6900 Deep Learning Homework #1

Alex Darwiche

I used a series of fully connected layers, ReLU activation functions, and batch normalization to achieve test accuracy of over 45%.

Training Screenshot(s):

A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

Testing Screenshot(s):





**Code is below:**

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

# First of the first, please start writing it early!

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

import torch

import torch.nn as nn

import torch.nn.functional as F

import torch.utils.data as Data

import torchvision

import torch.optim as optim

import os

from PIL import Image

import torchvision.transforms as transforms

import matplotlib.pyplot as plt

import sys

import pandas as pd

# Normalize the CIFAR10 Dataset (from Pytorch Website)

transform = transforms.Compose([

transforms.Resize((32, 32)), # Resize image to 32x32

transforms.ToTensor(), # Convert image to tensor

transforms.Normalize(mean=[0.4914, 0.4822, 0.4465], std=[0.2470, 0.2435, 0.2616]) # CIFAR-10 normalization

])

# ----------------- prepare training data -----------------------

train\_data = torchvision.datasets.CIFAR10(

root='./data.cifar10', # location of the dataset

train=True, # this is training data

transform=transform, # Converts a PIL.Image or numpy.ndarray to torch.FloatTensor of shape (C x H x W)

download=True # if you haven't had the dataset, this will automatically download it for you

)

# Define a batch size to set with, using 128

batch\_size = 128

# Load the training data from the dataset, breaking it into batches

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

# ----------------- prepare testing data -----------------------

test\_data = torchvision.datasets.CIFAR10(root='./data.cifar10/', train=False, transform=transform)

# Load the training data from the dataset, breaking it into batches

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

# ----------------- build the model ------------------------

# Define a Deep Nueral Network with only Fully Connected Layers or Batch Normalization

class net(nn.Module):

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

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

self.fc1 = nn.Linear(32\*32\*3,2000)

self.fc2 = nn.Linear(2000,1000)

self.fc3 = nn.Linear(1000,500)

self.fc4 = nn.Linear(500,100)

self.fc5 = nn.Linear(100,10)

self.bn1 = nn.BatchNorm1d(2000)

self.bn2 = nn.BatchNorm1d(100)

# Define a forward pass through the model

def forward(self, x):

x = torch.flatten(x, 1)

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

x = self.bn1(x)

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

x = F.relu(self.fc3(x))

x = F.relu(self.fc4(x))

x = self.bn2(x)

x = self.fc5(x)

return x

# ------ maybe some helper functions -----------

def test\_accuracy():

'''

This function will return the accuracy of the model

on the testing data.

'''

model.eval() # switch the model to evaluation mode

correct = 0

total = 0

with torch.no\_grad():

for data in test\_loader:

images, target = data # split data into images/target

outputs = model(images) # Determine the "prediction" or "output" from the model based on the images

loss = loss\_func(outputs,target) # Find the loss (difference between ground truth and prediction)

\_, predicted = torch.max(outputs, 1) # Determine the most likely label for the image

total += target.size(0) # Find the amount of total images in batch

correct += (predicted == target).sum().item() # Find the total images that are correctly labeled

# Return Accuracy and Loss

return [(100 \* correct / total),loss.item()]

def save\_model(test\_accuracy):

'''

This function will save the model in the event that it exceeds

a previous model's test accuracy, or if a model does not exist.

'''

model\_path = "./model/model.pt" # Where to save model and what to name it.

# Determine if a model already exists, save model if not

if os.path.exists(model\_path):

model2 = torch.load(model\_path) # Load the model from the save. This isn't exactly the "load\_state\_dict" as I added test\_accuracy to the model save

if model2['test\_accuracy'] < test\_accuracy: # If the "new model" has better test accuracy, then continue

os.remove(model\_path) # Remove the old model

torch.save({ # save the new model parameters AND its test accuracy

'model\_state\_dict': model.state\_dict(),

'test\_accuracy': test\_accuracy

}, model\_path)

else:

torch.save({ # save the new model parameters AND its test accuracy

'model\_state\_dict': model.state\_dict(),

'test\_accuracy': test\_accuracy

}, model\_path)

def load\_model():

'''

This function loads the model\_state\_dict or model state dictionary from the saved

model.

'''

model = net()

model.load\_state\_dict(torch.load("./model/model.pt")['model\_state\_dict'])

return model

# This if statement dictates the interactions depending on the arugments passed to command line.

# IF 'Train', train the model.

# IF 'Test' or 'Predict', try to predict the command line image's class

if 'train' in sys.argv[1:]:

model = net()

loss\_func = nn.CrossEntropyLoss()

optimizer = optim.SGD(model.parameters(), lr = .001, momentum = .9, weight\_decay=0.05)

epochs = 10

# Print the Columns for the results table

results = pd.DataFrame({"Epoch": [], "Step": [], "Train\_Loss": [], "Train\_Acc": [], "Test\_Loss": [], "Test\_Acc": []})

print(" | ".join("{:<10}".format(col) for col in results.columns.to\_list()))

for epoch in range(epochs):

for step, (input, target) in enumerate(train\_loader):

train\_loss = 0

model.train() # set the model in training mode

optimizer.zero\_grad() # Set all gradients back to 0

output = model(input) # Forward pass through the model

loss = loss\_func(output,target) # Evaluate the loss/difference from ground truth

loss.backward() # Calculate gradients to back propogate

optimizer.step() # Back-propogate the losses

train\_loss += loss.item() # Accumulate losses

\_, predicted = output.max(1)

if step == 0:

test\_acc, test\_loss = test\_accuracy() # Get test accuracy and loss

correct = 0

total = 0

total += target.size(0)

correct += (predicted == target).sum().item() # Calculate train accuracy and loss, exactly as done for test, just on the training data.

train\_acc = 100 \* correct / total

# Create the new row of Results Dataframe

new\_row = pd.DataFrame([{"Epoch": epoch, "Step": step, "Train\_Loss": round(train\_loss, 2), "Train\_Acc": round(train\_acc, 2), "Test\_Loss": round(test\_loss, 2), "Test\_Acc": round(test\_acc, 2)}])

results = pd.concat([results,new\_row], ignore\_index=True)

# Print the new row of data from the first step of the new epoch

print(" | ".join("{:<10}".format(str(value)) for value in results.iloc[-1].values))

# Attempt to save model if needed

save\_model(test\_acc)

elif 'predict' in sys.argv[1:] or 'test' in sys.argv[1:]:

# Need to "normalize" the picture that is being input, ensuring its the correct size

# and the channels are normalized based on mean/std.

transform = transforms.Compose([

transforms.Resize((32, 32)), # Resize image to 32x32

transforms.ToTensor(), # Convert image to tensor

transforms.Normalize(mean=[0.4914, 0.4822, 0.4465], std=[0.2470, 0.2435, 0.2616]) # CIFAR-10 normalization

])

# Load image from file

image\_path = sys.argv[2] # Replace with your file path

image = Image.open(image\_path).convert("RGB")

# Apply transformations

image\_tensor = transform(image).unsqueeze(0) # Add batch dimension (1, 3, 32, 32)

model = load\_model()

# Run inference

with torch.no\_grad():

model.eval()

output = model(image\_tensor)

predicted\_class = output.argmax(1).item()

# CIFAR-10 Class Labels

labels\_map = {

0: 'airplane', 1: 'automobile', 2: 'bird', 3: 'cat', 4: 'deer',

5: 'dog', 6: 'frog', 7: 'horse', 8: 'ship', 9: 'truck'

}

print(f"prediction result: {labels\_map[predicted\_class]}")