Dropout原理

Dropout是防止过拟合的一种方法（过拟合overfitting指：模型在训练数据上损失函数较小，预测准确率较高；但是在测试数据上损失函数比较大，预测准确率较低。） 训练神经网络模型时，如果训练样本较少，为了防止模型过拟合，Dropout可以作为一种优化方法。

Dropout是指在神经网络的每次训练中以一个参数p为概率，使部分隐层部分神经元失活，以此来解决过拟合问题，效果可以当作用多个不同的神经网络模型在同一训练集上进行训练，最后集成求平均。Dropout还可以消除某些神经元之间的联系，增强模型的鲁棒性。

用代码实现正则化(L1、L2、Dropout）

L1范数

L1范数是参数矩阵W中元素的绝对值之和，L1范数相对于L0范数不同点在于，L0范数求解是NP问题，而L1范数是L0范数的最优凸近似，求解较为容易。L1常被称为LASSO.

regularization\_loss = 0

for param in model.parameters():

regularization\_loss += torch.sum(abs(param))

for epoch in range(EPOCHS):

y\_pred = model(x\_train)

classify\_loss = criterion(y\_pred, y\_train.float().view(-1, 1))

loss = classify\_loss + 0.001 \* regularization\_loss # 引入L1正则化项

L2范数

L2范数是参数矩阵W中元素的平方之和，这使得参数矩阵中的元素更稀疏，与前两个范数不同的是，它不会让参数变为0，而是使得参数大部分都接近于0。L1追求稀疏化，从而丢弃了一部分特征（参数为0），而L2范数只是使参数尽可能为0，保留了特征。L2被称为Rigde.

PyTorch中实现Dropout

import torch

from torch.autograd import Variable

import matplotlib.pyplot as plt

# torch.manual\_seed(1) # reproducible

N\_SAMPLES = 20

N\_HIDDEN = 300

# training data

x = torch.unsqueeze(torch.linspace(-1, 1, N\_SAMPLES), 1)

y = x + 0.3\*torch.normal(torch.zeros(N\_SAMPLES, 1), torch.ones(N\_SAMPLES, 1))

x, y = Variable(x), Variable(y)

# test data

test\_x = torch.unsqueeze(torch.linspace(-1, 1, N\_SAMPLES), 1)

test\_y = test\_x + 0.3\*torch.normal(torch.zeros(N\_SAMPLES, 1), torch.ones(N\_SAMPLES, 1))

test\_x, test\_y = Variable(test\_x, volatile=True), Variable(test\_y, volatile=True)

# show data

'''

plt.scatter(x.data.numpy(), y.data.numpy(), c='magenta', s=50, alpha=0.5, label='train')

plt.scatter(test\_x.data.numpy(), test\_y.data.numpy(), c='cyan', s=50, alpha=0.5, label='test')

plt.legend(loc='upper left')

plt.ylim((-2.5, 2.5))

plt.show()

'''

net\_overfitting = torch.nn.Sequential(

torch.nn.Linear(1, N\_HIDDEN),

torch.nn.ReLU(),

torch.nn.Linear(N\_HIDDEN, N\_HIDDEN),

torch.nn.ReLU(),

torch.nn.Linear(N\_HIDDEN, 1),

)

net\_dropped = torch.nn.Sequential(

torch.nn.Linear(1, N\_HIDDEN),

torch.nn.Dropout(0.5), # drop 50% of the neuron

torch.nn.ReLU(),

torch.nn.Linear(N\_HIDDEN, N\_HIDDEN),

torch.nn.Dropout(0.5), # drop 50% of the neuron

torch.nn.ReLU(),

torch.nn.Linear(N\_HIDDEN, 1),

)

print(net\_overfitting) # net architecture

print(net\_dropped)

optimizer\_ofit = torch.optim.Adam(net\_overfitting.parameters(), lr=0.01)

optimizer\_drop = torch.optim.Adam(net\_dropped.parameters(), lr=0.01)

loss\_func = torch.nn.MSELoss()

plt.ion() # something about plotting

for t in range(500):

pred\_ofit = net\_overfitting(x)

pred\_drop = net\_dropped(x)

loss\_ofit = loss\_func(pred\_ofit, y)

loss\_drop = loss\_func(pred\_drop, y)

optimizer\_ofit.zero\_grad()

optimizer\_drop.zero\_grad()

loss\_ofit.backward()

loss\_drop.backward()

optimizer\_ofit.step()

optimizer\_drop.step()

if t % 10 == 0:

# change to eval mode in order to fix drop out effect

net\_overfitting.eval()

net\_dropped.eval() # parameters for dropout differ from train mode

# plotting

plt.cla()

test\_pred\_ofit = net\_overfitting(test\_x)

test\_pred\_drop = net\_dropped(test\_x)

plt.scatter(x.data.numpy(), y.data.numpy(), c='magenta', s=50, alpha=0.3, label='train')

plt.scatter(test\_x.data.numpy(), test\_y.data.numpy(), c='cyan', s=50, alpha=0.3, label='test')

plt.plot(test\_x.data.numpy(), test\_pred\_ofit.data.numpy(), 'r-', lw=3, label='overfitting')

plt.plot(test\_x.data.numpy(), test\_pred\_drop.data.numpy(), 'b--', lw=3, label='dropout(50%)')

plt.text(0, -1.2, 'overfitting loss=%.4f' % loss\_func(test\_pred\_ofit, test\_y).data[0], fontdict={'size': 20, 'color': 'red'})

plt.text(0, -1.5, 'dropout loss=%.4f' % loss\_func(test\_pred\_drop, test\_y).data[0], fontdict={'size': 20, 'color': 'blue'})

plt.legend(loc='upper left'); plt.ylim((-2.5, 2.5));plt.pause(0.1)

# change back to train mode

net\_overfitting.train()

net\_dropped.train()

plt.ioff()

plt.show()