知乎 首发于 Lawbda半生记

# PyTorch深度学习：60分钟入门(Translation)

胡莱人形
直脑筋·缺心眼·智障·博士

关注他

红色石头、mpcv 等 860 人赞同了该文章

某天在微博上看到@爱可可-爱生活 老师推了Pytorch的入门教程，就顺手下来翻了。虽然完工的比较早但是手头菜的没有linux服务器没法子运行结果。开学以来终于在师兄的机器装上了Torch，中间的运行结果也看明白了。所以现在发一下这篇两周之前做的教程翻译。

首先惯例上原文链接，特别的原作者是以ipyhton notebook来写的教程，运行相当的方便。但带来的问题就是翻译作为专栏文章的效果实在是太差。

原文档链接在此：Deep Learning with PyTorch.ipynb

**特别注明：原教程是以ipython notebook写就，因此代码部分非常零散，我在翻译的过程中将部分代码进行了整合以保证文章的紧凑，翻译的目的是将教程说明部分表达完整，具体的代码运行步骤请移步Github下载源文件进行代码的运行。**

## 以下是教程的主体部分：

本教程的目的:

- 更高层级地理解PyTorch的Tensor库以及神经网络。
- 训练一个小的神经网络来对图像进行分类。

*本教程以您拥有一定的numpy基础的前提下展开*

## PyTorch是什么?

这是一个基于Python的科学计算包，其旨在服务两类场合：

- 替代numpy发挥GPU潜能
- 一个提供了高度灵活性和效率的深度学习实验性平台

## 我们开搞

### Tensors

Tensors和numpy中的ndarrays较为相似, 与此同时Tensor也能够使用GPU来加速运算。

```
from __future__ import print_function
import torch
x = torch.Tensor(5, 3)  # 构造一个未初始化的5*3的矩阵
x = torch.rand(5, 3)  # 构造一个随机初始化的矩阵
x # 此处在notebook中输出x的值来查看具体的x内容
x.size()

#NOTE: torch.Size 事实上是一个tuple, 所以其支持相关的操作*
y = torch.rand(5, 3)

#此处 将两个同形矩阵相加有两种语法结构
x + y # 语法一
torch.add(x, y) # 语法二

# 另外输出tensor也有两种写法
result = torch.Tensor(5, 3) # 语法一
torch.add(x, y, out=result) # 语法二
y.add_(x) # 将y与x相加

# 特别注明：任何可以改变tensor内容的操作都会在方法名后加一个下划线'_'
# 例如：x.copy_(y), x.t_(), 这俩都会改变x的值。

#另外python中的切片操作也是资次的。
x[:,1] #这一操作会输出x矩阵的第二列的所有值
```

100+ Tensor的操作，包括换位、索引、切片、数学运算、线性算法和随机数等等。

详见：torch - PyTorch 0.1.9 documentation

**Numpy桥**

将Torch的Tensor和numpy的array相互转换简直就是洒洒水啦。注意Torch的Tensor和numpy的array会共享他们的存储空间，修改一个会导致另外的一个也被修改。

```
# 此处演示tensor和numpy数据结构的相互转换
a = torch.ones(5)
b = a.numpy()

# 此处演示当修改numpy数组之后,与之相关联的tensor也会相应的被修改
a.add_(1)
print(a)
print(b)

# 将numpy的Array转换为torch的Tensor
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)

# 另外除了CharTensor之外，所有的tensor都可以在CPU运算和GPU预算之间相互转换
# 使用CUDA函数来将Tensor移动到GPU上
# 当CUDA可用时会进行GPU的运算
if torch.cuda.is_available():
    x = x.cuda()
    y = y.cuda()
    x + y
```

## PyTorch中的神经网络

接下来介绍pytorch中的神经网络部分。PyTorch中所有的神经网络都来自于autograd包

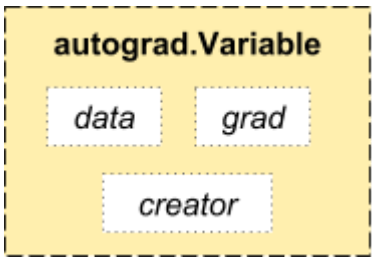
## Autograd: 自动求导

autograd 包提供Tensor所有操作的自动求导方法。
这是一个运行时定义的框架，这意味着你的反向传播是根据你代码运行的方式来定义的，因此每一轮迭代都可以各不相同。

以这些例子来讲，让我们用更简单的术语来看看这些特性。

autograd.Variable 这是这个包中最核心的类。 它包装了一个Tensor，并且几乎支持所有的定义在其上的操作。一旦完成了你的运算，你可以调用 .backward()来自动计算出所有的梯度。

你可以通过属性 .data 来访问原始的tensor，而关于这一Variable的梯度则集中于 .grad 属性中。




还有一个在自动求导中非常重要的类 Function。

Variable 和 Function 二者相互联系并且构建了一个描述整个运算过程的无环图。每个Variable拥有一个 .creator 属性，其引用了一个创建Variable的 Function。(除了用户创建的Variable其 creator 部分是 None)。

如果你想要进行求导计算，你可以在Variable上调用.backward()。 如果Variable是一个标量（例如它包含一个单元素数据），你无需对backward()指定任何参数，然而如果它有更多的元素，你需要指定一个和tensor的形状想匹配的grad_output参数。

```
from torch.autograd import Variable
x = Variable(torch.ones(2, 2), requires_grad = True)
y = x + 2
y.creator

# y 是作为一个操作的结果创建的因此y有一个creator
z = y * y * 3
out = z.mean()
```

```
# out.backward()和操作out.backward(torch.Tensor([1.0]))是等价的
# 在此处输出 d(out)/dx
x.grad
```

最终得出的结果应该是一个全是4.5的矩阵。设置输出的变量为o。我们通过这一公式来计算：

$o=\frac{1}{4}\sum_i z_i$，$z_i=3(x_i+2)^2$，$z_i\big|_{x_i=1}=27$，因此，$\frac{\partial o}{\partial x_i}=\frac{3}{2}(x_i+2)$，最后有 $\frac{\partial o}{\partial x_i}\big|_{x_i=1}=\frac{9}{2}=4.5$

> 你可以使用自动求导来做许多疯狂的事情。

```
x = torch.randn(3)
x = Variable(x, requires_grad = True)
y = x * 2
while y.data.norm() < 1000:
    y = y * 2
gradients = torch.FloatTensor([0.1, 1.0, 0.0001])
y.backward(gradients)
x.grad
```

阅读材料：

你可以在这读更多关于Variable 和 Function的文档: pytorch.org/docs/autograd.html

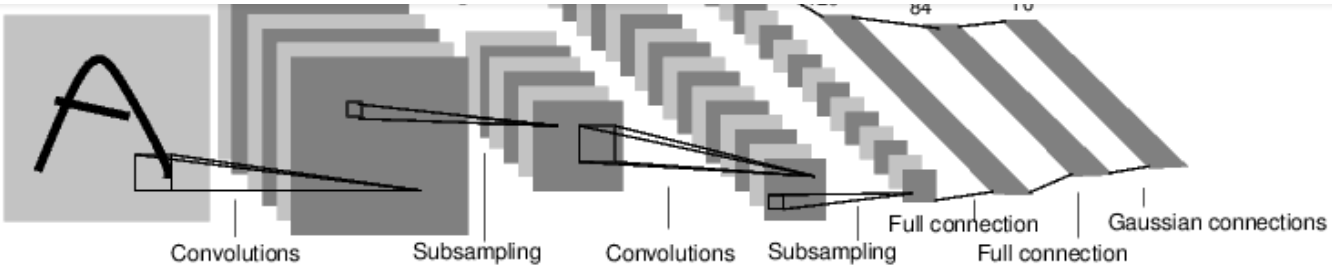
## 神经网络

使用 torch.nn 包可以进行神经网络的构建。

现在你对autograd有了初步的了解，而nn建立在autograd的基础上来进行模型的定义和微分。

nn.Module中包含着神经网络的层，同时forward(input)方法能够将output进行返回。

举个例子，来看一下这个数字图像分类的神经网络。




这是一个简单的前馈神经网络。 从前面获取到输入的结果，从一层传递到另一层，最后输出最后结果。

一个典型的神经网络的训练过程是这样的：

- 定义一个有着可学习的参数（或者权重）的神经网络
- 对着一个输入的数据集进行迭代:
  - 用神经网络对输入进行处理
  - 计算代价值 (对输出值的修正到底有多少)
  - 将梯度传播回神经网络的参数中
  - 更新网络中的权重
    - 通常使用简单的更新规则: weight = weight + learning_rate * gradient

让我们来定义一个神经网络:

```
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(1, 6, 5) # 1 input image channel, 6 output chan
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1   = nn.Linear(16*5*5, 120) # an affine operation: y = Wx + b
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2)) # Max pooling over a
```

```
        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()
net

'''神经网络的输出结果是这样的
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 (400 -> 120)
  (fc2): Linear (120 -> 84)
  (fc3): Linear (84 -> 10)
)
'''
```

仅仅需要定义一个forward函数就可以了，backward会自动地生成。

你可以在forward函数中使用所有的Tensor中的操作。

模型中可学习的参数会由net.parameters()返回。

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

input = Variable(torch.randn(1, 1, 32, 32))
out = net(input)
'''out 的输出结果如下
```

```
[torch.FloatTensor of size 1x10]
'''

net.zero_grad() # 对所有的参数的梯度缓冲区进行归零
out.backward(torch.randn(1, 10)) # 使用随机的梯度进行反向传播
```

> **注意: torch.nn 只接受小批量的数据**
> 整个torch.nn包只接受那种小批量样本的数据，而非单个样本。 例如，nn.Conv2d能够结构一个四维的TensornSamples x nChannels x Height x Width。
> *如果你拿的是单个样本，使用input.unsqueeze(0)来加一个假维度就可以了。*

**复习一下前面我们学到的：**

- torch.Tensor - 一个多维数组
- autograd.Variable - 改变Tensor并且记录下来操作的历史记录。和Tensor拥有相同的API，以及backward()的一些API。同时包含着和张量相关的梯度。
- nn.Module - 神经网络模块。便捷的数据封装，能够将运算移往GPU，还包括一些输入输出的东西。
- nn.Parameter - 一种变量，当将任何值赋予Module时自动注册为一个参数。
- autograd.Function - 实现了使用自动求导方法的前馈和后馈的定义。每个Variable的操作都会生成至少一个独立的Function节点，与生成了Variable的函数相连之后记录下操作历史。

到现在我们已经明白的部分:

- 定义了一个神经网络。
- 处理了输入以及实现了反馈。

仍然没整的:

- 计算代价。
- 更新网络中的权重。

一个代价函数接受（输出，目标）对儿的输入，并计算估计出输出与目标之间的差距。

nn package包中一些不同的代价函数.

一个简单的代价函数：nn.MSELoss计算输入和目标之间的均方误差。

```
output = net(input)
target = Variable(torch.range(1, 10))  # a dummy target, for example
criterion = nn.MSELoss()
loss = criterion(output, target)
'''loss的值如下
Variable containing:
 38.5849
[torch.FloatTensor of size 1]
'''
```

现在，如果你跟随loss从后往前看，使用.creator属性你可以看到这样的一个计算流程图：

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

因此当我们调用loss.backward()时整个图通过代价来进行区分，图中所有的变量都会以.grad来累积梯度。

```
# For illustration, let us follow a few steps backward
print(loss.creator) # MSELoss
print(loss.creator.previous_functions[0][0]) # Linear
print(loss.creator.previous_functions[0][0].previous_functions[0][0]) # ReLU

'''
<torch.nn._functions.thnn.auto.MSELoss object at 0x7fe8102dd7c8>
<torch.nn._functions.linear.Linear object at 0x7fe8102dd708>
<torch.nn._functions.thnn.auto.Threshold object at 0x7fe8102dd648>
'''

# 现在我们应当调用loss.backward(), 之后来看看 conv1's在进行反馈之后的偏置梯度如何
net.zero_grad() # 归零操作
print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)
loss.backward()
print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

```
conv1.bias.grad before backward
Variable containing:
 0
 0
 0
 0
 0
 0
[torch.FloatTensor of size 6]

conv1.bias.grad after backward
Variable containing:
 0.0346
-0.0141
 0.0544
-0.1224
-0.1677
 0.0908
[torch.FloatTensor of size 6]
'''
```

现在我们已经了解如何使用代价函数了。

阅读材料：

> 神经网络包中包含着诸多用于神经网络的模块和代价函数，带有文档的完整清单在这里：
> torch.nn - PyTorch 0.1.9 documentation

**只剩下一个没学了：**

- 更新网络的权重

最简单的更新的规则是随机梯度下降法(SGD):

> weight = weight - learning_rate * gradient

我们可以用简单的python来表示:

```
    f.data.sub_(f.grad.data * learning_rate)
```

然而在你使用神经网络的时候你想要使用不同种类的方法诸如：SGD, Nesterov-SGD, Adam, RMSProp, etc.

我们构建了一个小的包torch.optim来实现这个功能，其中包含着所有的这些方法。 用起来也非常简单:

```
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
```

就是这样。

但你现在也许会想。

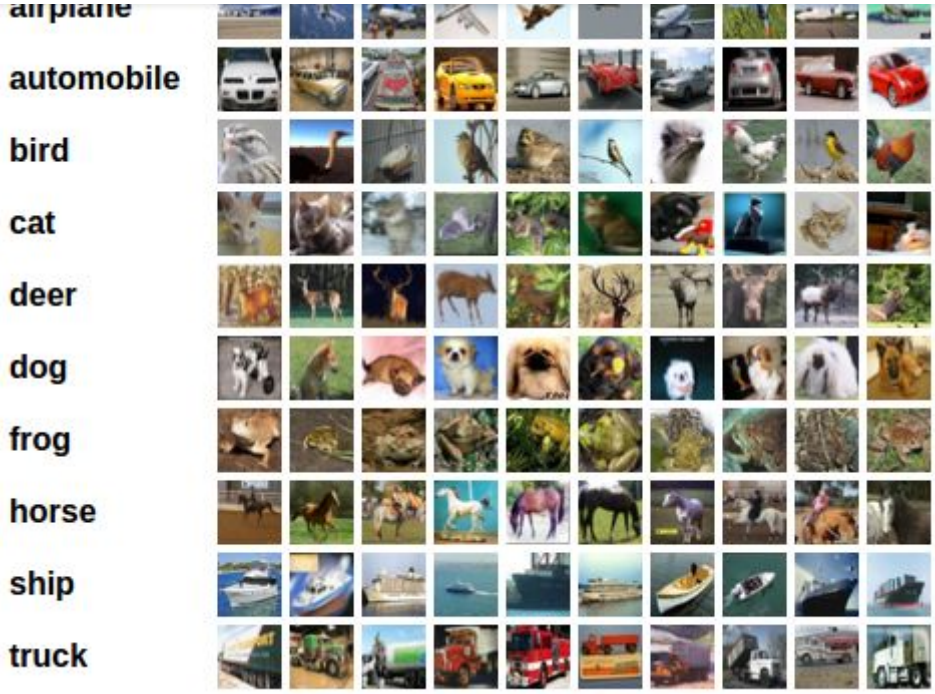
**那么数据怎么办呢?**

通常来讲，当你处理图像，声音，文本，视频时需要使用python中其他独立的包来将他们转换为numpy中的数组，之后再转换为torch.*Tensor。

- 图像的话，可以用Pillow, OpenCV。
- 声音处理可以用scipy和librosa。
- 文本的处理使用原生Python或者Cython以及NLTK和SpaCy都可以。

特别的对于图像，我们有torchvision这个包可用,其中包含了一些现成的数据集如：Imagenet, CIFAR10, MNIST等等。同时还有一些转换图像用的工具。 这非常的方便并且避免了写样板代码。

本教程使用CIFAR10数据集。 我们要进行的分类的类别有：'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'。 这个数据集中的图像都是3通道，32x32像素的图片




下面是对torch神经网络使用的一个实战练习。

## 训练一个图片分类器

我们要按顺序做这几个步骤：

1. 使用torchvision来读取并预处理CIFAR10数据集
2. 定义一个卷积神经网络
3. 定义一个代价函数
4. 在神经网络中训练训练集数据
5. 使用测试集数据测试神经网络

## 1. 读取并预处理CIFAR10

使用torchvision读取CIFAR10相当的方便。

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

```
transform=transforms.Compose([transforms.ToTensor(),
                              transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
                             ])
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=Tru
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=Tru
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')
'''注：这一部分需要下载部分数据集 因此速度可能会有一些慢 同时你会看到这样的输出

Downloading http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/c
Extracting tar file
Done!
Files already downloaded and verified
'''
```

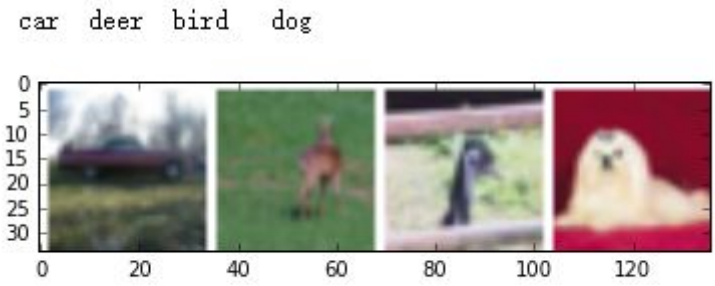
我们来从中找几张图片看看。

```
# functions to show an image
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
def imshow(img):
    img = img / 2 + 0.5 # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1,2,0)))

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

# print images
imshow(torchvision.utils.make_grid(images))
```

结果是这样的：




## 2. 定义一个卷积神经网络

```
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()
```

## 3. 定义代价函数和优化器

## 4. 训练网络

事情变得有趣起来了。 我们只需一轮一轮迭代然后不断通过输入来进行参数调整就行了。

```
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

        # wrap them in Variable
        inputs, labels = Variable(inputs), Variable(labels)

        # 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.data[0]
        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')
'''这部分的输出结果为
[1,  2000] loss: 2.212
[1,  4000] loss: 1.892
[1,  6000] loss: 1.681
[1,  8000] loss: 1.590
[1, 10000] loss: 1.515
[1, 12000] loss: 1.475
[2,  2000] loss: 1.409
```

```
[2,  8000] loss: 1.334
[2, 10000] loss: 1.313
[2, 12000] loss: 1.264
Finished Training
'''
```

我们已经训练了两遍了。 此时需要测试一下到底结果如何。

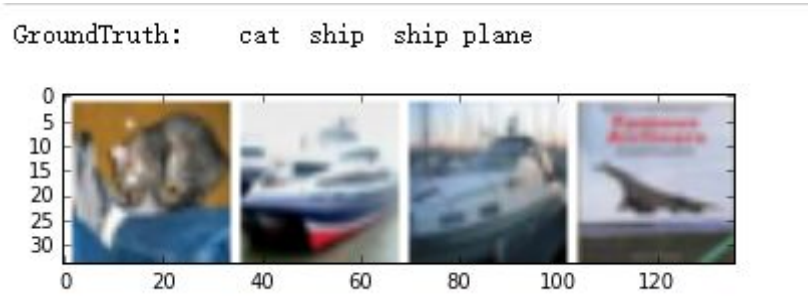
通过对比神经网络给出的分类和已知的类别结果，可以得出正确与否，如果预测的正确，我们可以将样本加入正确预测的结果的列表中。

好的第一步，让我们展示几张照片来熟悉一下。

```
dataiter = iter(testloader)
images, labels = dataiter.next()

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

结果是这样的：




好的，接下来看看神经网络如何看待这几个照片。

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

# the outputs are energies for the 10 classes.
# Higher the energy for a class, the more the network
# thinks that the image is of the particular class
```

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

print('Predicted: ', ' '.join('%5s'% classes[predicted[j][0]] for j in range(4)

'''输出结果为
Predicted:    cat plane   car plane
'''
```

结果看起来挺好。

看看神经网络在整个数据集上的表现结果如何。

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

print('Accuracy of the network on the 10000 test images: %d %%' % (100 * correc

'''输出结果为
Accuracy of the network on the 10000 test images: 54 %
'''
```

看上去这玩意输出的结果比随机整的要好，随机选择的话从十个中选择一个出来，准确率大概只有10%。

看上去神经网络学到了点东西。

嗯。。。那么到底哪些类别表现良好又是哪些类别不太行呢?

```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
for data in testloader:
```

```
        _, predicted = torch.max(outputs.data, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i]
            class_total[label] += 1

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

'''输出结果为
Accuracy of plane : 73 %
Accuracy of   car : 70 %
Accuracy of  bird : 52 %
Accuracy of   cat : 27 %
Accuracy of  deer : 34 %
Accuracy of   dog : 37 %
Accuracy of  frog : 62 %
Accuracy of horse : 72 %
Accuracy of  ship : 64 %
Accuracy of truck : 53 %
'''
```

好吧，接下来该怎么搞了？

我们该如何将神经网络运行在GPU上呢？

## 在GPU上进行训练

就像你把Tensor传递给GPU进行运算一样，你也可以将神经网络传递给GPU。

这一过程将逐级进行操作，直到所有组件全部都传递到GPU上。

```
net.cuda()

'''输出结果为
Net (
```

```
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear (400 -> 120)
  (fc2): Linear (120 -> 84)
  (fc3): Linear (84 -> 10)
)
'''
```

记住，每一步都需要把输入和目标传给GPU。

```
inputs, labels = Variable(inputs.cuda()), Variable(labels.cuda())
```

我为什么没有进行CPU运算和GPU运算的对比呢？因为神经网络实在太小了，其中的差距并不明显。

## 目标达成：

- 在更高层级上理解PyTorch的Tensor库和神经网络。
- 训练一个小的神经网络。

## 接下来我该去哪？

- Train neural nets to play video games
- Train a state-of-the-art ResNet network on imagenet
- Train an face generator using Generative Adversarial Networks
- Train a word-level language model using Recurrent LSTM networks
- More examples
- More tutorials
- Discuss PyTorch on the Forums
- Chat with other users on Slack

Trans by lawbda，edit in 2017.03.05 15:38

最后放上广告：


编辑于 2018-01-22

深度学习（Deep Learning） Torch (深度学习框架)



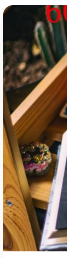
## 文章被以下专栏收录

Lawbda半生记
Thug postgraduate life.

## 推荐阅读


适合PyTorch小白的官网教程：Learning PyTorch With...

刘昕宸 发表于小刘学编程...



60