

PYTORCH

PyTorch (0.3.0) 是基于Python的开源深度学习框架，它包括了支持GPUs计算的Tensor模块以及自动求导等先进的模块，被广泛应用于科学研究中，是最流行的动态图框架。

涉及包

PyTorch (0.3.0)

导入包

import torch

Tensors

Tensors是PyTorch中最重要的高维数据结构，与NumPy的ndarrays数据结构类似，它的数学运算、索引、切片接口与ndarrays也极为相似，不同的是tensors可以使用GPU加速计算。

创建Tensors

x, y | 表示一个torch.FloatTensor对象。

x.size() | 查看x的形状。

torch.Tensor(5, 3) | 创建一个未初始化的5×3矩阵。

torch.rand(5, 3) | 创建一个均匀分布随机初始化的5×3矩阵。

tensors的其他创建方法	说明
torch.Tensor()	基础构造函数
torch.ones()	全 1 tensor
torch.zeros()	全 0 tensor
torch.eye()	对角线为1，其他为0
torch.rand/randn()	[0, 1]均匀/标准分布采样
torch.normal(0, 1)	$\mathcal{N}(0,1)$ 正态分布采样
torch.uniform(0, 2)	[0, 2]区间均匀分布采样

Tensors基本操作和运算

x.view(-1, 3) | 调整x形状并保留元素总数，-1表示自动推测。

x.resize(2, 4) | 修改x形状，可以改变x的元素总数。

x.type(torch.IntTensor) | 将x类型设置为IntTensor。

x.numpy() | 将tensor转换为NumPy的ndarray结构。

torch.from_numpy(arr) | 将ndarray结构的arr转换为tensor。

x.cuda() | 将tensor转换为可使用GPU加速运算的tensor。

torch.mm(x, y) | 矩阵乘法运算。

torch.dot(x, y) | 求内积，注意它与numpy.dot的不同。

torch.add(x, y) | 求和，等价于x + y和y.add(x)。

y.add_(x) | 求和，并使用求和结果替换y。

tensors的逐元素操作方法	说明
torch.abs/torch.sqrt(x)	求绝对值/平方根
torch.exp/fmod/log(x)	求指数/求余/对数
torch.clamp(x, min, max)	截断函数
torch.mul(x, 3)	每个元素乘以2
torch.pow(x, 2)	每个元素求2次幂
torch.sigmoid(x)/tanh(x)	常用激活函数

注：逐元素操作即对每一个元素做同样的操作（element-wise），其输出与输入的形状一致。

Tensors的持久化

torch.save(x, 'x.pth') | 保存tensor在pickle文件中。

torch.load('x.pth') | 加载保存在pickle文件中的tensor。

Tensors中的广播法则实现

广播法则（Broadcast）是科学计算的一个常用技巧，它使得在快速执行向量化的同时不会占用额外的内存/显存。即使PyTorch支持自动广播法则，我们也仍需要通过一些函数手动实现广播法则，使得程序更直观，并且运算不容易出错。

x = torch.ones(2, 3)

y = torch.randn(3, 2, 1)

torch.unsqueeze(x, dim=0) | 等价于x.unsqueeze(0)，在索引为0的位置添加长度为1的维度，广播后x变为1×2×3（与y维度一致）。

x.expand(3, 2, 3); y.expand(3, 2, 3) | 对于长度为1的维度，通过沿此维度复制数据的方式扩展成一个高维结构，最终使得参与计算的x与y的维度及各维度的长度一致（计算前提）。

torch.squeeze(x) | 等价于x.squeeze()，将长度为1的维度删除。

Autograd

from torch.autograd import Variable

Autograd模块可以对在Tensors上的所有操作自动微分，并实现了计算图的相关功能，利于神经网络结构中的反向传播计算。

autograd.Variable

autograd.Variable是Autograd的核心类，它对Tensors进行了封装，被封装后的Tensors可调用.backward实现反向传播，自动计算所有梯度。

autograd.Variable属性

data

grad

grad_fn

✓ data | 保存Variable所包含的tensor。

✓ grad | 保存data对应的梯度，grad也是一个Variable。

✓ grad_fn | 一个Function对象，记录tensors的操作历史，用于构建计算图，并反向传播计算对于输入的梯度。

x = Variable(torch.randn(2, 2), requires_grad=True)

使用tensor创建一个Variable，并且包含梯度grad属性。

x = Variable(torch.randn(2, 2), volatile=True) | 使用tensor创建一个Variable，不求导，无法进行反向传播，在测试推理场景下可以提升运行速度，节省大量内存/显存。

y = x**2; y.grad_fn | 得到由x产生y的Function类。

y.backward() | 由y反向传播并计算梯度。

x.grad | 得到y对x的梯度值，每个值为2x。

x.grad.data.zero_() | 清空历史梯度值。

动态计算图构建

① 定义Variables

x = Variable(torch.randn(3, 4), requires_grad=True)

y = Variable(torch.randn(3, 4), requires_grad=True)

z = Variable(torch.randn(3, 4), requires_grad=True)

② 构建计算图，并前向传播

a = x * y

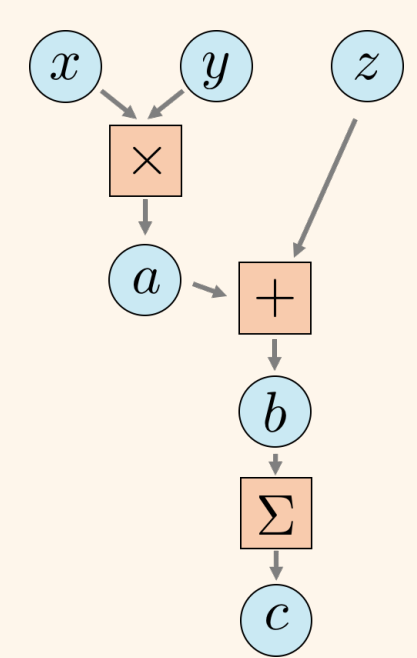
b = a + z

c = torch.sum(b)

③ 后向传播并计算所有梯度

c.backward()

print x.grad, y.grad, z.grad



神经网络模块nn

import torch.nn as nn

import torch.nn.functional as F

torch.nn是PyTorch中专门为神经网络开发的模块，它包含多种神经网络功能层和常用函数，并封装好了可学习参数。

nn.Module

torch.nn的核心数据结构，既可以表示网络中的某层（layer），也可以表示一个包含多层的神经网络，实际使用常继承nn.Module撰写自己的网络/层，定义时只需实现前向传播函数。

class MyModule(nn.Module):

def __init__(self, in, hidden, out):

super(MyModule, self).__init__()

self.w = nn.Parameter(torch.randn(in, out))

self.b = nn.Parameter(torch.randn(out))

def forward(self, x):

x = x.mm(self.w)

return x + self.b.expand_as(x)

对nn.Module中已经实现好的所有神经网络/层，需要关注：

✓ 构造函数的参数，如nn.Linear中的三个参数in_features, out_features和bias。

✓ 属性、可学习参数和子module，如nn.Linear中有weight和bias两个可学习参数，不包括子module。

✓ 输入输出的形状，如nn.Linear的输入形状为(N, input_features)，输出为(N, ouput_features)，N是batch_size，若输入为单个样本，则需要调用unsqueeze(0)将数据伪装成batch_size=1的batch。

神经网络层

nn.Linear(6, 2) | 全连接层，输入为6个特征，输出为2个特征。

nn.Conv2d(1, 1, (3, 3)) | 卷积层，单通道且卷积核为3×3。

nn.ConvTranspose2d(1, 1, (3, 3)) | 逆卷积。

nn.MaxPool2d(3) | 最大值池化层，窗口大小为3×3。

nn.AvgPool2d(3, stride=2) | 平均池化层，窗口大小为3×3，步长为2。

nn.AdaptiveMaxPool2d((5, 7)) | 自适应最大值池化层，目标输出大小为5×7，另外有自适应平均池化层。

nn.BatchNorm2d(100, affine=True) | 批规范化层，特征数量为100，且带有可学习参数， $y = \frac{x - \text{mean}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \text{gamma} + \text{beta}$ 。

nn.InstanceNorm2d(100, affine=True) | 实例规范化层，常用在风格迁移中。

nn.Dropout2d(0.5) | Dropout层，丢弃概率为0.5，与其他功能层不同，一维情况下为nn.Dropout。

nn.RNN(10, 20, 2) | 多层循环神经网络，输入向量为10维，隐层单元为20个，层数为2的Elman循环神经网络。

nn.LSTM(10, 20, 2) | 多层LSTM RNN。

nn.GRU(10, 20, 2) | 多层门控单元RNN。

nn.RNNCell(10, 20) | 单层循环神经网络，可与nn.LSTMCell和nn.GRUCell灵活组合构成复杂的循环神经网络

nn.Embedding(8, 5) | 词向量，8个词且每个词用5维向量表示。

from collections import OrderedDict

nn.Sequential(OrderedDict([

(‘conv1’, nn.Conv2d(3, 3, 3)),

(‘bn1’, nn.BatchNorm2d(3)),

(‘relu1’, nn.ReLU())

]))

按照顺序将多个神经网络层组合起来。

✓ 以上功能层只展示了二维，nn还提供了一维1d和三维3d功能层。

✓ 将神经网络层实例化后，我们需要注意每个输入或者隐层的大小。

✓ 在自定义网络结构时，定义前向传播函数可以结合不带可学习参数的功能函数如最大值池化函数F.max_pool2d、激活函数F.relu等。

激活函数

nn.ReLU()

ReLU(x) = max(0, x).

nn.Tanh()

Tanh(x) = ($\exp(x) - \exp(-x)$)/($\exp(x) + \exp(-x)$).

nn.Sigmoid()

Sigmoid(x) = 1/(1 + $\exp(-x)$).

nn.Softmax()

Softmax(x_i) = $\exp(x_i)/\sum_j \exp(x_j)$.

nn.PReLU()

PReLU(x) = max(0, x) + a * min(0, x).

nn.LeakyReLU(0.1)

LeakyReLU(x) = max(0, x) + 0.1 * min(0, x).

nn.Threshold(0.1, 20)

阈值激活函数，if $x > 0.1$, then $y = x$; if $x \leq 0.1$, then $y = 20$.

损失函数

nn.L1Loss()

L1损失。

nn.MSELoss()

均方误差损失。

nn.NLLLoss()

负对数似然损失。

nn.CrossEntropyLoss()

交叉熵损失，注意它是LogSoftMax和NLLLoss的结合，所以无需在输出层加上SoftMax。

优化器

from torch import optim

optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9) | 带动量的随机梯度下降。

optimizer.zero_grad()

梯度清零。

optimizer.step()

执行优化。

optim.Adadelta/RMSprop/Adagrad/Adam/Adamax()

自适应学习率优化器。

初始化策略

from torch.nn import init

init.xavier_normal(para) | 使用xavier中的初始化方法。

init.kaiming_normal(para) | 使用Dr.He提出的初始化方法。

完整示例

import torch

from torch.autograd import Variable

from torch import nn, optim

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))

y_true = Variable(torch.randn(N, D_in), requires_grad=False)

model = nn.Sequential(nn.Linear(D_in, H),

nn.ReLU(),

nn.Linear(H, D_out))

criterion = nn.MSELoss(size_average=False)

optimizer = optim.Adam(model.parameters(), lr=1e-6)

for epoch in range(500):

y_pred = model(x)

loss = criterion(y_pred, y_true)


optimizer.zero_grad()

loss.backward()

optimizer.step()

一个成为数据科学家的理由

www.hackdata.cn



Yan Xiaodong

数据嗨客出品