Exercise 2-7 MNIST

Jirayu Petchhan, D10907801

Coding

import torch

import torchvision

import torchvision.transforms as transforms

import torch.optim as optim

import torch.nn as nn

import torch.nn.functional as F

import matplotlib.pyplot as plt

import numpy as np

########################################################################

# The output of torchvision datasets are PILImage images of range [0, 1].

# We transform them to Tensors of normalized range [-1, 1].

if \_\_name\_\_ == "\_\_main\_\_":

def imshow(img):

img = img / 2 + 0.5 # unnormalize

npimg = img.numpy()

plt.imshow(np.transpose(npimg, (1, 2, 0)))

plt.show()

transform = transforms.Compose(

[ transforms.ToTensor(),

transforms.Normalize([0.5], [0.5])])

# trainset = torchvision.datasets.CIFAR10(root='./data', train=True,

# download=True, transform=transform)

trainset = torchvision.datasets.MNIST(root="./data1", train=True, transform=transform, target\_transform=None,

download=True)

trainloader = torch.utils.data.DataLoader(trainset, batch\_size=4,

shuffle=True, num\_workers=2)

# testset = torchvision.datasets.CIFAR10(root='./data', train=False,

# download=True, transform=transform)

testset = torchvision.datasets.MNIST(root="./data1", train=False, transform=transform, target\_transform=None,

download=True)

testloader = torch.utils.data.DataLoader(testset, batch\_size=4,

shuffle=False, num\_workers=2)

classes = ('0', '1', '2', '3',

'4', '5', '6', '7', '8', '9')

########################################################################

# 2. Define a Convolutional Neural Network

# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

# Copy the neural network from the Neural Networks section before and modify it to

# take 3-channel images (instead of 1-channel images as it was defined).

class Net(nn.Module):

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

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

self.conv1 = nn.Conv2d(1, 6, 5)

self.pool = nn.MaxPool2d(2, 2)

self.conv2 = nn.Conv2d(6, 16, 5)

self.fc1 = nn.Linear(16 \* 4 \* 4, 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)))

# print(x.shape)

x = x.view(-1, 16 \* 4 \* 4)

x = F.relu(self.fc1(x))

x = F.relu(self.fc2(x))

x = self.fc3(x)

return x

net = Net()

########################################################################

# 3. Define a Loss function and optimizer

# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

# Let's use a Classification Cross-Entropy loss and SGD with momentum.

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(), lr=0.05, momentum=0.9)

########################################################################

# 4. Train the network

# ^^^^^^^^^^^^^^^^^^^^

#

# This is when things start to get interesting.

# We simply have to loop over our data iterator, and feed the inputs to the

# network and optimize.

for epoch in range(3): # loop over the dataset multiple times

running\_loss = 0.0

for i, data in enumerate(trainloader, 0):

# get the inputs; data is a list of [inputs, labels]

inputs, labels = data

# 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.item()

if i % 2000 == 1999: # print every 2000 mini-batches

print("Epoch : {} steps : {} Training Loss : {}".format(epoch + 1, i + 1, running\_loss / 2000))

running\_loss = 0.0

print('Finished Training')

########################################################################

# 5. Test the network on the test data

# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

#

# We have trained the network for 2 passes over the training dataset.

# But we need to check if the network has learnt anything at all.

#

# We will check this by predicting the class label that the neural network

# outputs, and checking it against the ground-truth. If the prediction is

# correct, we add the sample to the list of correct predictions.

#

# Okay, first step. Let us display an image from the test set to get familiar.

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)))

#

# ########################################################################

# # Okay, now let us see what the neural network thinks these examples above are:

#

outputs = net(images)

#

# ########################################################################

# # The outputs are energies for the 10 classes.

# # The higher the energy for a class, the more the network

# # thinks that the image is of the particular class.

# # So, let's get the index of the highest energy:

\_, predicted = torch.max(outputs, 1)

#

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]

for j in range(4)))

########################################################################

# The results seem pretty good.

#

# Let us look at how the network performs on the whole dataset.

correct = 0

total = 0

with torch.no\_grad():

for data in testloader:

images, labels = data

outputs = net(images)

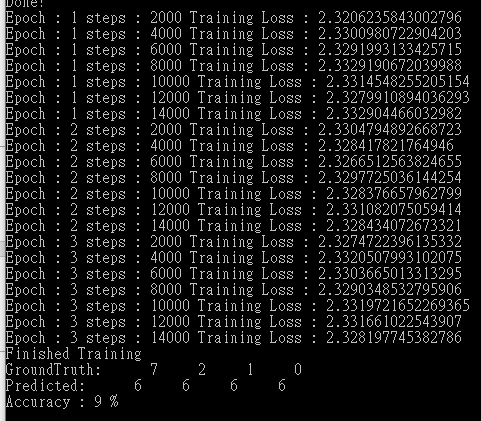
\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

print('Accuracy : %d %%' % (100 \* correct / total))

Training Model



Acuuracy is only 9 %

Visualize (Ground Truth)

