区。你应该期望看

使用上述步骤，您可以轻松地在移动设备上部署模型。 另外，有关caffe2移动后端的更多信息，请查看caffe2-android-demo。

PyTorch简介

1.Torch张量库介绍

深度学习的所有计算都是在张量上进行的,其中张量是一个可以被超过二维索引的矩阵的一般表示形式。稍后我们将详细讨论这意味着什么。首 先，我们先来看一下我们可以用张量来干什么。

```
# 作者: Robert Guthrie

import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

1.1 创建张量

张量可以在Python list形式下通过 `torch.Tensor()` 函数创建。

```
# 利用给定数据创建一个torch.Tensor对象.这是一个一维向量
V_data = [1., 2., 3.]
V = torch.Tensor(V_data)
print(V)

# 创建一个矩阵
M_data = [[1., 2., 3.], [4., 5., 6]]
M = torch.Tensor(M_data)
print(M)

# 创建2x2x2形式的三维张量.
T_data = [[[1., 2.], [3., 4.]],
          [[5., 6.], [7., 8.]]]
T = torch.Tensor(T_data)
print(T)
```

- 输出结果：

```
tensor([1., 2., 3.])
tensor([[1., 2., 3.],
```

```
        [4., 5., 6.]])
tensor([[[1., 2.],
         [3., 4.]],

        [[5., 6.],
         [7., 8.]]])
```

什么是三维张量？让我们这样想象。如果你有一个向量,那么对这个向量索引就会得到一个标量。如果你有一个矩阵，对这个矩阵索引那么就会 得到一个向量。如果你有一个三维张量，那么对其索引就会得到一个矩阵!

针对术语的说明：当我在本教程内使用“tensor”，它针对的是所有 `torch.Tensor` 对象。矩阵和向量是特殊的 `torch.Tensors` ，他们的维 度分别是1和2。当我说到三维张量，我会简洁的使用“3D tensor”。

```
# 索引V得到一个标量 (0维张量)
print(V[0])

# 从向量V中获取一个数字
print(V[0].item())

# 索引M得到一个向量
print(M[0])

# 索引T得到一个矩阵
print(T[0])
```

- 输出结果：

```
tensor(1.)
1.0
tensor([1., 2., 3.])
tensor([[1., 2.],
        [3., 4.]])
```

你也可以创建其他数据类型的tensors。默认的数据类型为浮点型。可以使用 `torch.LongTensor()` 来创建一个整数类型的张量。你可以在文 件中寻找更多的数据类型，但是浮点型和长整形是最常用的。

你可以使用 `torch.randn()` 创建一个张量。这个张量拥有随机数据和需要指定的维度。

```

```

```
x = torch.randn((3, 4, 5))
print(x)
```

- 输出结果：

```
tensor([[[-1.5256, -0.7502, -0.6540, -1.6095, -0.1002],
         [-0.6092, -0.9798, -1.6091, -0.7121,  0.3037],
         [-0.7773, -0.2515, -0.2223,  1.6871,  0.2284],
         [ 0.4676, -0.6970, -1.1608,  0.6995,  0.1991]],

        [[ 0.8657,  0.2444, -0.6629,  0.8073,  1.1017],
         [-0.1759, -2.2456, -1.4465,  0.0612, -0.6177],
         [-0.7981, -0.1316,  1.8793, -0.0721,  0.1578],
         [-0.7735,  0.1991,  0.0457,  0.1530, -0.4757]],

        [[-0.1110,  0.2927, -0.1578, -0.0288,  0.4533],
         [ 1.1422,  0.2486, -1.7754, -0.0255, -1.0233],
         [-0.5962, -1.0055,  0.4285,  1.4761, -1.7869],
         [ 1.6103, -0.7040, -0.1853, -0.9962, -0.8313]]])
```

1.2 张量操作

你可以以你想要的方式操作张量。

```
x = torch.Tensor([1., 2., 3.])
y = torch.Tensor([4., 5., 6.])
z = x + y
print(z)
```

- 输出结果：

```
tensor([5., 7., 9.])
```

可以查阅文档获取大量可用操作的完整列表,这些操作不仅局限于数学操作范围。

接下来一个很有帮助的操作就是连接。

```
# 默认情况下, 它沿着第一个行进行连接 (连接行)
x_1 = torch.randn(2, 5)
y_1 = torch.randn(3, 5)
z_1 = torch.cat([x_1, y_1])
print(z_1)
```

```
# 连接列：
x_2 = torch.randn(2, 3)
y_2 = torch.randn(2, 5)
# 第二个参数指定了沿着哪条轴连接
z_2 = torch.cat([x_2, y_2], 1)
print(z_2)

# 如果你的tensors是不兼容的，torch会报错。取消注释来查看错误。
# torch.cat([x_1, x_2])
```

- 输出结果：

```
tensor([[-0.8029,  0.2366,  0.2857,  0.6898, -0.6331],
        [ 0.8795, -0.6842,  0.4533,  0.2912, -0.8317],
        [-0.5525,  0.6355, -0.3968, -0.6571, -1.6428],
        [ 0.9803, -0.0421, -0.8206,  0.3133, -1.1352],
        [ 0.3773, -0.2824, -2.5667, -1.4303,  0.5009]])
tensor([[ 0.5438, -0.4057,  1.1341, -0.1473,  0.6272,  1.0935,  0.0939,  1.2381],
        [-1.1115,  0.3501, -0.7703, -1.3459,  0.5119, -0.6933, -0.1668, -0.9999]])
```

1.3 重构张量

使用 `.view()` 去重构张量。这是一个高频方法，因为许多神经网络的神经元对输入格式有明确的要求。你通常需要先将数据重构再输入到神经 元中。

```
x = torch.randn(2, 3, 4)
print(x)
print(x.view(2, 12))  # 重构为2行12列
# 同上。如果维度为-1,那么它的大小可以根据数据推断出来
print(x.view(2, -1))
```

- 输出结果：

```
tensor([[[ 0.4175, -0.2127, -0.8400, -0.4200],
         [-0.6240, -0.9773,  0.8748,  0.9873],
         [-0.0594, -2.4919,  0.2423,  0.2883]],

        [[-0.1095,  0.3126,  1.5038,  0.5038],
         [ 0.6223, -0.4481, -0.2856,  0.3880],
         [-1.1435, -0.6512, -0.1032,  0.6937]]])
tensor([[ 0.4175, -0.2127, -0.8400, -0.4200, -0.6240, -0.9773,  0.8748,  0.9873,
         -0.0594, -2.4919,  0.2423,  0.2883],
        [-0.1095,  0.3126,  1.5038,  0.5038,  0.6223, -0.4481, -0.2856,  0.3880,
         -1.1435, -0.6512, -0.1032,  0.6937]])
```

```
tensor([[ 0.4175, -0.2127, -0.8400, -0.4200, -0.6240, -0.9773,  0.8748,  0.9873,
         -0.0594, -2.4919,  0.2423,  0.2883],
        [-0.1095,  0.3126,  1.5038,  0.5038,  0.6223, -0.4481, -0.2856,  0.3880,
         -1.1435, -0.6512, -0.1032,  0.6937]])
```

2.计算图和自动求导

计算图的思想对于有效率的深度学习编程是很重要的，因为它可以使你不必去自己写反向梯度传播。计算图只是简单地说明了如何将数据组合 在一起以输出结果。因为图完全指定了操作所包含的参数，因此它包含了足够的信息去求导。这可能听起来很模糊，所以让我们看看使用Pytorch 的基本标记（属性）：`requires_grad`。

首先，从程序员的角度来思考。我们在上面刚刚创建的`torch.Tensor`对象中存储了什么？显然，是数据和结构，也很可能是其他的东西。 但是当我们将两个张量相加，我们得到了一个输出张量。这个输出所能体现出的只有数据和结构，并不能体现出是由两个张量加之和得到的（因为它可能是从一个文件中读取的, 也可能是其他操作的结果等）。

如果`requires_grad=True`，张量对象可以一直跟踪它是如何创建的。让我们在实际中来看。

```
# 张量对象带有“requires_grad”标记
x =torch.Tensor([1., 2., 3], requires_grad=True)

# 通过requires_grad=True，您也可以做之前所有的操作。
y = torch.Tensor([4., 5., 6], requires_grad=True)
z = x + y
print(z)

# 但是z还有一些额外的东西.
print(z.grad_fn)
```

- 输出结果：

```
tensor([5., 7., 9.], grad_fn=<AddBackward0>)
<AddBackward0 object at 0x7fd66e2d9cf8>
```

既然变量知道怎么创建的它们。z知道它并非是从文件读取的，也不是乘法或指数或其他运算的结果。如果你继续跟踪`z.grad_fn`，你会从中 找到x和y的痕迹。

但是它如何帮助我们计算梯度?

```
# 我们来将z中所有项作和运算
s = z.sum()
print(s)
print(s.grad_fn)
```

- 输出结果：

```
tensor(21., grad_fn=<SumBackward0>)
<SumBackward0 object at 0x7fd73f1c7e48>
```

那么这个计算和对x的第一个分量的导数等于多少? 在数学上,我们求

$$\frac{\partial s}{\partial x_0}$$

s是被作为张量z的和创建的。张量z是x+y的和，因此

$$s = \overbrace{x_0 + y_0}^{z_0} + \overbrace{x_1 + y_1}^{z_1} + \overbrace{x_2 + y_2}^{z_2}$$

并且s包含了足够的信息去决定我们需要的导数为1!

当然它掩盖了如何计算导数的挑战。这是因为s携带了足够多的信息所以导数可以被计算。现实中，Pytorch 程序的开发人员用程序指令 `sum()` 和 `+` 操作以知道如何计算它们的梯度并且运行反向传播算法。深入讨论此算法超出了本教程的范围.

让我们用Pytorch计算梯度，发现我们是对的:(注意如果你运行这个模块很多次，它的梯度会上升，这是因为Pytorch累积梯度渐变为 `.grad` 属 性，而且对于很多模型它是很方便的.)

```
# 在任意变量上使用 .backward()将会运行反向,从它开始.
s.backward()
print(x.grad)
```

- 输出结果;

```
tensor([1., 1., 1.])
```

作为一个成功的深度学习程序员了解下面的模块如何运行是至关重要的。

```
x = torch.randn((2, 2))
y = torch.randn((2, 2))
#用户创建的张量在默认情况下“requires_grad=False”
print(x.requires_grad, y.requires_grad)
z = x + y
# 你不能通过z反向传播。
print(z.grad_fn)


# “.requires_grad_( ... )”改变了“requires_grad”属性
# 如果没有指定，标记默认为True
x = x.requires_grad_()
y = y.requires_grad_()
#正如我们在上面看到的一样，z包含足够的信息计算梯度。
z = x + y
print(z.grad_fn)
# 如果任何操作的输入部分带有“requires_grad=True”那么输出就会变为：
print(z.requires_grad)

# 现在z有关于x,y的历史信息
# 我们可以获取它的值，将其从历史中分离出来吗？
new_z = z.detach()

# new_z有足够的信息反向传播至x和y吗？
# 答案是没有
print(new_z.grad_fn)
# 怎么会这样？ “z.detach()”函数返回了一个与“z”相同存储的张量
#但是没有携带历史的计算信息。
# 它对于自己是如何计算得来的不知道任何事情。
# 从本质上讲，我们已经把这个变量从过去的历史中分离出来了。
```

- 输出结果：

```
False False
None
<AddBackward0 object at 0x7fd66c736470>
True
None
```

您也可以通过 `.requires_grad ``= True by wrapping the code block in ``with torch.no_grad()` :停止跟踪张量的历史记录中的自动 求导。

```
print(x.requires_grad)
print((x ** 2).requires_grad)
```

```
with torch.no_grad():
    print((x ** 2).requires_grad)
```

- 输出结果：

```
True
True
False
```

使用PyTorch进行深度学习

1.深度学习构建模块：仿射变换, 非线性函数以及目标函数

深度学习表现为使用更巧妙的方法将线性函数和非线性函数进行组合。非线性函数的引入使得训练出来的模型更加强大。在本节中，我们将学 习这些核心组件，建立目标函数，并理解模型是如何构建的。

1.1 仿射变换

深度学习的核心组件之一是仿射变换，仿射变换是一个关于矩阵A和向量x，b的$f(x)$函数，如下所示：

$$f(x)$$

对于矩阵A和向量x，b。 这里要学习的参数是A和b。 通常，b被称为偏差项。

PyTorch以及大多数的深度学习框架所做的事情都与传统的线性代数有些不同。它的映射输入是行而不是列。也就是说，下面代码输出的第i行 是输入的第i行进行A变换，并加上偏移项的结果。看下面的例子：

```
# Author: Robert Guthrie

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

torch.manual_seed(1)
```

```
lin = nn.Linear(5, 3)  # maps from R^5 to R^3, parameters A, b
# data is 2x5.  A maps from 5 to 3... can we map "data" under A?
data = torch.randn(2, 5)
print(lin(data))  # yes
```

- 输出结果：

```

```

```
tensor([[ 0.1755, -0.3268, -0.5069],
        [-0.6602,  0.2260,  0.1089]], grad_fn=<AddmmBackward>)
```

1.2 非线性函数

首先，注意以下这个例子，它将解释为什么我们需要非线性函数。假设我们有两个仿射变换$f(x) = Ax + b$和$g(x) = Cx + d$。 那么$f(g(x))$又是什么呢？

$$f(g(x)) = A(Cx + d) + b = ACx + (Ad + b)$$

AC是一个矩阵，$Ad + b$是一个向量，可以看出，两个仿射变换的组合还是一个仿射变换。

由此可以看出，使用以上方法将多个仿射变换组合成的长链式的神经网络，相对于单个仿射变换并没有性能上的提升。

但是如果我们在两个仿射变换之间引入非线性，那么结果就大不一样了，我们可以构建出一个高性能的模型。

最常用的核心的非线性函数有：$tanh(x)$, $\sigma(x)$, $ReLU(x)$。你可能会想："为什么是这些函数？明明有其他更多的非线性函数。"这些函数常用的原因是它们拥有可以容易计算的梯度，而计算梯度是学习的本质。例如:

$$\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x))$$

> 注意：尽管你可能在AI课程的介绍中学习了一些神经网络，在这些神经网络中$\sigma(x)$是默认非线性的，但是通常 在实际使用的过程中都会避开它们。这是因为当参数的绝对值增长时，梯度会很快消失。小梯度意味着很难学习。因此大部分人默认选择 tanh或者ReLU。

```
# 在pytorch中，大多数非线性都在torch.函数中（我们将它导入为F）
# 请注意，非线性通常没有像仿射图那样的参数。
# 也就是说，他们没有在训练期间更新的权重。
data = torch.randn(2, 2)
```

```
print(data)
print(F.relu(data))
```

- 输出结果：

```
tensor([[-0.5404, -2.2102],
        [ 2.1130, -0.0040]])
tensor([[0.0000, 0.0000],
        [2.1130, 0.0000]])
```

1.3 Softmax和概率

Softmax(x)也是一个非线性函数，但它的特殊之处在于，它通常是神经网络的最后一个操作。这是因为它接受实数向量，并且返回一个概率分 布。它的定义如下。设x为实数向量（正、负，无论什么，没有约束）。然后Softmax(x)的第i个分量是：

$$\frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

很明显，输出的是一个概率分布：每一个元素都非负且和为1。

你也可以认为这只是一个对输入的元素进行的求幂运算符，使所有的内容都非负，然后除以规范化常量。

```
# Softmax也在torch.nn.functional中
data = torch.randn(5)
print(data)
print(F.softmax(data, dim=0))
print(F.softmax(data, dim=0).sum())  # 总和为1，因为它是一个分布！
print(F.log_softmax(data, dim=0))  # theres also log_softmax
```

- 输出结果：

```
tensor([ 1.3800, -1.3505,  0.3455,  0.5046,  1.8213])
tensor([0.2948, 0.0192, 0.1048, 0.1228, 0.4584])
tensor(1.)
tensor([-1.2214, -3.9519, -2.2560, -2.0969, -0.7801])
```

1.4 目标函数

目标函数正是神经网络通过训练来最小化的函数（因此，它常常被称作损失函数或者成本函数）。这需要首先选择一个训练数据实例，通过神 经网络运行它并计算输出的损失。然后通过损失函数的导数来更新模型的参数。因此直观来讲，如果它的结果是错误的，而模型完全信任他，那么损失将会很高。反之，当模型信任计算结果而结果正确时，损失会很低。

在你的训练实例中最小化损失函数的目的是使你的网络拥有很好的泛化能力，可以在开发数据集，测试数据集以及实际生产中拥有很小的损失。 损失函数的一个例子是负对数似然损失函数，这个函数经常在多级分类中出现。在监督多级分类中，这意味着训练网络最小化正确输出的负对数概率（等效的于最大化正确输出的对数概率）。

2.优化和训练

那么，我们该怎么计算函数实例的损失函数呢？我们应该做什么呢？我们在之前了解到TensorFlow 中的 Tensor 知道如何计算梯度以及计算 梯度相关的东西。由于我们的损失正是一个Tensor ，因此我们可以使用所有与梯度有关的参数来计算梯度。然后我们可以进行标准梯度更新。 设θ为我们的参数，$L(\theta)$为损失函数，η一个正的学习率。然后,

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta L(\theta)$$

目前，有大量的算法和积极的研究试图做一些除了这种普通的梯度更新以外的事情。许多人尝试去基于训练时发生的事情来改变学习率。但是， 你不需要担心这些特殊的算法到底在干什么，除非你真的很感兴趣。Torch提供了大量的算法在torch.optim包中，且全部都是透明的。在语法 上使用复杂的算法和使用最简单的梯度更新一样简单。但是尝试不同的更新算法和在更新算法中使用不同的参数（例如不同的初始学习率）对 于优化你的网络的性能很重要。通常，仅仅将普通的SGD替换成一个例如*Adam*或者*RMSProp*优化器都可以显著的提升性能。

3.使用PyTorch创建网络组件

在我们继续关注 NLP 之前，让我们先使用PyTorch构建一个只用仿射变换和非线性函数组成的网络示例。我们也将了解如何计算损失函数，使 用PyTorch内置的负对数似然函数，以及通过反向传播更新参数。

所有的网络组件应该继承 `nn.Module` 并覆盖 `forward()` 方法。继承 `nn.Module` 提供给了一些方法给你的组件。例如，它可以跟踪可训练的 参数，你可以通过 `.to(device)`方 法在 CPU 和 GPU 之间交换它们。 `.to(device)` 方法中的 device 可以是CPU设备 `torch.device("cpu")` 或者 CUDA 设备 `torch.device("cuda:0")` 。

让我们写一个神经网络的示例，它接受一些稀疏的BOW(词袋模式)表示，然后输出分布在两个标签上的概率："English"和"Spanish"。这个模型只是一个逻辑回归。

3.1 示例: 基于逻辑回归与词袋模式的文本分类器

我们的模型将会把BOW表示映射成标签上的对数概率。我们为词汇中的每个词指定一个索引。例如，我们所有的词汇是两个单词"hello"和"world"， 用0和1表示。句子"hello hello hello hello"的表示是

```
[4,0]
```

对于"hello world world hello", 则表示成

```
[2,2]
```

通常表示成

```
[Count(hello),Count(world)]
```

用x来表示这个BOW向量。网络的输出是:

$$logSoftmax(Ax + b)$$

也就是说，我们数据传入一个仿射变换然后做对数归一化 `logsoftmax` 。

```
data = [("me gusta comer en la cafeteria".split(), "SPANISH"),
        ("Give it to me".split(), "ENGLISH"),
        ("No creo que sea una buena idea".split(), "SPANISH"),
```

```
        ("No it is not a good idea to get lost at sea".split(), "ENGLISH")]

test_data = [("Yo creo que si".split(), "SPANISH"),
             ("it is lost on me".split(), "ENGLISH")]

# word_to_ix maps each word in the vocab to a unique integer, which will be its
# index into the Bag of words vector
word_to_ix = {}
for sent, _ in data + test_data:
    for word in sent:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)
print(word_to_ix)

VOCAB_SIZE = len(word_to_ix)
NUM_LABELS = 2

class BoWClassifier(nn.Module):  # inheriting from nn.Module!

    def __init__(self, num_labels, vocab_size):
        # calls the init function of nn.Module.  Dont get confused by syntax,
        # just always do it in an nn.Module
        super(BoWClassifier, self).__init__()

        # Define the parameters that you will need.  In this case, we need A and
b,
        # the parameters of the affine mapping.
        # Torch defines nn.Linear(), which provides the affine map.
        # Make sure you understand why the input dimension is vocab_size
        # and the output is num_labels!
        self.linear = nn.Linear(vocab_size, num_labels)

        # NOTE! The non-linearity log softmax does not have parameters! So we
don't need
        # to worry about that here

    def forward(self, bow_vec):
        # Pass the input through the linear layer,
        # then pass that through log_softmax.
        # Many non-linearities and other functions are in torch.nn.functional
        return F.log_softmax(self.linear(bow_vec), dim=1)

def make_bow_vector(sentence, word_to_ix):
    vec = torch.zeros(len(word_to_ix))
    for word in sentence:
        vec[word_to_ix[word]] += 1
    return vec.view(1, -1)

def make_target(label, label_to_ix):
```

```
    return torch.LongTensor([label_to_ix[label]])

model = BoWClassifier(NUM_LABELS, VOCAB_SIZE)

# 模型知道它的参数。 下面的第一个输出是A，第二个输出是b。
# 无论何时将组件分配给模块的__init__函数中的类变量，都是使用self.linear = nn.Linear
 (...) 行完成的。
# 然后通过PyTorch，你的模块（在本例中为BoWClassifier）将存储nn.Linear参数的知识
for param in model.parameters():
    print(param)

# 要运行模型，请传入BoW矢量
# 这里我们不需要训练，所以代码包含在torch.no_grad () 中
with torch.no_grad():
    sample = data[0]
    bow_vector = make_bow_vector(sample[0], word_to_ix)
    log_probs = model(bow_vector)
    print(log_probs)
```

- 输出结果：

```
{'me': 0, 'gusta': 1, 'comer': 2, 'en': 3, 'la': 4, 'cafeteria': 5, 'Give': 6,
'it': 7, 'to': 8, 'No': 9, 'creo': 10, 'que': 11, 'sea': 12, 'una': 13, 'buena':
14, 'idea': 15, 'is': 16, 'not': 17, 'a': 18, 'good': 19, 'get': 20, 'lost': 21,
'at': 22, 'Yo': 23, 'si': 24, 'on': 25}
Parameter containing:
tensor([[ 0.1194,  0.0609, -0.1268,  0.1274,  0.1191,  0.1739, -0.1099, -0.0323,
         -0.0038,  0.0286, -0.1488, -0.1392,  0.1067, -0.0460,  0.0958,  0.0112,
          0.0644,  0.0431,  0.0713,  0.0972, -0.1816,  0.0987, -0.1379, -0.1480,
          0.0119, -0.0334],
        [ 0.1152, -0.1136, -0.1743,  0.1427, -0.0291,  0.1103,  0.0630, -0.1471,
          0.0394,  0.0471, -0.1313, -0.0931,  0.0669,  0.0351, -0.0834, -0.0594,
          0.1796, -0.0363,  0.1106,  0.0849, -0.1268, -0.1668,  0.1882,  0.0102,
          0.1344,  0.0406]], requires_grad=True)
Parameter containing:
tensor([0.0631, 0.1465], requires_grad=True)
tensor([[-0.5378, -0.8771]])
```

上面的哪一个值对应的是 ENGLISH 的对数概率，哪一个是SPANISH的对数概率？我们还没有定义，但是如果我必须要定义我们想要训练的东西。

```
label_to_ix = {"SPANISH": 0, "ENGLISH": 1}
```

让我们来训练吧! 我们将实例传入来获取对数概率，计算损失函数，计算损失函数的梯度，然后使用一个梯度步长来更新参数。在PyTorch的 `nn` 包里提供了损失函数。 `nn.NLLLoss()` 是我们想要的负对数似然损失函数。 `torch.optim` 中也定义了优化方法。这里，我们只使用SGD。

注意，因为 `NLLLoss` 的输入是一个对数概率的向量以及目标标签。它不会为我们计算对数概率。这也是为什么我们最后一层网络是 `log_softmax` 的原因。损失函数 `nn.CrossEntropyLoss()` 除了对结果额外计算了logsoftmax之外，和NLLLoss()没什么区别。

```
# 在我们训练之前运行测试数据，只是为了看到之前-之后
with torch.no_grad():
    for instance, label in test_data:
        bow_vec = make_bow_vector(instance, word_to_ix)
        log_probs = model(bow_vec)
        print(log_probs)

# 打印与“creo”对应的矩阵列
print(next(model.parameters())[:, word_to_ix["creo"]])

loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

# 通常，您希望多次传递训练数据.
# 100比实际数据集大得多，但真实数据集有两个以上的实例。
# 通常，在5到30个epochs之间是合理的。
for epoch in range(100):
    for instance, label in data:
        # 步骤1： 请记住，PyTorch会累积梯度。
        # We need to clear them out before each instance
        model.zero_grad()

        # 步骤2：制作我们的BOW向量，并且我们必须将目标作为整数包装在Tensor中。
        # 例如，如果目标是SPANISH，那么我们包装整数0.
        # 然后，loss函数知道对数概率的第0个元素是对应于SPANISH的对数概率
        bow_vec = make_bow_vector(instance, word_to_ix)
        target = make_target(label, label_to_ix)

        # 步骤3：运行我们的前向传递.
        log_probs = model(bow_vec)

        # 步骤4： 通过调用optimizer.step()来计算损失，梯度和更新参数
        loss = loss_function(log_probs, target)
        loss.backward()
        optimizer.step()

with torch.no_grad():
    for instance, label in test_data:
```

```
        bow_vec = make_bow_vector(instance, word_to_ix)
        log_probs = model(bow_vec)
        print(log_probs)

# 对应西班牙语的指数上升，英语下降！
print(next(model.parameters())[:, word_to_ix["creo"]])
```

- 输出结果：

```
tensor([[-0.9297, -0.5020]])
tensor([[-0.6388, -0.7506]])
tensor([-0.1488, -0.1313], grad_fn=<SelectBackward>)
tensor([[-0.2093, -1.6669]])
tensor([[-2.5330, -0.0828]])
tensor([ 0.2803, -0.5605], grad_fn=<SelectBackward>)
```

我们得到了正确的结果！你可以看到Spanish的对数概率比第一个例子中的高的多，English的对数概率在第二个测试数据中更高，结果也应该 是这样。

现在你了解了如何创建一个PyTorch组件，将数据传入并进行梯度更新。现在我们已经可以开始进行深度学习上的自然语言处理了。

词嵌入：编码形式的词汇语义

词嵌入是一种由真实数字组成的稠密向量，每个向量都代表了单词表里的一个单词。在自然语言处理中，总会遇到这样的情况：特征全是单词！ 但是，如何在电脑上表述一个单词呢？你在电脑上存储的单词的 ASCII 码，但是它仅仅代表单词怎么拼写，没有说明单词的内在含义(你也许 能够从词缀中了解它的词性，或者从大小写中得到一些属性，但仅此而已)。更重要的是，你能把这些 ASCII 码字符组合成什么含义？当V 代表词汇表、输入数据是$|V|$维的情况下，我们往往想从神经网络中得到数据密集的结果，但是结果只有很少的几个维度（例如，预测的数据只有几个 标签 时）。我们如何从大的数据维度空间中得到稍小一点的维度空间？

放弃使用 ASCII 码字符的形式表示单词，换用 one-hot encoding 会怎么样了？好吧，w这个单词就能这样表示：

$$\widehat{y}_i \epsilon T$$

其中，1表示的独有位置，其他位置全是0。其他的词都类似，在另外不一样的位置有一个1代表它，其他位置也都是0。这种表达除了占用巨大 的空间外，还有个很大的缺陷。它只是简单的把词看做一个单独个体，认为它们之间毫无联系。我们真正想要的是能够表达单词之间一些相似 的含义。为什么要这样做呢？来看下面的例子：

假如我们正在搭建一个语言模型，训练数据有下面一些句子：

- The mathematician ran to the store.
- The physicist ran to the store.
- The mathematician solved the open problem.

现在又得到一个没见过的新句子:

- The physicist solved the open problem.

我们的模型可能在这个句子上表现的还不错，但是，如果利用了下面两个事实，模型会表现更佳：

- 我们发现数学家和物理学家在句子里有相同的作用，所以在某种程度上，他们有语义的联系。
- 当看见物理学家在新句子中的作用时，我们发现数学家也有起着相同的作用。

然后我们就推测，物理学家在上面的句子里也类似于数学家吗？这就是我们所指的相似性理念：指的是语义相似，而不是简单的拼写相似。 这就是一种通过连接我们发现的和没发现的一些内容相似点、用于解决语言数据稀疏性的技术。这个例子依赖于一个基本的语言假设：那些在 相似语句中出现的单词，在语义上也是相互关联的。这就叫做distributional hypothesis（分布式假设）。

1. Getting Dense Word Embeddings（密集词嵌入）

我们如何解决这个问题呢？也就是，怎么编码单词中的语义相似性？ 也许我们会想到一些语义属性。 举个例子，我们发现数学家和物理学家 都能跑，所以也许可以给含有“能跑”语义属性的单词打高分，考虑一下其他的属性，想象一下你可能会在这些属性上给普通的单词打什么分。

如果每个属性都表示一个维度，那我们也许可以用一个向量表示一个单词，就像这样：

$$q_{\text{mathematician}} = \begin{bmatrix} \overbrace{2.3}^{\text{can run}}, & \overbrace{9.4}^{\text{likes coffee}}, & \overbrace{-5.5}^{\text{majored in Physics}}, \dots \end{bmatrix}$$

$$q_{\text{physicist}} = \begin{bmatrix} \overbrace{2.5}^{\text{can run}}, & \overbrace{9.1}^{\text{likes coffee}}, & \overbrace{6.4}^{\text{majored in Physics}}, \dots \end{bmatrix}$$

那么，我们就这可以通过下面的方法得到这些单词之间的相似性：

$$\text{Similarity}(\text{physicist}, \text{mathematician}) = q_{\text{physicist}} \cdot q_{\text{mathematician}}$$

尽管通常情况下需要进行长度归一化：

$$\text{Similarity}(\text{physicist}, \text{mathematician}) = \frac{q_{\text{physicist}} \cdot q_{\text{mathematician}}}{\|q_{\text{physicist}}\| \|q_{\text{mathematician}}\|} = \cos(\phi)$$

ϕ是两个向量的夹角。这就意味着，完全相似的单词相似度为1。完全不相似的单词相似度为-1。

你可以把本章开头介绍的 one-hot 稀疏向量看做是我们新定义向量的一种特殊形式，那里的单词相似度为0，现在我们给每个单词一些独特的 语义属性。这些向量数据密集，也就是说它们数字通常都非零。

但是新的这些向量存在一个严重的问题：你可以想到数千种不同的语义属性，它们可能都与决定相似性有关，而且，到底如何设置不同属性的 值呢？深度学习的中心思想是用神经网络来学习特征的表示，而不是程序员去设计它们。所以为什么不把词嵌入只当做模型参数，而是通过训 练来更新呢？这就才是我们要确切做的事。我们将用神经网络做一些潜在语义属性，但是原则上，学习才是关键。 注意，词嵌入可能无法解释。 就是说，尽管使用我们上面手动制作的向量，能够发现数学家和物理学家都喜欢喝咖啡的相似性，如果我们允许神经网络来学习词嵌入，那么 就会发现数学家和物理学家在第二维度有个较大的值，它所代表的含义很不清晰。它们在一些潜在语义上是相似的，但是对我们来说无法解释。

2. Pytorch中的词嵌入

在我们举例或练习之前，这里有一份关于如何在Pytorch和常见的深度学习中使用词嵌入的简要介绍。与制作 one-hot 向量时对每个单词定义 一个特殊的索引类似，当我们使用词向量时也需要为每个单词定义一个索引。这些索引将是查询表的关键点。意思就是，词嵌入被被存储在一 个$|V| \times D$的向量中，其中D是词嵌入的维度。词被被分配的索引 i，表示在向量的第i行存储它的嵌入。 在所有的代码中，从单词到索引的映射是一个叫 word_to_ix 的字典。

能使用词嵌入的模块是 `torch.nn.Embedding` ，这里面有两个参数：词汇表的大小和词嵌入的维度。

索引这张表时，你必须使用 `torch.LongTensor` （因为索引是整数，不是浮点数）。

```
# 作者: Robert Guthrie

import torch
```

```
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

```
word_to_ix = {"hello": 0, "world": 1}
embeds = nn.Embedding(2, 5)  # 2 words in vocab, 5 dimensional embeddings
lookup_tensor = torch.tensor([word_to_ix["hello"]], dtype=torch.long)
hello_embed = embeds(lookup_tensor)
print(hello_embed)
```

- 输出结果：

```
tensor([[ 0.6614,  0.2669,  0.0617,  0.6213, -0.4519]],
       grad_fn=<EmbeddingBackward>)
```

3.例子： N-Gram语言模型

回想一下，在 n-gram 语言模型中,给定一个单词序列向量，我们要计算的是:

$$P(w_i|w_{i-1},w_{i-2},...,w_{i-n+1})$$

w_i是单词序列的第 i 个单词。 在本例中，我们将在训练样例上计算损失函数，并且用反向传播算法更新参数。

```
CONTEXT_SIZE = 2
EMBEDDING_DIM = 10
# 我们用莎士比亚的十四行诗 Sonnet 2
test_sentence = """When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
Thy youth's proud livery so gazed on now,
Will be a totter'd weed of small worth held:
Then being asked, where all thy beauty lies,
Where all the treasure of thy lusty days;
To say, within thine own deep sunken eyes,
Were an all-eating shame, and thriftless praise.
How much more praise deserv'd thy beauty's use,
If thou couldst answer 'This fair child of mine
Shall sum my count, and make my old excuse,'
Proving his beauty by succession thine!
This were to be new made when thou art old,
```

```
And see thy blood warm when thou feel'st it cold.""".split()
# 应该对输入变量进行标记，但暂时忽略。
# 创建一系列的元组，每个元组都是([ word_i-2, word_i-1 ], target word)的形式。
trigrams = [([test_sentence[i], test_sentence[i + 1]], test_sentence[i + 2])
            for i in range(len(test_sentence) - 2)]
# 输出前3行，先看下是什么样子。
print(trigrams[:3])

vocab = set(test_sentence)
word_to_ix = {word: i for i, word in enumerate(vocab)}

class NGramLanguageModeler(nn.Module):

    def __init__(self, vocab_size, embedding_dim, context_size):
        super(NGramLanguageModeler, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(context_size * embedding_dim, 128)
        self.linear2 = nn.Linear(128, vocab_size)

    def forward(self, inputs):
        embeds = self.embeddings(inputs).view((1, -1))
        out = F.relu(self.linear1(embeds))
        out = self.linear2(out)
        log_probs = F.log_softmax(out, dim=1)
        return log_probs

losses = []
loss_function = nn.NLLLoss()
model = NGramLanguageModeler(len(vocab), EMBEDDING_DIM, CONTEXT_SIZE)
optimizer = optim.SGD(model.parameters(), lr=0.001)

for epoch in range(10):
    total_loss = 0
    for context, target in trigrams:

        # 步骤 1\. 准备好进入模型的数据 (例如将单词转换成整数索引,并将其封装在变量中)
        context_idxs = torch.tensor([word_to_ix[w] for w in context],
dtype=torch.long)

        # 步骤 2\. 回调torch累乘梯度
        # 在传入一个新实例之前，需要把旧实例的梯度置零。
        model.zero_grad()

        # 步骤 3\. 继续运行代码，得到单词的log概率值。
        log_probs = model(context_idxs)

        # 步骤 4\. 计算损失函数 (再次注意，Torch需要将目标单词封装在变量里) 。
        loss = loss_function(log_probs, torch.tensor([word_to_ix[target]],
```

```
dtype=torch.long))

        # 步骤 5\. 反向传播更新梯度
        loss.backward()
        optimizer.step()

        # 通过调tensor.item()得到单个Python数值。
        total_loss += loss.item()
    losses.append(total_loss)
print(losses)  # 用训练数据每次迭代，损失函数都会下降。
```

- 输出结果：

```
[(['When', 'forty'], 'winters'), (['forty', 'winters'], 'shall'), (['winters',
'shall'], 'besiege')]
[523.1487259864807, 520.6150465011597, 518.0996162891388, 515.6003141403198,
513.1156675815582, 510.645352602005, 508.1888840198517, 505.74565410614014,
503.314866065979, 500.8949146270752]
```

4.练习：计算连续词袋模型的词向量

连续词袋模型（CBOW）在NLP深度学习中使用很频繁。它是一个模型，尝试通过目标词前后几个单词的文本，来预测目标词。这有别于语言模型，因为CBOW不是序列的，也不必是概率性的。CBOW常用于快速地训练词向量，得到的嵌入用来初始化一些复杂模型的嵌入。通常情况下，这被称 为*预训练嵌入*。它几乎总能帮忙把模型性能提升几个百分点。

CBOW 模型如下所示：给定一个单词w_i，N代表两边的滑窗距，如$w_{i-1},...,w_{i-N}$和$w_{i+1},...,w_{i+N}$，并将所有的上下文词统称为C，CBOW 试图最小化

$$-logp(w_i|C) = -logSoftmax(A(\sum_{w \epsilon C} q_w) + b)$$

其中q_w是单词w_i的嵌入。

在 Pytorch 中，通过填充下面的类来实现这个模型，有两条需要注意：

- 考虑下你需要定义哪些参数。
- 确保你知道每步操作后的结构，如果想重构，请使用 `.view()` 。

```

```

```
CONTEXT_SIZE = 2  # 左右各两个词
raw_text = """We are about to study the idea of a computational process.
Computational processes are abstract beings that inhabit computers.
As they evolve, processes manipulate other abstract things called data.
The evolution of a process is directed by a pattern of rules
called a program. People create programs to direct processes. In effect,
we conjure the spirits of the computer with our spells.""".split()

# 通过对`raw_text`使用set()函数，我们进行去重操作
vocab = set(raw_text)
vocab_size = len(vocab)

word_to_ix = {word: i for i, word in enumerate(vocab)}
data = []
for i in range(2, len(raw_text) - 2):
    context = [raw_text[i - 2], raw_text[i - 1],
               raw_text[i + 1], raw_text[i + 2]]
    target = raw_text[i]
    data.append((context, target))
print(data[:5])

class CBOW(nn.Module):

    def __init__(self):
        pass

    def forward(self, inputs):
        pass

# 创建模型并且训练。这里有些函数帮你在使用模块之前制作数据。

def make_context_vector(context, word_to_ix):
    idxs = [word_to_ix[w] for w in context]
    return torch.tensor(idxs, dtype=torch.long)

make_context_vector(data[0][0], word_to_ix)  # example
```

- 输出结果：

```
[(['We', 'are', 'to', 'study'], 'about'), (['are', 'about', 'study', 'the'],
'to'), (['about', 'to', 'the', 'idea'], 'study'), (['to', 'study', 'idea',
'of'], 'the'), (['study', 'the', 'of', 'a'], 'idea')]
```

序列模型和长短句记忆（LSTM）模型

- 前馈网络

之前我们已经学过了许多的前馈网络。所谓前馈网络, 就是网络中不会保存状态。然而有时这并不是我们想要的效果。在自然语言处理 (NLP, Natural Language Processing) 中, 序列模型是一个核心的概念。

- 序列模型

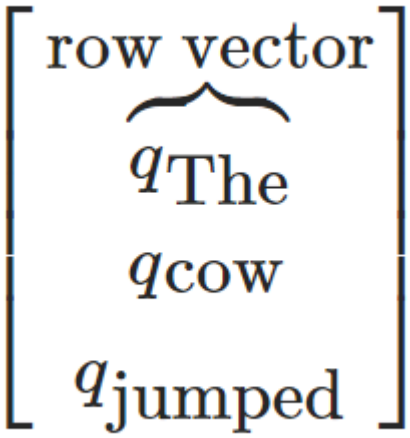
所谓序列模型, 即输入依赖于时间信息的模型。一个典型的序列模型是隐马尔科夫模型 (HMM, Hidden Markov Model)。另一个序列模型的例 子是条件随机场 (CRF, Conditional Random Field)。

- 循环神经网络

循环神经网络是指可以保存某种状态的神经网络。比如说, 神经网络中上个时刻的输出可以作为下个 时刻的输入的一部分, 以此信息就可以 通过序列在网络中一直往后传递。对于LSTM (Long-Short Term Memory) 来说, 序列中的每个元素都有一个相应的隐状态h_t, 该隐状态原则上可以包含序列当前结点之前的任一节点的信息。我们可以使用隐藏状态来预测语言模型中的单词, 词性标签以及其他。

1.Pytorch中的LSTM

在正式学习之前，有几个点要说明一下，Pytorch中 LSTM 的输入形式是一个 3D 的Tensor，每一个维度都有重要的意义，第一个维度就是序列本身， 第二个维度是 `mini-batch` 中实例的索引，第三个维度是输入元素的索引，我们之前没有接触过 `mini-batch` ，所以我们就先忽略它并假设第 二维的维度是1。如果要用”The cow jumped”这个句子来运行一个序列模型，那么就应该把它整理成如下的形式：



除了有一个额外的大小为1的第二维度。

此外, 你还可以向网络逐个输入序列, 在这种情况下, 第一个轴的大小也是1。

来看一个简单的例子。

```
# Author: Robert Guthrie

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

torch.manual_seed(1)
```

```
lstm = nn.LSTM(3, 3)  # 输入维度为3维, 输出维度为3维
inputs = [torch.randn(1, 3) for _ in range(5)]  # 生成一个长度为5的序列

# 初始化隐藏状态.
hidden = (torch.randn(1, 1, 3),
          torch.randn(1, 1, 3))
for i in inputs:
    # 将序列中的元素逐个输入到LSTM.
    # 经过每步操作,hidden 的值包含了隐藏状态的信息.
    out, hidden = lstm(i.view(1, 1, -1), hidden)

# 另外我们可以对一整个序列进行训练.
# LSTM第一个返回的第一个值是所有时刻的隐藏状态
# 第二个返回值是最后一个时刻的隐藏状态
#(所以"out"的最后一个和"hidden"是一样的)
```

```
# 之所以这样设计：
# 通过"out"你能取得任何一个时刻的隐藏状态，而"hidden"的值是用来进行序列的反向传播运算，具体
方式就是将它作为参数传入后面的 LSTM 网络.

# 增加额外的第二个维度.
inputs = torch.cat(inputs).view(len(inputs), 1, -1)
hidden = (torch.randn(1, 1, 3), torch.randn(1, 1, 3))  # 清空隐藏状态.
out, hidden = lstm(inputs, hidden)
print(out)
print(hidden)
```

- 输出结果：

```
tensor([[[-0.0187,  0.1713, -0.2944]],

        [[-0.3521,  0.1026, -0.2971]],

        [[-0.3191,  0.0781, -0.1957]],

        [[-0.1634,  0.0941, -0.1637]],

        [[-0.3368,  0.0959, -0.0538]]], grad_fn=<StackBackward>)
(tensor([[[-0.3368,  0.0959, -0.0538]]], grad_fn=<StackBackward>), tensor([[[-0.
9825,  0.4715, -0.0633]]], grad_fn=<StackBackward>))
```

2.例子:用LSTM来进行词性标注

在这部分, 我们将会使用一个 LSTM 网络来进行词性标注。在这里我们不会用到维特比算法, 前向-后向算法或者任何类似的算法,而是将这部 分内容作为一个 (有挑战) 的练习留给读者, 希望读者在了解了这部分的内容后能够实现如何将维特比算法应用到 LSTM 网络中来。

该模型如下:输入的句子是$w_1, ..., w_M$，其中$w_i \epsilon V$，标签的集合定义为 T，y_i 为单词w_i的标签，用$\hat{y}_i$表示对单词w_i词性的预测。

这是一个结构预测模型, 我们的输出是一个序列$\hat{y}_1, ..., \hat{y}_M$, 其中$\hat{y}_i \epsilon T$。

在进行预测时, 需将句子每个词输入到一个 LSTM 网络中。将时刻 i 的隐藏状态标记为h_i,同样地, 对每个标签赋一个 独一无二的索引 (类似 word embeddings 部分 word_to_ix 的设置). 然后就得到了$\hat{y}_i$的预测规则：

$$\hat{y}_i = argmax_j(logSoftmax(Ah_i + b))_j$$

即先对隐状态进行一个仿射变换, 然后计算一个对数 softmax, 最后得到的预测标签即为对数 softmax 中最大的值对应的标签. 注意, 这也 意味着 A 空间的维度是|T|。

2.1 准备数据

```
def prepare_sequence(seq, to_ix):
    idxs = [to_ix[w] for w in seq]
    return torch.tensor(idxs, dtype=torch.long)

training_data = [
    ("The dog ate the apple".split(), ["DET", "NN", "V", "DET", "NN"]),
    ("Everybody read that book".split(), ["NN", "V", "DET", "NN"])
]
word_to_ix = {}
for sent, tags in training_data:
    for word in sent:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)
print(word_to_ix)
tag_to_ix = {"DET": 0, "NN": 1, "V": 2}

# 实际中通常使用更大的维度如32维, 64维.
# 这里我们使用小的维度, 为了方便查看训练过程中权重的变化.
EMBEDDING_DIM = 6
HIDDEN_DIM = 6
```

- 输出结果：

```
{'The': 0, 'dog': 1, 'ate': 2, 'the': 3, 'apple': 4, 'Everybody': 5, 'read': 6,
'that': 7, 'book': 8}
```

2.1 创建模型

```
class LSTMTagger(nn.Module):

    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super(LSTMTagger, self).__init__()
        self.hidden_dim = hidden_dim

        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)

        # LSTM以word_embeddings作为输入, 输出维度为 hidden_dim 的隐藏状态值
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)

        # 线性层将隐藏状态空间映射到标注空间
```

```
        self.hidden2tag = nn.Linear(hidden_dim, tagset_size)
        self.hidden = self.init_hidden()

    def init_hidden(self):
        # 一开始并没有隐藏状态所以我们要先初始化一个
        # 关于维度为什么这么设计请参考Pytoch相关文档
        # 各个维度的含义是 (num_layers, minibatch_size, hidden_dim)
        return (torch.zeros(1, 1, self.hidden_dim),
                torch.zeros(1, 1, self.hidden_dim))

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        lstm_out, self.hidden = self.lstm(
            embeds.view(len(sentence), 1, -1), self.hidden)
        tag_space = self.hidden2tag(lstm_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_space, dim=1)
        return tag_scores
```

2.3 训练模型

```
model = LSTMTagger(EMBEDDING_DIM, HIDDEN_DIM, len(word_to_ix), len(tag_to_ix))
loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

# 查看训练前的分数
# 注意：输出的 i,j 元素的值表示单词 i 的 j 标签的得分
# 这里我们不需要训练不需要求导，所以使用torch.no_grad()
with torch.no_grad():
    inputs = prepare_sequence(training_data[0][0], word_to_ix)
    tag_scores = model(inputs)
    print(tag_scores)

for epoch in range(300):  # 实际情况下你不会训练300个周期，此例中我们只是随便设了一个值
    for sentence, tags in training_data:
        # 第一步：请记住Pytorch会累加梯度.
        # 我们需要在训练每个实例前清空梯度
        model.zero_grad()

        # 此外还需要清空 LSTM 的隐状态，
        # 将其从上个实例的历史中分离出来.
        model.hidden = model.init_hidden()

        # 准备网络输入，将其变为词索引的 Tensor 类型数据
        sentence_in = prepare_sequence(sentence, word_to_ix)
        targets = prepare_sequence(tags, tag_to_ix)

        # 第三步：前向传播.
```

```
        tag_scores = model(sentence_in)

        # 第四步：计算损失和梯度值，通过调用 optimizer.step() 来更新梯度
        loss = loss_function(tag_scores, targets)
        loss.backward()
        optimizer.step()

# 查看训练后的得分
with torch.no_grad():
    inputs = prepare_sequence(training_data[0][0], word_to_ix)
    tag_scores = model(inputs)

    # 句子是 "the dog ate the apple"，i,j 表示对于单词 i，标签 j 的得分.
    # 我们采用得分最高的标签作为预测的标签．从下面的输出我们可以看到，预测得
    # 到的结果是0 1 2 0 1．因为 索引是从0开始的，因此第一个值0表示第一行的
    # 最大值，第二个值1表示第二行的最大值，以此类推．所以最后的结果是 DET
    # NOUN VERB DET NOUN，整个序列都是正确的!
    print(tag_scores)
```

• 输出结果：

```
tensor([[-1.1389, -1.2024, -0.9693],
        [-1.1065, -1.2200, -0.9834],
        [-1.1286, -1.2093, -0.9726],
        [-1.1190, -1.1960, -0.9916],
        [-1.0137, -1.2642, -1.0366]])
tensor([[-0.0858, -2.9355, -3.5374],
        [-5.2313, -0.0234, -4.0314],
        [-3.9098, -4.1279, -0.0368],
        [-0.0187, -4.7809, -4.5960],
        [-5.8170, -0.0183, -4.1879]])
```

3.练习:使用字符级特征来增强 LSTM 词性标注器

在上面的例子中, 每个词都有一个词嵌入, 作为序列模型的输入. 接下来让我们使用每个的单词的 字符级别的表达来增强词嵌入。我们期望这 个操作对结果能有显著提升, 因为像词缀这样的字符级信息对于词性有很大的影响。比如说, 像包含词缀 -ly 的单词基本上都是被标注为副词。

具体操作如下：用c_w的字符级表达, 同之前一样，我们使用x_w来表示词嵌入。序列模型的输入就变成了x_w和c_w的拼接。因此, 如果x_w的维度是5，c_w 的维度是3，那么我们的 LSTM 网络的输入维度大小就是8。

为了得到字符级别的表达, 将单词的每个字符输入一个 LSTM 网络, 而c_w则为这个 LSTM 网络最后的隐状态。一些提示：

- 新模型中需要两个 LSTM, 一个跟之前一样, 用来输出词性标注的得分, 另外一个新增加的用来获取每个单词的字符级别表达。
- 为了在字符级别上运行序列模型，你需要用嵌入的字符来作为字符 LSTM 的输入。

高级：制定动态决策和BI-LSTM CRF

1.动态与静态深度学习工具包

Pytorch是一种*动态*神经网络套件。另一个动态套件的例子是Dynet（我之所以提到这一点，因为与 Pytorch和Dynet一起使用是相似的。如果你在Dynet中看到一个例子，它可能会帮助你在Pytorch中实现它）。相反的是*静态*工具包，其中包 括Theano，Keras，TensorFlow等。核心区别如下： * 在静态工具包中，您可以定义一次计算图，对其进行编译，然后将实例流式传输给它。 * 在动态工具包中，为每个实例定义计算图。它永远不会被编译并且是即时执行的。

在没有很多经验的情况下，很难理解其中的差异。一个例子是假设我们想要构建一个深层组成解析器。假设我们的模型大致涉及以下步骤： * 我们自底向上地建造树 * 标记根节点（句子的单词） * 从那里，使用神经网络和单词的嵌入来找到形成组成部分的组合。每当你形成一个新的成分时，使用某种技术来嵌入成分。在这种情况下， 我们的网络架构将完全取决于输入句子。在“The green cat scratched the wall”一句中，在模型中的某个点上，我们想要结合跨度$(i, j, r) = (1.3, NP)$（即，NP 组成部分跨越单词1到单词3，在这种情况下是“The green cat”）。

然而，另一句话可能是“Somewhere, the big fat cat scratched the wall”。在这句话中，我们希望在某个时刻形成组成$(2, 4, NP)$。 我们想要形成的成分将取决于实例。如果我们只编译计算图一次，就像在静态工具包中那样，但编写这个逻辑将非常困难或者说是不可能的。 但是，在动态工具包中，不仅有1个预定义的计算图。每个实例都可以有一个新的计算图，所以这个问题就消失了。

动态工具包还具有易于调试和代码更接近宿主语言的优点（我的意思是Pytorch和Dynet看起来更像是比Keras或Theano更实际的Python代码）。

2.Bi-LSTM条件随机场讨论

对于本节，我们将看到用于命名实体识别的Bi-LSTM条件随机场的完整复杂示例。虽然上面的LSTM标记符通常足以用于词性标注，但是像CRF这样的 序列模型对于NER上的强大性能非常重要。CRF，虽然这个名字听起来很可怕，但所有模型都是CRF，在LSTM中提供了这些功能。CRF

是一个高级模型，比本教程中的任何早期模型复杂得多。如果你想跳过它，也可以。要查看您是否准备好，请查看是否可以：

- 在步骤i中为标记k写出维特比变量的递归。
- 修改上述重复以计算转发变量。
- 再次修改上面的重复计算以计算日志空间中的转发变量（提示：log-sum-exp）

如果你可以做这三件事，你应该能够理解下面的代码。回想一下，CRF计算条件概率。设y为标签序列，x为字的输入序列。然后我们计算

$$P(y|x) = \frac{exp(Score(x,y))}{\sum_{y'} exp(Score(x,y))}$$

通过定义一些对数电位$log\psi_i(x,y)$来确定得分:

$$Score(x,y) = \sum_i log\psi_i(x,y)$$

为了使分区功能易于处理，电位必须仅查看局部特征。

在Bi-LSTM CRF中，我们定义了两种潜力：发射和过渡。索引i处的单词的发射电位来自时间步长i处 的Bi-LSTM的隐藏状态。转换分数存储在T矩阵P中，其中T是标记集。在我们的 实现中，$P_{j,k}$是从标签k转换到标签j的分数。所以：

$$\text{Score}(x,y) = \sum_i \log \psi_{\text{EMIT}}(y_i \to x_i) + \log \psi_{\text{TRANS}}(y_{i-1} \to y_i) = \sum_i h_i[y_i] +$$

$$\mathbf{P}y_i, yi - 1$$

在第二个表达式中，我们将标记视为分配了唯一的非负索引。

如果上面的讨论过于简短，你可以查看这个，是迈克尔柯林斯写的关于CRF的文章。

3.实现说明

下面的示例实现了日志空间中的前向算法来计算分区函数，以及用于解码的维特比算法。反向传播将自动为我们计算梯度。我们不需要手工做任何事情。

这个实现冰未优化。如果您了解发生了什么，您可能会很快发现在前向算法中迭代下一个标记可能是在一个大的操作中完成的。我想编码更具可读性。 如果您想进行相关更改，可以将此标记器用于实际任务。

3.1 导包

```
# Author: Robert Guthrie

import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.optim as optim

torch.manual_seed(1)
```

3.2 辅助函数

辅助函数的功能是使代码更具可读性。

```
def argmax(vec):
    # 将argmax作为python int返回
    _, idx = torch.max(vec, 1)
    return idx.item()


def prepare_sequence(seq, to_ix):
    idxs = [to_ix[w] for w in seq]
    return torch.tensor(idxs, dtype=torch.long)


# 以正向算法的数值稳定方式计算log sum exp
def log_sum_exp(vec):
    max_score = vec[0, argmax(vec)]
    max_score_broadcast = max_score.view(1, -1).expand(1, vec.size()[1])
    return max_score + \
        torch.log(torch.sum(torch.exp(vec - max_score_broadcast)))
```

3.3 创建模型

```
class BiLSTM_CRF(nn.Module):

    def __init__(self, vocab_size, tag_to_ix, embedding_dim, hidden_dim):
        super(BiLSTM_CRF, self).__init__()
        self.embedding_dim = embedding_dim
```

```
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size
        self.tag_to_ix = tag_to_ix
        self.tagset_size = len(tag_to_ix)

        self.word_embeds = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2,
                            num_layers=1, bidirectional=True)

        # 将LSTM的输出映射到标记空间。
        self.hidden2tag = nn.Linear(hidden_dim, self.tagset_size)

        # 转换参数矩阵。 输入i，j是得分从j转换到i。
        self.transitions = nn.Parameter(
            torch.randn(self.tagset_size, self.tagset_size))

        # 这两个语句强制执行我们从不转移到开始标记的约束
        # 并且我们永远不会从停止标记转移
        self.transitions.data[tag_to_ix[START_TAG], :] = -10000
        self.transitions.data[:, tag_to_ix[STOP_TAG]] = -10000

        self.hidden = self.init_hidden()

    def init_hidden(self):
        return (torch.randn(2, 1, self.hidden_dim // 2),
                torch.randn(2, 1, self.hidden_dim // 2))

    def _forward_alg(self, feats):
        # 使用前向算法来计算分区函数
        init_alphas = torch.full((1, self.tagset_size), -10000.)
        # START_TAG包含所有得分.
        init_alphas[0][self.tag_to_ix[START_TAG]] = 0.

        # 包装一个变量，以便我们获得自动反向提升
        forward_var = init_alphas

        # 通过句子迭代
        for feat in feats:
            alphas_t = []  # The forward tensors at this timestep
            for next_tag in range(self.tagset_size):
                # 广播发射得分：无论以前的标记是怎样的都是相同的
                emit_score = feat[next_tag].view(
                    1, -1).expand(1, self.tagset_size)
                # trans_score的第i个条目是从i转换到next_tag的分数
                trans_score = self.transitions[next_tag].view(1, -1)
                # next_tag_var的第i个条目是我们执行log-sum-exp之前的边（i -> next_tag
) 的值
                next_tag_var = forward_var + trans_score + emit_score
```

```
                # 此标记的转发变量是所有分数的log-sum-exp。
                alphas_t.append(log_sum_exp(next_tag_var).view(1))
            forward_var = torch.cat(alphas_t).view(1, -1)
        terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
        alpha = log_sum_exp(terminal_var)
        return alpha

    def _get_lstm_features(self, sentence):
        self.hidden = self.init_hidden()
        embeds = self.word_embeds(sentence).view(len(sentence), 1, -1)
        lstm_out, self.hidden = self.lstm(embeds, self.hidden)
        lstm_out = lstm_out.view(len(sentence), self.hidden_dim)
        lstm_feats = self.hidden2tag(lstm_out)
        return lstm_feats

    def _score_sentence(self, feats, tags):
        # Gives the score of a provided tag sequence
        score = torch.zeros(1)
        tags = torch.cat([torch.tensor([self.tag_to_ix[START_TAG]],
dtype=torch.long), tags])
        for i, feat in enumerate(feats):
            score = score + \
                self.transitions[tags[i + 1], tags[i]] + feat[tags[i + 1]]
        score = score + self.transitions[self.tag_to_ix[STOP_TAG], tags[-1]]
        return score

    def _viterbi_decode(self, feats):
        backpointers = []

        # Initialize the viterbi variables in log space
        init_vvars = torch.full((1, self.tagset_size), -10000.)
        init_vvars[0][self.tag_to_ix[START_TAG]] = 0

        # forward_var at step i holds the viterbi variables for step i-1
        forward_var = init_vvars
        for feat in feats:
            bptrs_t = []  # holds the backpointers for this step
            viterbivars_t = []  # holds the viterbi variables for this step

            for next_tag in range(self.tagset_size):
                # next_tag_var [i]保存上一步的标签i的维特比变量
                # 加上从标签i转换到next_tag的分数。
                # 我们这里不包括emission分数，因为最大值不依赖于它们（我们在下面添加它们）
                next_tag_var = forward_var + self.transitions[next_tag]
                best_tag_id = argmax(next_tag_var)
                bptrs_t.append(best_tag_id)
                viterbivars_t.append(next_tag_var[0][best_tag_id].view(1))
            # 现在添加emission分数，并将forward_var分配给我们刚刚计算的维特比变量集
```

```
            forward_var = (torch.cat(viterbivars_t) + feat).view(1, -1)
            backpointers.append(bptrs_t)

        # 过渡到STOP_TAG
        terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
        best_tag_id = argmax(terminal_var)
        path_score = terminal_var[0][best_tag_id]

        # 按照后退指针解码最佳路径。
        best_path = [best_tag_id]
        for bptrs_t in reversed(backpointers):
            best_tag_id = bptrs_t[best_tag_id]
            best_path.append(best_tag_id)
        # 弹出开始标记（我们不想将其返回给调用者）
        start = best_path.pop()
        assert start == self.tag_to_ix[START_TAG]  # Sanity check
        best_path.reverse()
        return path_score, best_path

    def neg_log_likelihood(self, sentence, tags):
        feats = self._get_lstm_features(sentence)
        forward_score = self._forward_alg(feats)
        gold_score = self._score_sentence(feats, tags)
        return forward_score - gold_score

    def forward(self, sentence):  # dont confuse this with _forward_alg above.
        # 获取BiLSTM的emission分数
        lstm_feats = self._get_lstm_features(sentence)

        # 根据功能，找到最佳路径。
        score, tag_seq = self._viterbi_decode(lstm_feats)
        return score, tag_seq
```

3.4 进行训练

```
START_TAG = "<START>"
STOP_TAG = "<STOP>"
EMBEDDING_DIM = 5
HIDDEN_DIM = 4

# 弥补一些训练数据
training_data = [(
    "the wall street journal reported today that apple corporation made
money".split(),
    "B I I I O O O B I O O".split()
), (
    "georgia tech is a university in georgia".split(),
```

```
    "B I O O O O B".split()
)]

word_to_ix = {}
for sentence, tags in training_data:
    for word in sentence:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)

tag_to_ix = {"B": 0, "I": 1, "O": 2, START_TAG: 3, STOP_TAG: 4}

model = BiLSTM_CRF(len(word_to_ix), tag_to_ix, EMBEDDING_DIM, HIDDEN_DIM)
optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=1e-4)

# 在训练前检查预测
with torch.no_grad():
    precheck_sent = prepare_sequence(training_data[0][0], word_to_ix)
    precheck_tags = torch.tensor([tag_to_ix[t] for t in training_data[0][1]],
dtype=torch.long)
    print(model(precheck_sent))

# 确保加载LSTM部分中较早的prepare_sequence
for epoch in range(
        300):  # again, normally you would NOT do 300 epochs, it is toy data
    for sentence, tags in training_data:
        # 步骤1. 请记住，Pytorch积累了梯度
        # We need to clear them out before each instance
        model.zero_grad()

        # 步骤2. 为我们为网络准备的输入，即将它们转换为单词索引的张量.
        sentence_in = prepare_sequence(sentence, word_to_ix)
        targets = torch.tensor([tag_to_ix[t] for t in tags], dtype=torch.long)

        # 步骤3. 向前运行
        loss = model.neg_log_likelihood(sentence_in, targets)

        # 步骤4.通过调用optimizer.step () 来计算损失，梯度和更新参数
        loss.backward()
        optimizer.step()

# 训练后检查预测
with torch.no_grad():
    precheck_sent = prepare_sequence(training_data[0][0], word_to_ix)
    print(model(precheck_sent))
# 得到结果
```

- **输出结果**

```
(tensor(2.6907), [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1])
(tensor(20.4906), [0, 1, 1, 1, 2, 2, 2, 0, 1, 2, 2])
```

4.练习：区分标记的新损失函数

我们没有必要在进行解码时创建计算图，因为我们不会从维特比路径得分反向传播。因为无论如何我们都有它，尝试训练标记器，其中损失函 数是Viterbi path得分和gold-standard得分之间的差异。应该清楚的是，当预测的标签序列是正确的标签序列时，该功能是非负值和0。这基 本上是*结构感知器*。

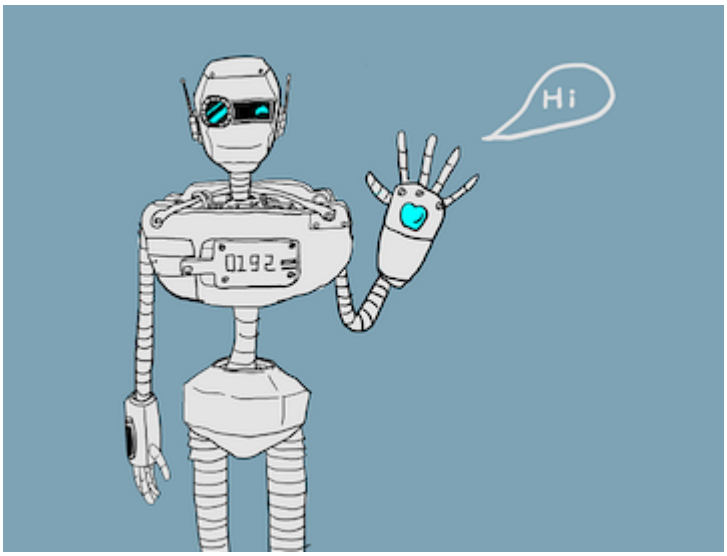
由于已经实现了Viterbi和score_sentence，因此这种修改应该很短。这是一个关于计算图形的形状*取决于训练实例*的示例。虽然我没有尝 试在静态工具包中实现它，但我想它是可以的但可能没有那么容易。

拿起一些真实数据并进行比较！

聊天机器人教程

在本教程中，我们探索一个好玩有趣的循环的序列到序列（sequence-to-sequence）的模型用例。我们将用Cornell Movie-Dialogs Corpus 处的电影剧本来训练一个简单的聊天机器人。

在人工智能研究领域中，对话模型是一个非常热门的话题。聊天机器人可以在各种设置中找到，包括客户服务应用和在线帮助。这些机器人通常 由基于检索的模型提供支持，这些模型的输出是某些形式问题预先定义的响应。在像公司IT服务台这样高度受限制的领域中，这些模型可能足够了，但是，对于更一般的用例它们还不够健壮。让一台机器与多领域的人进行有意义的对话是一个远未解决的研究问题。最近，深度学习热潮已经允许 强大的生成模型，如谷歌的神经对话模型 Neural Conversational Model，这标志着向多领域生成对话模型迈出了一大步。 在本教程中，我们将在PyTorch中实现这种模型。




```
> hello?
Bot: hello .
> where am I?
Bot: you re in a hospital .
> who are you?
Bot: i m a lawyer .
> how are you doing?
Bot: i m fine .
> are you my friend?
Bot: no .
> you're under arrest
Bot: i m trying to help you !
> i'm just kidding
```

```
Bot: i m sorry .
> where are you from?
Bot: san francisco .
> it's time for me to leave
Bot: i know .
> goodbye
Bot: goodbye .
```

教程要点

- 对Cornell Movie-Dialogs Corpus数据集的加载和预处理
- 用Luong attention mechanism(s)实现一个sequence-to-sequence模型
- 使用小批量数据联合训练解码器和编码器模型
- 实现贪婪搜索解码模块
- 与训练好的聊天机器人互动

鸣谢

本教程借鉴以下源码： * Yuan-Kuei Wu's pytorch-chatbot implementation: https://github.com/ywk991112/pytorch-chatbot * Sean Robertson's practical-pytorch seq2seq-translation example: https://github.com/spro/practical-pytorch/tree/master/seq2seq-translation * FloydHub's Cornell Movie Corpus preprocessing code: https://github.com/floydhub/textutil-preprocess-cornell-movie-corpus

1.下载数据文件

下载数据文件点击这里并将其放入到当前目录下的 `data/` 文件夹下。之后我们引入一些必须的包。

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import torch
from torch.jit import script, trace
import torch.nn as nn
from torch import optim
import torch.nn.functional as F
```

```
import csv
import random
import re
import os
import unicodedata
import codecs
from io import open
import itertools
import math

USE_CUDA = torch.cuda.is_available()
device = torch.device("cuda" if USE_CUDA else "cpu")
```

2.加载和预处理数据

下一步就是格式化处理我们的数据文件并将数据加载到我们可以使用的结构中。 Cornell Movie-Dialogs Corpus是一个丰富的电影角色对话数据集： * 10,292 对电影角色之间的220,579次对话 * 617部电影中的9,035个电影角色 * 总共304,713发言量

这个数据集庞大而多样，在语言形式、时间段、情感上等都有很大的变化。我们希望这种多样性使我们的模型能够适应多种形式的输入和查询。

首先，我们通过数据文件的某些行来查看原始数据的格式

```
corpus_name = "cornell movie-dialogs corpus"
corpus = os.path.join("data", corpus_name)

def printLines(file, n=10):
    with open(file, 'rb') as datafile:
        lines = datafile.readlines()
    for line in lines[:n]:
        print(line)

printLines(os.path.join(corpus, "movie_lines.txt"))
```

* 输出结果

```
b'L1045 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ They do not!\n'
b'L1044 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ They do to!\n'
b'L985 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ I hope so.\n'
b'L984 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ She okay?\n'
b"L925 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Let's go.\n"
b'L924 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ Wow\n'
```

```
b"L872 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Okay -- you're gonna need to
learn how to lie.\n"
b'L871 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ No\n'
b'L870 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ I\'m kidding.  You know how
sometimes you just become this "persona"?  And you don\'t know how to quit?\n'
b'L869 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Like my fear of wearing
pastels?\n'
```

2.1 创建格式化数据文件

为了方便起见，我们将创建一个格式良好的数据文件，其中每一行包含一个由 `tab` 制表符分隔的查询语句和响应语句对。

以下函数便于解析原始 movie_lines.txt 数据文件。* `loadLines`:将文件的每一行拆分为字段 `(lineID, characterID, movieID, character, text)` 组合的字典 * `loadConversations` :根据 `movie_conversations.txt` 将 `loadLines` 中的每一行数据进行归类 * `extractSentencePairs`: 从对话中提取句子对

```
# 将文件的每一行拆分为字段字典
def loadLines(fileName, fields):
    lines = {}
    with open(fileName, 'r', encoding='iso-8859-1') as f:
        for line in f:
            values = line.split(" +++$+++ ")
            # Extract fields
            lineObj = {}
            for i, field in enumerate(fields):
                lineObj[field] = values[i]
            lines[lineObj['lineID']] = lineObj
    return lines

# 将 `loadLines` 中的行字段分组为基于 *movie_conversations.txt* 的对话
def loadConversations(fileName, lines, fields):
    conversations = []
    with open(fileName, 'r', encoding='iso-8859-1') as f:
        for line in f:
            values = line.split(" +++$+++ ")
            # Extract fields
            convObj = {}
            for i, field in enumerate(fields):
                convObj[field] = values[i]
            # Convert string to list (convObj["utteranceIDs"] == "['L598485',
'L598486', ...]")
            lineIds = eval(convObj["utteranceIDs"])
            # Reassemble lines
```

```
            convObj["lines"] = []
            for lineId in lineIds:
                convObj["lines"].append(lines[lineId])
            conversations.append(convObj)
    return conversations

# 从对话中提取一对句子
def extractSentencePairs(conversations):
    qa_pairs = []
    for conversation in conversations:
        # Iterate over all the lines of the conversation
        for i in range(len(conversation["lines"]) - 1):  # We ignore the last line
(no answer for it)
            inputLine = conversation["lines"][i]["text"].strip()
            targetLine = conversation["lines"][i+1]["text"].strip()
            # Filter wrong samples (if one of the lists is empty)
            if inputLine and targetLine:
                qa_pairs.append([inputLine, targetLine])
    return qa_pairs
```

现在我们将调用这些函数来创建文件，我们命名为 `formatted_movie_lines.txt`。

```
# 定义新文件的路径
datafile = os.path.join(corpus, "formatted_movie_lines.txt")

delimiter = '\t'

delimiter = str(codecs.decode(delimiter, "unicode_escape"))

# 初始化行dict，对话列表和字段ID
lines = {}
conversations = []
MOVIE_LINES_FIELDS = ["lineID", "characterID", "movieID", "character", "text"]
MOVIE_CONVERSATIONS_FIELDS = ["character1ID", "character2ID", "movieID",
"utteranceIDs"]

# 加载行和进程对话
print("\nProcessing corpus...")
lines = loadLines(os.path.join(corpus, "movie_lines.txt"), MOVIE_LINES_FIELDS)
print("\nLoading conversations...")
conversations = loadConversations(os.path.join(corpus, "movie_conversations.txt"),
                                  lines, MOVIE_CONVERSATIONS_FIELDS)

# 写入新的csv文件
print("\nWriting newly formatted file...")
with open(datafile, 'w', encoding='utf-8') as outputfile:
    writer = csv.writer(outputfile, delimiter=delimiter, lineterminator='\n')
    for pair in extractSentencePairs(conversations):
```

```
        writer.writerow(pair)

# 打印一个样本的行
print("\nSample lines from file:")
printLines(datafile)
```

* 输出结果：

```
Processing corpus...

Loading conversations...

Writing newly formatted file...

Sample lines from file:
b"Can we make this quick?  Roxanne Korrine and Andrew Barrett are having an
incredibly horrendous public break- up on the quad.  Again.\tWell, I thought we'd
start with pronunciation, if that's okay with you.\n"
b"Well, I thought we'd start with pronunciation, if that's okay with you.\tNot
the hacking and gagging and spitting part.  Please.\n"
b"Not the hacking and gagging and spitting part.  Please.\tOkay... then how 'bout
we try out some French cuisine.  Saturday?  Night?\n"
b"You're asking me out.  That's so cute. What's your name again?\tForget it.\n"
b"No, no, it's my fault -- we didn't have a proper introduction ---\tCameron.\n"
b"Cameron.\tThe thing is, Cameron -- I'm at the mercy of a particularly hideous
breed of loser.  My sister.  I can't date until she does.\n"
b"The thing is, Cameron -- I'm at the mercy of a particularly hideous breed of
loser.  My sister.  I can't date until she does.\tSeems like she could get a date
easy enough...\n"
b'Why?\tUnsolved mystery.  She used to be really popular when she started high
school, then it was just like she got sick of it or something.\n'
b"Unsolved mystery.  She used to be really popular when she started high school,
then it was just like she got sick of it or something.\tThat's a shame.\n"
b'Gosh, if only we could find Kat a boyfriend...\tLet me see what I can do.\n'
```

2.2 加载和清洗数据

我们下一个任务是创建词汇表并将查询/响应句子对（对话）加载到内存。

注意我们正在处理**词序**，这些词序没有映射到离散数值空间。因此，我们必须通过数据集中的单词来创建一个索引。

为此我们创建了一个 `Voc` 类,它会存储从单词到索引的映射、索引到单词的反向映射、每个单词的计数和总单词量。这个类提供向词汇表中添 加单词的方法(`addWord`)、添加所有单词到句子中的方法 (`addSentence`) 和清洗不常见的单词方法(`trim`)。更多的数据清洗在后面进行。

```
# 默认词向量
PAD_token = 0  # Used for padding short sentences
SOS_token = 1  # Start-of-sentence token
EOS_token = 2  # End-of-sentence token

class Voc:
    def __init__(self, name):
        self.name = name
        self.trimmed = False
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
        self.num_words = 3  # Count SOS, EOS, PAD

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.num_words
            self.word2count[word] = 1
            self.index2word[self.num_words] = word
            self.num_words += 1
        else:
            self.word2count[word] += 1

    # 删除低于特定计数阈值的单词
    def trim(self, min_count):
        if self.trimmed:
            return
        self.trimmed = True

        keep_words = []

        for k, v in self.word2count.items():
            if v >= min_count:
                keep_words.append(k)

        print('keep_words {} / {} = {:.4f}'.format(
            len(keep_words), len(self.word2index), len(keep_words) /
len(self.word2index)
        ))

        # 重初始化字典
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
```

```
        self.num_words = 3 # Count default tokens

        for word in keep_words:
            self.addWord(word)
```

现在我们可以组装词汇表和查询/响应语句对。在使用数据之前，我们必须做一些预处理。

首先，我们必须使用 `unicodeToAscii` 将 unicode 字符串转换为 ASCII。然后，我们应该将所有字母转换为小写字母并清洗掉除基本标点之 外的所有非字母字符 (`normalizeString`)。最后，为了帮助训练收敛，我们将过滤掉长度大于MAX_LENGTH 的句子 (`filterPairs`)。

```
MAX_LENGTH = 10  # Maximum sentence length to consider

# 将Unicode字符串转换为纯ASCII, 多亏了
# https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# 初始化Voc对象 和 格式化pairs对话存放到list中
def readVocs(datafile, corpus_name):
    print("Reading lines...")
    # Read the file and split into lines
    lines = open(datafile, encoding='utf-8').read().strip().split('\n')
    # Split every line into pairs and normalize
    pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]
    voc = Voc(corpus_name)
    return voc, pairs

# 如果对 'p' 中的两个句子都低于 MAX_LENGTH 阈值, 则返回True
def filterPair(p):
    # Input sequences need to preserve the last word for EOS token
    return len(p[0].split(' ')) < MAX_LENGTH and len(p[1].split(' ')) < MAX_LENGTH

# 过滤满足条件的 pairs 对话
def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]

# 使用上面定义的函数, 返回一个填充的voc对象和对列表
def loadPrepareData(corpus, corpus_name, datafile, save_dir):
    print("Start preparing training data ...")
    voc, pairs = readVocs(datafile, corpus_name)
    print("Read {!s} sentence pairs".format(len(pairs)))
    pairs = filterPairs(pairs)
```

```
    print("Trimmed to {!s} sentence pairs".format(len(pairs)))
    print("Counting words...")
    for pair in pairs:
        voc.addSentence(pair[0])
        voc.addSentence(pair[1])
    print("Counted words:", voc.num_words)
    return voc, pairs

# 加载/组装voc和对
save_dir = os.path.join("data", "save")
voc, pairs = loadPrepareData(corpus, corpus_name, datafile, save_dir)
# 打印一些对进行验证
print("\npairs:")
for pair in pairs[:10]:
    print(pair)
```

* 输出结果：

```
Start preparing training data ...
Reading lines...
Read 221282 sentence pairs
Trimmed to 64271 sentence pairs
Counting words...
Counted words: 18008

pairs:
['there .', 'where ?']
['you have my word . as a gentleman', 'you re sweet .']
['hi .', 'looks like things worked out tonight huh ?']
['you know chastity ?', 'i believe we share an art instructor']
['have fun tonight ?', 'tons']
['well no . . .', 'then that s all you had to say .']
['then that s all you had to say .', 'but']
['but', 'you always been this selfish ?']
['do you listen to this crap ?', 'what crap ?']
['what good stuff ?', 'the real you .']
```

另一种有利于让训练更快收敛的策略是去除词汇表中很少使用的单词。减少特征空间也会降低模型学习目标函数的难度。我们通过以下两个步 骤完成这个操作: * 使用 `voc.trim` 函数去除 MIN_COUNT 阈值以下单词 。 * 如果句子中包含词频过小的单词，那么整个句子也被过滤掉。

```
MIN_COUNT = 3    # 修剪的最小字数阈值

def trimRareWords(voc, pairs, MIN_COUNT):
    # 修剪来自voc的MIN_COUNT下使用的单词
    voc.trim(MIN_COUNT)
```

```
    # Filter out pairs with trimmed words
    keep_pairs = []
    for pair in pairs:
        input_sentence = pair[0]
        output_sentence = pair[1]
        keep_input = True
        keep_output = True
        # 检查输入句子
        for word in input_sentence.split(' '):
            if word not in voc.word2index:
                keep_input = False
                break
        # 检查输出句子
        for word in output_sentence.split(' '):
            if word not in voc.word2index:
                keep_output = False
                break

        # 只保留输入或输出句子中不包含修剪单词的对
        if keep_input and keep_output:
            keep_pairs.append(pair)

    print("Trimmed from {} pairs to {}, {:.4f} of total".format(len(pairs),
len(keep_pairs), len(keep_pairs) / len(pairs)))
    return keep_pairs


# 修剪voc和对
pairs = trimRareWords(voc, pairs, MIN_COUNT)
```

* 输出结果：

```
keep_words 7823 / 18005 = 0.4345
Trimmed from 64271 pairs to 53165, 0.8272 of total
```

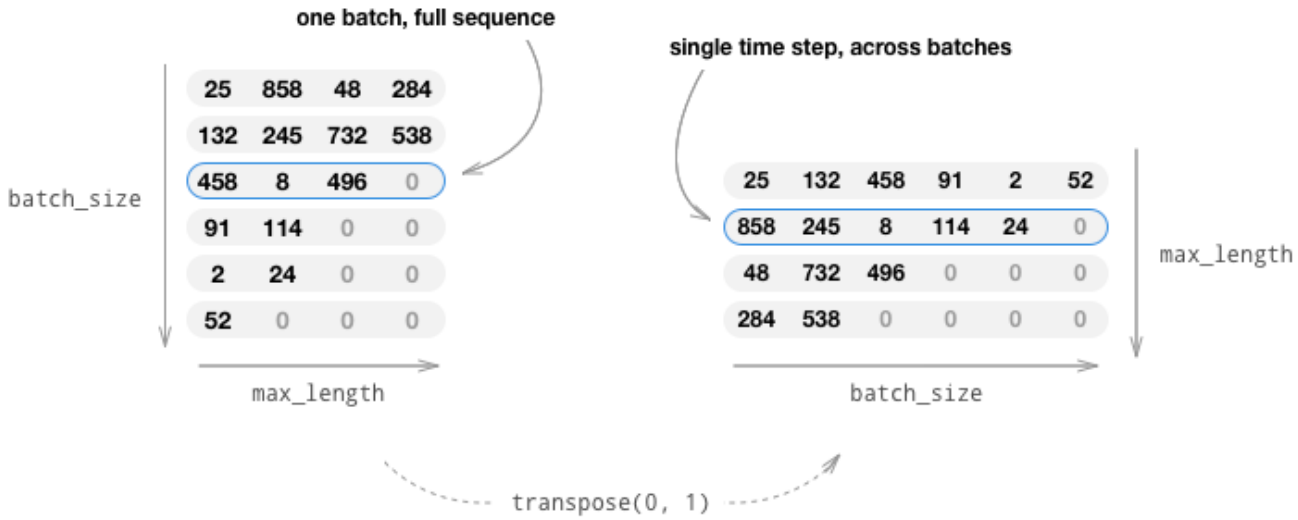
3.为模型准备数据

尽管我们已经投入了大量精力来准备和清洗我们的数据，将它变成一个很好的词汇对象和一系列的句子对，但我们的模型最终希望数据以 `numerical torch` 张量作为输入。可以在seq2seq translation tutorial 中找到为模型准备处理数据的一种方法。 在该教程中，我们使用 `batch size` 大小为1，这意味着我们所要做的就是将句子对中的单词转换为词汇表中的相应索引，并将其提供给模型。

但是，如果你想要加速训练或者想要利用GPU并行计算能力，则需要使用小批量 `mini-batches` 来训练。

使用小批量 `mini-batches` 也意味着我们必须注意批量处理中句子长度的变化。为了容纳同一batch中不同大小的句子，我们将使我们的批量输 入张量大小 `(max_length, batch_size)` ，其中短于*max_length*的句子在*EOS_token*之后进行零填充（ `zero padded` ）。

如果我们简单地将我们的英文句子转换为张量，通过将单词转换为索引 `indicesFromSentence` 和零填充 `zero-pad` ，我们的张量的大小将是 `(batch_size, max_length)` ，并且索引第一维将在所有时间步骤中返回完整序列。但是，我们需要沿着时间对我们批量数据进行索引并且包 括批量数据中所有序列。因此，我们将输入批处理大小转换为 `(max_length, batch_size)` ，以便跨第一维的索引返回批处理中所有句子的时 间步长。 我们在 `zeroPadding` 函数中隐式处理这个转置。



`inputvar` 函数是处理将句子转换为张量的过程，最终创建正确大小的零填充张量。它还返回批处理中每个序列的长度张量(tensor of lengths)， 长度张量稍后将传递给我们的解码器。

`outputvar` 函数执行与 `inputvar` 类似的函数，但他不返回长度张量，而是返回二进制 mask tensor 和最大目标句子长度。二进制 mask tensor 的大小与输出目标张量的大小相同，但作为 `PAD_token` 的每个元素都是0而其他元素都是1。

`batch2traindata` 只需要取一批句子对，并使用上述函数返回输入张量和目标张量。

```
def indexesFromSentence(voc, sentence):
    return [voc.word2index[word] for word in sentence.split(' ')] + [EOS_token]

# zip 对数据进行合并了，相当于行列转置了
def zeroPadding(l, fillvalue=PAD_token):
```

```
    return list(itertools.zip_longest(*l, fillvalue=fillvalue))

# 记录 PAD_token的位置为0,  其他的为1
def binaryMatrix(l, value=PAD_token):
    m = []
    for i, seq in enumerate(l):
        m.append([])
        for token in seq:
            if token == PAD_token:
                m[i].append(0)
            else:
                m[i].append(1)
    return m

# 返回填充前（加入结束index EOS_token做标记）的长度 和 填充后的输入序列张量
def inputVar(l, voc):
    indexes_batch = [indexesFromSentence(voc, sentence) for sentence in l]
    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])
    padList = zeroPadding(indexes_batch)
    padVar = torch.LongTensor(padList)
    return padVar, lengths

# 返回填充前（加入结束index EOS_token做标记）最长的一个长度 和 填充后的输入序列张量，和 填
充后的标记 mask
def outputVar(l, voc):
    indexes_batch = [indexesFromSentence(voc, sentence) for sentence in l]
    max_target_len = max([len(indexes) for indexes in indexes_batch])
    padList = zeroPadding(indexes_batch)
    mask = binaryMatrix(padList)
    mask = torch.ByteTensor(mask)
    padVar = torch.LongTensor(padList)
    return padVar, mask, max_target_len

# 返回给定batch对的所有项目
def batch2TrainData(voc, pair_batch):
    pair_batch.sort(key=lambda x: len(x[0].split(" ")), reverse=True)
    input_batch, output_batch = [], []
    for pair in pair_batch:
        input_batch.append(pair[0])
        output_batch.append(pair[1])
    inp, lengths = inputVar(input_batch, voc)
    output, mask, max_target_len = outputVar(output_batch, voc)
    return inp, lengths, output, mask, max_target_len

# 验证例子
small_batch_size = 5
batches = batch2TrainData(voc, [random.choice(pairs) for _ in
range(small_batch_size)])
```

```
input_variable, lengths, target_variable, mask, max_target_len = batches

print("input_variable:", input_variable)
print("lengths:", lengths)
print("target_variable:", target_variable)
print("mask:", mask)
print("max_target_len:", max_target_len)
```

* 输出结果：

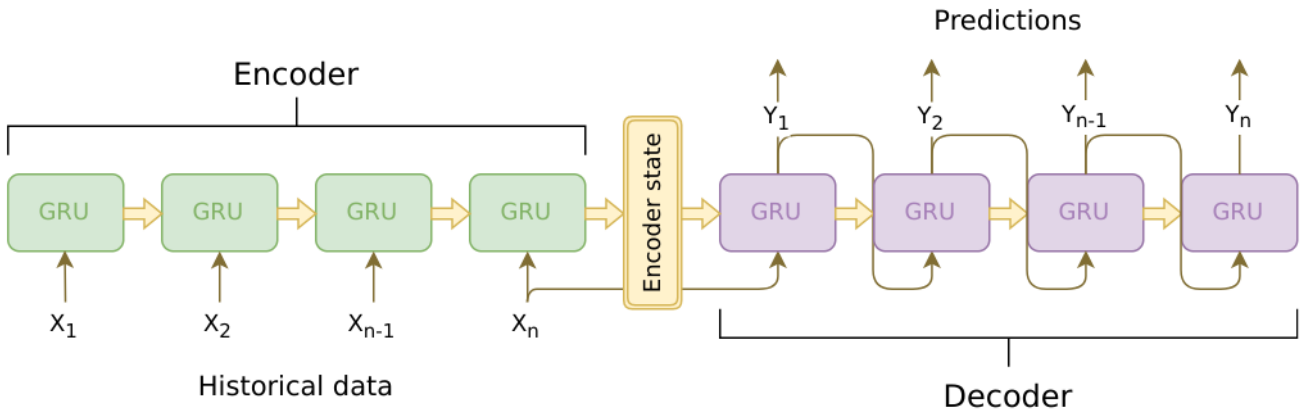
```
input_variable: tensor([[ 273,   64,   53,   25,   25],
        [ 188,  542, 4095,  200,  200],
        [  53,    4,  115,   67, 3644],
        [ 660,    4, 3600, 1531,    2],
        [1258,    4,    4,    4,    0],
        [   4,    2,    2,    2,    0],
        [   2,    0,    0,    0,    0]])
lengths: tensor([7, 6, 6, 6, 4])
target_variable: tensor([[ 147,  214,  219,  252,  122],
        [  47,    4,  389,  387,   27],
        [   7,    4,   25,   25,   14],
        [1026,    4,  222,    4,   53],
        [1034,    2,   53,    2,    4],
        [  12,    0, 4096,    0,    4],
        [1113,    0,    6,    0,    4],
        [   4,    0,    2,    0,    2],
        [   2,    0,    0,    0,    0]])
mask: tensor([[1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1],
        [1, 0, 0, 0, 0]], dtype=torch.uint8)
max_target_len: 9
```

4.定义模型

4.1 Seq2Seq模型

Seq2Seq模型 我们聊天机器人的大脑是序列到序列（seq2seq）模型。seq2seq模型的目标是将可变长度序列作为输入，并使用固定大小的模型将可变长度序 列作为输出返回。

Sutskever et al.发现通过一起使用两个独立的RNN，我们可以完成这项任务。第一个RNN充当**编码 器**，其将可变长度输入序列编码为固定长度上下文向量。理论上，该上下文向量（RNN的最终隐藏层）将包含关于输入到机器人的查询语句的 语义信息。第二个RNN是一个**解码器**，它接收输入文字和上下文矢量，并返回序列中下一句文字的概率和在下一次迭代中使用的隐藏状态。



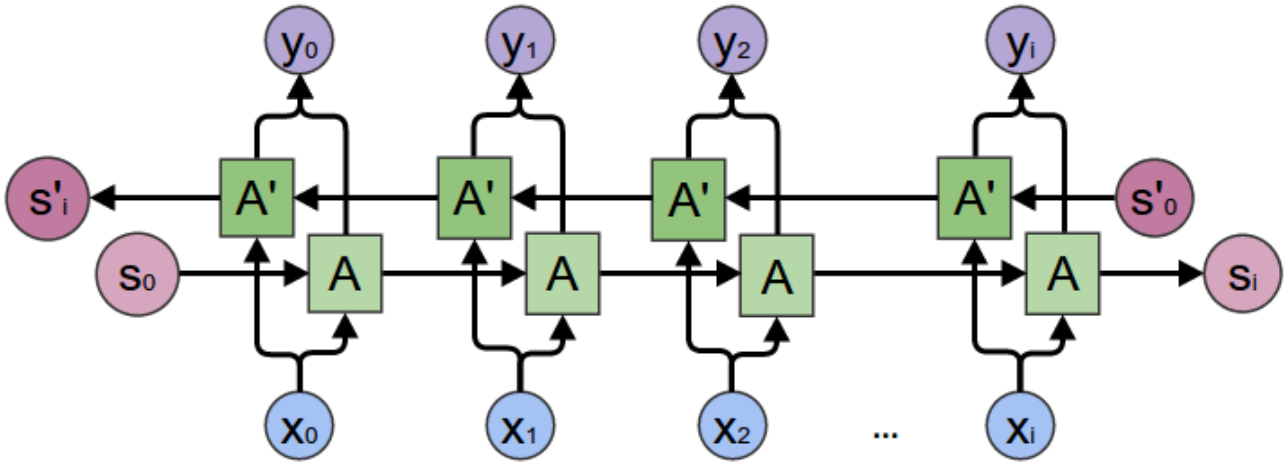
图片来源: https://jeddy92.github.io/JEddy92.github.io/ts_seq2seq_intro/

4.2 编码器

编码器RNN每次迭代中输入一个语句输出一个token（例如，一个单词），同时在这时间内输出“输出”向量和“隐藏状态”向量。然后将隐藏 状态向量传递到下一步，并记录输出向量。编码器将其在序列中的每一点处看到的上下文转换为高维空间中的一系列点，解码器将使用这些点 为给定任务生成有意义的输出。

我们的编码器的核心是由Cho et al.等人发明的多层门循环单元。在2014年，我们将使用 GRU的双向变体，这意味着基本上有两个独立的RNN：一个以正常的顺序输入输入序列，另一个以相反的顺序输入输入序列。每个网络的输出在 每个时间步骤求和。使用双向GRU将为我们提供编码过去和未来上下文的优势。

双向RNN：



图片来源: https://colah.github.io/posts/2015-09-NN-Types-FP/

> 注意: `embedding` 层用于在任意大小的特征空间中对我们的单词索引进行编码。对于我们的模型，此图层会将每个单词映射到大小为 `hidden_size` 的特征空间。训练后，这些值会被编码成和他们相似的有意义词语。

最后，如果将填充的一批序列传递给RNN模块，我们必须分别使用 `torch.nn.utils.rnn.pack_padded_sequence` 和 `torch.nn.utils.rnn.pad_packed_sequence` 在RNN传递时分别进行填充和反填充。

计算图

1.将单词索引转换为词嵌入 embeddings。
2.为RNN模块打包填充batch序列。
3.通过GRU进行前向传播。
4.反填充。
5.对双向GRU输出求和。
6.返回输出和最终隐藏状态。

输入

- `input_seq` ：一批输入句子; *shape = (max_length , batch_size*
- `input_lengths` ：batch中每个句子对应的句子长度列表;*shape=(batch_size)*
- `hidden` :隐藏状态;*shape =(n_layers x num_directions , batch_size , hidden_size)*

输出

- `outputs` ：GRU最后一个隐藏层的输出特征（双向输出之和）;*shape = (max_length , batch_size , hidden_size)*
- `hidden` ：从GRU更新隐藏状态;*shape = (n_layers x num_directions , batch_size , hidden_size)*

```
class EncoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding, n_layers=1, dropout=0):
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size
        self.embedding = embedding

        # 初始化GRU; input_size和hidden_size参数都设置为'hidden_size'
        # 因为我们的输入大小是一个嵌入了多个特征的单词== hidden_size
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers,
                          dropout=(0 if n_layers == 1 else dropout),
bidirectional=True)

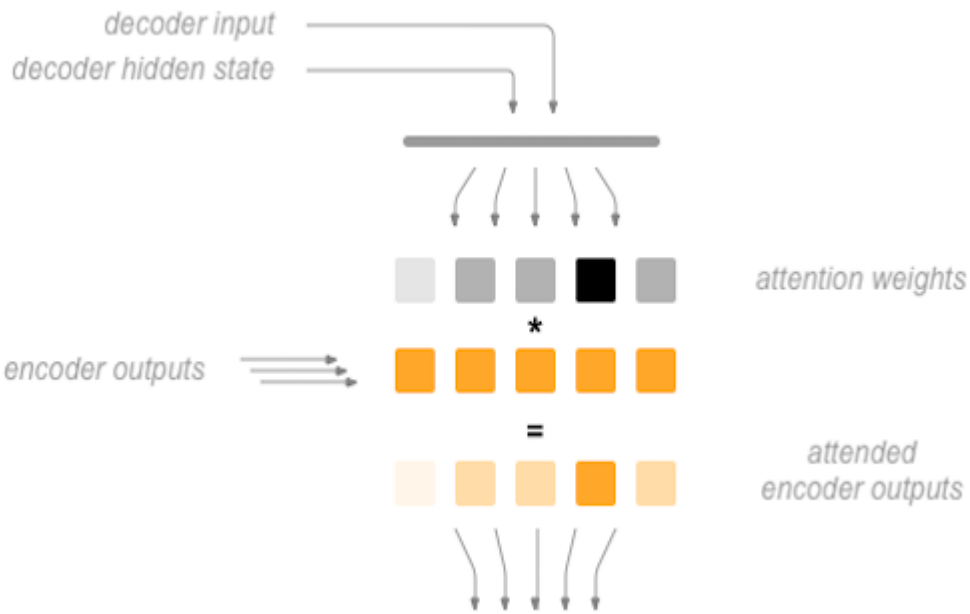
    def forward(self, input_seq, input_lengths, hidden=None):
        # 将单词索引转换为词向量
        embedded = self.embedding(input_seq)
        # 为RNN模块打包填充batch序列
        packed = nn.utils.rnn.pack_padded_sequence(embedded, input_lengths)
        # 正向通过GRU
        outputs, hidden = self.gru(packed, hidden)
        # 打开填充
        outputs, _ = nn.utils.rnn.pad_packed_sequence(outputs)
        # 总和双向GRU输出
        outputs = outputs[:, :, :self.hidden_size] +
outputs[:, : ,self.hidden_size:]
        # 返回输出和最终隐藏状态
        return outputs, hidden
```

4.3 解码器

解码器RNN以*token-by-token*的方式生成响应语句。它使用编码器的上下文向量和内部隐藏状态来生成序列中的下一个单词。它持续生成单词，直到输出是*EOS_token*，这个表示句子的结尾。一个 vanilla seq2seq 解码器的常见问题是，如果我们只依赖于上下文向量来编码整个输入 序列的含义，那么我们很可能会丢失信息。尤其是在处理长输入序列时，这极大地限制了我们的解码器的能力。

为了解决这个问题，Bahdanau et al.等人创建了一种“attention mechanism”，允许解码器关注 输入序列的某些部分，而不是在每一步都使用完全固定的上下文。

在一个高的层级中，用解码器的当前隐藏状态和编码器输出来计算注意力。输出注意力的权重与输入序列具有相同的大小，允许我们将它们乘 以编码器输出，给出一个加权和，表示要注意的编码器输出部分。Sean Robertson的图片很好地描述了这一点：

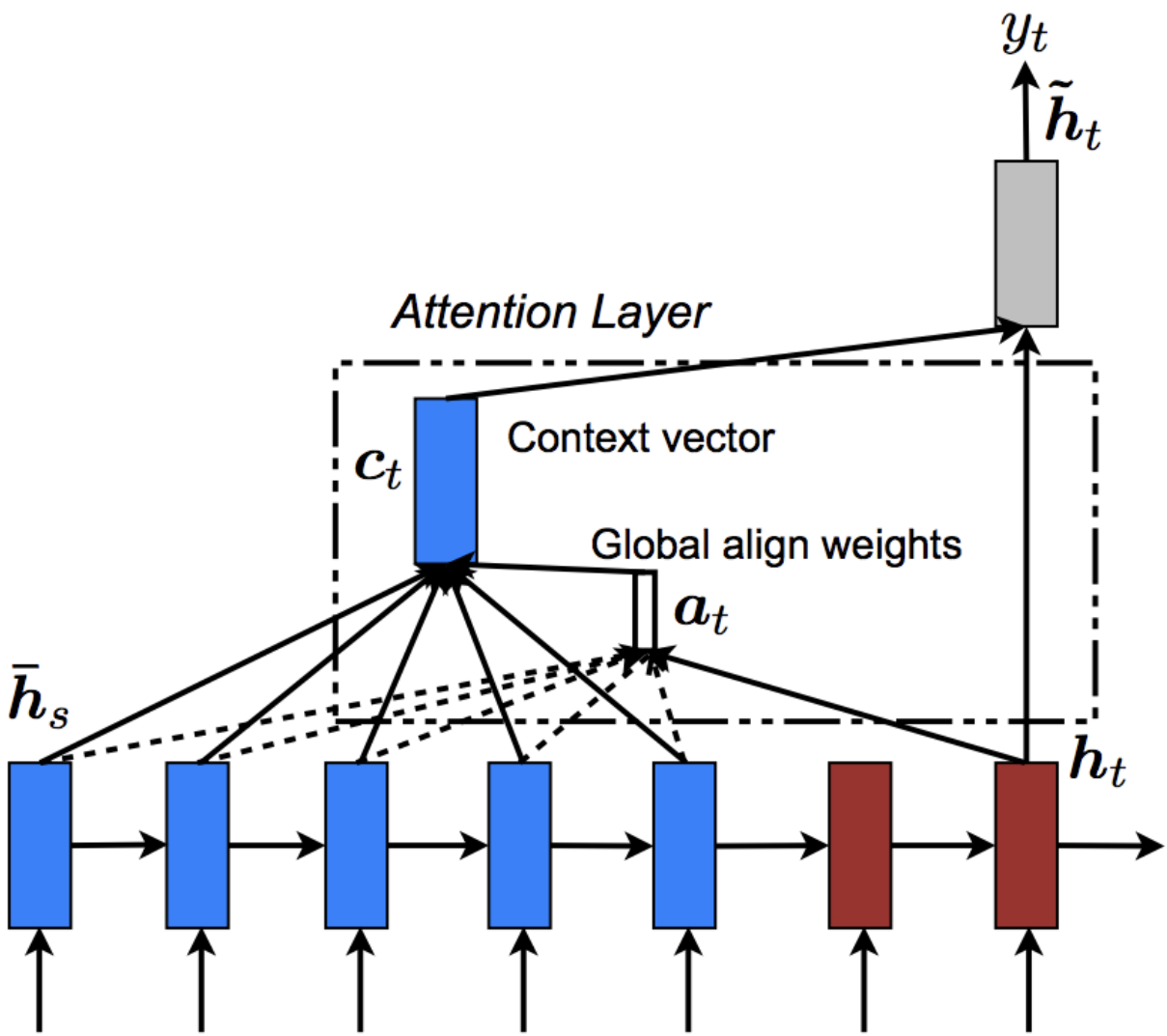


Luong et al.通过创造“Global attention”，改善了Bahdanau et al. 的基础工作。关键的区别在于，对于“Global attention”，我们考虑所有编码器的隐藏状态，而不是 Bahdanau 等人的“Local attention”，它只考虑当前步中编码器的隐藏状态。另一个区别在于，通过“Global attention”，我们仅使用当前步的解码器的隐藏状态来计算注意力权重（或者能量）。Bahdanau 等人的注意力计算需要知道前一步中解码器的状态。 此外，Luong等人提供各种方法来计算编码器输出和解码器输出 之间的注意权重（能量），称之为“score functions”：

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W}_a \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W}_a [\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

其中h_t=当前目标解码器状态，$\bar{h}_s$= 所有编码器状态。

总体而言，Global attention机制可以通过下图进行总结。请注意，我们将“Attention Layer”用一个名为 `Attn` 的 `nn.Module` 来单独实现。 该模块的输出是经过softmax标准化后权重张量的大小 * (batch_size，1，max_length) *。




```
# Luong的attention layer
class Attn(torch.nn.Module):
    def __init__(self, method, hidden_size):
        super(Attn, self).__init__()
        self.method = method
        if self.method not in ['dot', 'general', 'concat']:
            raise ValueError(self.method, "is not an appropriate attention
method.")
        self.hidden_size = hidden_size
```

```
        if self.method == 'general':
            self.attn = torch.nn.Linear(self.hidden_size, hidden_size)
        elif self.method == 'concat':
            self.attn = torch.nn.Linear(self.hidden_size * 2, hidden_size)
            self.v = torch.nn.Parameter(torch.FloatTensor(hidden_size))

    def dot_score(self, hidden, encoder_output):
        return torch.sum(hidden * encoder_output, dim=2)

    def general_score(self, hidden, encoder_output):
        energy = self.attn(encoder_output)
        return torch.sum(hidden * energy, dim=2)

    def concat_score(self, hidden, encoder_output):
        energy = self.attn(torch.cat((hidden.expand(encoder_output.size(0), -1,
-1), encoder_output), 2)).tanh()
        return torch.sum(self.v * energy, dim=2)

    def forward(self, hidden, encoder_outputs):
        # 根据给定的方法计算注意力（能量）
        if self.method == 'general':
            attn_energies = self.general_score(hidden, encoder_outputs)
        elif self.method == 'concat':
            attn_energies = self.concat_score(hidden, encoder_outputs)
        elif self.method == 'dot':
            attn_energies = self.dot_score(hidden, encoder_outputs)

        # Transpose max_length and batch_size dimensions
        attn_energies = attn_energies.t()

        # Return the softmax normalized probability scores (with added dimension)
        return F.softmax(attn_energies, dim=1).unsqueeze(1)
```

现在我们已经定义了注意力子模块，我们可以实现真实的解码器模型。对于解码器，我们将每次手动进行一批次的输入。这意味着我们的词嵌 入张量和GRU输出都将具有相同大小*（1，batch_size，hidden_size）*。

计算图

1.获取当前输入的词嵌入
2.通过单向GRU进行前向传播
3.通过2输出的当前GRU计算注意力权重
4.将注意力权重乘以编码器输出以获得新的“weighted sum”上下文向量
5.使用Luong eq.5连接加权上下文向量和GRU输出

6.使用Luong eq.6预测下一个单词（没有softmax）
7.返回输出和最终隐藏状态

输入

- `input_step` ：每一步输入序列batch（一个单词）;*shape = (1 , batch_size)*
- `last_hidden` ：GRU的最终隐藏层;*shape = (n_layers x num_directions , batch_size , hidden_size)*
- `encoder_outputs` ：编码器模型的输出;*shape = (max_length , batch_size , hidden_size)*

输出

- `output`：一个softmax标准化后的张量，代表了每个单词在解码序列中是下一个输出单词的概率;*shape = (batch_size , voc.num_words)*
- `hidden`：GRU的最终隐藏状态;*shape = (n_layers x num_directions , batch_size , hidden_size*

```
class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, embedding, hidden_size, output_size,
n_layers=1, dropout=0.1):
        super(LuongAttnDecoderRNN, self).__init__()

        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout = dropout

        # 定义层
        self.embedding = embedding
        self.embedding_dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers, dropout=(0 if
n_layers == 1 else dropout))
        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

        self.attn = Attn(attn_model, hidden_size)

    def forward(self, input_step, last_hidden, encoder_outputs):
        # 注意：我们一次运行这一步（单词）
        # 获取当前输入字的嵌入
        embedded = self.embedding(input_step)
        embedded = self.embedding_dropout(embedded)
        # 通过单向GRU转发
        rnn_output, hidden = self.gru(embedded, last_hidden)
```

```
        # 从当前GRU输出计算注意力
        attn_weights = self.attn(rnn_output, encoder_outputs)
        # 将注意力权重乘以编码器输出以获得新的“加权和”上下文向量
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1))
        # 使用Luong的公式五连接加权上下文向量和GRU输出
        rnn_output = rnn_output.squeeze(0)
        context = context.squeeze(1)
        concat_input = torch.cat((rnn_output, context), 1)
        concat_output = torch.tanh(self.concat(concat_input))
        # 使用Luong的公式6预测下一个单词
        output = self.out(concat_output)
        output = F.softmax(output, dim=1)
        # 返回输出和在最终隐藏状态
        return output, hidden
```

5.定义训练步骤

5.1 Masked 损失

由于我们处理的是批量填充序列，因此在计算损失时我们不能简单地考虑张量的所有元素。我们定义 `maskNLLLoss` 可以根据解码器的输出张量、 描述目标张量填充的 `binary mask` 张量来计算损失。该损失函数计算与 `mask tensor` 中的1对应的元素的平均负对数似然。

```
def maskNLLLoss(inp, target, mask):
    nTotal = mask.sum()
    crossEntropy = -torch.log(torch.gather(inp, 1, target.view(-1, 1)).squeeze(1))
    loss = crossEntropy.masked_select(mask).mean()
    loss = loss.to(device)
    return loss, nTotal.item()
```

5.2 单次训练迭代

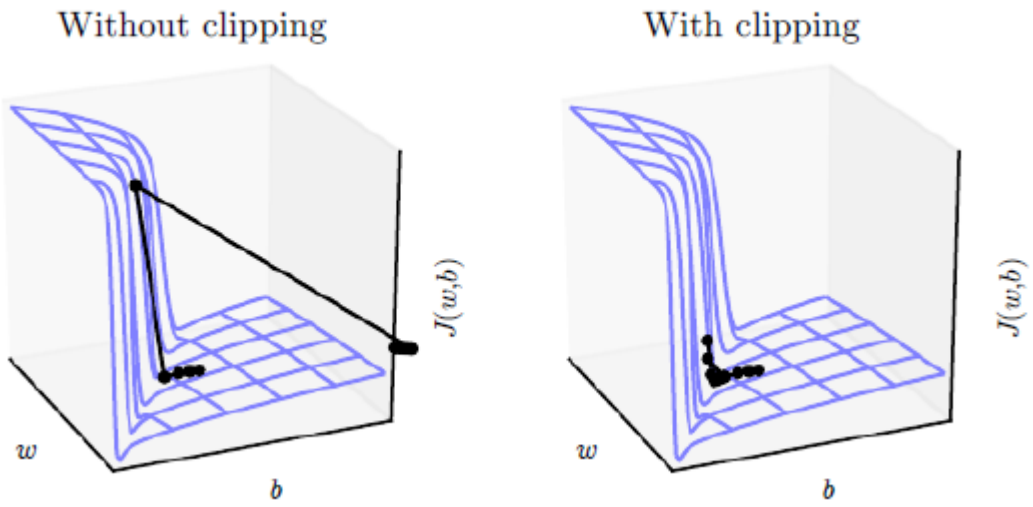
`train` 函数包含单次训练迭代的算法（单批输入）。

我们将使用一些巧妙的技巧来帮助融合：

- 第一个技巧是使用**teacher forcing**。 这意味着在一些概率是由 `teacher_forcing_ratio` 设置，我们使用当前目标单词作为解码器 的下一个输入，而不是使用解码器的当前推测。该技巧充当解码器的**training wheels**，有助于更有效的训练。然而，**teacher forcing** 可能导致推导中的模型不稳定，因为解码器可能没有足够的机会在训练期间真正地制作自己的输出序

列。因此，我们必须注意我们如何设置 `teacher_forcing_ratio` ，同时不要被快速的收敛所迷惑。

- 第二个技巧是梯度裁剪(**gradient clipping**)。这是一种用于对抗“爆炸梯度（exploding gradient）”问题的常用技术。本质上，通过将梯度剪切或阈值化到最大值，我们可以防止在损失函数中梯度以指数方式增长并发生溢出（NaN）或者越过梯度。



图片来源: Goodfellow et al. Deep Learning. 2016. https://www.deeplearningbook.org/

- **操作顺序**

1.通过编码器前向计算整个批次输入。
2.将解码器输入初始化为SOS_token，将隐藏状态初始化为编码器的最终隐藏状态。
3.通过解码器一次一步地前向计算输入一批序列。
4.如果是 teacher forcing 算法：将下一个解码器输入设置为当前目标;如果是 no teacher forcing 算法：将下一个解码器输入设置为当前解码器输出。
5.计算并累积损失。
6.执行反向传播。
7.裁剪梯度。
8.更新编码器和解码器模型参数。

> PyTorch的RNN模块（ `RNN` ， `LSTM` ， `GRU` ）可以像任何其他非重复层一样使用，只需将整个输入序列（或一批序列）传递给它们。 我们在 编码器 中使用GRU层就是这样的。实际情况是，在计算中有一个迭代过程循环计算隐藏状态的每一步。或者，你每次只运行一个 模块。在这种情

> 况下，我们在训练过程中手动循环遍历序列就像我们必须为解码器模型做的那样。只要你正确的维护这些模型的模块，就 可以非常简单的实现顺序模型。

```
def train(input_variable, lengths, target_variable, mask, max_target_len,
encoder, decoder, embedding,
          encoder_optimizer, decoder_optimizer, batch_size, clip,
max_length=MAX_LENGTH):

    # 零化梯度
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    # 设置设备选项
    input_variable = input_variable.to(device)
    lengths = lengths.to(device)
    target_variable = target_variable.to(device)
    mask = mask.to(device)

    # 初始化变量
    loss = 0
    print_losses = []
    n_totals = 0

    # 正向传递编码器
    encoder_outputs, encoder_hidden = encoder(input_variable, lengths)

    # 创建初始解码器输入（从每个句子的SOS令牌开始）
    decoder_input = torch.LongTensor([[SOS_token for _ in range(batch_size)]])
    decoder_input = decoder_input.to(device)

    # 将初始解码器隐藏状态设置为编码器的最终隐藏状态
    decoder_hidden = encoder_hidden[:decoder.n_layers]

    # 确定我们是否此次迭代使用`teacher forcing`
    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else
False

    # 通过解码器一次一步地转发一批序列
    if use_teacher_forcing:
        for t in range(max_target_len):
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden, encoder_outputs
            )
            # Teacher forcing: 下一个输入是当前的目标
            decoder_input = target_variable[t].view(1, -1)
            # 计算并累计损失
            mask_loss, nTotal = maskNLLLoss(decoder_output, target_variable[t],
```

```
mask[t])
            loss += mask_loss
            print_losses.append(mask_loss.item() * nTotal)
            n_totals += nTotal
    else:
        for t in range(max_target_len):
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden, encoder_outputs
            )
            # No teacher forcing: 下一个输入是解码器自己的当前输出
            _, topi = decoder_output.topk(1)
            decoder_input = torch.LongTensor([[topi[i][0] for i in
range(batch_size)]])
            decoder_input = decoder_input.to(device)
            # 计算并累计损失
            mask_loss, nTotal = maskNLLLoss(decoder_output, target_variable[t],
mask[t])
            loss += mask_loss
            print_losses.append(mask_loss.item() * nTotal)
            n_totals += nTotal

    # 执行反向传播
    loss.backward()

    # 剪辑梯度：梯度被修改到位
    _ = torch.nn.utils.clip_grad_norm_(encoder.parameters(), clip)
    _ = torch.nn.utils.clip_grad_norm_(decoder.parameters(), clip)

    # 调整模型权重
    encoder_optimizer.step()
    decoder_optimizer.step()

    return sum(print_losses) / n_totals
```

5.3 训练迭代

现在终于将完整的训练步骤与数据结合在一起了。给定传递的模型、优化器、数据等，`trainIters` 函数负责运行 `n_iterations` 的训练。 这个功能显而易见，因为我们通过 `train` 函数的完成了繁重工作。

需要注意的一点是，当我们保存模型时，我们会保存一个包含编码器和解码器state_dicts（参数）、优化器的state_dicts、损失、迭代等的 压缩包。以这种方式保存模型将为我们checkpoint,提供最大的灵活性。加载checkpoint后，我们将能够使用模型参数进行推理，或者我们可 以在我们中断的地方继续训练。

```
def trainIters(model_name, voc, pairs, encoder, decoder, encoder_optimizer,
decoder_optimizer, embedding, encoder_n_layers, decoder_n_layers, save_dir,
n_iteration, batch_size, print_every, save_every, clip, corpus_name,
loadFilename):

    # 为每次迭代加载batches
    training_batches = [batch2TrainData(voc, [random.choice(pairs) for _ in
range(batch_size)])
                        for _ in range(n_iteration)]

    # 初始化
    print('Initializing ...')
    start_iteration = 1
    print_loss = 0
    if loadFilename:
        start_iteration = checkpoint['iteration'] + 1

    # 训练循环
    print("Training...")
    for iteration in range(start_iteration, n_iteration + 1):
        training_batch = training_batches[iteration - 1]
        # 从batch中提取字段
        input_variable, lengths, target_variable, mask, max_target_len =
training_batch

        # 使用batch运行训练迭代
        loss = train(input_variable, lengths, target_variable, mask,
max_target_len, encoder,
                     decoder, embedding, encoder_optimizer, decoder_optimizer,
batch_size, clip)
        print_loss += loss

        # 打印进度
        if iteration % print_every == 0:
            print_loss_avg = print_loss / print_every
            print("Iteration: {}; Percent complete: {:.1f}%; Average loss: {:.
4f}".format(iteration, iteration / n_iteration * 100, print_loss_avg))
            print_loss = 0

        # 保存checkpoint
        if (iteration % save_every == 0):
            directory = os.path.join(save_dir, model_name, corpus_name, '{}-{}
_{}'.format(encoder_n_layers, decoder_n_layers, hidden_size))
            if not os.path.exists(directory):
                os.makedirs(directory)
            torch.save({
                'iteration': iteration,
                'en': encoder.state_dict(),
```

```
                'de': decoder.state_dict(),
                'en_opt': encoder_optimizer.state_dict(),
                'de_opt': decoder_optimizer.state_dict(),
                'loss': loss,
                'voc_dict': voc.__dict__,
                'embedding': embedding.state_dict()
            }, os.path.join(directory, '{}_{}.tar'.format(iteration,
'checkpoint')))
```

6.评估定义

在训练模型后，我们希望能够自己与机器人交谈。首先，我们必须定义我们希望模型如何解码编码输入。

6.1 贪婪解码

贪婪解码是我们在不使用 teacher forcing 时在训练期间使用的解码方法。换句话说，对于每一步，我们只需从具有最高 softmax 值的 decoder_output 中选择单词。该解码方法在单步长级别上是最佳的。

为了便于贪婪解码操作，我们定义了一个 `GreedySearchDecoder` 类。当运行时，类的实例化对象输入序列（ `input_seq` ）的大小是 `(input_seq length, 1)` ，标量输入（ `input_length` ）长度的张量和 max_length 来约束响应句子长度。使用以下计算图来评估输入句子：

- 计算图

1.通过编码器模型前向计算。
2.准备编码器的最终隐藏层，作为解码器的第一个隐藏输入。
3.将解码器的第一个输入初始化为 SOS_token。
4.将初始化张量追加到解码后的单词中。
5.一次迭代解码一个单词token：
  (i)通过解码器进行前向计算。
  (ii)获得最可能的单词token及其softmax分数。
  (iii)记录token和分数。
  (iv)准备当前token作为下一个解码器的输入。
6.返回收集到的词 tokens 和分数。

```
class GreedySearchDecoder(nn.Module):
    def __init__(self, encoder, decoder):
```

```
        super(GreedySearchDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, input_seq, input_length, max_length):
        # 通过编码器模型转发输入
        encoder_outputs, encoder_hidden = self.encoder(input_seq, input_length)
        # 准备编码器的最终隐藏层作为解码器的第一个隐藏输入
        decoder_hidden = encoder_hidden[:decoder.n_layers]
        # 使用SOS_token初始化解码器输入
        decoder_input = torch.ones(1, 1, device=device, dtype=torch.long) *
SOS_token
        # 初始化张量以将解码后的单词附加到
        all_tokens = torch.zeros([0], device=device, dtype=torch.long)
        all_scores = torch.zeros([0], device=device)
        # 一次迭代地解码一个词tokens
        for _ in range(max_length):
            # 正向通过解码器
            decoder_output, decoder_hidden = self.decoder(decoder_input,
decoder_hidden, encoder_outputs)
            # 获得最可能的单词标记及其softmax分数
            decoder_scores, decoder_input = torch.max(decoder_output, dim=1)
            # 记录token和分数
            all_tokens = torch.cat((all_tokens, decoder_input), dim=0)
            all_scores = torch.cat((all_scores, decoder_scores), dim=0)
            # 准备当前令牌作为下一个解码器输入（添加维度）
            decoder_input = torch.unsqueeze(decoder_input, 0)
        # 返回收集到的词tokens和分数
        return all_tokens, all_scores
```

6.2 评估我们的文本

现在我们已经定义了解码方法，我们可以编写用于评估字符串输入句子的函数。 `evaluate` 函数管理输入句子的低层级处理过程。我们首先使 用*batch_size == 1*将句子格式化为输入batch的单词索引。我们通过将句子的单词转换为相应的索引，并通过转换维度来为我们的模型准备 张量。我们还创建了一个 `lengths` 张量，其中包含输入句子的长度。在这种情况下， `lengths` 是标量，因为我们一次只评估一个句子（*batch_size == 1*）。 接下来，我们使用我们的 `GreedySearchDecoder` 实例化后的对象（searcher）获得解码响应句子的张量。最后，我们将响应的索引转换为单 词并返回已解码单词的列表。

`evaluateInput` 充当聊天机器人的用户接口。调用时，将生成一个输入文本字段，我们可以在其中输入查询语句。在输入我们的输入句子并 按 Enter 后，我们的文本以与训练数据相同的方式标准化，并最终被输入到评估函数以获得解码的输出句子。我们循环这个过程，这样我们可 以继续与我们的机器人聊天直到我们输入“q”或“quit”。

最后，如果输入的句子包含一个不在词汇表中的单词，我们会通过打印错误消息并提示用户输入另一个句子来优雅地处理。

```
def evaluate(encoder, decoder, searcher, voc, sentence, max_length=MAX_LENGTH):
    ### 格式化输入句子作为batch
    # words -> indexes
    indexes_batch = [indexesFromSentence(voc, sentence)]
    # 创建lengths张量
    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])
    # 转置batch的维度以匹配模型的期望
    input_batch = torch.LongTensor(indexes_batch).transpose(0, 1)
    # 使用合适的设备
    input_batch = input_batch.to(device)
    lengths = lengths.to(device)
    # 用searcher解码句子
    tokens, scores = searcher(input_batch, lengths, max_length)
    # indexes -> words
    decoded_words = [voc.index2word[token.item()] for token in tokens]
    return decoded_words


def evaluateInput(encoder, decoder, searcher, voc):
    input_sentence = ''
    while(1):
        try:
            # 获取输入句子
            input_sentence = input('> ')
            # 检查是否退出
            if input_sentence == 'q' or input_sentence == 'quit': break
            # 规范化句子
            input_sentence = normalizeString(input_sentence)
            # 评估句子
            output_words = evaluate(encoder, decoder, searcher, voc,
input_sentence)
            # 格式化和打印回复句
            output_words[:] = [x for x in output_words if not (x == 'EOS' or x ==
'PAD')]
            print('Bot:', ' '.join(output_words))

        except KeyError:
            print("Error: Encountered unknown word.")
```

7.运行模型

最后，是时候运行我们的模型了！

无论我们是否想要训练或测试聊天机器人模型，我们都必须初始化各个编码器和解码器模型。在接下来的部分，我们设置所需要的配置，选择 从头开始或设置检查点以从中加载，并构建和初始化模型。您可以随意使用不同的配置来优化性能。

```
# 配置模型
model_name = 'cb_model'
attn_model = 'dot'
#attn_model = 'general'
#attn_model = 'concat'
hidden_size = 500
encoder_n_layers = 2
decoder_n_layers = 2
dropout = 0.1
batch_size = 64

# 设置检查点以加载；如果从头开始，则设置为None
loadFilename = None
checkpoint_iter = 4000
#loadFilename = os.path.join(save_dir, model_name, corpus_name,
#                            '{}-{}_{}'.format(encoder_n_layers,
decoder_n_layers, hidden_size),
#                            '{}_checkpoint.tar'.format(checkpoint_iter))

# 如果提供了loadFilename，则加载模型
if loadFilename:
    # 如果在同一台机器上加载，则对模型进行训练
    checkpoint = torch.load(loadFilename)
    # If loading a model trained on GPU to CPU
    #checkpoint = torch.load(loadFilename, map_location=torch.device('cpu'))
    encoder_sd = checkpoint['en']
    decoder_sd = checkpoint['de']
    encoder_optimizer_sd = checkpoint['en_opt']
    decoder_optimizer_sd = checkpoint['de_opt']
    embedding_sd = checkpoint['embedding']
    voc.__dict__ = checkpoint['voc_dict']

print('Building encoder and decoder ...')
# 初始化词向量
embedding = nn.Embedding(voc.num_words, hidden_size)
if loadFilename:
    embedding.load_state_dict(embedding_sd)
```

```
# 初始化编码器 & 解码器模型
encoder = EncoderRNN(hidden_size, embedding, encoder_n_layers, dropout)
decoder = LuongAttnDecoderRNN(attn_model, embedding, hidden_size, voc.num_words,
decoder_n_layers, dropout)
if loadFilename:
    encoder.load_state_dict(encoder_sd)
    decoder.load_state_dict(decoder_sd)
# 使用合适的设备
encoder = encoder.to(device)
decoder = decoder.to(device)
print('Models built and ready to go!')
```

- 输出结果

```
Building encoder and decoder ...
Models built and ready to go!
```

7.1 执行训练

如果要训练模型，请运行以下部分。

首先我们设置训练参数，然后初始化我们的优化器，最后我们调用 `trainIters` 函数来运行我们的训练迭代。

```
# 配置训练/优化
clip = 50.0
teacher_forcing_ratio = 1.0
learning_rate = 0.0001
decoder_learning_ratio = 5.0
n_iteration = 4000
print_every = 1
save_every = 500

# 确保dropout layers在训练模型中
encoder.train()
decoder.train()

# 初始化优化器
print('Building optimizers ...')
encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)
decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate *
decoder_learning_ratio)
if loadFilename:
    encoder_optimizer.load_state_dict(encoder_optimizer_sd)
    decoder_optimizer.load_state_dict(decoder_optimizer_sd)
```

```
# 运行训练迭代
print("Starting Training!")
trainIters(model_name, voc, pairs, encoder, decoder, encoder_optimizer,
decoder_optimizer,
           embedding, encoder_n_layers, decoder_n_layers, save_dir, n_iteration,
batch_size,
           print_every, save_every, clip, corpus_name, loadFilename)
```

- 输出结果：

```
Building optimizers ...
Starting Training!
Initializing ...
Training...
Iteration: 1; Percent complete: 0.0%; Average loss: 8.9717
Iteration: 2; Percent complete: 0.1%; Average loss: 8.8521
Iteration: 3; Percent complete: 0.1%; Average loss: 8.6360
Iteration: 4; Percent complete: 0.1%; Average loss: 8.4234
Iteration: 5; Percent complete: 0.1%; Average loss: 7.9403
Iteration: 6; Percent complete: 0.1%; Average loss: 7.3892
Iteration: 7; Percent complete: 0.2%; Average loss: 7.0589
Iteration: 8; Percent complete: 0.2%; Average loss: 7.0130
Iteration: 9; Percent complete: 0.2%; Average loss: 6.7383
Iteration: 10; Percent complete: 0.2%; Average loss: 6.5343
...
Iteration: 3991; Percent complete: 99.8%; Average loss: 2.6607
Iteration: 3992; Percent complete: 99.8%; Average loss: 2.6188
Iteration: 3993; Percent complete: 99.8%; Average loss: 2.8319
Iteration: 3994; Percent complete: 99.9%; Average loss: 2.5817
Iteration: 3995; Percent complete: 99.9%; Average loss: 2.4979
Iteration: 3996; Percent complete: 99.9%; Average loss: 2.7317
Iteration: 3997; Percent complete: 99.9%; Average loss: 2.5969
Iteration: 3998; Percent complete: 100.0%; Average loss: 2.2275
Iteration: 3999; Percent complete: 100.0%; Average loss: 2.7124
Iteration: 4000; Percent complete: 100.0%; Average loss: 2.5975
```

7.2 运行评估

要与您的模型聊天，请运行以下代码：

```
# 将dropout layers设置为eval模式
encoder.eval()
decoder.eval()

# 初始化探索模块
```

```
searcher = GreedySearchDecoder(encoder, decoder)

# 开始聊天（取消注释并运行以下行开始）
# evaluateInput(encoder, decoder, searcher, voc)
```

8.结论

到这里，就是教程的全部了。恭喜，您现在知道构建生成聊天机器人模型的基础知识！如果您有兴趣，可以尝试通过调整模型和训练参数以及 自定义训练模型的数据来定制聊天机器人的行为。

查看其他教程，了解PyTorch中更酷的深度学习应用程序！

使用字符级RNN生成名字

在上一个教程中，中我们使用RNN网络对名字所属的语言进行分类。这一次我们会反过来根据语言生成名字。

```
> python sample.py Russian RUS
Rovakov
Uantov
Shavakov

> python sample.py German GER
Gerren
Ereng
Rosher

> python sample.py Spanish SPA
Salla
Parer
Allan

> python sample.py Chinese CHI
Chan
Hang
Iun
```

我们仍使用只有几层线性层的小型RNN。最大的区别在于，这里不是在读取一个名字的所有字母后预测类别，而是输入一个类别之后在每一时刻 输出一个字母。循环预测字符以形成语言通常也被称为“语言模型”。（也可以将字符换成单词或更高级的结构进行这一过程）

- **阅读建议**

开始本教程前，你已经安装好了PyTorch，并熟悉Python语言，理解“张量”的概念：

- https://pytorch.org/ PyTorch 安装指南
- Deep Learning with PyTorch：A 60 Minute Blitz :PyTorch的基本入门教程
- Learning PyTorch with Examples:得到深层而广泛的概述
- PyTorch for Former Torch Users Lua Torch:如果你曾是一个Lua张量的使用者

事先学习并了解RNN的工作原理对理解这个例子十分有帮助:

- The Unreasonable Effectiveness of Recurrent Neural Networks展示了很多实际的例子
- Understanding LSTM Networks是关于LSTM的，但也提供有关RNN的说明

1.准备数据

点击这里下载数据并将其解压到当前文件夹。

有关此过程的更多详细信息，请参阅上一个教程。简而言之，有一些纯文本文件 `data/names/[Language].txt` ，它们的每行都有一个名字。 我们按行将文本按行分割得到一个数组，将Unicode编码转化为ASCII编码，最终得到 `{language: [names ...]}` 格式存储的字典变量。

```
from __future__ import unicode_literals, print_function, division
from io import open
import glob
import os
import unicodedata
import string

all_letters = string.ascii_letters + " .,;'-"
n_letters = len(all_letters) + 1 # Plus EOS marker

def findFiles(path): return glob.glob(path)

# 将Unicode字符串转换为纯ASCII, 感谢https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )

# 读取文件并分成几行
def readLines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split('\n')
    return [unicodeToAscii(line) for line in lines]

# 构建category_lines字典, 列表中的每行是一个类别
category_lines = {}
all_categories = []
for filename in findFiles('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
```

```
    all_categories.append(category)
    lines = readLines(filename)
    category_lines[category] = lines

n_categories = len(all_categories)

if n_categories == 0:
    raise RuntimeError('Data not found. Make sure that you downloaded data '
        'from https://download.pytorch.org/tutorial/data.zip and extract it to '
        'the current directory.')

print('# categories:', n_categories, all_categories)
print(unicodeToAscii("O'Néàl"))
```

- 输出结果

```
# categories: 18 ['French', 'Czech', 'Dutch', 'Polish', 'Scottish', 'Chinese',
'English', 'Italian', 'Portuguese', 'Japanese', 'German', 'Russian', 'Korean',
'Arabic', 'Greek', 'Vietnamese', 'Spanish', 'Irish']
O'Neal
```

2.构造神经网络

这个神经网络比上一个RNN教程 中的网络增加了额外的类别张量参数，该参数与其他输入连接在一起。类别可以像字母一样组成 one-hot 向量构成张量输入。

我们将输出作为下一个字母是什么的可能性。采样过程中，当前输出可能性最高的字母作为下一时刻输入字母。

在组合隐藏状态和输出之后我们增加了第二个linear层 `o2o` ，使模型的性能更好。当然还有一个dropout层，参考这篇论文随机将输入部分替换为0 给出的参数（dropout=0.1）来模糊处理输入防止过拟合。 我们将它添加到网络的末端，故意添加一些混乱使采样特征增加。

```
import torch
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size

        self.i2h = nn.Linear(n_categories + input_size + hidden_size, hidden_size)
```

```
        self.i2o = nn.Linear(n_categories + input_size + hidden_size, output_size)
        self.o2o = nn.Linear(hidden_size + output_size, output_size)
        self.dropout = nn.Dropout(0.1)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, category, input, hidden):
        input_combined = torch.cat((category, input, hidden), 1)
        hidden = self.i2h(input_combined)
        output = self.i2o(input_combined)
        output_combined = torch.cat((hidden, output), 1)
        output = self.o2o(output_combined)
        output = self.dropout(output)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```

3.训练

3.1 训练准备

首先，构造一个可以随机获取成对训练数据(category, line)的函数。

```
import random

# 列表中的随机项
def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

# 从该类别中获取随机类别和随机行
def randomTrainingPair():
    category = randomChoice(all_categories)
    line = randomChoice(category_lines[category])
    return category, line
```

对于每个时间步长（即，对于要训练单词中的每个字母），网络的输入将是“`（类别，当前字母，隐藏状态）`”，输出将是“`（下一个字母，下一个隐藏状态）`”。因此，对于每个训练集，我们将需要类别、一组输入字母和一组输出/目标字母。

在每一个时间序列，我们使用当前字母预测下一个字母，所以训练用的字母对来自于一个单词。例如 对于 “ `ABCD<EOS>` “，我们将创建 (“A”，“B”)，（“B”，“C”），（“C”，“D”），（“D”，“EOS”））。

类别张量是一个 `<1 x n_categories>` 尺寸的one-hot张量。训练时，我们在每一个时间序 列都将其提供给神经网络。这是一种选择策略，也可选择将其作为初始隐藏状态的一部分，或者其他什么结构。

```
# 类别的One-hot张量
def categoryTensor(category):
    li = all_categories.index(category)
    tensor = torch.zeros(1, n_categories)
    tensor[0][li] = 1
    return tensor

# 用于输入的从头到尾字母（不包括EOS）的one-hot矩阵
def inputTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters)
    for li in range(len(line)):
        letter = line[li]
        tensor[li][0][all_letters.find(letter)] = 1
    return tensor

# 用于目标的第二个结束字母（EOS）的LongTensor
def targetTensor(line):
    letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
    letter_indexes.append(n_letters - 1) # EOS
    return torch.LongTensor(letter_indexes)
```

为了方便训练，我们将创建一个 `randomTrainingExample` 函数，该函数随机获取（类别，行）的对并将它们转换为所需要的（类别，输入，目标）格式张量。

```
# 从随机(类别, 行)对中创建类别, 输入和目标张量
def randomTrainingExample():
    category, line = randomTrainingPair()
    category_tensor = categoryTensor(category)
    input_line_tensor = inputTensor(line)
    target_line_tensor = targetTensor(line)
    return category_tensor, input_line_tensor, target_line_tensor
```

3.2 训练神经网络

和只使用最后一个时刻输出的分类任务相比，这次我们每一个时间序列都会进行一次预测，所以每一个时间序列我们都会计算损失。

autograd 的神奇之处在于您可以在每一步中简单地累加这些损失，并在最后反向传播。

```
criterion = nn.NLLLoss()

learning_rate = 0.0005

def train(category_tensor, input_line_tensor, target_line_tensor):
    target_line_tensor.unsqueeze_(-1)
    hidden = rnn.initHidden()

    rnn.zero_grad()

    loss = 0

    for i in range(input_line_tensor.size(0)):
        output, hidden = rnn(category_tensor, input_line_tensor[i], hidden)
        l = criterion(output, target_line_tensor[i])
        loss += l

    loss.backward()

    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.item() / input_line_tensor.size(0)
```

为了跟踪训练耗费的时间，我添加一个 `timeSince (timestamp)` 函数，它返回一个人类可读的字符串：

```
import time
import math

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)
```

训练过程和平时一样。多次运行训练，等待几分钟，每 `print_every` 次打印当前时间和损失。在 `all_losses` 中保留每 `plot_every` 次的平 均损失，以便稍后进行绘图。

```
rnn = RNN(n_letters, 128, n_letters)

n_iters = 100000
print_every = 5000
plot_every = 500
all_losses = []
total_loss = 0 # Reset every plot_every iters

start = time.time()

for iter in range(1, n_iters + 1):
    output, loss = train(*randomTrainingExample())
    total_loss += loss

    if iter % print_every == 0:
        print('%s (%d %d%%) %.4f' % (timeSince(start), iter, iter / n_iters *
100, loss))

    if iter % plot_every == 0:
        all_losses.append(total_loss / plot_every)
        total_loss = 0
```

- 输出结果：

```
0m 23s (5000 5%) 3.1569
0m 43s (10000 10%) 2.3132
1m 3s (15000 15%) 2.5069
1m 24s (20000 20%) 1.3100
1m 44s (25000 25%) 3.6083
2m 4s (30000 30%) 3.5398
2m 24s (35000 35%) 2.4387
2m 44s (40000 40%) 2.2262
3m 4s (45000 45%) 2.6500
3m 24s (50000 50%) 2.4559
3m 44s (55000 55%) 2.5030
4m 4s (60000 60%) 2.9417
4m 24s (65000 65%) 2.1571
4m 44s (70000 70%) 1.7415
5m 4s (75000 75%) 2.3649
5m 24s (80000 80%) 3.0096
5m 44s (85000 85%) 1.9196
6m 4s (90000 90%) 1.9468
```

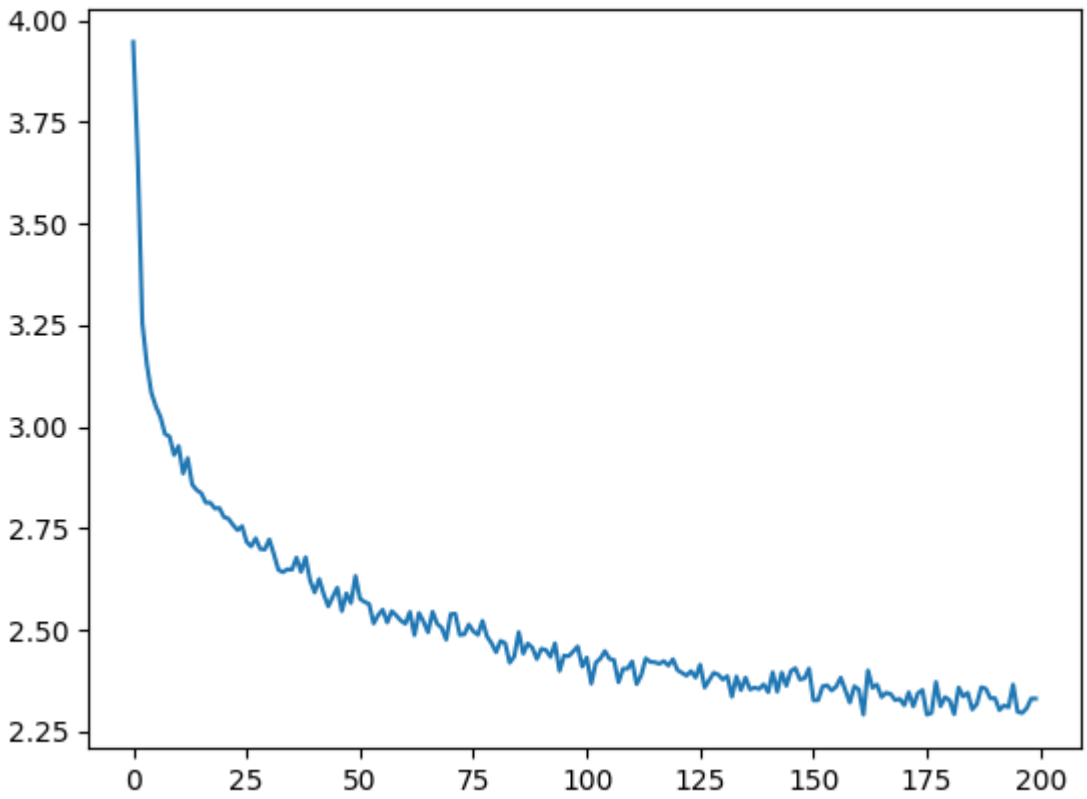
```
6m 25s (95000 95%) 2.1522
6m 45s (100000 100%) 2.0344
```

3.3 损失数据作图

从 `all_losses` 得到历史损失记录，反映了神经网络的学习情况：

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses)
```



4.网络采样

我们每次给网络提供一个字母并预测下一个字母是什么，将预测到的字母继续输入，直到得到EOS字符结束循环。

- 用输入类别、起始字母和空隐藏状态创建输入张量。
- 用起始字母构建一个字符串变量 output_name
- 得到最大输出长度，
    * 将当前字母传入神经网络
    * 从前一层得到下一个字母和下一个隐藏状态
    * 如果字母是EOS，在这里停止
    * 如果是一个普通的字母，添加到output_name变量并继续循环
- 返回最终得到的名字单词

另一种策略是，不必给网络一个起始字母，而是在训练中提供一个“字符串开始”的标记，并让网络自己选择起始的字母。

```
max_length = 20

# 来自类别和首字母的样本
def sample(category, start_letter='A'):
    with torch.no_grad():  # no need to track history in sampling
        category_tensor = categoryTensor(category)
        input = inputTensor(start_letter)
        hidden = rnn.initHidden()

        output_name = start_letter

        for i in range(max_length):
            output, hidden = rnn(category_tensor, input[0], hidden)
            topv, topi = output.topk(1)
            topi = topi[0][0]
            if topi == n_letters - 1:
                break
            else:
                letter = all_letters[topi]
                output_name += letter
            input = inputTensor(letter)

        return output_name
```

```
# 从一个类别和多个起始字母中获取多个样本
def samples(category, start_letters='ABC'):
    for start_letter in start_letters:
        print(sample(category, start_letter))

samples('Russian', 'RUS')

samples('German', 'GER')

samples('Spanish', 'SPA')

samples('Chinese', 'CHI')
```

- 输出结果：

```
Rovanik
Uakilovev
Shaveri
Garter
Eren
Romer
Santa
Parera
Artera
Chan
Ha
Iua
```

练习

- 尝试其它（类别->行）格式的数据集，比如:
    * 系列小说 -> 角色名称
    * 词性 -> 单词
    * 国家 -> 城市
- 尝试“start of sentence” 标记，使采样的开始过程不需要指定起始字母
- 通过更大和更复杂的网络获得更好的结果
    * 尝试 nn.LSTM 和 nn.GRU 层
    * 组合这些 RNN构造更复杂的神经网络

使用字符级RNN进行名字分类

我们将构建和训练字符级RNN来对单词进行分类。字符级RNN将单词作为一系列字符读取，在每一步输出预测和“隐藏状态”，将其先前的隐藏 状态输入至下一时刻。我们将最终时刻输出作为预测结果，即表示该词属于哪个类。

具体来说，我们将在18种语言构成的几千个名字的数据集上训练模型，根据一个名字的拼写预测它是哪种语言的名字：

```
$ python predict.py Hinton
(-0.47) Scottish
(-1.52) English
(-3.57) Irish

$ python predict.py Schmidhuber
(-0.19) German
(-2.48) Czech
(-2.68) Dutch
```

- **阅读建议**

开始本教程前，你已经安装好了PyTorch，并熟悉Python语言，理解“张量”的概念：

- https://pytorch.org/ PyTorch 安装指南
- Deep Learning with PyTorch：A 60 Minute Blitz :PyTorch的基本入门教程
- Learning PyTorch with Examples:得到深层而广泛的概述
- PyTorch for Former Torch Users Lua Torch:如果你曾是一个Lua张量的使用者

事先学习并了解RNN的工作原理对理解这个例子十分有帮助:

- The Unreasonable Effectiveness of Recurrent Neural Networks展示了很多实际的例子
- Understanding LSTM Networks是关于LSTM的，但也提供有关RNN的说明

1.准备数据

点击这里下载数据，并将其解压到当前文件夹。

在”`data/names`“文件夹下是名称为”[language].txt”的18个文本文件。每个文件的每一行都有一个名字，它们几乎都是罗马化的文本（但是我们仍需要将其从Unicode转换为ASCII编码）

我们最终会得到一个语言对应名字列表的字典，`{language: [names ...]}`。通用变量“category”和“line”（例子中的语言和名字单词）用于以后的可扩展性。

```
from __future__ import unicode_literals, print_function, division
from io import open
import glob
import os

def findFiles(path): return glob.glob(path)

print(findFiles('data/names/*.txt'))

import unicodedata
import string

all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)

# 将Unicode字符串转换为纯ASCII，感谢https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )

print(unicodeToAscii('Ślusàrski'))

# 构建category_lines字典，每种语言的名字列表
category_lines = {}
all_categories = []

# 读取文件并分成几行
def readLines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split('\n')
    return [unicodeToAscii(line) for line in lines]

for filename in findFiles('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    all_categories.append(category)
    lines = readLines(filename)
    category_lines[category] = lines

n_categories = len(all_categories)
```

- 输出结果：

```
['data/names/French.txt', 'data/names/Czech.txt', 'data/names/Dutch.txt', 'data/
names/Polish.txt', 'data/names/Scottish.txt', 'data/names/Chinese.txt', 'data/
names/English.txt', 'data/names/Italian.txt', 'data/names/Portuguese.txt', 'data/
```

```
names/Japanese.txt', 'data/names/German.txt', 'data/names/Russian.txt', 'data/
names/Korean.txt', 'data/names/Arabic.txt', 'data/names/Greek.txt', 'data/names/
Vietnamese.txt', 'data/names/Spanish.txt', 'data/names/Irish.txt']
Slusarski
```

现在我们有了 `category_lines` ，一个字典变量存储每一种语言及其对应的每一行文本(名字)列表的映射关系。变量 `all_categories` 是全部 语言种类的列表，变量 `n_categories` 是语言种类的数量，后续会使用。

```
print(category_lines['Italian'][:5])
```

- 输出结果：

```
['Abandonato', 'Abatangelo', 'Abatantuono', 'Abate', 'Abategiovanni']
```

单词转变为张量

现在我们已经加载了所有的名字，我们需要将它们转换为张量来使用它们。

我们使用大小为 `<1 x n_letters>` 的“one-hot 向量”表示一个字母。一个one-hot向量所有位置都填充为0，并在其表示的字母的位置表示为1， 例如 `"b" = <0 1 0 0 0 ...>` .（字母b的编号是2，第二个位置是1，其他位置是0）

我们使用一个 `<line_length x 1 x n_letters>` 的2D矩阵表示一个单词

额外的1维是batch的维度，PyTorch默认所有的数据都是成batch处理的。我们这里只设置了batch的大小为1。

```
import torch

# 从all_letters中查找字母索引，例如 "a" = 0
def letterToIndex(letter):
    return all_letters.find(letter)

# 仅用于演示，将字母转换为<1 x n_letters> 张量
def letterToTensor(letter):
    tensor = torch.zeros(1, n_letters)
    tensor[0][letterToIndex(letter)] = 1
    return tensor

# 将一行转换为<line_length x 1 x n_letters>，
# 或一个0ne-hot字母向量的数组
```

```
def lineToTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters)
    for li, letter in enumerate(line):
        tensor[li][0][letterToIndex(letter)] = 1
    return tensor

print(letterToTensor('J'))

print(lineToTensor('Jones').size())
```

- 输出结果：

```
tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
1.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
         0., 0., 0.]])
torch.Size([5, 1, 57])
```

2.构造神经网络

在autograd之前，要在Torch中构建一个可以复制之前时刻层参数的循环神经网络。layer的隐藏状态和梯度将交给计算图自己处理。这意味着 你可以像实现的常规的 feed-forward 层一样，以很纯粹的方式实现RNN。

这个RNN组件 (几乎是从这里复制的the PyTorch for Torch users tutorial) 仅使用两层 linear 层对输入和隐藏层做处理,在最后添加一层 LogSoftmax 层预测最终输出。

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
```

```
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

n_hidden = 128
rnn = RNN(n_letters, n_hidden, n_categories)
```

要运行此网络的一个步骤，我们需要传递一个输入（在我们的例子中，是当前字母的Tensor）和一个先前隐藏的状态（我们首先将其初始化为零）。 我们将返回输出（每种语言的概率）和下一个隐藏状态（为我们下一步保留使用）。

```
input = letterToTensor('A')
hidden =torch.zeros(1, n_hidden)

output, next_hidden = rnn(input, hidden)
```

为了提高效率，我们不希望为每一步都创建一个新的Tensor，因此我们将使用 `lineToTensor` 函数而不是 `letterToTensor` 函数，并使用切片 方法。这一步可以通过预先计算批量的张量进一步优化。

```
input = lineToTensor('Albert')
hidden = torch.zeros(1, n_hidden)

output, next_hidden = rnn(input[0], hidden)
print(output)
```

- 输出结果：

```
tensor([[-2.8857, -2.9005, -2.8386, -2.9397, -2.8594, -2.8785, -2.9361, -2.8270,
         -2.9602, -2.8583, -2.9244, -2.9112, -2.8545, -2.8715, -2.8328, -2.8233,
         -2.9685, -2.9780]], grad_fn=<LogSoftmaxBackward>)
```

可以看到输出是一个 `<1 x n_categories>` 的张量，其中每一条代表这个单词属于某一类的可能性（越高可能性越大）。

2.训练

2.1 训练前的准备

进行训练步骤之前我们需要构建一些辅助函数。* 第一个是当我们知道输出结果对应每种类别的可能性时，解析神经网络的输出。我们可以使用 `Tensor.topk` 函数得到最大值在结果中的位置索引：

```
def categoryFromOutput(output):
    top_n, top_i = output.topk(1)
    category_i = top_i[0].item()
    return all_categories[category_i], category_i

print(categoryFromOutput(output))
```

- 输出结果：

```
('Arabic', 13)
```

- 第二个是我们需要一种快速获取训练示例（得到一个名字及其所属的语言类别）的方法：

```
import random

def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

def randomTrainingExample():
    category = randomChoice(all_categories)
    line = randomChoice(category_lines[category])
    category_tensor = torch.tensor([all_categories.index(category)],
dtype=torch.long)
    line_tensor = lineToTensor(line)
    return category, line, category_tensor, line_tensor

for i in range(10):
    category, line, category_tensor, line_tensor = randomTrainingExample()
    print('category =', category, '/ line =', line)
```

- 输出结果：

```
category = Dutch / line = Tholberg
category = Irish / line = Murphy
```

```
category = Vietnamese / line = An
category = German / line = Von essen
category = Polish / line = Kijek
category = Scottish / line = Bell
category = Czech / line = Marik
category = Korean / line = Jeong
category = Korean / line = Choe
category = Portuguese / line = Alves
```

2.2 训练神经网络

现在，训练过程只需要向神经网络输入大量的数据，让它做出预测，并将对错反馈给它。

`nn.LogSoftmax` 作为最后一层layer时， `nn.NLLLoss` 作为损失函数是合适的。

```
criterion = nn.NLLLoss()
```

训练过程的每次循环将会发生：

- 构建输入和目标张量
- 构建0初始化的隐藏状态
- 读入每一个字母
    * 将当前隐藏状态传递给下一字母
- 比较最终结果和目标
- 反向传播
- 返回结果和损失

```
learning_rate = 0.005 # If you set this too high, it might explode. If too low, it might not learn

def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()

    rnn.zero_grad()

    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    loss = criterion(output, category_tensor)
    loss.backward()
```

```
    # 将参数的梯度添加到其值中，乘以学习速率
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.item()
```

现在我们只需要准备一些例子来运行程序。由于train函数同时返回输出和损失，我们可以打印其输出结果并跟踪其损失画图。由于有1000个 示例，我们每 `print_every` 次打印样例，并求平均损失。

```
import time
import math

n_iters = 100000
print_every = 5000
plot_every = 1000

# 跟踪绘图的损失
current_loss = 0
all_losses = []

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

start = time.time()

for iter in range(1, n_iters + 1):
    category, line, category_tensor, line_tensor = randomTrainingExample()
    output, loss = train(category_tensor, line_tensor)
    current_loss += loss

    # 打印迭代的编号，损失，名字和猜测
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '✓' if guess == category else '✗ (%s)' % category
        print('%d  %d%% (%s) %.4f  %s / %s  %s' % (iter, iter / n_iters * 100,
timeSince(start), loss, line, guess, correct))

    # 将当前损失平均值添加到损失列表中
    if iter % plot_every == 0:
```

```
        all_losses.append(current_loss / plot_every)
        current_loss = 0
```

- 输出结果：

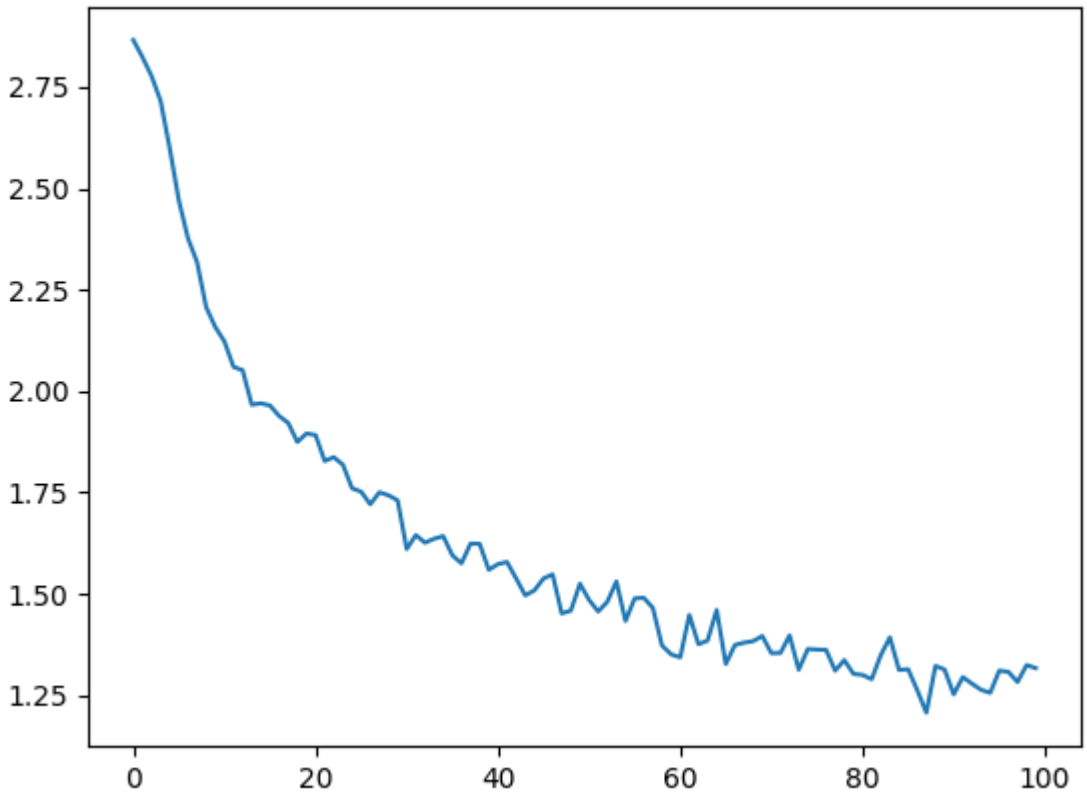
```
5000 5% (0m 8s) 2.7792 Verdon / Scottish ✗ (English)
10000 10% (0m 16s) 2.0748 Campos / Greek ✗ (Portuguese)
15000 15% (0m 25s) 2.0458 Kuang / Vietnamese ✗ (Chinese)
20000 20% (0m 33s) 1.1703 Nghiem / Vietnamese ✓
25000 25% (0m 41s) 2.6035 Boyle / English ✗ (Scottish)
30000 30% (0m 50s) 2.2823 Mozdzierz / Dutch ✗ (Polish)
35000 35% (0m 58s) nan Lagana / Irish ✗ (Italian)
40000 40% (1m 6s) nan Simonis / Irish ✗ (Dutch)
45000 45% (1m 15s) nan Nobunaga / Irish ✗ (Japanese)
50000 50% (1m 23s) nan Ingermann / Irish ✗ (English)
55000 55% (1m 31s) nan Govorin / Irish ✗ (Russian)
60000 60% (1m 39s) nan Janson / Irish ✗ (German)
65000 65% (1m 48s) nan Tsangaris / Irish ✗ (Greek)
70000 70% (1m 56s) nan Vlasenkov / Irish ✗ (Russian)
75000 75% (2m 4s) nan Needham / Irish ✗ (English)
80000 80% (2m 12s) nan Matsoukis / Irish ✗ (Greek)
85000 85% (2m 21s) nan Koo / Irish ✗ (Chinese)
90000 90% (2m 29s) nan Novotny / Irish ✗ (Czech)
95000 95% (2m 37s) nan Dubois / Irish ✗ (French)
100000 100% (2m 45s) nan Padovano / Irish ✗ (Italian)
```

2.3 绘画出结果

从 `all_losses` 得到历史损失记录，反映了神经网络的学习情况：

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses)
```



3.评价结果

为了了解网络在不同类别上的表现，我们将创建一个混淆矩阵，显示每种语言（行）和神经网络将其预测为哪种语言（列）。为了计算混淆矩 阵，使用 `evaluate()` 函数处理了一批数据，`evaluate()` 函数与去掉反向传播的 `train()` 函数大体相同。

```
# 在混淆矩阵中跟踪正确的猜测
confusion = torch.zeros(n_categories, n_categories)
n_confusion = 10000

# 只需返回给定一行的输出
def evaluate(line_tensor):
    hidden = rnn.initHidden()

    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    return output
```

```
# 查看一堆正确猜到的例子和记录
for i in range(n_confusion):
    category, line, category_tensor, line_tensor = randomTrainingExample()
    output = evaluate(line_tensor)
    guess, guess_i = categoryFromOutput(output)
    category_i = all_categories.index(category)
    confusion[category_i][guess_i] += 1

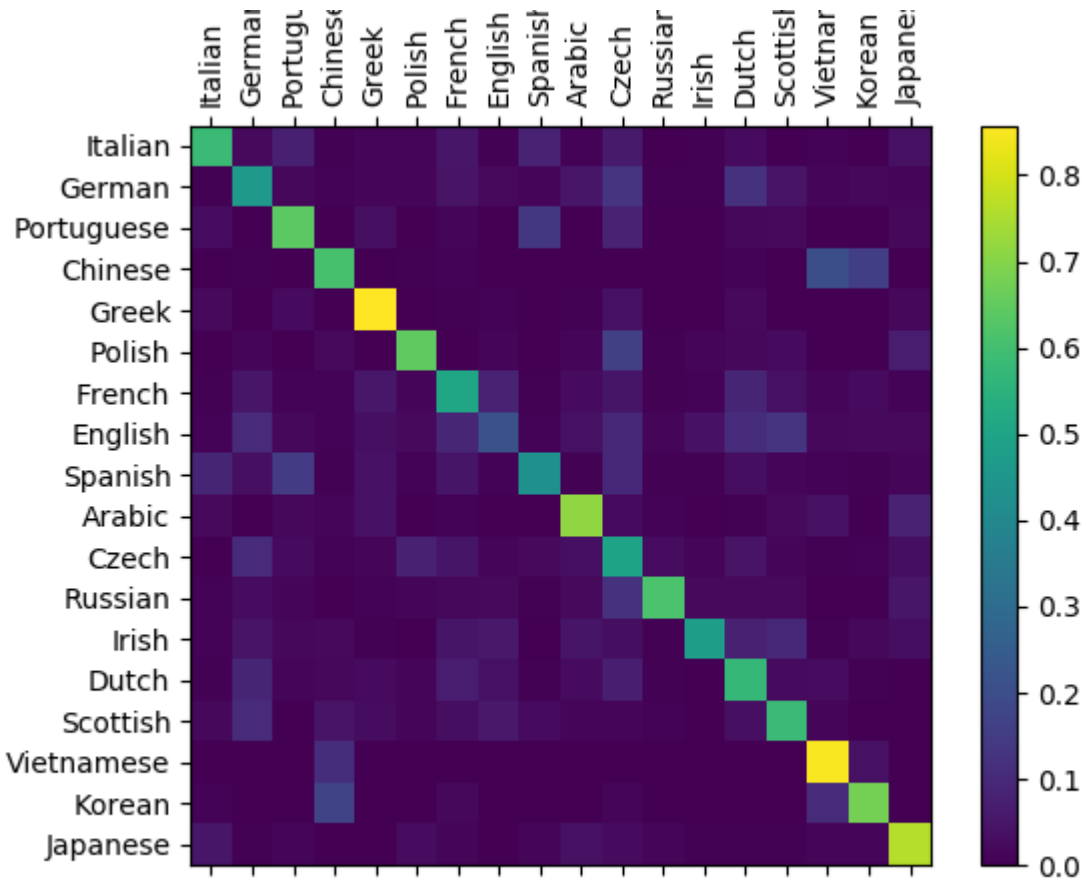
# 通过将每一行除以其总和来归一化
for i in range(n_categories):
    confusion[i] = confusion[i] / confusion[i].sum()

# 设置绘图
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(confusion.numpy())
fig.colorbar(cax)

# 设置轴
ax.set_xticklabels([''] + all_categories, rotation=90)
ax.set_yticklabels([''] + all_categories)

# 每个刻度线强制标签
ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

# sphinx_gallery_thumbnail_number = 2
plt.show()
```



你可以从主轴线以外挑出亮的点，显示模型预测错了哪些语言，例如汉语预测为了韩语，西班牙预测为了意大利。看上去在希腊语上效果很好，在英语上表现欠佳。（可能是因为英语与其他语言的重叠较多）。

处理用户输入

```
def predict(input_line, n_predictions=3):
    print('\n> %s' % input_line)
    with torch.no_grad():
        output = evaluate(lineToTensor(input_line))

        # 获得前N个类别
        topv, topi = output.topk(n_predictions, 1, True)
        predictions = []

        for i in range(n_predictions):
            value = topv[0][i].item()
            category_index = topi[0][i].item()
            print('(%.2f) %s' % (value, all_categories[category_index]))
            predictions.append([value, all_categories[category_index]])
```

```
predict('Dovesky')
predict('Jackson')
predict('Satoshi')
```

- 输出结果：

```
> Dovesky
(-0.74) Russian
(-0.77) Czech
(-3.31) English

> Jackson
(-0.80) Scottish
(-1.69) English
(-1.84) Russian

> Satoshi
(-1.16) Japanese
(-1.89) Arabic
(-1.90) Polish
```

最终版的脚本in the Practical PyTorch repo 将上述代码拆分为几个文件：

- data.py (读取文件)
- model.py (构造RNN网络)
- train.py (运行训练过程)
- predict.py (在命令行中和参数一起运行predict()函数)
- server.py (使用bottle.py构建JSON API的预测服务)

运行 `train.py` 来训练和保存网络

将 `predict.py` 和一个名字的单词一起运行查看预测结果：

```
$ python predict.py Hazaki
(-0.42) Japanese
(-1.39) Polish
(-3.51) Czech
```

运行 `server.py` 并访问http://localhost:5533/Yourname 得到JSON格式的预测输出

4.练习

- 尝试其它（类别->行）格式的数据集，比如:

* 任何单词 -> 语言
* 姓名 -> 性别
* 角色姓名 -> 作者
* 页面标题 -> blog 或 subreddit

- 通过更大和更复杂的网络获得更好的结果
    * 增加更多linear层
    * 尝试 nn.LSTM 和 nn.GRU 层
    * 组合这些 RNN构造更复杂的神经网络

在深度学习和 NLP 中使用 Pytorch

本文带您进入pytorch框架进行深度学习编程的核心思想。Pytorch的很多概念(比如计算图抽象和自动求导)并非它所独有的,和其他深度学习 框架相关。

我写这篇教程是专门针对那些从未用任何深度学习框架(例如：Tensorflow, Theano, Keras, Dynet)编写代码而从事NLP领域的人。我假设你 已经知道NLP领域要解决的核心问题：词性标注、语言模型等等。我也认为你通过AI这本书中所讲的知识熟悉了神经网络达到了入门的级别。 通常这些课程都会介绍反向传播算法和前馈神经网络，并指出它们是线性组合和非线性组合构成的链。本文在假设你已经有了这些知识的情况下， 教你如何开始写深度学习代码。

注意这篇文章主要关于 `_models_` ，而不是数据。对于所有的模型，我只创建一些数据维度较小的测试示例以便你可以看到权重在训练过程中 如何变化。如果你想要尝试一些真实数据，您有能力删除本示例中的模型并重新训练他们。

- PyTorch简介： PyTorch简介
- 使用PyTorch进行深度学习： 使用PyTorch进行深度学习
- 词嵌入：编码形式的词汇语义： 词嵌入：编码形式的词汇语义
- 序列模型和长短句记忆（LSTM）模型： 序列模型和长短句记忆（LSTM）模型
- 高级：制定动态决策和BI-LSTM CRF： 制定动态决策和BI-LSTM CRF

使用Sequence2Sequence网络和注意力进行翻译

在这个项目中，我们将讲解使用神经网络将法语翻译成英语。

```
[KEY: > input, = target, < output]

> il est en train de peindre un tableau .
= he is painting a picture .
< he is painting a picture .

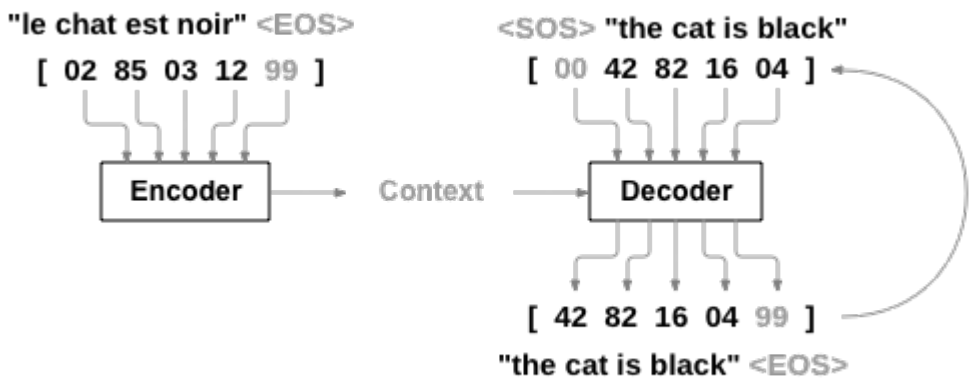
> pourquoi ne pas essayer ce vin delicieux ?
= why not try that delicious wine ?
< why not try that delicious wine ?

> elle n est pas poete mais romanciere .
= she is not a poet but a novelist .
< she not not a poet but a novelist .

> vous etes trop maigre .
= you re too skinny .
< you re all alone .
```

...取得了不同程度的成功。

这可以通过序列到序列网络来实现，其中两个递归神经网络一起工作以将一个序列转换成另一个序列。编码器网络将输入序列压缩成向量，并 且解码器网络将该向量展开成新的序列。



- **阅读建议**

开始本教程前，你已经安装好了PyTorch，并熟悉Python语言，理解“张量”的概念：

- https://pytorch.org/ PyTorch 安装指南
- Deep Learning with PyTorch：A 60 Minute Blitz :PyTorch的基本入门教程
- Learning PyTorch with Examples:得到深层而广泛的概述
- PyTorch for Former Torch Users Lua Torch:如果你曾是一个Lua张量的使用者

事先学习并了解序列到序列网络的工作原理对理解这个例子十分有帮助:

- Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation
- Sequence to Sequence Learning with Neural Networks
- Neural Machine Translation by Jointly Learning to Align and Translate
- A Neural Conversational Model

您还可以找到之前有关Classifying Names with a Character-Level RNN和 Generating Names with a Character-Level RNN 的教程，因为这些概念分别与编码器和解码器模型非常相似。

更多信息，请阅读介绍这些主题的论文：

- Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation
- Sequence to Sequence Learning with Neural Networks
- Neural Machine Translation by Jointly Learning to Align and Translate
- A Neural Conversational Model

1.导入必须的包

```
from __future__ import unicode_literals, print_function, division
from io import open
import unicodedata
import string
import re
import random

import torch
```

```
import torch.nn as nn
from torch import optim
import torch.nn.functional as F

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

2.加载数据文件

该项目的数据是成千上万的英语到法语的翻译对的集合。

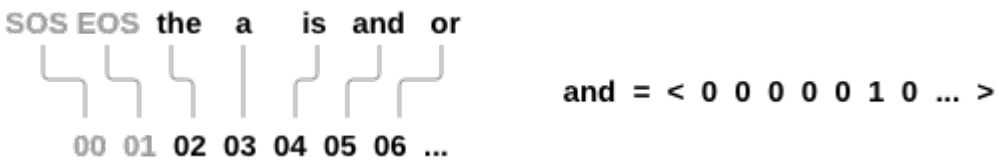
关于Open Data Stack Exchange的这个问题，开放式翻译网站 https://tatoeba.org/给出了指导，该网站的下载位于https://tatoeba.org/eng/downloads
- 更好的是，有人做了额外的拆分工作，将语言对分成单独的文本文件：https：//www.manythings.org/anki/

英语到法语对因为太大而无法包含在repo中，因此下载到data / eng-fra.txt再继续进行后续步骤。该文件是以制表符分隔的翻译对列表：

```
I am cold.    J'ai froid.
```

> 注意：从此处下载数据并将其解压缩到当前目录。

与字符级RNN教程中使用的字符编码类似，我们将语言中的每个单词表示为one-hot向量或零的巨向量，除了单个字符（在单词的索引处）。 与语言中可能存在的几十个字符相比，还有更多的字，因此编码向量很大。然而，我们投机取巧并修剪数据，每种语言只使用几千个单词。



我们将需要每个单词的唯一索引，以便稍后用作网络的输入和目标。为了跟踪所有这些，我们将使用一个名为 `Lang` 的辅助类，它具有 word→index（ `word2index` ）和index→word（ `index2word` ）的字典，以及用于稍后替换稀有单词的每个单词 `word2count` 的计数。

```
SOS_token = 0
EOS_token = 1
```

```
class Lang:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "SOS", 1: "EOS"}
        self.n_words = 2  # Count SOS and EOS

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1
```

这些文件都是Unicode格式，为了简化我们将Unicode字符转换为ASCII，使所有内容都小写，并去掉大多数标点符号。

```
# 将Unicode字符串转换为纯ASCII, 感谢https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# 小写, 修剪和删除非字母字符


def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    return s
```

2.1 读取数据文件

要读取数据文件，我们将文件拆分为行，然后将行拆分成对。 这些文件都是英语→其他语言，所以如果我们想翻译其他语言→英语，我添加 `reverse` 标志来反转对。

```
```

```
def readLangs(lang1, lang2, reverse=False):
    print("Reading lines...")

    # 读取文件并分成几行
    lines = open('data/%s-%s.txt' % (lang1, lang2), encoding='utf-8').\
        read().strip().split('\n')

    # 将每一行拆分成对并进行标准化
    pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]

    # 反向对，使Lang实例
    if reverse:
        pairs = [list(reversed(p)) for p in pairs]
        input_lang = Lang(lang2)
        output_lang = Lang(lang1)
    else:
        input_lang = Lang(lang1)
        output_lang = Lang(lang2)

    return input_lang, output_lang, pairs
```

由于有很多例句，我们想快速训练，我们会将数据集修剪成相对简短的句子。 这里最大长度是10个单词（包括结束标点符号），我们将过滤到 转换为“我是”或“他是”等形式的句子（考虑先前替换的撇号）。

```
MAX_LENGTH = 10

eng_prefixes = (
    "i am ", "i m ",
    "he is", "he s ",
    "she is", "she s ",
    "you are", "you re ",
    "we are", "we re ",
    "they are", "they re "
)


def filterPair(p):
    return len(p[0].split(' ')) < MAX_LENGTH and \
        len(p[1].split(' ')) < MAX_LENGTH and \
        p[1].startswith(eng_prefixes)


def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]
```

准备数据的完整过程是：

- 读取文本文件并拆分成行，将行拆分成对
- 规范化文本，按长度和内容进行过滤
- 从成对的句子中制作单词列表

```
def prepareData(lang1, lang2, reverse=False):
    input_lang, output_lang, pairs = readLangs(lang1, lang2, reverse)
    print("Read %s sentence pairs" % len(pairs))
    pairs = filterPairs(pairs)
    print("Trimmed to %s sentence pairs" % len(pairs))
    print("Counting words...")
    for pair in pairs:
        input_lang.addSentence(pair[0])
        output_lang.addSentence(pair[1])
    print("Counted words:")
    print(input_lang.name, input_lang.n_words)
    print(output_lang.name, output_lang.n_words)
    return input_lang, output_lang, pairs


input_lang, output_lang, pairs = prepareData('eng', 'fra', True)
print(random.choice(pairs))
```

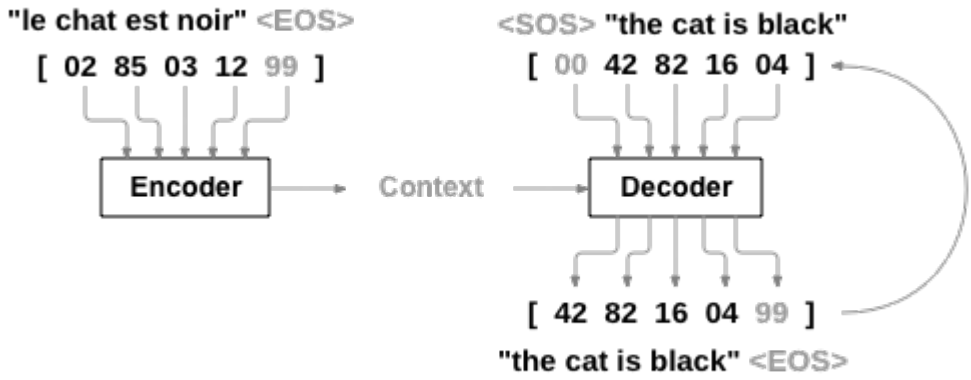
- 输出结果：

```
Reading lines...
Read 135842 sentence pairs
Trimmed to 10599 sentence pairs
Counting words...
Counted words:
fra 4345
eng 2803
['nous nous deshabillons .', 'we re undressing .']
```

3.Seq2Seq模型

递归神经网络（RNN）是一种对序列进行操作的网络，它使用自己的输出作为后续步骤的输入。

Sequence to Sequence network(seq2seq网络)或[Encoder Decoder network(https://arxiv.org/pdf/1406.1078v3.pdf) 是由称为编码器和解码器的两个RNN组成的模型。编码器读取输入序列并输出单个向量，并且解码器读取该向量以产生输出序列。



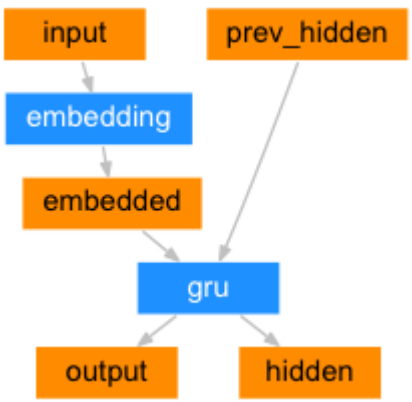
与使用单个RNN的序列预测不同，其中每个输入对应于输出，seq2seq模型使我们从序列长度和顺序中解放出来，这使其成为两种语言之间转换 的理想选择。

考虑一句“Je ne suis pas le chat noir”→“我不是黑猫”。输入句子中的大多数单词在输出句子中都有直接翻译，但顺序略有不同，例 如 “聊天黑色”和“黑猫”。由于“ne / pas”结构，输入句中还有一个单词。直接从输入字序列产生正确的翻译将是困难的。

使用seq2seq模型，编码器创建单个向量，在理想情况下，将输入序列的“含义”编码为单个向量 - 句子的某些N维空间中的单个点。

3.1 编码器

seq2seq网络的编码器是RNN，它为输入句子中的每个单词输出一些值。对于每个输入的词，编码器输出向量和隐藏状态，并将隐藏状态用于下 一个输入的单词。



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

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden

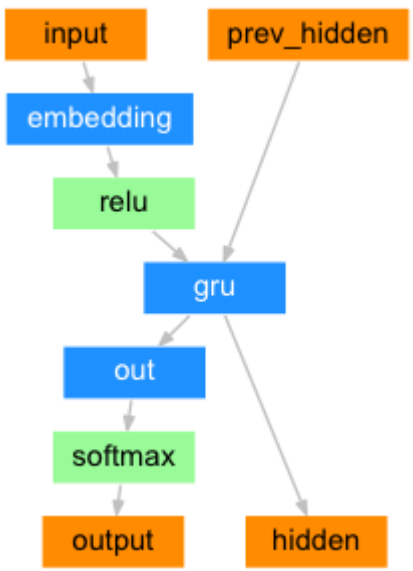
    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

3.2 解码器

码器是另一个RNN，它接收编码器输出向量并输出一系列字以创建转换。

简单的解码器 在最简单的seq2seq解码器中，我们仅使用编码器的最后一个输出。 最后一个输出有时称为上下文向量，因为它编码整个序列的上下文。 该上下文向量用作解码器的初始隐藏状态。

在解码的每个步骤中，给予解码器输入token和隐藏状态。初始输入token是开始字符串 `<SOS>` 标记，第一个隐藏状态是上下文向量（编码器 的最后隐藏状态）。



```
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

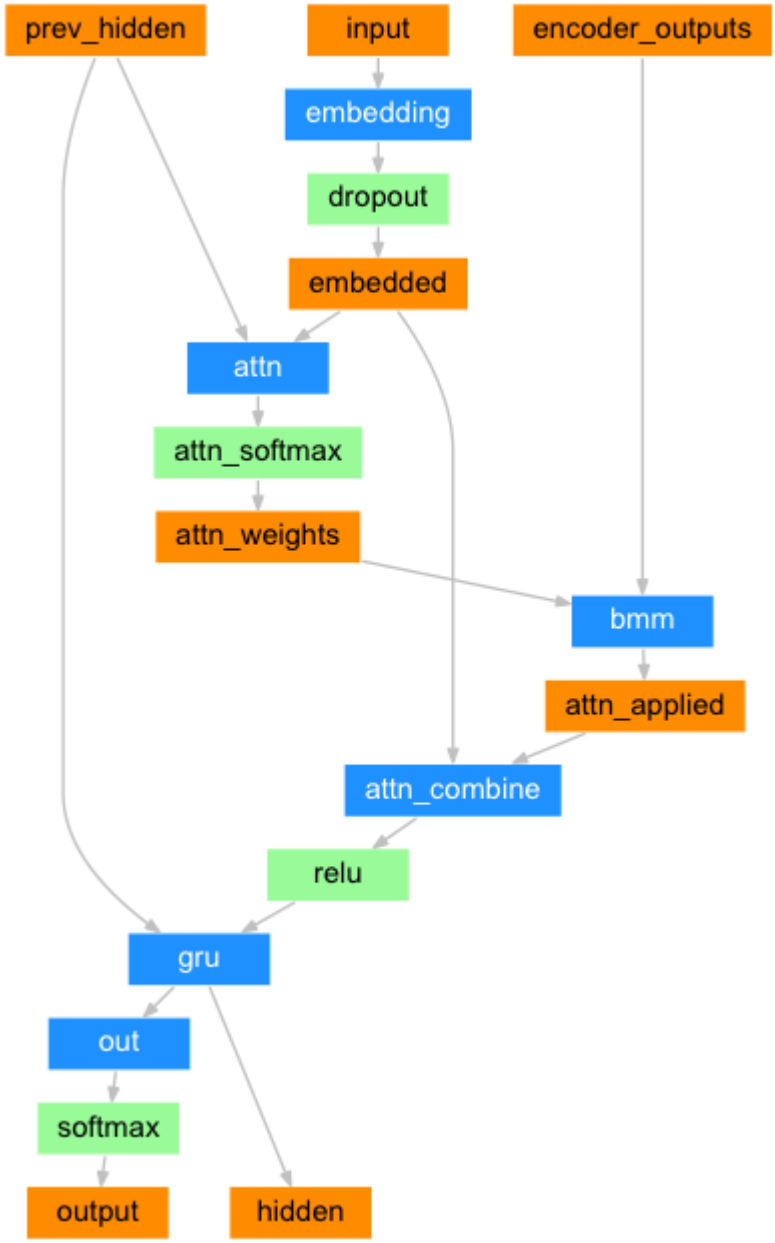
我鼓励你训练和观察这个模型的结果，但为了节省空间，我们将直接进入主题并引入注意机制。

3.3 注意力机制解码器

如果仅在编码器和解码器之间传递上下文向量，则该单个向量承担编码整个句子的信息。

注意力允许解码器网络针对解码器自身输出的每个步骤“聚焦”编码器输出的不同部分。首先，我们计算一组注意力权重。这些将乘以编码器 输出向量以创建加权组合。结果（在代码中称为 `attn_applied` ）应包含有关输入序列特定部分的信息，从而帮助解码器选择正确的输出单词。

使用解码器的输入和隐藏状态作为输入，使用另一个前馈层 `attn` 来计算注意力权重。因为训练数据中存在所有不同大小的句子，为了实际创建和 训练该层，我们必须选择它可以应用的最大句子长度（输入长度，对于编码器输出）。最大长度的句子将使用所有注意力权重，而较短的句子将仅使用前几个。




```
class AttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, dropout_p=0.1,
max_length=MAX_LENGTH):
        super(AttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size)
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)
```

```
        self.out = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        attn_weights = F.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))

        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        output = F.relu(output)
        output, hidden = self.gru(output, hidden)

        output = F.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

> 注意：
> 通过使用相对位置方法，还有其他形式的注意力可以解决长度限制问题。阅读[Effective Approaches to Attention-based Neural Machine Translation.] (https://arxiv.org/abs/1508.04025)的“本地注意”。

4.训练

4.1 准备训练数据

为了训练，对于每对翻译对，我们将需要输入张量（输入句子中的单词的索引）和目标张量（目标句子中的单词的索引）。在创建这些向量时，我们会将EOS标记附加到两个序列。

```
def indexesFromSentence(lang, sentence):
    return [lang.word2index[word] for word in sentence.split(' ')]


def tensorFromSentence(lang, sentence):
    indexes = indexesFromSentence(lang, sentence)
    indexes.append(EOS_token)
    return torch.tensor(indexes, dtype=torch.long, device=device).view(-1, 1)
```

```
def tensorsFromPair(pair):
    input_tensor = tensorFromSentence(input_lang, pair[0])
    target_tensor = tensorFromSentence(output_lang, pair[1])
    return (input_tensor, target_tensor)
```

4.2 训练模型

为了训练我们通过编码器运行的输入句子，并跟踪每个输出和最新的隐藏状态。然后，解码器被赋予标记作为其第一输入，并且编码器的 最后隐藏状态作为其第一隐藏状态。

“Teacher Forcing”是将真实目标输出用作每个下一个输入的概念，而不是使用解码器的猜测作为下一个输入。使用teacher forcing使模型 更快地收敛，但是当利用受过训练的网络时，它可能表现出不稳定性。

您可以观察teacher forcing网络的输出，这些网络使用连贯的语法阅读，但远离正确的翻译 - 直觉上它已经学会表示输出语法，并且一旦 老师告诉它前几个单词就可以“提取”意义，但是它没有正确地学习如何从翻译中创建句子。

由于PyTorch的 autograd 为我们提供了自由，我们可以随意选择使用teacher forcing或不使用简单的if语句。将 `teacher_forcing_ratio` 调高以使用更多。

```
teacher_forcing_ratio = 0.5


def train(input_tensor, target_tensor, encoder, decoder, encoder_optimizer,
decoder_optimizer, criterion, max_length=MAX_LENGTH):
    encoder_hidden = encoder.initHidden()

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

    loss = 0

    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(
            input_tensor[ei], encoder_hidden)
        encoder_outputs[ei] = encoder_output[0, 0]
```

```
    decoder_input = torch.tensor([[SOS_token]], device=device)

    decoder_hidden = encoder_hidden

    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else
False

    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # Teacher forcing

    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from history as
input

            loss += criterion(decoder_output, target_tensor[di])
            if decoder_input.item() == EOS_token:
                break

    loss.backward()

    encoder_optimizer.step()
    decoder_optimizer.step()

    return loss.item() / target_length
```

辅助函数

这是一个辅助函数，用于打印经过的时间和估计的剩余时间给定当前时间和进度%。

```
import time
import math


def asMinutes(s):
    m = math.floor(s / 60)
    s -= m * 60
```

```
    return '%dm %ds' % (m, s)


def timeSince(since, percent):
    now = time.time()
    s = now - since
    es = s / (percent)
    rs = es - s
    return '%s (- %s)' % (asMinutes(s), asMinutes(rs))
```

整个训练过程如下：

- 启动计时器
- 初始化优化器和标准
- 创建一组训练对
- 启动空损数组进行绘图

然后我们调用 `train` ，偶尔打印进度（例子的百分比，到目前为止的时间，估计的时间）和平均损失。

```
def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=100,
learning_rate=0.01):
    start = time.time()
    plot_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every

    encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)
    training_pairs = [tensorsFromPair(random.choice(pairs))
                      for i in range(n_iters)]
    criterion = nn.NLLLoss()

    for iter in range(1, n_iters + 1):
        training_pair = training_pairs[iter - 1]
        input_tensor = training_pair[0]
        target_tensor = training_pair[1]

        loss = train(input_tensor, target_tensor, encoder,
                     decoder, encoder_optimizer, decoder_optimizer, criterion)
        print_loss_total += loss
        plot_loss_total += loss

        if iter % print_every == 0:
```

```
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (timeSince(start, iter / n_iters),
                                         iter, iter / n_iters * 100,
print_loss_avg))

        if iter % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0

    showPlot(plot_losses)
```

结果绘图函数

绘图使用 matplotlib 库完成，使用在训练时保存的 `plot_losses` 的损失值数组。

```
import matplotlib.pyplot as plt
plt.switch_backend('agg')
import matplotlib.ticker as ticker
import numpy as np


def showPlot(points):
    plt.figure()
    fig, ax = plt.subplots()
    # this locator puts ticks at regular intervals
    loc = ticker.MultipleLocator(base=0.2)
    ax.yaxis.set_major_locator(loc)
    plt.plot(points)
```

评价函数

评估与训练大致相同，但没有目标，因此我们只需将解码器的预测反馈给每个步骤。每次它预测一个单词时我们都会将它添加到输出字符串中，如果它预测了EOS标记，我们就会停在那里。我们还存储解码器的注意力输出以供稍后显示。

```
def evaluate(encoder, decoder, sentence, max_length=MAX_LENGTH):
    with torch.no_grad():
        input_tensor = tensorFromSentence(input_lang, sentence)
        input_length = input_tensor.size()[0]
        encoder_hidden = encoder.initHidden()

        encoder_outputs = torch.zeros(max_length, encoder.hidden_size,
device=device)
```

```
        for ei in range(input_length):
            encoder_output, encoder_hidden = encoder(input_tensor[ei],
                                                     encoder_hidden)
            encoder_outputs[ei] += encoder_output[0, 0]

        decoder_input = torch.tensor([[SOS_token]], device=device)  # SOS

        decoder_hidden = encoder_hidden

        decoded_words = []
        decoder_attentions = torch.zeros(max_length, max_length)

        for di in range(max_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            decoder_attentions[di] = decoder_attention.data
            topv, topi = decoder_output.data.topk(1)
            if topi.item() == EOS_token:
                decoded_words.append('<EOS>')
                break
            else:
                decoded_words.append(output_lang.index2word[topi.item()])

            decoder_input = topi.squeeze().detach()

        return decoded_words, decoder_attentions[:di + 1]
```

我们可以从训练集中评估随机句子并打印输入、目标和输出以做出一些直观质量判断：

```
def evaluateRandomly(encoder, decoder, n=10):
    for i in range(n):
        pair = random.choice(pairs)
        print('>', pair[0])
        print('=', pair[1])
        output_words, attentions = evaluate(encoder, decoder, pair[0])
        output_sentence = ' '.join(output_words)
        print('<', output_sentence)
        print('')
```

5.训练和评价

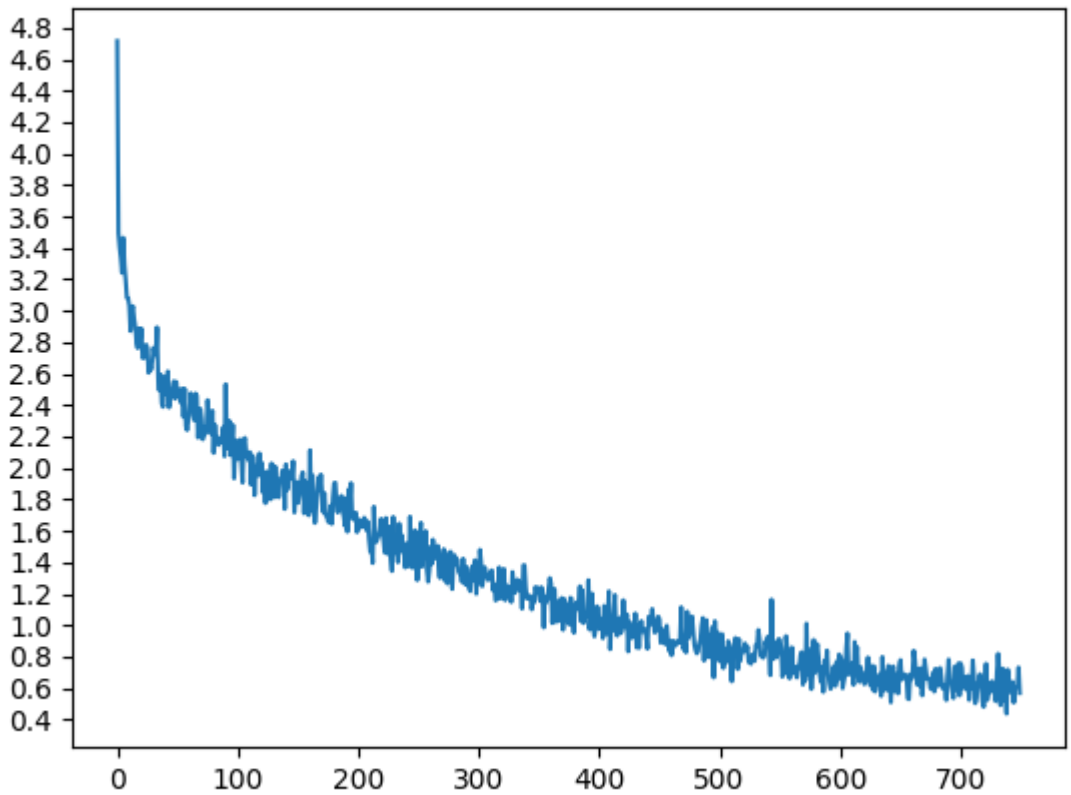
有了所有这些辅助函数（它看起来像是额外的工作，但它使得运行多个实验更容易）我们实际上可以初始化网络并开始训练。

请记住，输入句子被严格过滤。对于这个小数据集，我们可以使用256个隐藏节点和单个GRU层的相对较小的网络。在MacBook CPU上大约40分 钟后，我们将得到一些合理的结果。

> 注意：
> 如果你运行这个笔记，你可以训练、中断内核、评估，并在以后继续训练。注释掉编码器和解码器初始化的行并再次运行trainIters。

```
hidden_size = 256
encoder1 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
attn_decoder1 = AttnDecoderRNN(hidden_size, output_lang.n_words, dropout_p=0.
1).to(device)

trainIters(encoder1, attn_decoder1, 75000, print_every=5000)
```



- 输出结果：

```
1m 53s (- 26m 24s) (5000 6%) 2.8558
3m 42s (- 24m 3s) (10000 13%) 2.2832
5m 31s (- 22m 6s) (15000 20%) 1.9841
```

```
7m 19s (- 20m 8s) (20000 26%) 1.7271
9m 7s (- 18m 15s) (25000 33%) 1.5487
10m 54s (- 16m 21s) (30000 40%) 1.3461
12m 41s (- 14m 30s) (35000 46%) 1.2251
14m 30s (- 12m 41s) (40000 53%) 1.0956
16m 16s (- 10m 51s) (45000 60%) 1.0126
18m 5s (- 9m 2s) (50000 66%) 0.9212
19m 52s (- 7m 13s) (55000 73%) 0.7952
21m 41s (- 5m 25s) (60000 80%) 0.7481
23m 29s (- 3m 36s) (65000 86%) 0.6882
25m 17s (- 1m 48s) (70000 93%) 0.6190
27m 6s (- 0m 0s) (75000 100%) 0.5745
```

```
evaluateRandomly(encoder1, attn_decoder1)
```

- 输出结果：

```
> je pars en vacances pour quelques jours .
= i m taking a couple of days off .
< i m taking a couple of days off . <EOS>

> je ne me panique pas .
= i m not panicking .
< i m not panicking . <EOS>

> je recherche un assistant .
= i am looking for an assistant .
< i m looking a call . <EOS>

> je suis loin de chez moi .
= i m a long way from home .
< i m a little friend . <EOS>

> vous etes en retard .
= you re very late .
< you are late . <EOS>

> j ai soif .
= i am thirsty .
< i m thirsty . <EOS>

> je suis fou de vous .
= i m crazy about you .
< i m crazy about you . <EOS>

> vous etes vilain .
= you are naughty .
```

```
< you are naughty . <EOS>

> il est vieux et laid .
= he s old and ugly .
< he s old and ugly . <EOS>

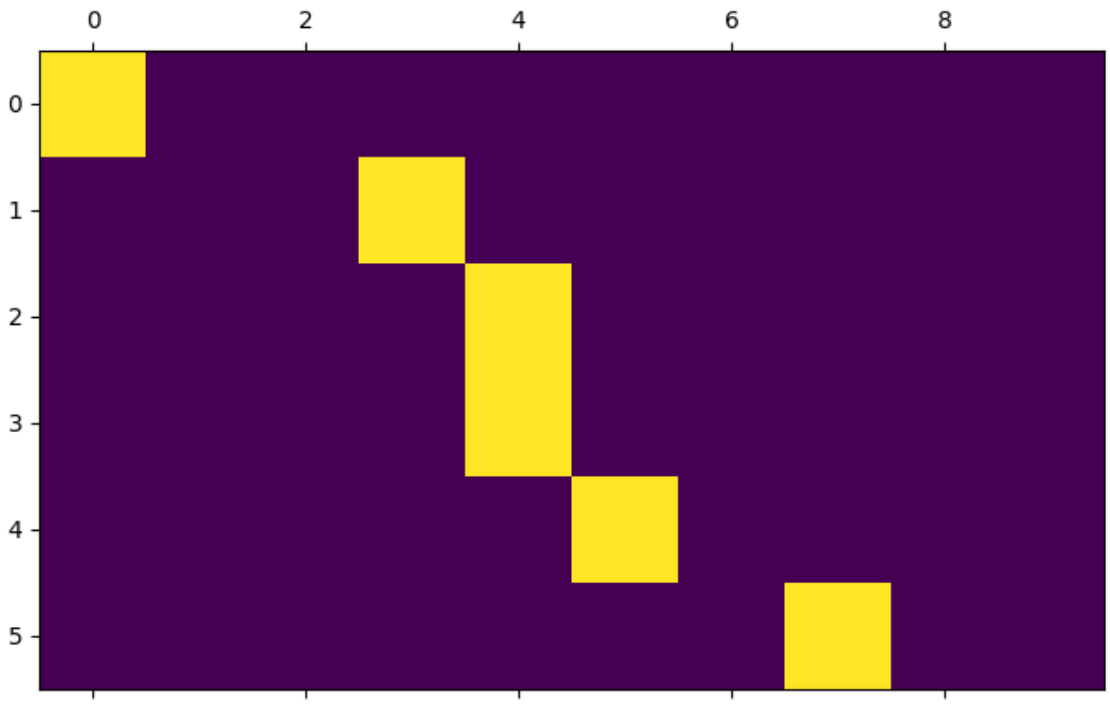
> je suis terrifiee .
= i m terrified .
< i m touched . <EOS>
```

6.可视化注意力

注意力机制的一个有用特性是其高度可解释的输出。因为它用于对输入序列的特定编码器输出进行加权，所以我们可以想象在每个时间步长看 网络最关注的位置。

您可以简单地运行plt.matshow（attention）以将注意力输出显示为矩阵，其中列是输入步骤，行是输出步骤：

```
output_words, attentions = evaluate(
    encoder1, attn_decoder1, "je suis trop froid .")
plt.matshow(attentions.numpy())
```



为了获得更好的观看体验，我们将额外添加轴和标签：

```
def showAttention(input_sentence, output_words, attentions):
    # 用colorbar设置图
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(attentions.numpy(), cmap='bone')
    fig.colorbar(cax)

    # 设置坐标
    ax.set_xticklabels(['']  input_sentence.split(' ') +
                       ['<EOS>'], rotation=90)
    ax.set_yticklabels(['']  output_words)

    # 在每个刻度处显示标签
    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    plt.show()

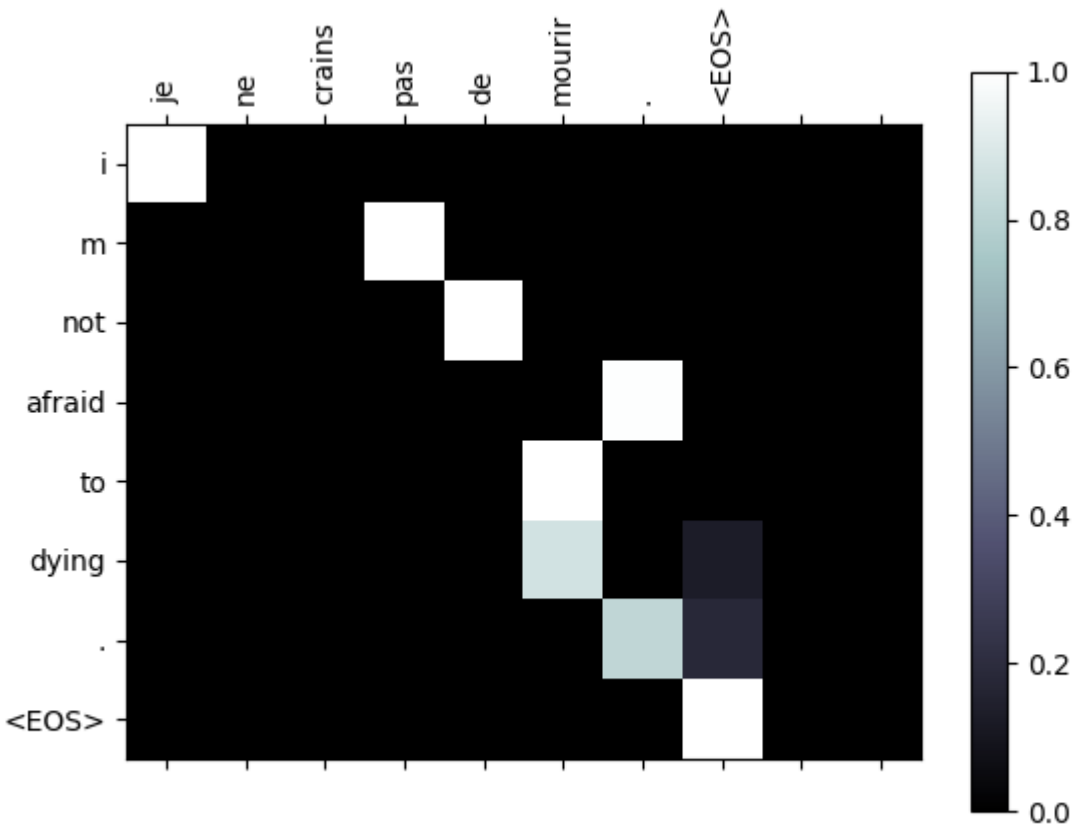

def evaluateAndShowAttention(input_sentence):
    output_words, attentions = evaluate(
        encoder1, attn_decoder1, input_sentence)
    print('input =', input_sentence)
    print('output =', ' '.join(output_words))
    showAttention(input_sentence, output_words, attentions)

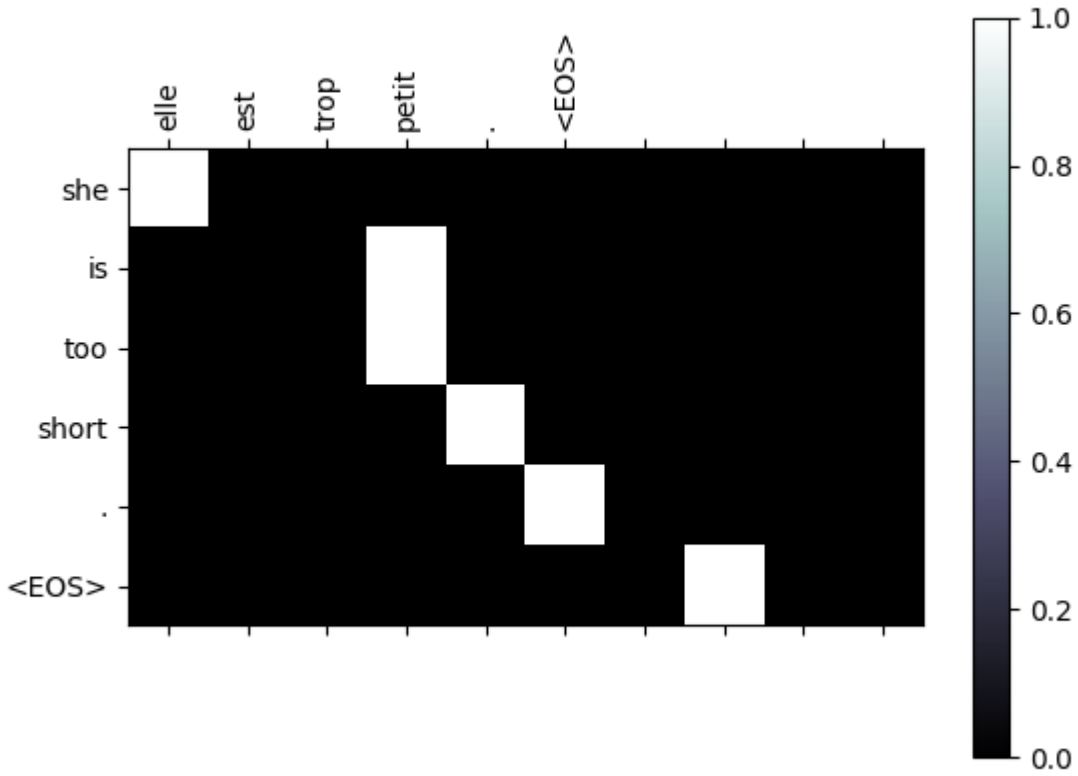

evaluateAndShowAttention("elle a cinq ans de moins que moi .")

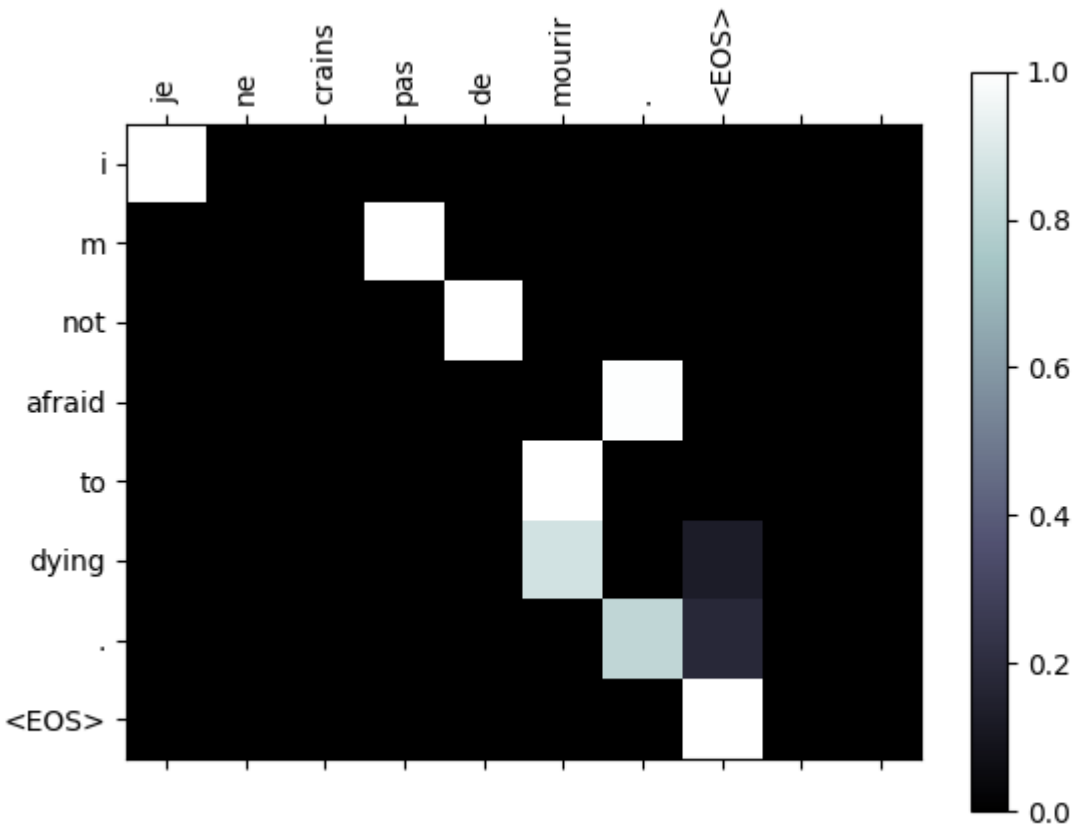
evaluateAndShowAttention("elle est trop petit .")

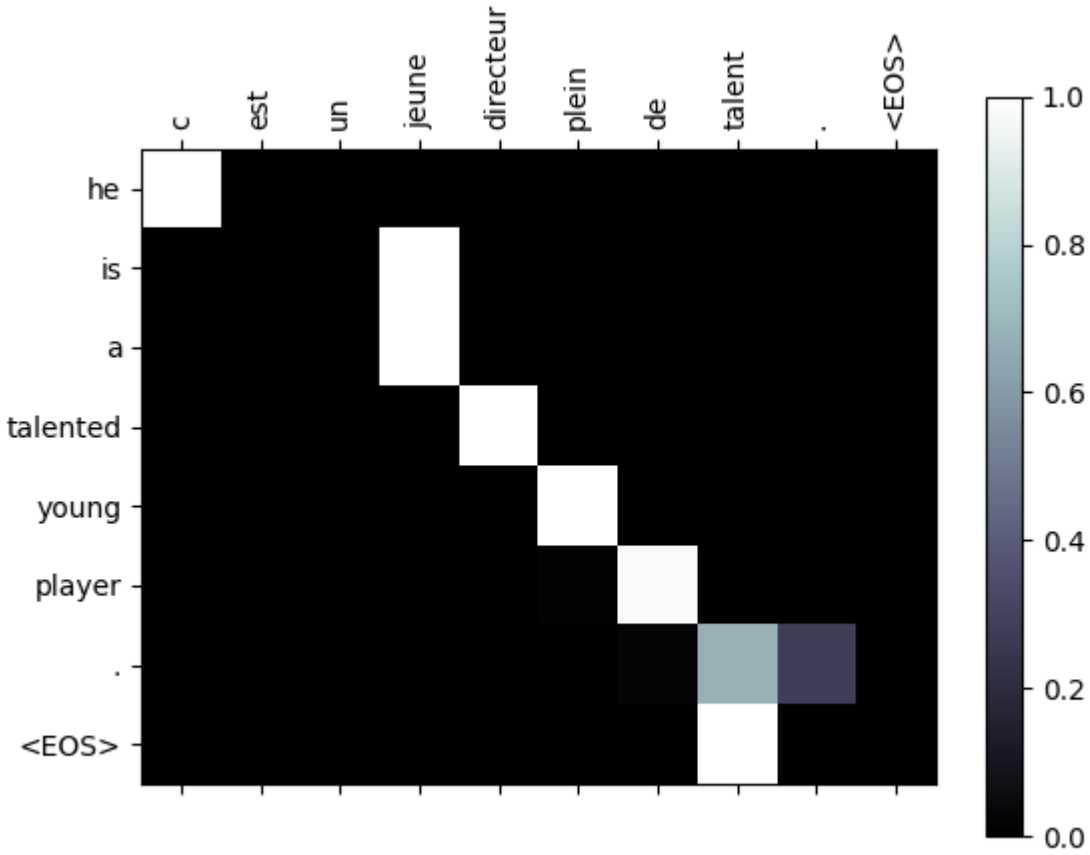
evaluateAndShowAttention("je ne crains pas de mourir .")

evaluateAndShowAttention("c est un jeune directeur plein de talent .")
```


je
ne
crains
pas
de
mourir
.
<EOS>
i
m
not
afraid
to
dying
.
<EOS>
1.0
0.8
0.6
0.4
0.2
0.0


elle
est
trop
petit
.
<EOS>
she
is
too
short
.
<EOS>
1.0
0.8
0.6
0.4
0.2
0.0


je
ne
crains
pas
de
mourir
.
<EOS>
i
m
not
afraid
to
dying
.
<EOS>
1.0
0.8
0.6
0.4
0.2
0.0



- 输出结果：

```
input = elle a cinq ans de moins que moi .
output = she is two years younger than me . <EOS>
input = elle est trop petit .
output = she s too trusting . <EOS>
input = je ne crains pas de mourir .
output = i m not afraid of dying . <EOS>
input = c est un jeune directeur plein de talent .
output = he s a fast person . <EOS>
```

7.练习

- 尝试使用其他数据集：
  * 另一种语言对
  * 人→机器（例如IOT命令）

    * 聊天→回复
    * 问题→答案

- 用预先训练过的字嵌入（例如word2vec或GloVe）替换嵌入
- 尝试使用更多图层，更多隐藏单元和更多句子。比较训练时间和结果。
- 如果你使用翻译文件，其中对有两个相同的短语（`I am test \t I am tes`），你可以使用它作为自动编码器。试试这个：
    * 训练为自动编码器
    * 仅保存编码器网络
    * 从那里训练一个新的解码器进行翻译

DCGAN教程

1. 简介

本教程通过一个例子来对 DCGANs 进行介绍。我们将会训练一个生成对抗网络（GAN）用于在展示了许多真正的名人的图片后产生新的名人。 这里的大部分代码来自pytorch/examples中的 dcgan 实现，本文档将对实现进行进行全面 的介绍，并阐明该模型的工作原理以及为什么如此。但是不需要担心，你并不需要事先了解 GAN，但可能需要花一些事件来推理一下底层 实际发生的事情。此外，为了有助于节省时间，最好是使用一个GPU，或者两个。让我们从头开始。

2. 生成对抗网络（Generative Adversarial Networks）

2.1 什么是 GAN

GANs是用于 DL （Deep Learning）模型去捕获训练数据分布情况的框架，以此我们可以从同一分布中生成新的数据。GANs是有Ian Goodfellow 于2014年提出，并且首次在论文Generative Adversarial Nets中 描述。它们由两个不同的模型组成，一个是生成器一个是判别器。生成器的工作是产生看起来像训练图像的“假”图像；判别器的工作是 查看图像并输出它是否是真实的训练图像或来自生成器的伪图像。在训练期间，产生器不断尝试通过产生越来越好的假动作来超越判别器， 而判别器则是为了更好地检测并准确地对真实和假图像进行分类。这个游戏的平衡是当生成器产生完美的假动作以使假图像看起来像是来自 训练数据，而判别器总是猜测生成器输出图像为真或假的概率为50%。

现在，我们开始定义在这个教程中使用到的一些符号。

- 判别器的符号定义

设x表示代表一张图像的数据，$D(x)$ 是判别器网络，它输出x来自训练数据而不是生成器的（标量）概率。这里，由于我们处理图像，$D(x)$ 的输入是 CHW 大小为3x64x64的图像。直观地，当x来自训练数据时$D(x)$应该是 HIGH ，而当x来 自生成器时$D(x)$应该是 LOW。$D(x)$也可以被认为是传统的二元分类器。

- 生成器的符号定义

对于生成器的符号，让z是从标准正态分布中采样的潜在空间矢量，$G(z)$表示将潜在向量z 映射到数据空间的生成器函数，G的目标是估计训练数据来自什么分布（p_{data}），以便它可以 根据估计的分布（p_g）生成假样本。

因此，$D(G(z))$是生成器G的输出是真实图像的概率（标量）。正如Goodfellow 的论文中所描述的，D和G玩一个极小极大的游戏，其中D试图最大化它正确地 分类真实数据和假样本$logD(x)$的概率，并且G试图最小化D预测其输出是假的 概率$log(1 - D(G(x)))$。从论文来看，GAN 损失函数是:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[logD(x)] + \mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z)))]$$

理论上，这个极小极大游戏的解决方案是$p_g = p_{data}$，如果输入的是真实的或假的，则判别器会随机猜测。然而，GAN 的收敛理论仍在积极研究中，实际上模型并不总是训练到这一点。

2.2 什么是 DCGAN

DCGAN 是上述 GAN 的直接扩展，区别的是它分别在判别器和生成器中明确地使用了卷积和卷积转置层。它首先是由Radford等人在论文 Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks 中提出。判别器由 *strided convolution layers*、*batch norm layers* 和 *LeakyReLU activations* 组成，它输入 3x64x64 的图像， 然后输出的是一个代表输入是来自实际数据分布的标量概率。生成器则是由 *convolutional-transpose layers*、 *batch norm layers* 和 *ReLU activations* 组成。它的输入是从标准正态分布中绘制的潜在向量z，输出是 3x64x64 的 RGB 图像。*strided conv-transpose layers* 允许潜在标量转换成具有与图像相同形状的体积。在本文中，作者还提供了一些有关如何设置优化器，如何计算损失函数以及如何初始化 模型权重的提示，所有这些都将在后面的章节中进行说明。

```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
```

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# 为再现性设置随机seem
manualSeed = 999
#manualSeed = random.randint(1, 10000) # 如果你想要新的结果就是要这段代码
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
```

- 输出结果：

```
Random Seed:  999
```

3. DCGAN实现过程

3.1 输入

让我们定义输入数据去运行我们的教程： * **dataroot**：存放数据集根目录的路径。我们将在下一节中详细讨论数据集 * **workers**：使用DataLoader加载数据的工作线程数 * **batch_size**：训练中使用的batch大小。在DCGAN论文中batch的大小为128 * **image_size**：用于训练的图像的空间大小。此实现默认 64x64。如果需要其他尺寸，则必须改变D和G 的结构。有关详细信息，请参见此处 。 * **nc**：输入图像中的颜色通道数。对于彩色图像，这是参数设置为3 * **nz**：潜在向量的长度 * **ngf**：与通过生成器携带的特征图的深度有关 * **ndf**：设置通过判别器传播的特征映射的深度 * **num_epochs**：要运行的训练的epoch数量。长时间的训练可能会带来更好的结果，但也需要更长的时间 * **lr**：学习速率。如DCGAN论文中所述，此数字应为0.0002 * **beta1**：适用于Adam优化器的beta1超参数。如论文所述，此数字应为0.5 * **ngpu**：可用的GPU数量。如果为0，则代码将以CPU模式运行。如果此数字大于0，它将在该数量的GPU上运行

```buildoutcfg
# 数据集的根目录

dataroot = “data/celeba”
```

加载数据的工作线程数

workers = 2

训练期间的batch大小

batch_size = 128

训练图像的空间大小。所有图像将使用变压器调整为此大小。

image_size = 64

训练图像中的通道数。对于彩色图像，这是3

nc = 3

潜在向量 z 的大小(例如： 生成器输入的大小)

nz = 100

生成器中特征图的大小

ngf = 64

判别器中的特征映射的大小

ndf = 64

训练epochs的大小

num_epochs = 5

优化器的学习速率

lr = 0.0002

适用于Adam优化器的Beta1超级参数

beta1 = 0.5

可用的GPU数量。使用0表示CPU模式。

ngpu = 1

```
#### 3.2 数据
在本教程中，我们将使用[Celeb-A Faces](http://mmlab.ie.cuhk.edu.hk/projects/
CelebA.html)数据集，该数据集可以在链接或Google Drive中下载。
数据集将下载为名为*img_align_celeba.zip*的文件。下载后，创建名为*celeba*的目录并将zip文件
解压缩到该目录中。然后，将此笔记中
的数据对象输入设置为刚刚创建的celeba目录。生成的目录结构应该是：

```buildoutcfg
/path/to/celeba
    -> img_align_celeba
        -> 188242.jpg
        -> 173822.jpg
        -> 284702.jpg
        -> 537394.jpg
```

这是一个重要的步骤，因为我们将使用 `ImageFolder` 数据集类，它要求在数据集的根文件夹中有子目录。现在，我们可以创建数据集，创 建数据加载器，设置要运行的设备，以及最后可视化一些训练数据。

```
```
```

```
# 我们可以按照设置的方式使用图像文件夹数据集。
# 创建数据集
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.
5)),
                           ]))
# 创建加载器
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# 选择我们运行在上面的设备
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else
"cpu")

# 绘制部分我们的输入图像
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],
padding=2, normalize=True).cpu(),(1,2,0)))
```

3.3 实现

通过设置输入参数和准备好的数据集，我们现在可以进入真正的实现步骤。我们将从权重初始化策略开始，然后详细讨论生成器，鉴别器， 损失函数和训练循环。

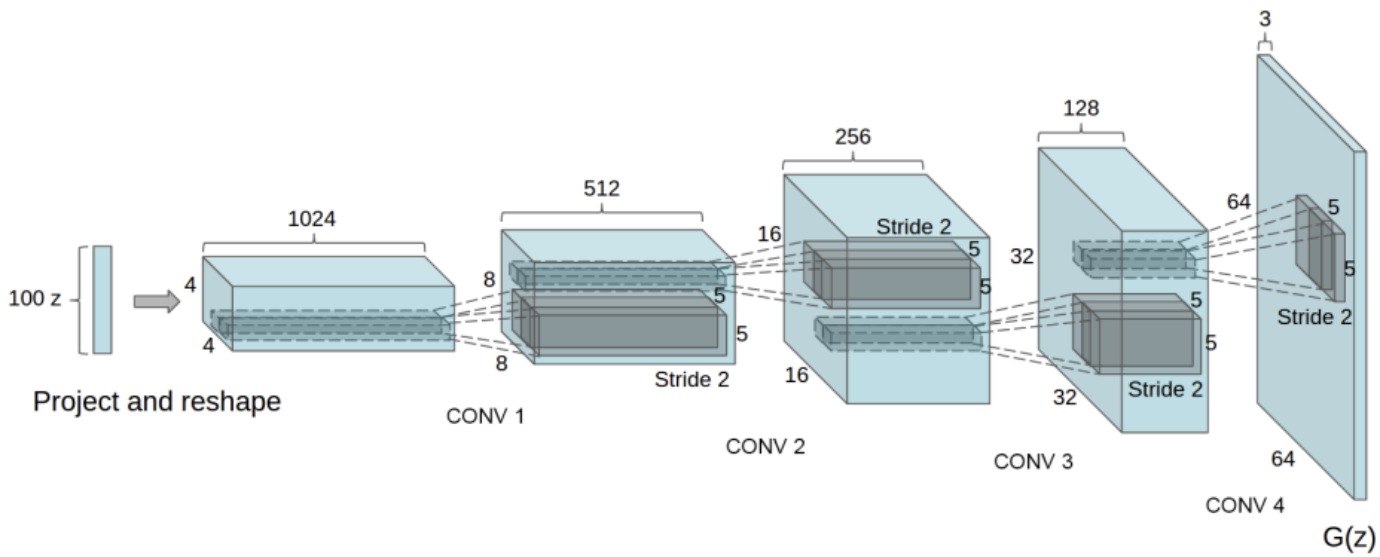
3.3.1 权重初始化

在 DCGAN 论文中，作者指出所有模型权重应从正态分布中随机初始化，mean = 0，stdev = 0.02。 weights_init 函数将初始化模型作为 输入，并重新初始化所有卷积，卷积转置和batch标准化层以满足此标准。初始化后立即将此函数应用于模型。

```
# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

3.3.2 生成器

生成器G用于将潜在空间矢量（z）映射到数据空间。由于我们的数据是图像，因此将z 转换为数据空间意味着最终创建与训练图像具有相同大小的RGB图像（即3x64x64）。实际上，这是通过一系列跨步的二维卷积转置层实现的， 每个转换层与二维批量标准层和relu activation进行配对。生成器的输出通过 tanh 函数输入，使其返回到[-1,1]范围的输入数据。值得 注意的是在转换层之后存在批量范数函数，因为这是DCGAN论文的关键贡献。这些层有助于训练期间的梯度流动。DCGAN论文中的生成器中 的图像如下所示：



> 请注意，我们对输入怎么设置（*nz*，*ngf*和*nc*）会影响代码中的生成器体系结构。*nz* 是z输入向量的长度，*ngf*与通过生成器传播的特征图的大小有关，*nc*是输出图像中的通道数（对于RGB图像，设置为3）。下面是生成器的代码。

- 生成器代码

```
# 生成器代码
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # 输入是Z，进入卷积
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
        return self.main(input)
```

现在，我们可以实例化生成器并应用 `weights_init` 函数。查看打印的模型以查看生成器对象的结构。

```buildoutcfg
```

创建生成器

netG = Generator(ngpu).to(device)

如果需要，管理multi-gpu

if (device.type == 'cuda') and (ngpu > 1): netG = nn.DataParallel(netG, list(range(ngpu)))

应用weights_init函数随机初始化所有权重，mean=0，stdev = 0.2。

netG.apply(weights_init)

打印模型

print(netG)

```
* 输出结果：

```buildoutcfg
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (8): ReLU(inplace=True)
```
```

```
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (13): Tanh()
  )
)
```

3.3.3 判别器

如上所述，判别器D是二进制分类网络，它将图像作为输入并输出输入图像是真实的标量概率（与假的相反）。这里，D采用 3x64x64 的输入图像，通过一系列Conv2d，BatchNorm2d和LeakyReLU层处理它，并通过Sigmoid激活函数输出 最终概率。如果问题需要，可以使用更多层扩展此体系结构，但使用strided convolution（跨步卷积），BatchNorm和LeakyReLU具有重要 意义。DCGAN论文提到使用跨步卷积而不是池化到降低采样是一种很好的做法，因为它可以让网络学习自己的池化功能。批量标准和 leaky relu函数也促进良好的梯度流，这对于和的学习过程都是至关重要的。

- 判别器代码

```
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
```

```
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

现在，与生成器一样，我们可以创建判别器，应用 `weights_init` 函数，并打印模型的结构。

```
# 创建判别器
netD = Discriminator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# 应用weights_init函数随机初始化所有权重，mean= 0, stdev = 0.2
netD.apply(weights_init)

# 打印模型
print(netD)
```

- 输出结果：

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
```

```
    )
)
```

3.3.4 损失函数和优化器

通过D和G设置，我们可以指定他们如何通过损失函数和优化器学习。我们将使用PyTorch中定义的 二进制交叉熵损失（BCELoss）函数：

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)]$$

注意该函数如何计算目标函数中的两个对数分量（即$log D(x)$和$log(1 - D(G(z)))$。我们可以指定用于输入y 的BCE方程的哪个部分。这是在即将出现的训练循环中完成的，但重要的是要了解我们如何通过改变y（即GT标签）来选择我们希望计算的组件。

接下来，我们将真实标签定义为1，将假标签定义为0。这些标签将在计算D和G的损失时使用，这 也是原始 GAN 论文中使用的惯例。最后，我们设置了两个单独的优化器，一个用于D，一个用于G。 如 DCGAN 论文中所述，两者都是Adam优化器，学习率为0.0002，Beta1 = 0.5。 为了跟踪生成器的学习进度，我们将生成一组固定的潜在 向量，这些向量是从高斯分布（即 `fixed_noise` ）中提取的。在训练循环中，我们将周期性地将此 `fixed_noise` 输入到G 中，并且在迭代中我们将看到图像形成于噪声之外。

```
# 初始化BCELoss函数
criterion = nn.BCELoss()

# 创建一批潜在的向量，我们将用它来可视化生成器的进程
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# 在训练期间建立真假标签的惯例
real_label = 1
fake_label = 0

# 为 G 和 D 设置 Adam 优化器
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

3.3.4 训练

最后，既然已经定义了 GAN 框架的所有部分，我们就可以对其进行训练了。请注意，训练GAN在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对错误的解释很少的模式崩溃，在这里，我们将密切关注Goodfellow的论文中的算法1，同时遵守ganhacks 中展示的一些最佳实

践。也就是说，我们将“为真实和虚假”图像构建不同的 mini-batches ，并且还调整G的目标函 数以最大化$log(D(G(z)))$。训练分为两个主要部分，第1部分更新判别器，第2部分更新生成器。

*** 第一部分：训练判别器**

回想一下，训练判别器的目的是最大化将给定输入正确分类为真实或假的概率。就Goodfellow而言，我们希望“通过提升其随机梯度来更 新判别器”。实际上，我们希望最大化$log(D(x) + log(1 - D(G(z)))$。由于ganhacks的独立 mini-batch 建议，我们将分两步计算。首先，我们将从训练集构建一批实际样本，向前通过D，计算损失$logD(x)$，然后计算向后传递的梯度。其次，我们将用当前生成器构造一批假样本，通过D向前传递该 batch，计算损失$log(1 - D(G(z)))$， 并通过反向传递累积梯度。现在，随着从全实时和全实时批量累积的梯度，我们称之为Discriminator优化器的一步。

*** 第一部分：训练判别器**

正如原始论文所述，我们希望通过最小化$log(1 - D(G(z)))$来训练生成器，以便产生更好的伪样本。如上所述，Goodfellow 表 明这不会提供足够的梯度，尤其是在学习过程的早期阶段。作为修复，我们希望最大化$log(D(G(z)))$。在代码中，我们通过 以下方式实现此目的：使用判别器对第1部分的生成器中的输出进行分类，使用真实标签： GT 标签计算G的损失， 在向后传递中计算G的梯度，最后使用优化器步骤更新G的参数。使用真实标签作为损失函数的GT 标签似乎是违反直觉的，但是这允许我们使用 BCELoss的$log(x)$部分（而不是$log(1 - x)$部分）， 这正是我们想要。

最后，我们将进行一些统计报告，在每个epoch结束时，我们将通过生成器推送我们的fixed_noise batch，以直观地跟踪G 训练的进度。训练的统计数据是：

- **Loss_D**：判别器损失计算为所有实际批次和所有假批次的损失总和($log(D(x) + log(D(G(z)))$)
- **Loss_G**：计算生成器损失($log(D(G(z)))$)
- **D(x)**：所有实际批次的判别器的平均输出（整批）。当G变好时这应该从接近1开始，然后理论上收敛到0.5。 想想为什么会这样。
- **D(G(z))**：所有假批次的平均判别器输出。第一个数字是在D更新之前，第二个数字是在D 更新之后。当G变好时，这些数字应该从0开始并收敛到0.5。想想为什么会这样。

> 此步骤可能需要一段时间，具体取决于您运行的epoch数以及是否从数据集中删除了一些数据。

```
# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # 对于数据加载器中的每个batch
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Add the gradients from the all-real and all-fake batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()
```

```
        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake batch
through D
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.
4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i ==
len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1
```

- 输出结果：

```
Starting Training Loop...
[0/5][0/1583]   Loss_D: 2.0937  Loss_G: 5.2059  D(x): 0.5704    D(G(z)): 0.6680 /
0.0090
[0/5][50/1583]  Loss_D: 0.3567  Loss_G: 12.2064 D(x): 0.9364    D(G(z)): 0.1409 /
0.0000
[0/5][100/1583] Loss_D: 0.3519  Loss_G: 8.8873  D(x): 0.8714    D(G(z)): 0.0327 /
0.0004
[0/5][150/1583] Loss_D: 0.5300  Loss_G: 6.6410  D(x): 0.8918    D(G(z)): 0.2776 /
```

```
0.0030
[0/5][200/1583] Loss_D: 0.2543  Loss_G: 4.3581  D(x): 0.8662    D(G(z)): 0.0844 /
0.0218
[0/5][250/1583] Loss_D: 0.7170  Loss_G: 4.2652  D(x): 0.8285    D(G(z)): 0.3227 /
0.0370
[0/5][300/1583] Loss_D: 0.5739  Loss_G: 4.2060  D(x): 0.8329    D(G(z)): 0.2577 /
0.0305
[0/5][350/1583] Loss_D: 0.8139  Loss_G: 6.5680  D(x): 0.9163    D(G(z)): 0.3844 /
0.0062
[0/5][400/1583] Loss_D: 0.4089  Loss_G: 5.0794  D(x): 0.8580    D(G(z)): 0.1221 /
0.0243
[0/5][450/1583] Loss_D: 0.4785  Loss_G: 4.1612  D(x): 0.7154    D(G(z)): 0.0514 /
0.0258
[0/5][500/1583] Loss_D: 0.3748  Loss_G: 4.2888  D(x): 0.8135    D(G(z)): 0.0955 /
0.0264
[0/5][550/1583] Loss_D: 0.5247  Loss_G: 5.9952  D(x): 0.8347    D(G(z)): 0.1580 /
0.0075
[0/5][600/1583] Loss_D: 0.7765  Loss_G: 2.2662  D(x): 0.5977    D(G(z)): 0.0408 /
0.1708
[0/5][650/1583] Loss_D: 0.6914  Loss_G: 4.4091  D(x): 0.6502    D(G(z)): 0.0266 /
0.0238
[0/5][700/1583] Loss_D: 0.5679  Loss_G: 5.3386  D(x): 0.8476    D(G(z)): 0.2810 /
0.0098
[0/5][750/1583] Loss_D: 0.3717  Loss_G: 5.1295  D(x): 0.9221    D(G(z)): 0.2207 /
0.0106
[0/5][800/1583] Loss_D: 0.4423  Loss_G: 3.1339  D(x): 0.8418    D(G(z)): 0.1655 /
0.0820
[0/5][850/1583] Loss_D: 0.3391  Loss_G: 4.8393  D(x): 0.7920    D(G(z)): 0.0315 /
0.0169
[0/5][900/1583] Loss_D: 0.4346  Loss_G: 4.3887  D(x): 0.8883    D(G(z)): 0.2270 /
0.0202
[0/5][950/1583] Loss_D: 0.5315  Loss_G: 4.6233  D(x): 0.8393    D(G(z)): 0.2490 /
0.0188
[0/5][1000/1583]        Loss_D: 0.5281  Loss_G: 6.1465  D(x): 0.9643    D(G(z)):
0.3270 / 0.0049
[0/5][1050/1583]        Loss_D: 0.5515  Loss_G: 6.4457  D(x): 0.9262    D(G(z)):
0.3361 / 0.0033
[0/5][1100/1583]        Loss_D: 0.4430  Loss_G: 4.7469  D(x): 0.7306    D(G(z)):
0.0184 / 0.0202
[0/5][1150/1583]        Loss_D: 0.7336  Loss_G: 2.6978  D(x): 0.6552    D(G(z)):
0.1293 / 0.1059
[0/5][1200/1583]        Loss_D: 0.2927  Loss_G: 4.7480  D(x): 0.8858    D(G(z)):
0.1329 / 0.0173
[0/5][1250/1583]        Loss_D: 2.0790  Loss_G: 5.1077  D(x): 0.2722    D(G(z)):
0.0036 / 0.0172
[0/5][1300/1583]        Loss_D: 0.2431  Loss_G: 5.0027  D(x): 0.8812    D(G(z)):
0.0816 / 0.0169
[0/5][1350/1583]        Loss_D: 0.2969  Loss_G: 4.6160  D(x): 0.9126    D(G(z)):
0.1609 / 0.0183
```

```
[0/5][1400/1583]        Loss_D: 0.7158  Loss_G: 2.9825  D(x): 0.6117    D(G(z)):
0.0292 / 0.0900
[0/5][1450/1583]        Loss_D: 0.7513  Loss_G: 1.9396  D(x): 0.6186    D(G(z)):
0.0559 / 0.2414
[0/5][1500/1583]        Loss_D: 0.4366  Loss_G: 3.9122  D(x): 0.8736    D(G(z)):
0.2231 / 0.0325
[0/5][1550/1583]        Loss_D: 0.3204  Loss_G: 4.2434  D(x): 0.8395    D(G(z)):
0.0929 / 0.0271
[1/5][0/1583]   Loss_D: 0.5077  Loss_G: 4.8872  D(x): 0.9331    D(G(z)): 0.3082 /
0.0122
[1/5][50/1583]  Loss_D: 0.5637  Loss_G: 3.6652  D(x): 0.8525    D(G(z)): 0.2684 /
0.0414
[1/5][100/1583] Loss_D: 0.4047  Loss_G: 3.6624  D(x): 0.8323    D(G(z)): 0.1508 /
0.0473
[1/5][150/1583] Loss_D: 0.3858  Loss_G: 3.3070  D(x): 0.7873    D(G(z)): 0.0826 /
0.0583
[1/5][200/1583] Loss_D: 0.4348  Loss_G: 3.6292  D(x): 0.8390    D(G(z)): 0.1908 /
0.0417
[1/5][250/1583] Loss_D: 0.5953  Loss_G: 2.1992  D(x): 0.6572    D(G(z)): 0.0649 /
0.1540
[1/5][300/1583] Loss_D: 0.4062  Loss_G: 3.8770  D(x): 0.8655    D(G(z)): 0.2012 /
0.0310
[1/5][350/1583] Loss_D: 0.9472  Loss_G: 1.4837  D(x): 0.4979    D(G(z)): 0.0322 /
0.2947
[1/5][400/1583] Loss_D: 0.5269  Loss_G: 2.6842  D(x): 0.9150    D(G(z)): 0.2922 /
0.1248
[1/5][450/1583] Loss_D: 0.6091  Loss_G: 3.8100  D(x): 0.8194    D(G(z)): 0.2720 /
0.0360
[1/5][500/1583] Loss_D: 0.5674  Loss_G: 3.2716  D(x): 0.8279    D(G(z)): 0.2452 /
0.0610
[1/5][550/1583] Loss_D: 0.8366  Loss_G: 5.5266  D(x): 0.9263    D(G(z)): 0.4840 /
0.0076
[1/5][600/1583] Loss_D: 0.6098  Loss_G: 2.2626  D(x): 0.6424    D(G(z)): 0.0640 /
0.1451
[1/5][650/1583] Loss_D: 0.3970  Loss_G: 3.4130  D(x): 0.8347    D(G(z)): 0.1613 /
0.0491
[1/5][700/1583] Loss_D: 0.5422  Loss_G: 3.1208  D(x): 0.7889    D(G(z)): 0.1972 /
0.0699
[1/5][750/1583] Loss_D: 0.9114  Loss_G: 1.3789  D(x): 0.5066    D(G(z)): 0.0350 /
0.3440
[1/5][800/1583] Loss_D: 1.1917  Loss_G: 5.6081  D(x): 0.9548    D(G(z)): 0.6084 /
0.0064
[1/5][850/1583] Loss_D: 0.4852  Loss_G: 1.9158  D(x): 0.7103    D(G(z)): 0.0636 /
0.1943
[1/5][900/1583] Loss_D: 0.5322  Loss_G: 2.8350  D(x): 0.7762    D(G(z)): 0.1994 /
0.0868
[1/5][950/1583] Loss_D: 0.7765  Loss_G: 1.7411  D(x): 0.5553    D(G(z)): 0.0732 /
0.2260
[1/5][1000/1583]        Loss_D: 0.5518  Loss_G: 4.5488  D(x): 0.9244    D(G(z)):
```

```
0.3354 / 0.0161
[1/5][1050/1583]        Loss_D: 0.4237  Loss_G: 3.2012  D(x): 0.8118    D(G(z)):
0.1651 / 0.0583
[1/5][1100/1583]        Loss_D: 1.1245  Loss_G: 5.5327  D(x): 0.9483    D(G(z)):
0.5854 / 0.0090
[1/5][1150/1583]        Loss_D: 0.5543  Loss_G: 1.9609  D(x): 0.6777    D(G(z)):
0.0933 / 0.1936
[1/5][1200/1583]        Loss_D: 0.4945  Loss_G: 2.0234  D(x): 0.7580    D(G(z)):
0.1329 / 0.1742
[1/5][1250/1583]        Loss_D: 0.5637  Loss_G: 2.9421  D(x): 0.7701    D(G(z)):
0.2123 / 0.0780
[1/5][1300/1583]        Loss_D: 0.6178  Loss_G: 2.5512  D(x): 0.7828    D(G(z)):
0.2531 / 0.1068
[1/5][1350/1583]        Loss_D: 0.4302  Loss_G: 2.5266  D(x): 0.8525    D(G(z)):
0.2053 / 0.1141
[1/5][1400/1583]        Loss_D: 1.5730  Loss_G: 1.4042  D(x): 0.2854    D(G(z)):
0.0183 / 0.3325
[1/5][1450/1583]        Loss_D: 0.6962  Loss_G: 3.3562  D(x): 0.8652    D(G(z)):
0.3732 / 0.0534
[1/5][1500/1583]        Loss_D: 0.7635  Loss_G: 1.4343  D(x): 0.5765    D(G(z)):
0.0807 / 0.3056
[1/5][1550/1583]        Loss_D: 0.4228  Loss_G: 3.3460  D(x): 0.8169    D(G(z)):
0.1671 / 0.0522
[2/5][0/1583]   Loss_D: 0.8332  Loss_G: 1.5990  D(x): 0.6355    D(G(z)): 0.2409 /
0.2433
[2/5][50/1583]  Loss_D: 0.4681  Loss_G: 2.0920  D(x): 0.7295    D(G(z)): 0.0978 /
0.1626
[2/5][100/1583] Loss_D: 0.7995  Loss_G: 2.8227  D(x): 0.7766    D(G(z)): 0.3675 /
0.0828
[2/5][150/1583] Loss_D: 0.3804  Loss_G: 2.6037  D(x): 0.8523    D(G(z)): 0.1729 /
0.1016
[2/5][200/1583] Loss_D: 0.9238  Loss_G: 0.8758  D(x): 0.5284    D(G(z)): 0.1343 /
0.4542
[2/5][250/1583] Loss_D: 0.5205  Loss_G: 2.6795  D(x): 0.7778    D(G(z)): 0.1875 /
0.0934
[2/5][300/1583] Loss_D: 0.7720  Loss_G: 3.8033  D(x): 0.9307    D(G(z)): 0.4405 /
0.0384
[2/5][350/1583] Loss_D: 0.5825  Loss_G: 3.3677  D(x): 0.9309    D(G(z)): 0.3609 /
0.0470
[2/5][400/1583] Loss_D: 0.4290  Loss_G: 2.5963  D(x): 0.7495    D(G(z)): 0.1047 /
0.0976
[2/5][450/1583] Loss_D: 0.7161  Loss_G: 4.0053  D(x): 0.8270    D(G(z)): 0.3655 /
0.0252
[2/5][500/1583] Loss_D: 0.5238  Loss_G: 2.3543  D(x): 0.8084    D(G(z)): 0.2320 /
0.1330
[2/5][550/1583] Loss_D: 0.7724  Loss_G: 2.2096  D(x): 0.6645    D(G(z)): 0.2238 /
0.1417
[2/5][600/1583] Loss_D: 0.4897  Loss_G: 2.8286  D(x): 0.7776    D(G(z)): 0.1738 /
0.0832
```

```
[2/5][650/1583] Loss_D: 1.2680  Loss_G: 4.7502  D(x): 0.8977    D(G(z)): 0.6179 /
0.0149
[2/5][700/1583] Loss_D: 0.7054  Loss_G: 3.3908  D(x): 0.8692    D(G(z)): 0.3753 /
0.0490
[2/5][750/1583] Loss_D: 0.4933  Loss_G: 3.6839  D(x): 0.8933    D(G(z)): 0.2845 /
0.0368
[2/5][800/1583] Loss_D: 0.6246  Loss_G: 2.7728  D(x): 0.8081    D(G(z)): 0.2968 /
0.0821
[2/5][850/1583] Loss_D: 1.2216  Loss_G: 1.1784  D(x): 0.3819    D(G(z)): 0.0446 /
0.3623
[2/5][900/1583] Loss_D: 0.6578  Loss_G: 1.7445  D(x): 0.6494    D(G(z)): 0.1271 /
0.2173
[2/5][950/1583] Loss_D: 0.8333  Loss_G: 1.2805  D(x): 0.5193    D(G(z)): 0.0543 /
0.3210
[2/5][1000/1583]        Loss_D: 0.7348  Loss_G: 0.7953  D(x): 0.5920    D(G(z)):
0.1265 / 0.4815
[2/5][1050/1583]        Loss_D: 0.6809  Loss_G: 3.7259  D(x): 0.8793    D(G(z)):
0.3686 / 0.0401
[2/5][1100/1583]        Loss_D: 0.7728  Loss_G: 2.1345  D(x): 0.5886    D(G(z)):
0.1234 / 0.1626
[2/5][1150/1583]        Loss_D: 0.9383  Loss_G: 3.7146  D(x): 0.8942    D(G(z)):
0.5075 / 0.0355
[2/5][1200/1583]        Loss_D: 0.4951  Loss_G: 2.8725  D(x): 0.8084    D(G(z)):
0.2163 / 0.0764
[2/5][1250/1583]        Loss_D: 0.6952  Loss_G: 2.1559  D(x): 0.6769    D(G(z)):
0.2063 / 0.1561
[2/5][1300/1583]        Loss_D: 0.4560  Loss_G: 2.6873  D(x): 0.7993    D(G(z)):
0.1710 / 0.0908
[2/5][1350/1583]        Loss_D: 0.9185  Loss_G: 3.9262  D(x): 0.8631    D(G(z)):
0.4938 / 0.0276
[2/5][1400/1583]        Loss_D: 0.5935  Loss_G: 1.2768  D(x): 0.6625    D(G(z)):
0.1064 / 0.3214
[2/5][1450/1583]        Loss_D: 0.8836  Loss_G: 4.0820  D(x): 0.9368    D(G(z)):
0.5101 / 0.0251
[2/5][1500/1583]        Loss_D: 0.5268  Loss_G: 2.1486  D(x): 0.7462    D(G(z)):
0.1701 / 0.1450
[2/5][1550/1583]        Loss_D: 0.5581  Loss_G: 3.0543  D(x): 0.8082    D(G(z)):
0.2489 / 0.0644
[3/5][0/1583]   Loss_D: 0.6875  Loss_G: 2.3447  D(x): 0.7796    D(G(z)): 0.3180 /
0.1182
[3/5][50/1583]  Loss_D: 0.7772  Loss_G: 1.2497  D(x): 0.5569    D(G(z)): 0.0763 /
0.3372
[3/5][100/1583] Loss_D: 1.8087  Loss_G: 0.8440  D(x): 0.2190    D(G(z)): 0.0213 /
0.4701
[3/5][150/1583] Loss_D: 0.6292  Loss_G: 2.8794  D(x): 0.8807    D(G(z)): 0.3623 /
0.0741
[3/5][200/1583] Loss_D: 0.5880  Loss_G: 2.2299  D(x): 0.8279    D(G(z)): 0.3026 /
0.1316
[3/5][250/1583] Loss_D: 0.7737  Loss_G: 1.2797  D(x): 0.5589    D(G(z)): 0.0836 /
```

```
0.3363
[3/5][300/1583] Loss_D: 0.5120  Loss_G: 1.5623  D(x): 0.7216    D(G(z)): 0.1406 /
0.2430
[3/5][350/1583] Loss_D: 0.5651  Loss_G: 3.2310  D(x): 0.8586    D(G(z)): 0.3048 /
0.0518
[3/5][400/1583] Loss_D: 1.3554  Loss_G: 5.0320  D(x): 0.9375    D(G(z)): 0.6663 /
0.0112
[3/5][450/1583] Loss_D: 0.5939  Loss_G: 1.9385  D(x): 0.6931    D(G(z)): 0.1538 /
0.1785
[3/5][500/1583] Loss_D: 1.5698  Loss_G: 5.0469  D(x): 0.9289    D(G(z)): 0.7124 /
0.0106
[3/5][550/1583] Loss_D: 0.5496  Loss_G: 1.7024  D(x): 0.6891    D(G(z)): 0.1171 /
0.2172
[3/5][600/1583] Loss_D: 2.0152  Loss_G: 6.4814  D(x): 0.9824    D(G(z)): 0.8069 /
0.0031
[3/5][650/1583] Loss_D: 0.6249  Loss_G: 2.9602  D(x): 0.8547    D(G(z)): 0.3216 /
0.0707
[3/5][700/1583] Loss_D: 0.4448  Loss_G: 2.3997  D(x): 0.8289    D(G(z)): 0.2034 /
0.1153
[3/5][750/1583] Loss_D: 0.5768  Loss_G: 2.5956  D(x): 0.8094    D(G(z)): 0.2721 /
0.1032
[3/5][800/1583] Loss_D: 0.5314  Loss_G: 2.9121  D(x): 0.8603    D(G(z)): 0.2838 /
0.0724
[3/5][850/1583] Loss_D: 0.9673  Loss_G: 4.2585  D(x): 0.9067    D(G(z)): 0.5233 /
0.0206
[3/5][900/1583] Loss_D: 0.7076  Loss_G: 2.7892  D(x): 0.7294    D(G(z)): 0.2625 /
0.0909
[3/5][950/1583] Loss_D: 0.4336  Loss_G: 2.8206  D(x): 0.8736    D(G(z)): 0.2363 /
0.0770
[3/5][1000/1583]        Loss_D: 0.6914  Loss_G: 1.9334  D(x): 0.6811    D(G(z)):
0.2143 / 0.1734
[3/5][1050/1583]        Loss_D: 0.6618  Loss_G: 1.8457  D(x): 0.6486    D(G(z)):
0.1421 / 0.2036
[3/5][1100/1583]        Loss_D: 0.6517  Loss_G: 3.2499  D(x): 0.8540    D(G(z)):
0.3491 / 0.0532
[3/5][1150/1583]        Loss_D: 0.6688  Loss_G: 3.9172  D(x): 0.9389    D(G(z)):
0.4170 / 0.0269
[3/5][1200/1583]        Loss_D: 0.9467  Loss_G: 0.8899  D(x): 0.4853    D(G(z)):
0.1028 / 0.4567
[3/5][1250/1583]        Loss_D: 0.6048  Loss_G: 3.3952  D(x): 0.8353    D(G(z)):
0.3150 / 0.0425
[3/5][1300/1583]        Loss_D: 0.4915  Loss_G: 2.5383  D(x): 0.7663    D(G(z)):
0.1622 / 0.1071
[3/5][1350/1583]        Loss_D: 0.7804  Loss_G: 1.5018  D(x): 0.5405    D(G(z)):
0.0719 / 0.2701
[3/5][1400/1583]        Loss_D: 0.6432  Loss_G: 1.5893  D(x): 0.6069    D(G(z)):
0.0576 / 0.2577
[3/5][1450/1583]        Loss_D: 0.7720  Loss_G: 3.8510  D(x): 0.9291    D(G(z)):
0.4558 / 0.0299
```

```
[3/5][1500/1583]        Loss_D: 0.9340  Loss_G: 4.6210  D(x): 0.9556    D(G(z)):
0.5341 / 0.0141
[3/5][1550/1583]        Loss_D: 0.7278  Loss_G: 4.0992  D(x): 0.9071    D(G(z)):
0.4276 / 0.0231
[4/5][0/1583]   Loss_D: 0.4672  Loss_G: 1.9660  D(x): 0.7085    D(G(z)): 0.0815 /
0.1749
[4/5][50/1583]  Loss_D: 0.5710  Loss_G: 2.3229  D(x): 0.6559    D(G(z)): 0.0654 /
0.1285
[4/5][100/1583] Loss_D: 0.8091  Loss_G: 0.8053  D(x): 0.5301    D(G(z)): 0.0609 /
0.4987
[4/5][150/1583] Loss_D: 0.5661  Loss_G: 1.4238  D(x): 0.6836    D(G(z)): 0.1228 /
0.2842
[4/5][200/1583] Loss_D: 0.6187  Loss_G: 1.6628  D(x): 0.6178    D(G(z)): 0.0744 /
0.2292
[4/5][250/1583] Loss_D: 0.9808  Loss_G: 2.0649  D(x): 0.5769    D(G(z)): 0.2623 /
0.1706
[4/5][300/1583] Loss_D: 0.6530  Loss_G: 2.7874  D(x): 0.8024    D(G(z)): 0.3063 /
0.0804
[4/5][350/1583] Loss_D: 0.5535  Loss_G: 2.5154  D(x): 0.7744    D(G(z)): 0.2165 /
0.1023
[4/5][400/1583] Loss_D: 0.5277  Loss_G: 2.1542  D(x): 0.6766    D(G(z)): 0.0801 /
0.1474
[4/5][450/1583] Loss_D: 0.5995  Loss_G: 2.6477  D(x): 0.7890    D(G(z)): 0.2694 /
0.0902
[4/5][500/1583] Loss_D: 0.7183  Loss_G: 1.2993  D(x): 0.5748    D(G(z)): 0.1000 /
0.3213
[4/5][550/1583] Loss_D: 0.4708  Loss_G: 2.0671  D(x): 0.7286    D(G(z)): 0.1094 /
0.1526
[4/5][600/1583] Loss_D: 0.5865  Loss_G: 1.9083  D(x): 0.7084    D(G(z)): 0.1745 /
0.1867
[4/5][650/1583] Loss_D: 1.5298  Loss_G: 4.2918  D(x): 0.9623    D(G(z)): 0.7240 /
0.0197
[4/5][700/1583] Loss_D: 0.9155  Loss_G: 0.9452  D(x): 0.4729    D(G(z)): 0.0575 /
0.4395
[4/5][750/1583] Loss_D: 0.7500  Loss_G: 1.7498  D(x): 0.5582    D(G(z)): 0.0772 /
0.2095
[4/5][800/1583] Loss_D: 0.5993  Loss_G: 2.5779  D(x): 0.7108    D(G(z)): 0.1829 /
0.1063
[4/5][850/1583] Loss_D: 0.6787  Loss_G: 3.6855  D(x): 0.9201    D(G(z)): 0.4084 /
0.0347
[4/5][900/1583] Loss_D: 1.2792  Loss_G: 2.2909  D(x): 0.6365    D(G(z)): 0.4471 /
0.1575
[4/5][950/1583] Loss_D: 0.6995  Loss_G: 3.3548  D(x): 0.9201    D(G(z)): 0.4188 /
0.0488
[4/5][1000/1583]        Loss_D: 0.6913  Loss_G: 3.9969  D(x): 0.8630    D(G(z)):
0.3771 / 0.0242
[4/5][1050/1583]        Loss_D: 0.7620  Loss_G: 1.7744  D(x): 0.6668    D(G(z)):
0.2290 / 0.2204
[4/5][1100/1583]        Loss_D: 0.6901  Loss_G: 3.1660  D(x): 0.8472    D(G(z)):
```

```
0.3595 / 0.0593
[4/5][1150/1583]        Loss_D: 0.5866  Loss_G: 2.4580  D(x): 0.7962    D(G(z)):
0.2695 / 0.1049
[4/5][1200/1583]        Loss_D: 0.8830  Loss_G: 3.9824  D(x): 0.9264    D(G(z)):
0.5007 / 0.0264
[4/5][1250/1583]        Loss_D: 0.4750  Loss_G: 2.1389  D(x): 0.8004    D(G(z)):
0.1933 / 0.1464
[4/5][1300/1583]        Loss_D: 0.4972  Loss_G: 2.3561  D(x): 0.8266    D(G(z)):
0.2325 / 0.1285
[4/5][1350/1583]        Loss_D: 0.6721  Loss_G: 1.1904  D(x): 0.6042    D(G(z)):
0.0839 / 0.3486
[4/5][1400/1583]        Loss_D: 0.4447  Loss_G: 2.7106  D(x): 0.8540    D(G(z)):
0.2219 / 0.0852
[4/5][1450/1583]        Loss_D: 0.4864  Loss_G: 2.5237  D(x): 0.7153    D(G(z)):
0.1017 / 0.1036
[4/5][1500/1583]        Loss_D: 0.7662  Loss_G: 1.1344  D(x): 0.5429    D(G(z)):
0.0600 / 0.3805
[4/5][1550/1583]        Loss_D: 0.4294  Loss_G: 2.9664  D(x): 0.8335    D(G(z)):
0.1943 / 0.0689
```

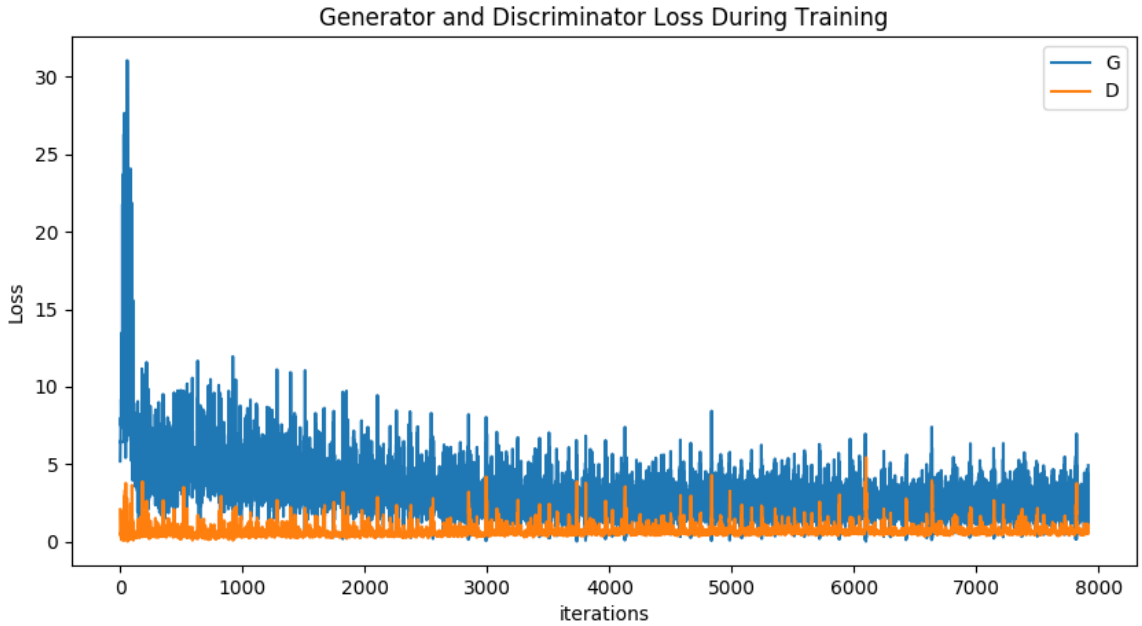
3.3.5 结果

最后，让我们看看我们是如何做到的。在这里，我们将看看三个不同的结果。首先，我们将看到D和G的损失在训练期间是如何变化的。其次，我们将可视化在每个epoch的 fixed_noise batch中G的输出。第三，我们将 查看来自G的紧邻一批实际数据的一批假数据。

损失与训练迭代

下面是D & G的损失与训练迭代的关系图。

```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

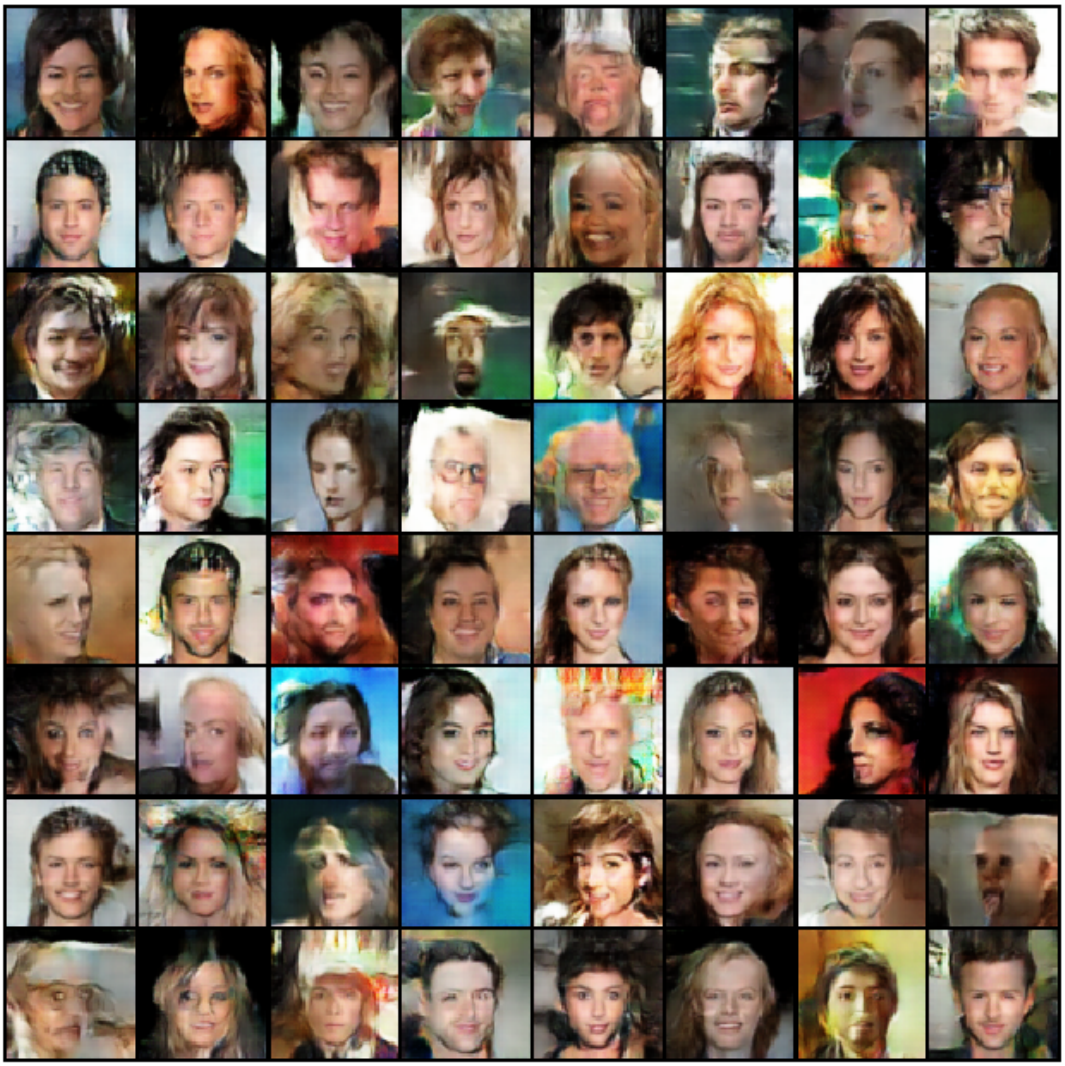


G的过程可视化

记住在每个训练epoch之后我们如何在fixed_noise batch中保存生成器的输出。现在，我们可以通过动画可视化G的训练进度。按播放按钮 开始动画。

```
#%%capture
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000,
blit=True)

HTML(ani.to_jshtml())
```



真实图像 vs 伪图像

最后，让我们一起看看一些真实的图像和伪图像。

```
# 从数据加载器中获取一批真实图像
real_batch = next(iter(dataloader))

# 绘制真实图像
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
```

```
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],
padding=5, normalize=True).cpu(),(1,2,0)))

# 在最后一个epoch中绘制伪图像
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()
```

进一步的工作

我们已经完成了我们的整个教程，但是你可以从下面几个方向进一步探讨。 你可以：

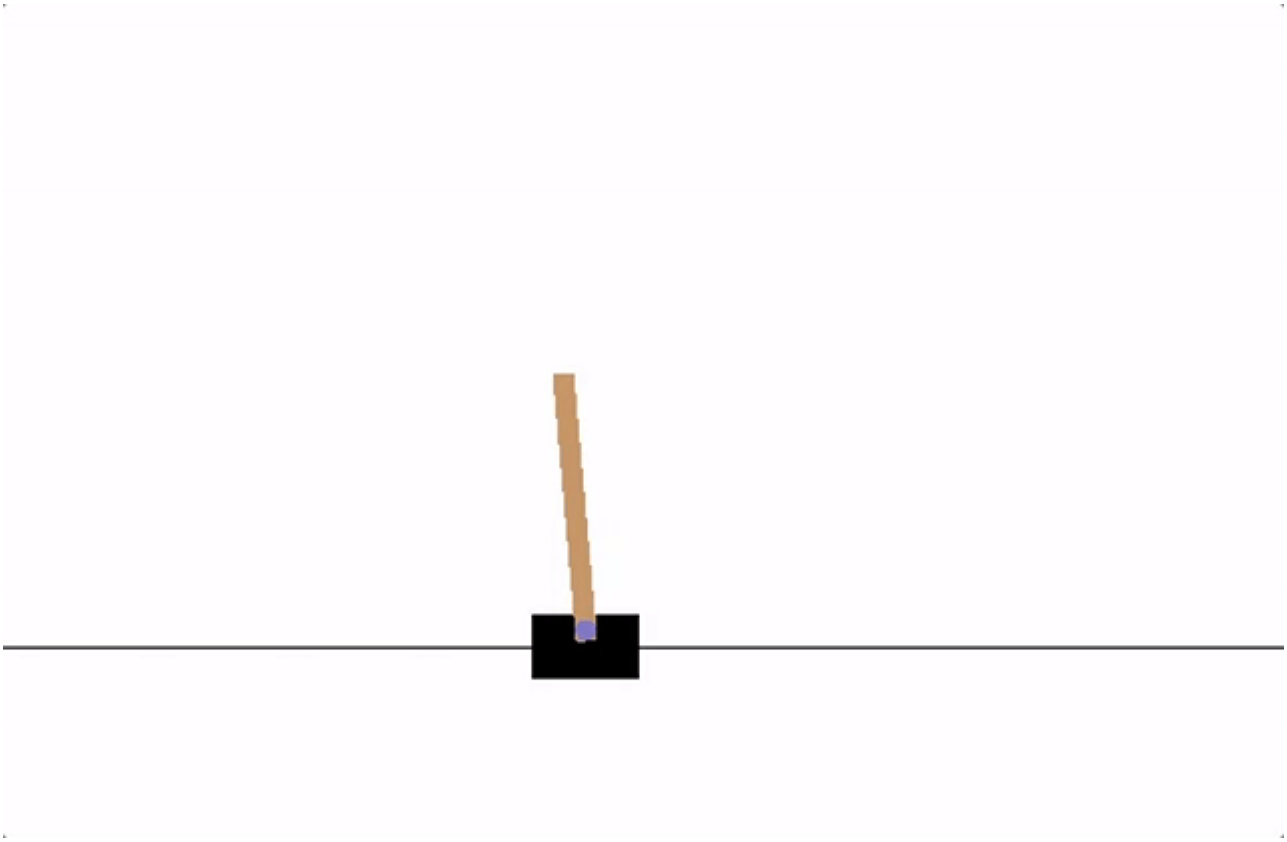
- 训练更长时间，看看结果有多好
- 修改此模型以获取不同的数据集，并或者更改图像和模型体系结构的大小
- 在这里查看其他一些很酷的GAN项目
- 创建生成音乐的GAN

强化学习（DQN）教程

本教程介绍如何使用PyTorch从OpenAI Gym中的 CartPole-v0 任务上训练一个Deep Q Learning (DQN) 代理。

1.任务

代理人必须在两个动作之间做出决定 - 向左或向右移动推车 - 以使连接到它的杆保持直立。您可以在Gym 网站上找到官方排行榜，里面包含各种算法以及可视化。



当代理观察环境的当前状态并选择动作时，环境转换到新状态，并且还返回指示动作的后果的奖励。在此任务中，每增加一个时间步长的 奖励为+1，如果杆落得太远或者推车距离中心超过2.4个单位，则环境终止。这意味着更好的表现场景将持续更长的时间，以及积累更大的回报。

CartPole任务的设计使得代理的输入是4个实际值，表示环境状态（位置，速度等）。然而，神经网络可以纯粹通过观察场景来解决任务， 因此我们将使用以cart为中心的屏幕补丁作为输入。也

因为如此，我们的结果与官方排行榜的结果无法直接比较 - 因为我们的任务 要困难得多。而且不幸的是，这确实减慢了训练速度，因为我们必须渲染所有帧。

严格地说，我们将状态显示为当前屏幕补丁与前一个补丁之间的差异。这将允许代理从一个图像中考虑杆的速度。

2.需要的包

首先，让我们导入所需的包。首先，我们需要gym来得到环境（使用 `pip install gym` ）。我们还将 使用PyTorch中的以下内容：

- 神经网络(`torch.nn`)
- 优化(`torch.optim`)
- 自动分化（ `torch.autograd` ）
- 视觉任务的实用程序(`torchvision`)- 一个单独的包

```
import gym
import math
import random
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple
from itertools import count
from PIL import Image

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision.transforms as T


env = gym.make('CartPole-v0').unwrapped

# set up matplotlib
is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython:
    from IPython import display

plt.ion()
```

```
# if gpu is to be used
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

3. 复现记忆（Replay Memory）

我们将使用经验重播记忆来训练我们的DQN。它存储代理观察到的转换，允许我们之后重用此数据。通过随机抽样，转换构建相关的一个批次。 已经表明经验重播记忆极大地稳定并改善了DQN训练程序。

为此，我们需要两个阶段：* `Transition` ：一个命名元组，表示我们环境中的单个转换。它实际上将（状态，动作）对映射到它们的（next_state，reward）结果， 状态是屏幕差异图像，如稍后所述
* `ReplayMemory` ：有界大小的循环缓冲区，用于保存最近观察到的过渡。它还实现了一个 `.sample()` 方法，用于为训练选择随机batch 的转换。

```
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))


class ReplayMemory(object):

    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, *args):
        """Saves a transition."""
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

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

现在，让我们定义我们的模型。但首先，让我们快速回顾一下DQN是什么。

4. DQN 算法

我们的环境是确定性的，因此为了简单起见，这里给出的所有方程式也是确定性的。在强化学习文献中，它们还包含对环境中随机转变的 期望。

我们的目标是训练出一种政策，试图最大化折现累积奖励 $R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$，其中$R_{t_0}$也称为回报。折扣$\gamma$ 应该是介于0和1之间的常数，以确保总和收敛。对于我们的代理来说，对比不确定的远期未来，它更看重它们相当有信心的不久的将来。

Q-learning背后的主要思想是，如果我们有一个函数$Q^* : State \times Action \to \mathbb{R}$，它可以告诉我们的回报是什么，如果我们要在给定状态下 采取行动，那么我们可以轻松地构建最大化我们奖励的政策：

$$\pi^*(s) = \operatorname{argmax}_a Q^*(s,a)$$

但是，我们不了解世界的一切，因此我们无法访问Q^*。但是，由于神经网络是通用函数逼近器，我们可以简单地创建 一个并训练从而使得它类似于Q^*。

对于我们的训练更新规则，我们将使用一个事实，即某些策略的每个Q函数都服从 Bellman 方程：

$$Q^{\pi}(s,a) = r + \gamma Q^{\pi}(s', \pi(s'))$$

平等的两边之间的差异被称为时间差异误差,δ:

$$\delta = Q(s,a) - (r + \gamma \max_a Q(s', a))$$

为了最大限度地降低此错误，我们将使用Huber损失。当误差很小时，Huber损失就像均 方误差一样，但是当误差很大时，就像平均绝对误差一样 - 当Q的估计噪声很多时，这使得它对异常值更加鲁棒。 我们通过从重放内存中采样的一批转换B来计算：

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,s',r) \in B} \mathcal{L}(\delta)$$

$$\text{where} \quad \mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

5. Q_网络（Q_network）

我们的模型将是一个卷积神经网络，它接收当前和之前的屏幕补丁之间的差异。它有两个输出，分别代表$Q(s, left)$和$Q(s, right)$（其中s是网络的输入）。实际上，网络正在尝试预测在给定当前输入的情况下采取每个动作的预期回报。

```
class DQN(nn.Module):

    def __init__(self, h, w, outputs):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
        self.bn3 = nn.BatchNorm2d(32)

        # 线性输入连接的数量取决于conv2d层的输出，因此取决于输入图像的大小，因此请对其进行计
算。
        def conv2d_size_out(size, kernel_size = 5, stride = 2):
            return (size - (kernel_size - 1) - 1) // stride  + 1
        convw = conv2d_size_out(conv2d_size_out(conv2d_size_out(w)))
        convh = conv2d_size_out(conv2d_size_out(conv2d_size_out(h)))
        linear_input_size = convw * convh * 32
        self.head = nn.Linear(linear_input_size, outputs)

    # 使用一个元素调用以确定下一个操作，或在优化期间调用batch。返回
tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        return self.head(x.view(x.size(0), -1))
```

6. 输入提取

下面的代码是用于从环境中提取和处理渲染图像的实用程序。它使用了 `torchvision` 软件包，可以轻松构成图像变换。运行单元后，它将 显示一个提取的示例补丁。

```
resize = T.Compose([T.ToPILImage(),
                    T.Resize(40, interpolation=Image.CUBIC),
                    T.ToTensor()])


def get_cart_location(screen_width):
    world_width = env.x_threshold * 2
    scale = screen_width / world_width
    return int(env.state[0] * scale + screen_width / 2.0)  # MIDDLE OF CART

def get_screen():
    # gym要求的返回屏幕是400x600x3，但有时更大，如800x1200x3。 将其转换为torch order (
CHW) 。
    screen = env.render(mode='rgb_array').transpose((2, 0, 1))
    # cart位于下半部分，因此不包括屏幕的顶部和底部
    _, screen_height, screen_width = screen.shape
    screen = screen[:, int(screen_height*0.4):int(screen_height * 0.8)]
    view_width = int(screen_width * 0.6)
    cart_location = get_cart_location(screen_width)
    if cart_location < view_width // 2:
        slice_range = slice(view_width)
    elif cart_location > (screen_width - view_width // 2):
        slice_range = slice(-view_width, None)
    else:
        slice_range = slice(cart_location - view_width // 2,
                            cart_location + view_width // 2)
    # 去掉边缘，使得我们有一个以cart为中心的方形图像
    screen = screen[:, :, slice_range]
    # 转换为float类型，重新缩放，转换为torch张量
    # (this doesn't require a copy)
    screen = np.ascontiguousarray(screen, dtype=np.float32) / 255
    screen = torch.from_numpy(screen)
    # 调整大小并添加batch维度 (BCHW)
    return resize(screen).unsqueeze(0).to(device)


env.reset()
plt.figure()
plt.imshow(get_screen().cpu().squeeze(0).permute(1, 2, 0).numpy(),
           interpolation='none')
```

```
plt.title('Example extracted screen')
plt.show()
```

7. 训练

7.1 超参数和实用程序

这个单元实例化我们的模型及其优化器，并定义了一些实用程序：* `select_action` ：将根据epsilon贪婪政策选择一项行动。简而言之，我们有时会使用我们的模型来选择动作，有时我们只会统一采样。 选择随机操作的概率将从 `EPS_START` 开始，并将以指数方式向 `EPS_END` 衰减。 `EPS_DECAY` 控制衰减的速度
* `plot_durations` ：帮助绘制episodes的持续时间，以及过去100个episodes的平均值（官方评估中使用的度量）。该图将位于包含主 要训练循环的单元下方，并将在每个episodes后更新。

```
BATCH_SIZE = 128
GAMMA = 0.999
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200
TARGET_UPDATE = 10

# 获取屏幕大小，以便我们可以根据AI gym返回的形状正确初始化图层。
# 此时的典型尺寸接近3x40x90
# 这是get_screen () 中的限幅和缩小渲染缓冲区的结果
init_screen = get_screen()
_, _, screen_height, screen_width = init_screen.shape

# 从gym行动空间中获取行动数量
n_actions = env.action_space.n

policy_net = DQN(screen_height, screen_width, n_actions).to(device)
target_net = DQN(screen_height, screen_width, n_actions).to(device)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

optimizer = optim.RMSprop(policy_net.parameters())
memory = ReplayMemory(10000)


steps_done = 0


def select_action(state):
```

```
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
        math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1)将返回每行的最大列值。
            # 最大结果的第二列是找到最大元素的索引，因此我们选择具有较大预期奖励的行动。
            return policy_net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[random.randrange(n_actions)]], device=device,
dtype=torch.long)


episode_durations = []


def plot_durations():
    plt.figure(2)
    plt.clf()
    durations_t = torch.tensor(episode_durations, dtype=torch.float)
    plt.title('Training...')
    plt.xlabel('Episode')
    plt.ylabel('Duration')
    plt.plot(durations_t.numpy())
    # 取100个episode的平均值并绘制它们
    if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
        plt.plot(means.numpy())

    plt.pause(0.001)  # 暂停一下，以便更新图表
    if is_ipython:
        display.clear_output(wait=True)
        display.display(plt.gcf())
```

8. 训练循环

在这里，您可以找到执行优化的单个步骤的 `optimize_model` 函数。它首先对一个batch进行采样，将所有张量连接成一个整体，计算$Q(s_t, a_t)$和$V(s_{t+1}) = max_a Q(s_{t+1}, a)$，并将它们组合成我们的损失。通过定义，如果s是终端状态，则设置$V(s) = 0$。我们还使 用目标网络来计算$V(s_{t+1})$以增加稳定性。目标网络的权重在大多数时间保持冻结状态，但每隔一段时间就会更新策略 网络的权重。这通常是一系列步骤，但为了简单起见，我们将使用episodes。

```
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # 转置batch（有关详细说明，请参阅https://stackoverflow.com/a/19343/3343043）。
    # 这会将过渡的batch数组转换为batch数组的过渡。
    batch = Transition(*zip(*transitions))

    # 计算非最终状态的掩码并连接batch元素（最终状态将是模拟结束后的状态）
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)), device=device,
dtype=torch.uint8)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                       if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # 计算Q(s_t, a) - 模型计算Q(s_t)，然后我们选择所采取的动作列。
    # 这些是根据policy_net对每个batch状态采取的操作
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # 计算所有下一个状态的V(s_{t+1})
    # non_final_next_states的操作的预期值是基于“较旧的”target_net计算的;
    # 用max(1)[0]选择最佳奖励。这是基于掩码合并的，这样我们就可以得到预期的状态值，或者在状
态是最终的情况下为0。
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)
[0].detach()
    # 计算预期的Q值
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # 计算Huber损失
    loss = F.smooth_l1_loss(state_action_values,
expected_state_action_values.unsqueeze(1))

    # 优化模型
    optimizer.zero_grad()
    loss.backward()
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1)
    optimizer.step()
```

下面，您可以找到主要的训练循环。在开始时，我们重置环境并初始`state`张量。然后，我们采样一个动作并执行它，观察下一个屏幕和奖励（总是1），并优化我们的模型一次。当episode结束时（我们的模型失败），我们重新开始循环。

下面，*num_episodes*设置为小数值。您应该下载笔记本并运行更多的epsiodes，例如300+以进行有意义的持续时间改进。

```
num_episodes = 50
for i_episode in range(num_episodes):
    # 初始化环境和状态
    env.reset()
    last_screen = get_screen()
    current_screen = get_screen()
    state = current_screen - last_screen
    for t in count():
        # 选择动作并执行
        action = select_action(state)
        _, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)

        # 观察新的状态
        last_screen = current_screen
        current_screen = get_screen()
        if not done:
            next_state = current_screen - last_screen
        else:
            next_state = None

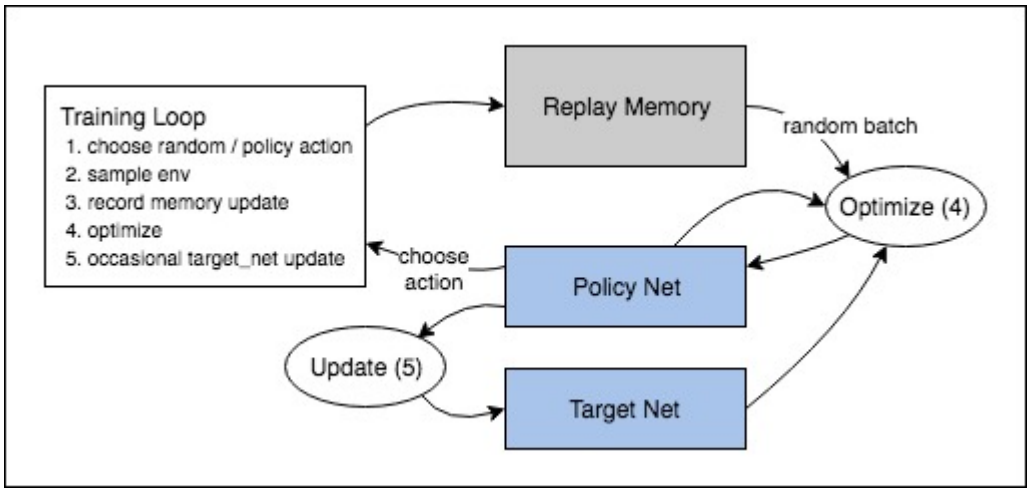
        # 在记忆中存储过渡
        memory.push(state, action, next_state, reward)

        # 移动到下一个状态
        state = next_state

        # 执行优化的一个步骤（在目标网络上）
        optimize_model()
        if done:
            episode_durations.append(t + 1)
            plot_durations()
            break
    # 更新目标网络，复制DQN中的所有权重和偏差
    if i_episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict())

print('Complete')
env.render()
env.close()
plt.ioff()
plt.show()
```

以下是说明整体结果数据流的图表。



可以随机选择或根据策略选择操作，从gym环境中获取下一步样本。我们将结果记录在重放内存中，并在每次迭代时运行优化步骤。优化从 重放内存中选择一个随机batch来进行新策略的训练。“较旧的”target_net也用于优化以计算预期的Q值; 它会偶尔更新以保持最新状态。

通过带Flask的REST API在Python中部署PyTorch

在本教程中，我们将使用Flask来部署PyTorch模型，并用讲解用于模型推断的 REST API。特别是，我们将部署一个预训练的DenseNet 121模 型来检测图像。

> 备注： 可在GitHub上获取本文用到的完整代码

这是在生产中部署PyTorch模型的系列教程中的第一篇。到目前为止，以这种方式使用Flask是开始为PyTorch模型提供服务的最简单方法， 但不适用于具有高性能要求的用例。因此： * 如果您已经熟悉TorchScript，则可以直接进入我们的Loading a TorchScript Model in C++教程。 * 如果您首先需要复习TorchScript，请查看我们的Intro a TorchScript教程。

1.定义API 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` ，它接受带有包含图像的 `file` 参数的HTTP POST请求。响应 将是包含预测的JSON响应： ```buildoutcfg {“class_id”: “n02124075”, “class_name”: “Egyptian_cat”}

```
## 2.依赖（包）
运行下面的命令来下载我们需要的依赖：
```buildoutcfg
$ pip install Flask==1.0.3 torchvision-0.3.0
```

3.简单的Web服务器

以下是一个简单的Web服务器，摘自Flask文档

```
from flask import Flask
app = Flask(__name__)


@app.route('/')
def hello():
    return 'Hello World!'
```

将以上代码段保存在名为 `app.py` 的文件中，您现在可以通过输入以下内容来运行Flask开发服务器：

```
```
```

```
$ FLASK_ENV=development FLASK_APP=app.py flask run
```

当您在web浏览器中访问 `http://localhost:5000/` 时，您会收到文本 `Hello World` 的问候！

我们将对以上代码片段进行一些更改，以使其适合我们的API定义。首先，我们将重命名 `predict` 方法。我们将端点路径更新为 `/predict`。 由于图像文件将通过HTTP POST请求发送，因此我们将对其进行更新，使其也仅接受POST请求：

```
@app.route('/predict', methods=['POST'])
def predict():
    return 'Hello World!'
```

我们还将更改响应类型，以使其返回包含ImageNet类的id和name的JSON响应。更新后的 `app.py` 文件现在为：

```
from flask import Flask, jsonify
app = Flask(__name__)

@app.route('/predict', methods=['POST'])
def predict():
    return jsonify({'class_id': 'IMAGE_NET_XXX', 'class_name': 'Cat'})
```

4.推理

在下一部分中，我们将重点介绍编写推理代码。这将涉及两部分，第一部分是准备图像，以便可以将其馈送到DenseNet；第二部分，我们将编 写代码以从模型中获取实际的预测。

4.1 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。你可以点击 这里来了解更多关于它的内容。

我们将使用来自 `torchvision` 库的 `transforms` 来建立转换管道，该转换管道可根据需要转换图像。您可以在此处 阅读有关转换的更多信息。

```
import io

import torchvision.transforms as transforms
```

```
from PIL import Image

def transform_image(image_bytes):
    my_transforms = transforms.Compose([transforms.Resize(255),
                                        transforms.CenterCrop(224),
                                        transforms.ToTensor(),
                                        transforms.Normalize(
                                            [0.485, 0.456, 0.406],
                                            [0.229, 0.224, 0.225])])
    image = Image.open(io.BytesIO(image_bytes))
    return my_transforms(image).unsqueeze(0)
```

上面的方法以字节为单位获取图像数据，应用一系列变换并返回张量。要测试上述方法，请以字节模式读取图像文件（首先将../_static/img/ sample_file.jpeg替换为计算机上文件的实际路径），然后查看是否获得了张量：

```
with open("../_static/img/sample_file.jpeg", 'rb') as f:
    image_bytes = f.read()
    tensor = transform_image(image_bytes=image_bytes)
    print(tensor)
```

- 输出结果：

```
tensor([[[[ 0.4508,  0.4166,  0.3994,  ..., -1.3473, -1.3302, -1.3473],
          [ 0.5364,  0.4851,  0.4508,  ..., -1.2959, -1.3130, -1.3302],
          [ 0.7077,  0.6392,  0.6049,  ..., -1.2959, -1.3302, -1.3644],
          ...,
          [ 1.3755,  1.3927,  1.4098,  ...,  1.1700,  1.3584,  1.6667],
          [ 1.8893,  1.7694,  1.4440,  ...,  1.2899,  1.4783,  1.5468],
          [ 1.6324,  1.8379,  1.8379,  ...,  1.4783,  1.7352,  1.4612]],

         [[ 0.5728,  0.5378,  0.5203,  ..., -1.3704, -1.3529, -1.3529],
          [ 0.6604,  0.6078,  0.5728,  ..., -1.3004, -1.3179, -1.3354],
          [ 0.8529,  0.7654,  0.7304,  ..., -1.3004, -1.3354, -1.3704],
          ...,
          [ 1.4657,  1.4657,  1.4832,  ...,  1.3256,  1.5357,  1.8508],
          [ 2.0084,  1.8683,  1.5182,  ...,  1.4657,  1.6583,  1.7283],
          [ 1.7458,  1.9384,  1.9209,  ...,  1.6583,  1.9209,  1.6408]],

         [[ 0.7228,  0.6879,  0.6531,  ..., -1.6476, -1.6302, -1.6476],
          [ 0.8099,  0.7576,  0.7228,  ..., -1.6476, -1.6476, -1.6650],
          [ 1.0017,  0.9145,  0.8797,  ..., -1.6476, -1.6650, -1.6999],
          ...,
          [ 1.6291,  1.6291,  1.6465,  ...,  1.6291,  1.8208,  2.1346],
          [ 2.1868,  2.0300,  1.6814,  ...,  1.7685,  1.9428,  2.0125],
          [ 1.9254,  2.0997,  2.0823,  ...,  1.9428,  2.2043,  1.9080]]]])
```

4.2 预测

现在将使用预训练的DenseNet 121模型来预测图像的类别。我们将使用 `torchvision` 库中的一个库，加载模型并进行推断。在此示例中，我们 将使用预训练的模型，但您可以对自己的模型使用相同的方法。在这个教程 中了解有关加载模型的更多信息。

```
from torchvision import models

# 确保使用`pretrained`作为`True`来使用预训练的权重：
model = models.densenet121(pretrained=True)
# 由于我们仅将模型用于推理，因此请切换到“eval”模式：
model.eval()


def get_prediction(image_bytes):
    tensor = transform_image(image_bytes=image_bytes)
    outputs = model.forward(tensor)
    _, y_hat = outputs.max(1)
    return y_hat
```

张量 `y_hat` 将包含预测的类的id的索引。但是，我们需要一个易于阅读的类名。为此，我们需要一个类id来命名映射。将该文件 下载为 `imagenet_class_index.json` 并记住它的保存位置（或者，如果您按照本教程中的确切步骤操作，请将其保存在 `tutorials/_static` 中）。 此文件包含ImageNet类的id到ImageNet类的name的映射。我们将加载此JSON文件并获取预测索引的类的name。

```
import json

imagenet_class_index = json.load(open('../_static/imagenet_class_index.json'))

def get_prediction(image_bytes):
    tensor = transform_image(image_bytes=image_bytes)
    outputs = model.forward(tensor)
    _, y_hat = outputs.max(1)
    predicted_idx = str(y_hat.item())
    return imagenet_class_index[predicted_idx]
```

在使用字典 `imagenet_class_index` 之前，首先我们将张量值转换为字符串值，因为字典 `imagenet_class_index` 中的keys是字符串。我们将 测试上述方法：

```
with open("../_static/img/sample_file.jpeg", 'rb') as f:
    image_bytes = f.read()
    print(get_prediction(image_bytes=image_bytes))
```

* 输出结果：

```
['n02124075', 'Egyptian_cat']
```

你会得到这样的一个响应：

```
['n02124075', 'Egyptian_cat']
```

数组中的第一项是ImageNet类的id，第二项是人类可读的name。

> 注意：您是否注意到模型变量不是 `get_prediction` 方法的一部分？或者为什么模型是全局变量？就内存和计算而言，加载模型可能是 一项昂贵的操作。如果将模型加载到 `get_prediction` 方法中，则每次调用该方法时都会不必要地加载该模型。由于我们正在构建Web服务 器，因此每秒可能有成千上万的请求，因此我们不应该浪费时间为每个推断重复加载模型。因此，我们仅将模型加载到内存中一次。在生 产系统中，必须有效利用计算以能够大规模处理请求，因此通常应在处理请求之前加载模型。

5.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：

```
from flask import request

@app.route('/predict', methods=['POST'])
def predict():
    if request.method == 'POST':
        # 从请求中获得文件
        file = request.files['file']
        # 转化为字节
        img_bytes = file.read()
        class_id, class_name = get_prediction(image_bytes=img_bytes)
        return jsonify({'class_id': class_id, 'class_name': class_name})
```

`app.py` 文件现已完成。以下是完整版本；将路径替换为保存文件的路径，它的运行应是如下：

```
import io
import json

from torchvision import models
import torchvision.transforms as transforms
from PIL import Image
from flask import Flask, jsonify, request


app = Flask(__name__)
imagenet_class_index = json.load(open('<PATH/TO/.json/FILE>/
imagenet_class_index.json'))
model = models.densenet121(pretrained=True)
model.eval()


def transform_image(image_bytes):
    my_transforms = transforms.Compose([transforms.Resize(255),
                                        transforms.CenterCrop(224),
                                        transforms.ToTensor(),
                                        transforms.Normalize(
                                            [0.485, 0.456, 0.406],
                                            [0.229, 0.224, 0.225])])
    image = Image.open(io.BytesIO(image_bytes))
    return my_transforms(image).unsqueeze(0)
```

```
def get_prediction(image_bytes):
    tensor = transform_image(image_bytes=image_bytes)
    outputs = model.forward(tensor)
    _, y_hat = outputs.max(1)
    predicted_idx = str(y_hat.item())
    return imagenet_class_index[predicted_idx]


@app.route('/predict', methods=['POST'])
def predict():
    if request.method == 'POST':
        file = request.files['file']
        img_bytes = file.read()
        class_id, class_name = get_prediction(image_bytes=img_bytes)
        return jsonify({'class_id': class_id, 'class_name': class_name})


if __name__ == '__main__':
    app.run()
```

让我们测试一下我们的web服务器，运行：

```
$ FLASK_ENV=development FLASK_APP=app.py flask run
```

我们可以使用requests库来发送一个POST请求到我们的app：

```
import requests

resp = requests.post("http://localhost:5000/predict",
                     files={"file": open('<PATH/TO/.jpg/FILE>/cat.jpg','rb')})
```

打印 `resp.json()` 会显示下面的结果：

```
{"class_id": "n02124075", "class_name": "Egyptian_cat"}
```

6.下一步工作

我们编写的服务器非常琐碎，可能无法完成生产应用程序所需的一切。因此，您可以采取一些措施来改善它：

- 端点 `/predict` 假定请求中总会有一个图像文件。这可能不适用于所有请求。我们的用户可能发送带有其他参数的图像，或者根本不发送任何图像。
- 用户也可以发送非图像类型的文件。由于我们没有处理错误，因此这将破坏我们的服务器。添加显式的错误处理路径来引发异常，这将使我们 能够更好地处理错误的输入
- 即使模型可以识别大量类别的图像，也可能无法识别所有图像。增强实现以处理模型无法识别图像中的任何情况的情况。
- 我们在开发模式下运行Flask服务器，该服务器不适合在生产中进行部署。您可以查看教程 以在生产环境中部署Flask服务器。
- 您还可以通过创建一个带有表单的页面来添加UI，该表单可以拍摄图像并显示预测。查看类似项目的演示及其源代码。
- 在本教程中，我们仅展示了如何构建可以一次返回单个图像预测的服务。我们可以修改服务以能够一次返回多个图像的预测。此外，service-streamer 库自动将对服务的请求排队，并将它们采样到可用于模型的min-batches中。您可以查看此教程。
- 最后，我们鼓励您在页面顶部查看链接到的有关部署PyTorch模型的其他教程。

TorchScript简介

本教程是对TorchScript的简介，TorchScript是PyTorch模型（`nn.Module`的子类）的中间表示，可以在高性能环境（例如C ++）中运行。

在本教程中，我们将介绍：

1. PyTorch中的模型创作基础，包括：
2. 模组
3. 定义前向功能
4. 将模块组成模块的层次结构
5. 将PyTorch模块转换为TorchScript（我们的高性能部署运行时）的特定方法
6. 跟踪现有模块
7. 使用脚本直接编译模块
8. 如何组合这两种方法
9. 保存和加载TorchScript模块

我们希望在完成本教程之后，您将继续阅读后续教程，该教程将引导您真正地从C ++调用TorchScript模型的示例。

```
import torch  # 这是同时使用PyTorch和TorchScript所需的全部导入！
print(torch.__version__)
```

- 输出结果

```
1.3.0
```

1.PyTorch模型创作的基础

让我们开始定义一个简单的模块。模块是PyTorch中组成的基本单位。它包含：

- 构造函数，为调用准备模块

- 一组参数和子模块。这些由构造函数初始化，并且可以在调用期间由模块使用。
- 前进功能。这是调用模块时运行的代码。 我们来看一个小例子：

```
class MyCell(torch.nn.Module):
    def __init__(self):
        super(MyCell, self).__init__()

    def forward(self, x, h):
        new_h = torch.tanh(x + h)
        return new_h, new_h

my_cell = MyCell()
x = torch.rand(3, 4)
h = torch.rand(3, 4)
print(my_cell(x, h))
```

- 输出结果

```
(tensor([[0.5139, 0.6451, 0.3697, 0.7738],
        [0.7936, 0.5864, 0.8063, 0.9324],
        [0.6479, 0.8408, 0.8062, 0.7263]]), tensor([[0.5139, 0.6451, 0.3697, 0.
7738],
        [0.7936, 0.5864, 0.8063, 0.9324],
        [0.6479, 0.8408, 0.8062, 0.7263]]))
```

因此，我们已经：

1. 创建了一个子类 `torch.nn.Module` 的类。
2. 定义一个构造函数。构造函数没有做太多事情，只是将构造函数称为super。
3. 定义了一个正向功能，该功能需要两个输入并返回两个输出。前向函数的实际内容并不是很重要，但是它是一种伪造的RNN单元 -即，该函数应用于循环。

我们实例化了该模块，并制作了 `x` 和 `y` ，它们只是3x4的随机值矩阵。 然后，我们使用 `my_cell (x, h)` 调用该单元格。 这又调用了我们的转发功能。

让我们做一些更有趣的事情：

```
class MyCell(torch.nn.Module):
    def __init__(self):
        super(MyCell, self).__init__()
        self.linear = torch.nn.Linear(4, 4)
```

```
    def forward(self, x, h):
        new_h = torch.tanh(self.linear(x) + h)
        return new_h, new_h

my_cell = MyCell()
print(my_cell)
print(my_cell(x, h))
```

- 输出结果

```
MyCell(
  (linear): Linear(in_features=4, out_features=4, bias=True)
)
(tensor([[ 0.3941,  0.4160, -0.1086,  0.8432],
        [ 0.5604,  0.4003,  0.5009,  0.6842],
        [ 0.7084,  0.7147,  0.1818,  0.8296]], grad_fn=<TanhBackward>),
tensor([[ 0.3941,  0.4160, -0.1086,  0.8432],
        [ 0.5604,  0.4003,  0.5009,  0.6842],
        [ 0.7084,  0.7147,  0.1818,  0.8296]], grad_fn=<TanhBackward>))
```

我们已经重新定义了模块 `MyCell` ，但是这次我们添加了 `self.linear` 属性，并在前进（forward）函数中调用了 `self.linear` 。

这里到底发生了什么？ `torch.nn.Linear` 是PyTorch标准库中的模块。就像 `MyCell` 一样，可以使用调用语法来调用它。我们正在建立模块的层次结构。

在模块上打印可以直观地表示该模块的子类层次结构。在我们的示例中，我们可以看到我们的线性子类及其参数。

通过以这种方式组合模块，我们可以简洁而易读地编写具有可重用组件的模型。

您可能已经在输出中注意到 `grad_fn` 。这是PyTorch的自动区分方法（称为autograd）的详细信息。简而言之，该系统允许我们通过潜在的 复杂程序来计算导数。该设计为模型创作提供了极大的灵活性。

现在，让我们检查一下它的灵活性：

```
class MyDecisionGate(torch.nn.Module):
    def forward(self, x):
        if x.sum() > 0:
            return x
        else:
            return -x
```

```
class MyCell(torch.nn.Module):
    def __init__(self):
        super(MyCell, self).__init__()
        self.dg = MyDecisionGate()
        self.linear = torch.nn.Linear(4, 4)

    def forward(self, x, h):
        new_h = torch.tanh(self.dg(self.linear(x)) + h)
        return new_h, new_h

my_cell = MyCell()
print(my_cell)
print(my_cell(x, h))
```

- 输出结果

```
MyCell(
  (dg): MyDecisionGate()
  (linear): Linear(in_features=4, out_features=4, bias=True)
)
(tensor([[0.0850, 0.2812, 0.5188, 0.8523],
        [0.1233, 0.3948, 0.6615, 0.7466],
        [0.7072, 0.6103, 0.6953, 0.7047]], grad_fn=<TanhBackward>), tensor([[0.
0850, 0.2812, 0.5188, 0.8523],
        [0.1233, 0.3948, 0.6615, 0.7466],
        [0.7072, 0.6103, 0.6953, 0.7047]], grad_fn=<TanhBackward>))
```

我们再次重新定义了 `MyCell` 类，但是在这里我们定义了 `MyDecisionGate` 。该模块利用控制流程。控制流包括循环和if语句之类的东西。

给定完整的程序表示形式，许多框架都采用计算符号派生的方法。但是，在PyTorch中，我们使用梯度带。我们记录操作发生时的操作，并在 计算衍生产品时向后回放。这样，框架不必为语言中的所有构造显式定义派生类。

2.TorchScript的基础

现在，让我们以正在运行的示例为例，看看如何应用TorchScript。

简而言之，即使PyTorch具有灵活和动态的特性，TorchScript也提供了捕获模型定义的工具。让我们开始研究所谓的*跟踪*。

2.1 跟踪（Tracing）模块

```
class MyCell(torch.nn.Module):
    def __init__(self):
        super(MyCell, self).__init__()
        self.linear = torch.nn.Linear(4, 4)

    def forward(self, x, h):
        new_h = torch.tanh(self.linear(x) + h)
        return new_h, new_h

my_cell = MyCell()
x, h = torch.rand(3, 4), torch.rand(3, 4)
traced_cell = torch.jit.trace(my_cell, (x, h))
print(traced_cell)
traced_cell(x, h)
```

- 输出结果

```
TracedModule[MyCell](
  original_name=MyCell
  (linear): TracedModule[Linear](original_name=Linear)
)
```

我们倒退了一点，并选择了 `MyCell` 类的第二个版本。和以前一样，我们实例化了它，但是这次，我们调用了 `torch.jit.trace` ，在 `Module` (模块)中 传递了该示例，并在示例中传递了网络可能看到的输入。

这到底是做什么的？ 它已调用模块，记录了模块运行时发生的操作，并创建了 `torch.jit.ScriptModule` 的实例（ `TracedModule` 是其实例）

TorchScript将其定义记录在中间表示（或IR）中，在深度学习中通常称为图形。我们可以检查带有 `.graph` 属性的图：

```
print(traced_cell.graph)
```

- 输出结果

```
graph(%self : ClassType<MyCell>,
      %input : Float(3, 4),
      %h : Float(3, 4)):
```

```
  %1 : ClassType<Linear> = prim::GetAttr[name="linear"](%self)
  %weight : Tensor = prim::GetAttr[name="weight"](%1)
  %bias : Tensor = prim::GetAttr[name="bias"](%1)
  %6 : Float(4, 4) = aten::t(%weight), scope: MyCell/Linear[linear] # /opt/
conda/lib/python3.6/site-packages/torch/nn/functional.py:1370:0
  %7 : int = prim::Constant[value=1](), scope: MyCell/Linear[linear] # /opt/
conda/lib/python3.6/site-packages/torch/nn/functional.py:1370:0
  %8 : int = prim::Constant[value=1](), scope: MyCell/Linear[linear] # /opt/
conda/lib/python3.6/site-packages/torch/nn/functional.py:1370:0
  %9 : Float(3, 4) = aten::addmm(%bias, %input, %6, %7, %8), scope: MyCell/
Linear[linear] # /opt/conda/lib/python3.6/site-packages/torch/nn/functional.py:
1370:0
  %10 : int = prim::Constant[value=1](), scope: MyCell # /var/lib/jenkins/
workspace/beginner_source/Intro_to_TorchScript_tutorial.py:188:0
  %11 : Float(3, 4) = aten::add(%9, %h, %10), scope: MyCell # /var/lib/jenkins/
workspace/beginner_source/Intro_to_TorchScript_tutorial.py:188:0
  %12 : Float(3, 4) = aten::tanh(%11), scope: MyCell # /var/lib/jenkins/
workspace/beginner_source/Intro_to_TorchScript_tutorial.py:188:0
  %13 : (Float(3, 4), Float(3, 4)) = prim::TupleConstruct(%12, %12)
  return (%13)
```

但是，这是一个非常低级的表示形式，图中包含的大多数信息对最终用户没有用。相反，我们可以使用 `.code` 属性来给出代码的Python语法解释：

```
print(traced_cell.code)
```

- 输出结果

```
import __torch__
import __torch__.torch.nn.modules.linear
def forward(self,
    input: Tensor,
    h: Tensor) -> Tuple[Tensor, Tensor]:
  _0 = self.linear
  weight = _0.weight
  bias = _0.bias
  _1 = torch.addmm(bias, input, torch.t(weight), beta=1, alpha=1)
  _2 = torch.tanh(torch.add(_1, h, alpha=1))
  return (_2, _2)
```

那么为什么我们要做所有这些呢？ 有以下几个原因：

1. TorchScript代码可以在其自己的解释器中调用，该解释器基本上是受限制的Python解释器。该解释器不被全局解释器锁定，因此可以在同 一实例上同时处理许多请求。

2. 这种格式使我们可以将整个模型保存到磁盘上，并将其加载到另一个环境中，例如在以Python以外的语言编写的服务器中
3. TorchScript为我们提供了一种表示形式，其中我们可以对代码进行编译器优化以提供更有效的执行
4. TorchScript允许我们与许多后端/设备运行时进行接口，这些运行时比单个操作员需要更广泛的程序视图。

我们可以看到，调用 `traced_cell` 产生的结果与Python模块相同：

```
print(my_cell(x, h))
print(traced_cell(x, h))
```

- 输出结果

```
(tensor([[-0.3983,  0.5954,  0.2587, -0.3748],
        [-0.5033,  0.4471,  0.8264,  0.2135],
        [ 0.3430,  0.5561,  0.6794, -0.2273]], grad_fn=<TanhBackward>),
tensor([[-0.3983,  0.5954,  0.2587, -0.3748],
        [-0.5033,  0.4471,  0.8264,  0.2135],
        [ 0.3430,  0.5561,  0.6794, -0.2273]], grad_fn=<TanhBackward>))
(tensor([[-0.3983,  0.5954,  0.2587, -0.3748],
        [-0.5033,  0.4471,  0.8264,  0.2135],
        [ 0.3430,  0.5561,  0.6794, -0.2273]],
       grad_fn=<DifferentiableGraphBackward>), tensor([[-0.3983,  0.5954,  0.
2587, -0.3748],
        [-0.5033,  0.4471,  0.8264,  0.2135],
        [ 0.3430,  0.5561,  0.6794, -0.2273]],
       grad_fn=<DifferentiableGraphBackward>))
```

3.使用脚本转换模块

有一个原因是我们使用了模块的第二版，而不是使用带有大量控制流的子模块。现在让我们检查一下：

```
class MyDecisionGate(torch.nn.Module):
    def forward(self, x):
        if x.sum() > 0:
            return x
        else:
            return -x
```

```
class MyCell(torch.nn.Module):
    def __init__(self, dg):
        super(MyCell, self).__init__()
        self.dg = dg
        self.linear = torch.nn.Linear(4, 4)

    def forward(self, x, h):
        new_h = torch.tanh(self.dg(self.linear(x)) + h)
        return new_h, new_h

my_cell = MyCell(MyDecisionGate())
traced_cell = torch.jit.trace(my_cell, (x, h))
print(traced_cell.code)
```

- 输出结果

```
import __torch__.___torch_mangle_0
import __torch__
import __torch__.torch.nn.modules.linear.___torch_mangle_1
def forward(self,
    input: Tensor,
    h: Tensor) -> Tuple[Tensor, Tensor]:
  _0 = self.linear
  weight = _0.weight
  bias = _0.bias
  x = torch.addmm(bias, input, torch.t(weight), beta=1, alpha=1)
  _1 = torch.tanh(torch.add(x, h, alpha=1))
  return (_1, _1)
```

查看 `.code` 输出，我们可以发现在哪里找不到 `if-else` 分支！ 为什么？ 跟踪完全按照我们所说的去做：运行代码，记录发生的操作，并构 造一个可以做到这一点的 `ScriptModule` 。不幸的是，诸如控制流之类的东西被抹去了。

我们如何在TorchScript中忠实地表示此模块？我们提供了一个*脚本编译器*，它可以直接分析您的Python源代码以将其转换为TorchScript。 让我们使用脚本编译器转换 `MyDecisionGate` ：

```
scripted_gate = torch.jit.script(MyDecisionGate())

my_cell = MyCell(scripted_gate)
traced_cell = torch.jit.script(my_cell)
print(traced_cell.code)
```

- 输出结果

```
import __torch__.___torch_mangle_3
import __torch__.___torch_mangle_2
import __torch__.torch.nn.modules.linear.___torch_mangle_4
def forward(self,
    x: Tensor,
    h: Tensor) -> Tuple[Tensor, Tensor]:
  _0 = self.linear
  _1 = _0.weight
  _2 = _0.bias
  if torch.eq(torch.dim(x), 2):
    _3 = torch.__isnot__(_2, None)
  else:
    _3 = False
  if _3:
    bias = ops.prim.unchecked_unwrap_optional(_2)
    ret = torch.addmm(bias, x, torch.t(_1), beta=1, alpha=1)
  else:
    output = torch.matmul(x, torch.t(_1))
    if torch.__isnot__(_2, None):
      bias0 = ops.prim.unchecked_unwrap_optional(_2)
      output0 = torch.add_(output, bias0, alpha=1)
    else:
      output0 = output
    ret = output0
  _4 = torch.gt(torch.sum(ret, dtype=None), 0)
  if bool(_4):
    _5 = ret
  else:
    _5 = torch.neg(ret)
  new_h = torch.tanh(torch.add(_5, h, alpha=1))
  return (new_h, new_h)
```

现在，我们已经忠实地捕获了我们在TorchScript中程序的行为。 现在，让我们尝试运行该程序：

```
# New inputs
x, h = torch.rand(3, 4), torch.rand(3, 4)
traced_cell(x, h)
```

3.1 混合脚本(Scripting)和跟踪(Tracing)

在某些情况下，需要使用跟踪而不是脚本（例如，模块具有许多架构决策，这些决策是基于我们希望不会出现在TorchScript中的恒定Python 值做出的）。在这种情况下，可以通过跟踪来编写脚本：`torch.jit.script` 将内联被跟踪模块的代码，而跟踪将内联脚本模块的代码。

- 第一种情况的示例：

```
class MyRNNLoop(torch.nn.Module):
    def __init__(self):
        super(MyRNNLoop, self).__init__()
        self.cell = torch.jit.trace(MyCell(scripted_gate), (x, h))

    def forward(self, xs):
        h, y = torch.zeros(3, 4), torch.zeros(3, 4)
        for i in range(xs.size(0)):
            y, h = self.cell(xs[i], h)
        return y, h

rnn_loop = torch.jit.script(MyRNNLoop())
print(rnn_loop.code)
```

- 输出结果

```
import __torch__
import __torch__.___torch_mangle_5
import __torch__.___torch_mangle_2
import __torch__.torch.nn.modules.linear.___torch_mangle_6
def forward(self,
    xs: Tensor) -> Tuple[Tensor, Tensor]:
  h = torch.zeros([3, 4], dtype=None, layout=None, device=None,
pin_memory=None)
  y = torch.zeros([3, 4], dtype=None, layout=None, device=None,
pin_memory=None)
  y0 = y
  h0 = h
  for i in range(torch.size(xs, 0)):
    _0 = self.cell
    _1 = torch.select(xs, 0, i)
    _2 = _0.linear
    weight = _2.weight
    bias = _2.bias
    _3 = torch.addmm(bias, _1, torch.t(weight), beta=1, alpha=1)
    _4 = torch.gt(torch.sum(_3, dtype=None), 0)
    if bool(_4):
```

```
      _5 = _3
    else:
      _5 = torch.neg(_3)
    _6 = torch.tanh(torch.add(_5, h0, alpha=1))
    y0, h0 = _6, _6
  return (y0, h0)
```

- 第二种情况的示例：

```
class WrapRNN(torch.nn.Module):
    def __init__(self):
        super(WrapRNN, self).__init__()
        self.loop = torch.jit.script(MyRNNLoop())

    def forward(self, xs):
        y, h = self.loop(xs)
        return torch.relu(y)

traced = torch.jit.trace(WrapRNN(), (torch.rand(10, 3, 4)))
print(traced.code)
```

- 输出结果

```
import __torch__
import __torch__.___torch_mangle_9
import __torch__.___torch_mangle_7
import __torch__.___torch_mangle_2
import __torch__.torch.nn.modules.linear.___torch_mangle_8
def forward(self,
    argument_1: Tensor) -> Tensor:
  _0 = self.loop
  h = torch.zeros([3, 4], dtype=None, layout=None, device=None,
pin_memory=None)
  h0 = h
  for i in range(torch.size(argument_1, 0)):
    _1 = _0.cell
    _2 = torch.select(argument_1, 0, i)
    _3 = _1.linear
    weight = _3.weight
    bias = _3.bias
    _4 = torch.addmm(bias, _2, torch.t(weight), beta=1, alpha=1)
    _5 = torch.gt(torch.sum(_4, dtype=None), 0)
    if bool(_5):
      _6 = _4
    else:
      _6 = torch.neg(_4)
```

```
    h0 = torch.tanh(torch.add(_6, h0, alpha=1))
  return torch.relu(h0)
```

这样，当情况需要它们时，可以使用脚本和跟踪并将它们一起使用。

4.保存和加载模型

我们提供API，以存档格式将TorchScript模块保存到磁盘或从磁盘加载TorchScript模块。这种格式包括代码，参数，属性和调试信息，这意 味着归档文件是模型的独立表示形式，可以在完全独立的过程中加载。让我们保存并加载包装好的 `RNN` 模块：

```
traced.save('wrapped_rnn.zip')

loaded = torch.jit.load('wrapped_rnn.zip')

print(loaded)
print(loaded.code)
```

- 输出结果

```
ScriptModule(
  original_name=WrapRNN
  (loop): ScriptModule(
    original_name=MyRNNLoop
    (cell): ScriptModule(
      original_name=MyCell
      (dg): ScriptModule(original_name=MyDecisionGate)
      (linear): ScriptModule(original_name=Linear)
    )
  )
)
import __torch__
import __torch__.___torch_mangle_9
import __torch__.___torch_mangle_7
import __torch__.___torch_mangle_2
import __torch__.torch.nn.modules.linear.___torch_mangle_8
def forward(self,
    argument_1: Tensor) -> Tensor:
  _0 = self.loop
  h = torch.zeros([3, 4], dtype=None, layout=None, device=None,
pin_memory=None)
  h0 = h
  for i in range(torch.size(argument_1, 0)):
```

```
    _1 = _0.cell
    _2 = torch.select(argument_1, 0, i)
    _3 = _1.linear
    weight = _3.weight
    bias = _3.bias
    _4 = torch.addmm(bias, _2, torch.t(weight), beta=1, alpha=1)
    _5 = torch.gt(torch.sum(_4, dtype=None), 0)
    if bool(_5):
      _6 = _4
    else:
      _6 = torch.neg(_4)
    h0 = torch.tanh(torch.add(_6, h0, alpha=1))
  return torch.relu(h0)
```

如您所见，序列化保留了模块层次结构和我们一直在研究的代码。例如，也可以将模型加载到C++中以实现不依赖Python的执行。

进一步阅读

我们已经完成了教程！有关更多涉及的演示，请查看NeurIPS演示，以使用TorchScript转换机器翻译模型： https://colab.research.google.com/drive/1HiICg6jRkBnr5hvK2-VnMi88Vi9pUzEJ

脚本的总运行时间：（0分钟0.247秒）

在C++中加载TorchScript模型

本教程已更新为可与PyTorch 1.2一起使用

顾名思义，PyTorch的主要接口是Python编程语言。尽管Python是合适于许多需要动态性和易于迭代的场景，并且是首选的语言，但同样的，在 许多情况下，Python的这些属性恰恰是不利的。后者通常适用的一种环境是要求生产-低延迟和严格部署。对于生产场景，即使只将C ++绑定到Java， Rust或Go之类的另一种语言中，它也是经常选择的语言。以下各段将概述PyTorch提供的从现有Python模型到可以完全从C ++加载和执行的序 列化表示形式的路径，而无需依赖Python。

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 Script启动，Torch Script是PyTorch模型的一种表示形式，可以由Torch Script编译器理解， 编译和序列化。如果您是从使用vanilla“eager” 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。在最常见的情况 下（如下所述），这只需要花费很少的功夫。如果您已经有了Torch脚本模块，则可以跳到本教程的下一部分。

有两种将PyTorch模型转换为Torch脚本的方法。第一种称为跟踪，一种机制，其中通过使用示例输入对模型的结构进行一次评估，并记录这些 输入在模型中的流量，从而捕获模型的结构。这适用于有限使用控制流的模型。第二种方法是在模型中添加显式批注，以告知Torch Script编 译器可以根据Torch Script语言施加的约束直接解析和编译模型代码。

> 提示：您可以在官方Torch脚本参考中找到有关这两种方法的完整文档，以及使用方法的进一步指导。

方法1：通过跟踪转换为Torch脚本

要将PyTorch模型通过跟踪转换为Torch脚本，必须将模型的实例以及示例输入传递给 `torch.jit.trace` 函数。这将产生一个 `torch.jit.ScriptModule` 对象，该对象的模型评估痕迹将嵌入模块的 `forward` 方法中：

```
import torch
import torchvision
```

```
# 你模型的一个实例.
model = torchvision.models.resnet18()

# 您通常会提供给模型的forward()方法的示例输入。
example = torch.rand(1, 3, 224, 224)

# 使用`torch.jit.trace `来通过跟踪生成`torch.jit.ScriptModule`
traced_script_module = torch.jit.trace(model, example)
```

现在可以对跟踪的 `ScriptModule` 进行评估，使其与常规PyTorch模块相同：

```
In[1]: output = traced_script_module(torch.ones(1, 3, 224, 224))
In[2]: output[0, :5]
Out[2]: tensor([-0.2698, -0.0381,  0.4023, -0.3010, -0.0448],
grad_fn=<SliceBackward>)
```

方法2：通过注释转换为Torch脚本

在某些情况下，例如，如果模型采用特定形式的控制流，则可能需要直接在Torch脚本中编写模型并相应地注释模型。例如，假设您具有以下 vanilla Pytorch模型：

```
import torch

class MyModule(torch.nn.Module):
    def __init__(self, N, M):
        super(MyModule, self).__init__()
        self.weight = torch.nn.Parameter(torch.rand(N, M))

    def forward(self, input):
        if input.sum() > 0:
          output = self.weight.mv(input)
        else:
          output = self.weight + input
        return output
```

因为此模块的前向方法使用取决于输入的控制流，所以它不适合跟踪。相反，我们可以将其转换为 `ScriptModule` 。为了将模块转换为 `ScriptModule` ，需要使用 `torch.jit.script` 编译模块，如下所示：

```
class MyModule(torch.nn.Module):
    def __init__(self, N, M):
        super(MyModule, self).__init__()
```

```
        self.weight = torch.nn.Parameter(torch.rand(N, M))

    def forward(self, input):
        if input.sum() > 0:
          output = self.weight.mv(input)
        else:
          output = self.weight + input
        return output

my_module = MyModule(10,20)
sm = torch.jit.script(my_module)
```

如果您需要在 `nn.Module` 中排除某些方法，因为它们使用了 `TorchScript` 尚不支持的Python功能，则可以使用 `@torch.jit.ignore` 对其进行注释

`my_module` 是 `ScriptModule` 的实例，可以序列化。

步骤2：将脚本模块序列化为文件

一旦有了ScriptModule（通过跟踪或注释PyTorch模型），您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。假设我们要序列化先前在跟踪示例中显示的 `ResNet18` 模型。要执行此序列化，只需在模块上调用save并传递一个文件名即可：

```
traced_script_module.save("traced_resnet_model.pt")
```

这将在您的工作目录中生成 `traced_resnet_model.pt` 文件。如果您还想序列化 `my_module` ，请调用 `my_module.save("my_module_model.pt")` 我们现在已经正式离开Python领域，并准备跨入C ++领域。

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，并且将来会得到很好的支持。 对于本教程，我们将使用CMake和LibTorch构建一个最小的C ++应用程序，该应用程序简单地加载并执行序列化的PyTorch模型。

最小的C ++应用程序

让我们从讨论加载模块的代码开始。以下将已经做：

```
include <torch/script.h> // One-stop header.

#include <iostream>
#include <memory>

int main(int argc, const char* argv[]) {
  if (argc != 2) {
    std::cerr << "usage: example-app <path-to-exported-script-module>\n";
    return -1;
  }


  torch::jit::script::Module module;
  try {
    // 使用以下命令从文件中反序列化脚本模块: torch::jit::load().
    module = torch::jit::load(argv[1]);
  }
  catch (const c10::Error& e) {
    std::cerr << "error loading the model\n";
    return -1;
  }

  std::cout << "ok\n";
}
```

`<torch/script.h>` 标头包含运行示例所需的LibTorch库中的所有相关包含。我们的应用程序接受序列化的PyTorch ScriptModule的文件路径 作为其唯一的命令行参数，然后使用 `torch::jit::load ()` 函数继续对该模块进行反序列化，该函数将此文件路径作为输入。作为返回，我们 收到一个 `Torch::jit::script::Module` 对象。我们将稍后讨论如何执行它。

取决于LibTorch和构建应用程序

假设我们将以上代码存储在名为 `example-app.cpp` 的文件中。最小的 `CMakeLists.txt` 可能看起来很简单：

```
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(custom_ops)
```

```
find_package(Torch REQUIRED)

add_executable(example-app example-app.cpp)
target_link_libraries(example-app "${TORCH_LIBRARIES}")
set_property(TARGET example-app PROPERTY CXX_STANDARD 11)
```

建立示例应用程序的最后一件事是LibTorch发行版。您可以随时从PyTorch网站的下载页面上获取最新的稳定版本。 如果下载并解压缩最新的归档文件，则应收到具有以下目录结构的文件夹：

```
libtorch/
  bin/
  include/
  lib/
  share/
```

- lib/ 文件夹包含您必须链接的共享库，
- include/ 文件夹包含程序需要包含的头文件，
- share/ 文件夹包含必要的CMake配置，以启用上面的简单 `find_package(Torch)` 命令。

> 提示;在Windows上，调试和发行版本不兼容ABI。 如果您打算以调试模式构建项目，请尝试使用LibTorch的调试版本。

最后一步是构建应用程序。为此，假定示例目录的布局如下：

```
example-app/
  CMakeLists.txt
  example-app.cpp
```

现在，我们可以运行以下命令从 `example-app/` 文件夹中构建应用程序：

```
mkdir build
cd build
cmake -DCMAKE_PREFIX_PATH=/path/to/libtorch ..
make
```

`/path/to/libtorch` 应该是解压缩的LibTorch发行版的完整路径。如果一切顺利，它将看起来像这样：

```
root@4b5a67132e81:/example-app# mkdir build
root@4b5a67132e81:/example-app# cd build
root@4b5a67132e81:/example-app/build# cmake -DCMAKE_PREFIX_PATH=/path/to/
```

```
libtorch ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /example-app/build
root@4b5a67132e81:/example-app/build# make
Scanning dependencies of target example-app
[ 50%] Building CXX object CMakeFiles/example-app.dir/example-app.cpp.o
[100%] Linking CXX executable example-app
[100%] Built target example-app
```

如果我们提供了我们之前创建的到示例应用程序二进制文件的跟踪ResNet18模型 `traced_resnet_model.pt` 的路径，则应该以友好的“ ok” 作为奖励。 请注意，如果尝试使用 `my_module_model.pt` 运行此示例，则会收到一条错误消息，提示您输入的形状不兼容。 `my_module_model.pt` 需要1D而不是4D。

```
root@4b5a67132e81:/example-app/build# ./example-app <path_to_model>/
traced_resnet_model.pt
ok
```

步骤4：在C ++中执行脚本模块

成功用C ++加载了序列化的ResNet18之后，我们现在只需执行几行代码即可！让我们将这些行添加到C ++应用程序的 `main()` 函数中：

```
// 创建输入向量
std::vector<torch::jit::IValue> inputs;
inputs.push_back(torch::ones({1, 3, 224, 224}));

// 执行模型并将输出转化为张量
at::Tensor output = module.forward(inputs).toTensor();
std::cout << output.slice(/*dim=*/1, /*start=*/0, /*end=*/5) << '\n';
```

前两行设置了我们模型的输入。我们创建一个 `torch::jit::IValue` 的向量（类型为type-erased的值 `Script::Module` 方法接受并返回），并添加单个输入。要创建输入张量，我们使用 `torch::ones()` ，等效于C ++ API中的 `torch.ones` 。然后，我们运行 `script::Module` 的 `forward` 方法，并向其传递我们创建的输入向量。作为回报，我们得到一个新的IValue，通过调用 `toTensor()` 将其转换为张量。

> 提示：要总体上了解有关torch::ones和PyTorch C ++ API之类的功能的更多信息，请参阅其文档，网址为https://pytorch.org/cppdocs。 PyTorch C ++ API提供了与Python API几乎相同的功能奇偶校验，使您可以像在Python中一样进一步操纵和处理张量。

在最后一行中，我们打印输出的前五个条目。由于在本教程前面的部分中，我们向Python中的模型提供了相同的输入，因此理想情况下，我们 应该看到相同的输出。让我们通过重新编译我们的应用程序并以相同的序列化模型运行它来进行尝试：

```
root@4b5a67132e81:/example-app/build# make
Scanning dependencies of target example-app
[ 50%] Building CXX object CMakeFiles/example-app.dir/example-app.cpp.o
[100%] Linking CXX executable example-app
[100%] Built target example-app
root@4b5a67132e81:/example-app/build# ./example-app traced_resnet_model.pt
-0.2698 -0.0381  0.4023 -0.3010 -0.0448
[ Variable[CPUFloatType]{1,5} ]
```

作为参考，Python以前的输出为：

```
tensor([-0.2698, -0.0381,  0.4023, -0.3010, -0.0448], grad_fn=<SliceBackward>)
```

看来匹配得很好！

> 提示：要将模型移至GPU内存，可以编写model.to（at::kCUDA）;。通过调用tensor.to（at::kCUDA），确保模型的输入也位于CUDA内存中，这将在CUDA内存中返回新的张量。

步骤5：获取帮助并探索API

本教程有望使您对PyTorch模型从Python到C ++的路径有一个大致的了解。使用本教程中描述的概念，您应该能够从vanilla, “eager” PyTorch模型，到Python中的已编译`ScriptModule`，再到磁盘上的序列化文件，以及–结束循环–到可执行脚本：C ++中的模块。

当然，有许多我们没有介绍的概念。例如，您可能会发现自己想要使用以C ++或CUDA实现的自定义运算符扩展`ScriptModule`，并在加载到纯 C ++生产环境中的ScriptModule中执行此自定义运算符。好消息是：这是可能的，并且得到了很好的支持！现在，您可以浏览此文件夹中的示例，我们将很快提供一个教程。目前，以下链接通常可能会有所帮助：

- Torch Script参考：https://pytorch.org/docs/master/jit.html
- PyTorch C ++ API文档：https://pytorch.org/cppdocs/
- PyTorch Python API文档：https://pytorch.org/docs/

与往常一样，如果您遇到任何问题或疑问，可以使用我们的论坛或GitHub issues进行联系。

小分队

翻译作者

磐创 News and PytorchChina

原文地址

https://pytorch.org/