PyTorch的基本概念

1.什么是Pytorch，为什么选择Pytroch？

2.Pytroch的安装

3.配置Python环境

4.准备Python管理器

5.通过命令行安装PyTorch

6.PyTorch基础概念

7.通用代码实现流程(实现一个深度学习的代码流程)PyTorch的基本概念  
1.什么是Pytorch，为什么选择Pytroch？  
2.Pytroch的安装  
3.配置Python环境  
4.准备Python管理器  
5.通过命令行安装PyTorch  
6.PyTorch基础概念  
7.通用代码实现流程(实现一个深度学习的代码流程)yTorch的基本概念  
1.什么是Pytorch，为什么选择Pytroch？  
2.Pytroch的安装  
3.配置Python环境  
4.准备Python管理器  
5.通过命令行安装PyTorch  
6.PyTorch基础概念  
7.通用代码实现流程(实现一个深度学习的代码流程) PyTorch的基本概念  
1.什么是Pytorch，为什么选择Pytroch？  
2.Pytroch的安装  
3.配置Python环境  
4.准备Python管理器  
5.通过命令行安装PyTorch  
6.PyTorch基础概念  
7.通用代码实现流程(实现一个深度学习的代码流程)

PyTorch的基本概念  
1.什么是Pytorch，为什么选择Pytroch？  
2.Pytroch的安装  
3.配置Python环境  
4.准备Python管理器  
5.通过命令行安装PyTorch  
6.PyTorch基础概念  
7.通用代码实现流程(实现一个深度学习的代码流程) **import** **numpy** **as** **np**

**import** **torch**

**import** **torch.nn** **as** **nn** *#torch.nn提供了一系列神经网络工具*

**import** **torch.optim** **as** **optim** *#torch.optim是一个实现了各种优化算法的库。大部分常用的方法得到支持*

**from** **torch.autograd** **import** Variable *#torch.autograd提供了类和函数用来对任意标量函数进行求导。*

*# torch.FloatTensor(2,3) 构建一个2\*3 Float类型的张量*

*# torch.DoubleTensor(2,3) 构建一个2\*3 Double类型的张量*

*# torch.ByteTensor(2,3) 构建一个2\*3 Byte类型的张量*

*# torch.CharTensor(2,3) 构建一个2\*3 Char类型的张量*

*# torch.ShortTensor(2,3) 构建一个2\*3 Short类型的张量*

*# torch.IntTensor(2,3) 构建一个2\*3 Int类型的张量*

*# torch.LongTensor(2,3) 构建一个2\*3 Long类型的张量*

*# torch.Tensor是默认的tensor类型（torch.FlaotTensor）的简称。*

dtype = torch.FloatTensor

sentences = [ "i like dog", "i love coffee", "i hate milk"]

word\_list = " ".join(sentences).split()

word\_list = list(set(word\_list))

word\_dict = {w: i **for** i, w **in** enumerate(word\_list)} *#单词：索引集*

number\_dict = {i: w **for** i, w **in** enumerate(word\_list)}*#索引：单词集*

n\_class = len(word\_dict) *# number of Vocabulary*

*#NNLM Parameter*

n\_step=2 *#n-1 in paper*

n\_hidden = 2 *# h in paper*

m=2 *#m in paper*

**def** make\_batch(sentences):

input\_batch=[]

target\_batch=[]

**for** sen **in** sentences:

word =sen.split()

input =[word\_dict[n] **for** n **in** word[:-1]]

target =word\_dict[word[-1]]

input\_batch.append(input)

target\_batch.append(target)

**return** input\_batch,target\_batch

*#Model*

**class** **NNLM**(nn.Module):

**def** \_\_init\_\_(self):

super(NNLM,self).\_\_init\_\_()

*#n\_class\*m的矩阵，代表n\_class个词语，m维度*

self.C = nn.Embedding(n\_class,m)

*#当Paramenters赋值给Module的属性的时候，他会自动的被加到 Module的 参数列表中*

self.H = nn.Parameter

self.H = nn.Parameter(torch.randn(n\_step \* m, n\_hidden).type(dtype))

self.W = nn.Parameter(torch.randn(n\_step \* m, n\_class).type(dtype))

self.d = nn.Parameter(torch.randn(n\_hidden).type(dtype))

self.U = nn.Parameter(torch.randn(n\_hidden, n\_class).type(dtype))

self.b = nn.Parameter(torch.randn(n\_class).type(dtype))

**def** forward(self,X):

X=self.C(X) *#转换成词向量*

*# [batch\_size, n\_step \* n\_class],这里-1在行，表示列固定的情况下，选择合适的行数*

*#行数=元素总个数/列数*

X=X.view(-1,n\_step\*m)

*#选择tanh激活函数*

tanh=torch.tanh(self.d+torch.mm(X,self.H))*#[batch\_size,n\_hidden]*

output =self.b+torch.mm(X,self.W)+torch.mm(tanh,self.U) *#[batch\_size,n\_class]*

**return** output

model=NNLM()

criterion = nn.CrossEntropyLoss() *#交叉熵*

optimizer = optim.Adam(model.parameters(), lr=0.001) *#Adam优化 ,并将先前定义的参数传入优化器*

input\_batch, target\_batch = make\_batch(sentences)

*# 如果用一个 Variable 进行计算, 那返回的也是一个同类型的 Variable.*

input\_batch = Variable(torch.LongTensor(input\_batch))

target\_batch = Variable(torch.LongTensor(target\_batch))

*# Training*

**for** epoch **in** range(5000):

optimizer.zero\_grad()

output = model(input\_batch)

*# output : [batch\_size, n\_class], target\_batch : [batch\_size] (LongTensor, not one-hot)*

loss = criterion(output, target\_batch)

**if** (epoch + 1)%1000 == 0:

print('Epoch:', '**%04d**' % (epoch + 1), 'cost =', '**{:.6f}**'.format(loss))

loss.backward()

optimizer.step()

*# Predict*

predict = model(input\_batch).data.max(1, keepdim=**True**)[1]

*# Test*

print([sen.split()[:2] **for** sen **in** sentences], '->', [number\_dict[n.item()] **for** n **in** predict.squeeze()])