**Pytorch人工智能**

[1. 神经网络NN 1](#_Toc64830185)

[1.1 拟合 1](#_Toc64830186)

[1.2 分类 2](#_Toc64830187)

[1.3 用Sequence搭建网络 4](#_Toc64830188)

[1.4 神经网络的保存和读取 4](#_Toc64830189)

[1.5 数据分批 5](#_Toc64830190)

[1.6 构建数据集 6](#_Toc64830191)

[2. 卷积神经网络CNN 8](#_Toc64830192)

[3. 循环神经网络RNN 11](#_Toc64830193)

[3.1 RNN介绍 11](#_Toc64830194)

[3.2 RNN拟合 11](#_Toc64830195)

[3.3 RNN分类 11](#_Toc64830196)

[4. 自编码AutoEncoder 11](#_Toc64830197)

[5. 强化学习Reinforcement Learning 11](#_Toc64830198)

[6. 生成对抗网络GAN 11](#_Toc64830199)

## 1. 神经网络NN

### 1.1 拟合

import torch

import torch.nn.functional as F

import matplotlib.pyplot as plt

import numpy as np

from torch.autograd import Variable

x = torch.unsqueeze(torch.linspace(0, 2, 100), dim=1) # 必须是2维的

y = torch.sin(3.14 \* x) + 0.1 \* torch.rand(x.size())

x, y = Variable(x), Variable(y) # 变成Variable才能放到神经网络

class Net(torch.nn.Module):

def \_\_init\_\_(self, nfeature, nhidden, noutput):

super(Net, self).\_\_init\_\_() # 固定模式

self.input = torch.nn.Linear(nfeature, nhidden) # 全连接层，参数分别是输入和输出

self.hidden = torch.nn.Linear(nhidden, nhidden)

self.predict = torch.nn.Linear(nhidden, noutput) # 输出层

def forward(self, x): # 定义层与层之间的连接

x = F.relu(self.input(x))

x = F.relu(self.hidden(x))

x = self.predict(x)

return x

net = Net(1, 30, 1) # 网格的一个实例

print(net)

plt.ion() # 绘图相关

plt.show()

optimizer = torch.optim.SGD(net.parameters(), lr=0.1) # 定义优化器，学习效率0.1

loss\_func = torch.nn.MSELoss() # 定义损失函数，对于拟合用MSELoss，对分类用CrossEntropyLoss

# 训练过程

for t in range(10000):

prediction = net(x) # 计算结果

loss = loss\_func(prediction, y) # 计算损失

# 反向传播

optimizer.zero\_grad()

loss.backward()

optimizer.step()

if t % 10 == 0:

plt.cla()

plt.scatter(x.data.numpy(), y.data.numpy())

plt.plot(x.data.numpy(), prediction.data.numpy(), 'r-', lw=5)

plt.ylim(-1.2,1.2)

plt.text(0.5,0,t+10)

plt.pause(0.1)

plt.ioff()

plt.show()

### 1.2 分类

import torch

import torch.nn.functional as F

import matplotlib.pyplot as plt

from torch.autograd import Variable

n\_data = torch.ones(100, 2)

# 生成数据点

x0 = torch.normal(2 \* n\_data, 1)

y0 = torch.zeros(100)

x1 = torch.normal(-2 \* n\_data, 1)

y1 = torch.ones(100)

x2 = torch.normal(2 \* n\_data, 1)

x2[:,0] = x2[:,0] \* -1

y2 = torch.ones(100) \* 2

# 数据拼接

x = torch.cat((x0, x1, x2), 0).type(torch.FloatTensor) # 记住这个形式即可

y = torch.cat((y0, y1, y2), ).type(torch.LongTensor) # 记住这个形式即可

x, y = Variable(x), Variable(y) # 放到Variable里面

# plt.scatter(x.data.numpy()[:, 0], x.data.numpy()[:, 1], c=y.data.numpy())

# plt.show()

# 定义网络

class Net(torch.nn.Module):

def \_\_init\_\_(self, nfeature, nhidden, nout):

super(Net, self).\_\_init\_\_() # 继承

# 定义层，可以自己起名字

self.ein = torch.nn.Linear(nfeature, nhidden) # Input Layer

# Linear是一种形式，后面的两个参数是输入和输出的维度

# 注意前一层的输出和后一层的输入的维度要一样

self.hidden = torch.nn.Linear(nhidden, nhidden) # Hidden Layer

self.predict = torch.nn.Linear(nhidden, nout) # Output Layer

# 定义网络结构

def forward(self, x):

x = F.relu(self.ein(x)) # 输入到隐藏层

# F.relu是ReLu激活函数，还有sigmoid, tahn等

x = F.relu(self.hidden(x)) # 隐藏层之间的连接

x = F.relu(self.hidden(x)) # 可以自己改变层数和结构

x = F.relu(self.hidden(x)) # 比如设置hidden1，2等等

x = self.predict(x) # 隐藏层到输出层

# 一般没有激活函数

return x

# 调用神经网络

net = Net(2, 20, 3)

# 定义优化器，这里是SGD，学习效率0.02

optimizer = torch.optim.SGD(net.parameters(), lr=0.02)

# 定义损失函数，在分类器里用CrossEntropyLoss，在拟合里用MSELoss

loss\_function = torch.nn.CrossEntropyLoss()

# 绘图相关

plt.ion()

plt.show()

# 开始训练

for i in range(5000): # 训练步数5000

out = net(x) # out是神经网络的判断

loss = loss\_function(out, y) # 计算误差，结果放前面，GroundTruth放后面

optimizer.zero\_grad() # 梯度归零

loss.backward() # 反向传播误差

optimizer.step() # 计算梯度

# 绘图相关

if i % 2 == 0:

plt.cla()

# 用softmax得到分类结果，softmax将out转换成分类和概率

# prediction是概率最大的分类

prediction = torch.max(F.softmax(out), 1)[1]

pred\_y = prediction.data.numpy()

target\_y = y.data.numpy()

plt.scatter(x.data.numpy()[:,0], x.data.numpy()[:,1], c=pred\_y)

accuracy = sum(pred\_y == target\_y) / 300

plt.text(1.5, -2, 'Accuracy %.2f' % accuracy)

n = i + 2

plt.text(1.5, -3, 'Iteration %d / 100' % n)

plt.pause(0.1)

# print(prediction)

# 绘图相关

plt.ioff()

plt.show()

# 储存

torch.save(net, 'nn.pkl')

# 读取神经网络

net2 = torch.load('nn.pkl')

# 使用读取的网络

out2 = net2(x)

prediction2 = torch.max(F.softmax(out2), 1)[1]

pred\_y2 = prediction2.data.numpy()

target\_y = y.data.numpy()

plt.scatter(x.data.numpy()[:,0], x.data.numpy()[:,1], c=pred\_y2)

plt.show()

### 1.3 用Sequence搭建网络

不用建立新的Class

net = torch.nn.Sequential(

torch.nn.Linear(2, 20), # 输入层

torch.nn.ReLU(), # 激活函数

torch.nn.Linear(20, 20), # 隐藏层1

torch.nn.ReLU(),

torch.nn.Linear(20, 20), # 隐藏层2

torch.nn.ReLU(),

torch.nn.Linear(20, 20), # 隐藏层3

torch.nn.ReLU(),

torch.nn.Linear(20, 3), # 输出层

)

### 1.4 神经网络的保存和读取

不仅保存神经网络的结构，也保存训练后的参数。

比如已经存在网络net1:

net1 = nn.Sequential(

torch.nn.Linear(2, 30),

torch.nn.ReLU(),

torch.nn.Linear(30, 30),

torch.nn.ReLU(),

torch.Linear(30, 1),

)

optimizer = torch.optim.SGD(net1.parameters(), lr=0.05)

loss\_func = torch.nn.MSELoss()

并且训练好（/训练到一半）了

for t in range(100):

prediction = net1(x)

loss = loss\_func(prediction, y)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

储存起来

torch.save(net1, 'myneuronetwork.pkl')

读取出来

net2 = torch.load('myneuronetwork.pkl')

这样net2就和net1一样了，也就是说

y1 = net1(x\_test)

y2 = net2(x\_test)

y2和y1一样

### 1.5 数据分批

import torch

import torch.utils.data as Data

BATCH\_SIZE = 3 # 一小批的个数

x = torch.linspace(0,1,10) # GroundTruth的例子

y = torch.normal(x)

# print(x.size(), y.size())

torch\_dataset = Data.TensorDataset(x, y) # 建立总的数据库

loader = Data.DataLoader( # 生成loader

dataset=torch\_dataset, # 数据库

batch\_size=BATCH\_SIZE, # 一小批的数量

shuffle=True, # 随机选取

)

for epoch in range(3): # 数据库的使用总次数

for step, (batch\_x, batch\_y) in enumerate(loader):

# 训练

b\_x = Variable(batch\_x)

b\_y = Variable(batch\_y)

output = net(b\_x)

loss = loss\_func(output, b\_y)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

# 打印

print(

'Epoch: ', epoch,

'\tStep: ', step,

'\tx: ', batch\_x.data.numpy(),

'\ty: ', batch\_y.data.numpy()

)

句子for step, (batch\_x, batch\_y) in enumerate(loader)会遍历每一个GroundTruth，比如x和y有10个，则每一个step会选取BATCH\_SIZE=3个，而第4步则会选取最后剩下的1个。也就是说，在1个epoch里，每一个GroundTruth都会训练一次。

### 1.6 构建数据集

from torch.utils.data import Dataset

import matplotlib.pyplot as plt

import numpy as np

import os

class MyData(Dataset): # MyData可以自己起名，必须继承与Dataset

def \_\_init\_\_(self, root\_dir, type='train', transformer=None): # 路径，训练/测试，类型转换

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

type\_indicator = False;

self.type = type

self.transformer = transformer

if self.type == 'train': # 根据训练/测试改变路径

self.path = root\_dir + '/train'

type\_indicator = True

elif self.type == 'test':

self.path = root\_dir + '/test'

type\_indicator = True

else:

print('''The second parameter of MyData() should be 'train' or 'test'.''')

if type\_indicator:

filenames = os.listdir(self.path) # 获取所有文件名

self.imgnames = []

self.labels = []

for filename in filenames:

if filename[-3:] == 'jpg': # 如果是图片，就把名字和Label记下来

self.imgnames.append(filename)

self.labels.append(filename.split('.')[0])

else:

pass

else:

pass

def \_\_getitem\_\_(self, item): # 必须有

img\_path = self.path + '/' + self.imgnames[item]

img = plt.imread(img\_path) # 读取第item个图片

img = self.transformer(np.array(img)) # 转换成标准类型

label = self.labels[item] # 获取第item个标签

if label == 'cat': # 要转换成数字

label = 0

else:

label = 1

return img, label # 返回图片和item

def \_\_len\_\_(self): # 返回长度，必须有

return len(self.imgnames)

联合DataLoader使用数据集

# 定义transformer

imgtotenor = transforms.ToTensor()

imgresize = transforms.Resize([50, 50])

transformer = transforms.Compose([imgtotensor, imgresize])

# 获取训练/测试集

train\_data = readdata.MyData(PATH, type='train', transformer=transformer)

test\_data = readdata.MyData(PATH, type='test', transformer=transformer)

# 建立DataLoader

train\_loader = Data.DataLoader(dataset=train\_data, batch\_size=9, shuffle=True)

test\_loader = Data.DataLoader(dataset=test\_data, batch\_size=4, shuffle=True)

# 使用

for step, (b\_x, b\_y) in enumerate(train\_loader):

# 可以用画图来测试：

if step == 0:

imgs = torchvision.utils.make\_grid(b\_x,3)

imgs = np.transpose(imgs, (1,2,0))

plt.imshow(imgs)

plt.show()

## 2. 卷积神经网络CNN

import torch

import torch.nn as nn

import torch.utils.data as Data

from torch.autograd import Variable

from torchvision import transforms

import matplotlib.pyplot as plt

import readdata

import os

# 定义一些常数

EPOCH = 1000

LR = 0.0025

PATH = '/Users/haibinzhao/Desktop/TECO/Software/Pytorch/cnn/dataset'

# 做个文件记录损失函数和测试准确性

if os.path.exists('loss.txt'):

os.remove('loss.txt')

if os.path.exists('acc.txt'):

os.remove('acc.txt')

# 定义transformer

imgtotensor = transforms.ToTensor()

imgresize = transforms.Resize([64, 64])

transformer = transforms.Compose([imgtotensor, imgresize])

# 搭建数据库和Loader

train\_data = readdata.MyData(PATH, type='train', transformer=transformer)

test\_data = readdata.MyData(PATH, type='test', transformer=transformer)

train\_loader = Data.DataLoader(dataset=train\_data, batch\_size=6, shuffle=True)

test\_loader = Data.DataLoader(dataset=test\_data, batch\_size=30, shuffle=True)

# 定义卷积神经网络

class CNN(nn.Module):

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

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

self.conv1 = nn.Sequential(

nn.Conv2d(

in\_channels=3, # 图片的维度，黑白1，RGB3

out\_channels=10, # 输出特征

kernel\_size=5,

stride=1, # 卷积核每次移动的像素

padding=2, # 扩边

),

nn.ReLU(),

nn.MaxPool2d(kernel\_size=2),

)

self.conv2 = nn.Sequential(

nn.Conv2d(10,20,5,1,2),

nn.ReLU(),

nn.MaxPool2d(2),

)

self.conv3 = nn.Sequential(

nn.Conv2d(20, 30, 5, 1, 2),

nn.ReLU(),

nn.MaxPool2d(2),

)

self.out1 = nn.Sequential(

nn.Linear(30 \* 8 \* 8, 30),

nn.ReLU()

)

self.out2 = nn.Sequential(

nn.Linear(30, 30),

nn.ReLU()

)

self.out3 = nn.Sequential(

nn.Linear(30, 2),

)

def forward(self, x): # 组合起来

x = self.conv1(x)

x = self.conv2(x)

x = self.conv3(x)

x = x.view(x.size(0), -1)

x = self.out1(x)

x = self.out2(x)

output = self.out3(x)

return output

cnn = CNN()

# 定义优化器和损失函数

optimizer = torch.optim.SGD(cnn.parameters(), lr=LR)

loss\_func = nn.CrossEntropyLoss()

# 训练过程

for epoch in range(EPOCH):

for step, (x, y) in enumerate(train\_loader):

b\_x = Variable(x)

b\_y = Variable(y)

output = cnn(b\_x)

loss = loss\_func(output, b\_y)

file\_loss = open('loss.txt', 'a') # 记录loss

file\_loss.write(str(loss.detach().numpy())) # 用detach取出loss的值来

file\_loss.write('\n')

file\_loss.close()

optimizer.zero\_grad() # 更新权重

loss.backward()

optimizer.step()

for i, (test\_x, test\_y) in enumerate(test\_loader): # 测试

b\_x\_test = Variable(test\_x)

b\_y\_test = Variable(test\_y)

output\_test = cnn(b\_x\_test)

prediction\_y = torch.max(output\_test, 1)[1].data.squeeze()

accuracy = sum(prediction\_y == b\_y\_test) / b\_y\_test.size(0)

print('Epoch: ', epoch, '| Accuracy: ', accuracy.numpy())

file\_acc = open('acc.txt', 'a') # 记录accuracy

file\_acc.write(str(accuracy.numpy()))

file\_acc.write('\n')

file\_acc.close()

# 绘图

def draw\_them():

file\_loss = 'loss.txt'

file\_acc = 'acc.txt'

loss\_record = []

accuracy\_record = []

with open(file\_loss, 'r') as f: # 逐行读取

lines = f.readlines()

for line in lines:

loss\_record.append(float(line))

with open(file\_acc, 'r') as f:

lines = f.readlines()

for line in lines:

accuracy\_record.append(float(line))

plt.subplot(1,2,1) # 绘制

plt.plot(loss\_record)

plt.title('Loss', fontsize='large')

plt.xlabel('Step')

plt.ylabel('Loss')

plt.subplot(1,2,2)

plt.plot(accuracy\_record)

plt.title('Accuracy', fontsize='large')

plt.xlabel('Epoch')

plt.ylabel('Accuracy (%)')

plt.show()

draw\_them() # 调用

## 3. 循环神经网络RNN

### 3.1 RNN拟合

import torch

from torch import nn

from torch.autograd import Variable

import numpy as np

import matplotlib.pyplot as plt

TIME\_STEP = 10

INPUT\_SIZE = 1

LR = 0.01

# 搭建RNN网络

class RNN(nn.Module):

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

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

self.rnn = nn.RNN( # RNN层

input\_size=INPUT\_SIZE,

hidden\_size=32,

num\_layers=1,

batch\_first=True, # batch在第1个维度就选True，否则False

)

self.out = nn.Linear(32, 1) # 输出层

def forward(self, x, h\_state): # 2个参数，除了输入x还有h（记忆）

# x, h\_state, r\_out的维度

# x (batch, time\_step, input\_size)

# h\_state (n\_layers, batch, hidden\_size)

# r\_out (batch, time\_step, hidden\_size)

r\_out, h\_state = self.rnn(x, h\_state)

# 取出r\_out中的值

outs = []

for time\_step in range(r\_out.size(1)):

outs.append(self.out(r\_out[:, time\_step, :]))

return torch.stack(outs, dim=1), h\_state

rnn = RNN()

optimizer = torch.optim.Adam(rnn.parameters(), lr=LR)

loss\_func = nn.MSELoss()

# 初始化一个h\_state，给一个None即可

h\_state = None

for step in range(150):

#训练数据

start, end = step \* np.pi, (step + 1) \* np.pi

t\_train = np.linspace(start, end, TIME\_STEP, dtype=np.float32)

x\_train = np.sin(t\_train)

y\_train = np.cos(t\_train)

# 变成Variable，维度要符合之前的要求

x = Variable(torch.from\_numpy(x\_train[np.newaxis, :, np.newaxis]))

y = Variable(torch.from\_numpy(y\_train[np.newaxis, :, np.newaxis]))

prediction, h\_state = rnn(x, h\_state)

h\_state = Variable(h\_state.data) # h\_state要变成Variable才能用于下一次训练

loss = loss\_func(prediction, y)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

plt.plot(t\_train, y\_train, 'b-')

plt.plot(t\_train, prediction.detach().view(-1).numpy(), 'r-')

plt.show()

### 3.2 RNN分类

import torch

from torch import nn

from torch.autograd import Variable

import torchvision.datasets as ds

import torchvision.transforms as transforms

import matplotlib.pyplot as plt

EPOCH = 2

BATCH\_SIZE = 64

TIME\_STEP = 28

INPUT\_SIZE = 28

LR = 0.01

DOWNLOAD\_MINST = False

# 建立训练集

train\_data = ds.MNIST(root='./mnist', train=True, transform=transforms.ToTensor(), download=DOWNLOAD\_MINST)

train\_loader = torch.utils.data.DataLoader(dataset=train\_data, batch\_size=BATCH\_SIZE, shuffle=True)

# 建立测试集

test\_data = ds.MNIST(root='./mnist', train=False, transform=transforms.ToTensor())

test\_x = Variable(test\_data.data).type(torch.FloatTensor)[:2000]/255.

test\_y = test\_data.targets.numpy().squeeze()[:2000]

class RNN(nn.Module):

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

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

self.rnn = nn.LSTM( # 使用LSTM形式的RNN

input\_size=INPUT\_SIZE,

hidden\_size=64,

num\_layers=1,

batch\_first=True,

)

self.out = nn.Linear(64, 10)

def forward(self, x):

# h\_n 是主线的hidden state, h\_c是分线的hidden state

r\_out, (h\_n, h\_c) = self.rnn(x, None) # x (batch\_size, time\_step, input\_size)

out = self.out(r\_out[:, -1, :]) # (batch, time\_step, input)

return out

rnn = RNN()

optimizer = torch.optim.Adam(rnn.parameters(), lr=LR)

loss\_func = nn.CrossEntropyLoss()

for epoch in range(EPOCH):

for step, (x, y) in enumerate(train\_loader):

b\_x = Variable(x.view(-1, 28, 28))

b\_y = Variable(y)

output = rnn(b\_x)

loss = loss\_func(output, b\_y)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

if step % 50 == 0:

test\_output = rnn(test\_x)

pred\_y = torch.max(test\_output, 1)[1].data.numpy().squeeze()

accuracy = sum(pred\_y == test\_y) / test\_y.size

print('Epoch: ', epoch, '| Accuracy: %.2f' % accuracy)

print(pred\_y[:10])

print(test\_y[:10])

## 4. 自编码Autoencoder

## 5. 强化学习Reinforcement Learning

## 6. 生成对抗网络GAN

## 7.图神经网络GNN

## 8. 深度学习DL

感谢周沫凡（<https://mofanpy.com>）

手机屏幕的截图

描述已自动生成

（原图中存在大量空白，因此图片经过适当剪裁拼接）