Lab Assignment 1

Student Number : 8551145

Name : Edbert Taidy

Date : 10 July 2024

Tutorial Group : T03

Before starting the training, it is vital to install torchvision using pip. Otherwise, the program itself won't work!

```
!pip install torch torchvision
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (2.3.1+cu121)
Requirement already satisfied: torchvision in /usr/local/lib/python3.10/dist-packages (0.18.1+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch) (3.15.4)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (from torch) (4.12.2)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch) (1.13.0)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch) (3.3)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch) (2023.6.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch) (12.1.105)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch) (12.1.105
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch) (12.1.105)
Requirement already satisfied: nvidia-cudnn-cu12==8.9.2.26 in /usr/local/lib/python3.10/dist-packages (from torch) (8.9.2.26)
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in /usr/local/lib/python3.10/dist-packages (from torch) (12.1.3.1)
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in /usr/local/lib/python3.10/dist-packages (from torch) (11.0.2.54)
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in /usr/local/lib/python3.10/dist-packages (from torch) (10.3.2.106)
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in /usr/local/lib/python3.10/dist-packages (from torch) (11.4.5.107
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in /usr/local/lib/python3.10/dist-packages (from torch) (12.1.0.106
Requirement already satisfied: nvidia-nccl-cu12==2.20.5 in /usr/local/lib/python3.10/dist-packages (from torch) (2.20.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch) (12.1.105)
Requirement already satisfied: triton==2.3.1 in /usr/local/lib/python3.10/dist-packages (from torch) (2.3.1)
Requirement already satisfied: nvidia-nvjitlink-cu12 in /usr/local/lib/python3.10/dist-packages (from nvidia-cusolver-cu12==11.4.5.1
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from torchvision) (1.25.2)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.10/dist-packages (from torchvision) (9.4.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch) (2.1.5)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy->torch) (1.3.0)
```

How to train a neural network to classify images?

1. Upload the neccessary packages required for the program.

```
#Import statements that are essential for the program.

import torch
import torchvision
import torchvision.transforms as transforms

import matplotlib.pyplot as plt
import numpy as np
```

Some important lines in the code:

1. torchvision.datasets.CIFAR10() The following line provides an interface for CIFAR10 dataset. CIFAR10 is a image dataset used for machine learning, especially in image recognition. It has 60,000 images that are divided into 10 classes.
2. torch.utils.data.DataLoader() This class provides a representative as a python iterable over a dataset.
3. torchvision.transforms.Normalize() This command normalizes a image tensor by calculating the mean and standard deviation.

2. Normalize the dataset that will be tested for the program. In this case, CIFAR10 will be used to train the model.

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

size = 6 #The size determines how many images we want the program to recognize at the same time.

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch_size=size,
                                         shuffle=True, num_workers=2)

classes = ('airplane','automobile','bird','cat','deer'
,'dog','frog','horse','ship','truck')
#CIFAR10 has 10 classes of images.

print(f"There are {len(trainset)} images in the trainset")
print(f"There are {len(testset)} images in the testset")
```

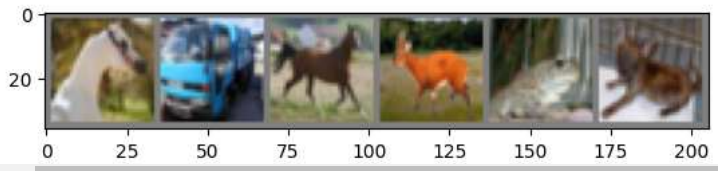
```
Files already downloaded and verified
Files already downloaded and verified
There are 50000 images in the trainset
There are 10000 images in the testset
```

Some sample images:

```
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(size)))
```



Define the neural network that is needed

```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 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)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

3. Define the loss function and the optimizer

```
import torch.nn as nn
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

4. Train and test the network

```
for epoch in range(6):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data

        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 1000 == 999:    # print every 1000 mini-batches
            print(f'[Epoch {epoch + 1}, Batch {i + 1}] loss: {running_loss / 1000:.3f}')
            running_loss = 0.0

print('Finished Training')
```

```
[Epoch 1, Batch 1000] loss: 2.289
[Epoch 1, Batch 2000] loss: 2.091
[Epoch 1, Batch 3000] loss: 1.887
[Epoch 1, Batch 4000] loss: 1.734
[Epoch 1, Batch 5000] loss: 1.687
[Epoch 1, Batch 6000] loss: 1.603
[Epoch 1, Batch 7000] loss: 1.560
[Epoch 1, Batch 8000] loss: 1.514
[Epoch 2, Batch 1000] loss: 1.448
[Epoch 2, Batch 2000] loss: 1.426
[Epoch 2, Batch 3000] loss: 1.391
[Epoch 2, Batch 4000] loss: 1.377
[Epoch 2, Batch 5000] loss: 1.336
[Epoch 2, Batch 6000] loss: 1.330
[Epoch 2, Batch 7000] loss: 1.331
[Epoch 2, Batch 8000] loss: 1.314
[Epoch 3, Batch 1000] loss: 1.226
[Epoch 3, Batch 2000] loss: 1.244
[Epoch 3, Batch 3000] loss: 1.232
[Epoch 3, Batch 4000] loss: 1.222
[Epoch 3, Batch 5000] loss: 1.202
[Epoch 3, Batch 6000] loss: 1.185
```

```
[Epoch 3, Batch 7000] loss: 1.199
[Epoch 3, Batch 8000] loss: 1.201
[Epoch 4, Batch 1000] loss: 1.120
[Epoch 4, Batch 2000] loss: 1.120
[Epoch 4, Batch 3000] loss: 1.124
[Epoch 4, Batch 4000] loss: 1.110
[Epoch 4, Batch 5000] loss: 1.129
[Epoch 4, Batch 6000] loss: 1.124
[Epoch 4, Batch 7000] loss: 1.105
[Epoch 4, Batch 8000] loss: 1.089
[Epoch 5, Batch 1000] loss: 1.025
[Epoch 5, Batch 2000] loss: 1.043
[Epoch 5, Batch 3000] loss: 1.003
[Epoch 5, Batch 4000] loss: 1.036
[Epoch 5, Batch 5000] loss: 1.040
[Epoch 5, Batch 6000] loss: 1.058
[Epoch 5, Batch 7000] loss: 1.037
[Epoch 5, Batch 8000] loss: 1.053
[Epoch 6, Batch 1000] loss: 0.955
[Epoch 6, Batch 2000] loss: 0.978
[Epoch 6, Batch 3000] loss: 0.978
[Epoch 6, Batch 4000] loss: 0.970
[Epoch 6, Batch 5000] loss: 0.935
[Epoch 6, Batch 6000] loss: 0.985
[Epoch 6, Batch 7000] loss: 0.982
[Epoch 6, Batch 8000] loss: 1.019
Finished Training
```

```
# Test the network
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(f'Accuracy of the network on the 10000 test images: {100 * correct / total} %')
```

```
Accuracy of the network on the 10000 test images: 62.89 %
```

The results above are just for the whole trainset, how about the accuracy of all images in each class?

```
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

```
Accuracy for class: airplane is 67.4 %
Accuracy for class: automobile is 69.9 %
Accuracy for class: bird  is 47.6 %
Accuracy for class: cat   is 51.6 %
Accuracy for class: deer  is 54.7 %
Accuracy for class: dog   is 47.8 %
Accuracy for class: frog  is 79.3 %
Accuracy for class: horse is 71.3 %
Accuracy for class: ship  is 67.3 %
Accuracy for class: truck is 72.0 %
```

Questions

---

1. Can you explain what is the normalization about?

In this following context, we are using torchvision.transforms.Normalize() for our model. It accepts tuples of RGB channel, the first parameter being calculating the mean, while the second parameter of the function calculates the standard deviation of each RGB channel.

Normalization is essential in image recognition, as it stablizes training, improves performance and reducing the risk of outliers.

2. Do the trainset and testset created using random sampling or stratification? Is the data distribution even?

The trainset and the testset are not created through random sampling or stratification. It loads the dataset as it was provided (CIFAR10).

3.Why a 'classes' tuple is created?

The tuple is created for the purpose of storing the ten sub-classes of the CIFAR10 dataset. It also increases accuracy and precission of the training model.

4.How many images are there in the trainset and testset?

```
numOfimagesInTrainset = len(trainset)
numOfimagesInTestset = len(testset)
print(f'Number of images in the trainset: {numOfimagesInTrainset}') #There are 50000 images in the trainset
print(f'Number of images in the testset: {numOfimagesInTestset}')   #There are 10000 images in the testset
```

```
Number of images in the trainset: 50000
Number of images in the testset: 10000
```

5. Explain briefly your understanding of the defined model (CNN) -> Convolutional Neural Network

Import statements

'torch.nn' is a basic package from pytorch for developing neural networks.

'torch.nn.functional' is a sub-package from torch.nn where it imports all functions that are related to the package

The following python code snippet defines a class named Net, and it takes nn.Module as a parameter.

The constructor itself calls another constructor from it's subclass. It also contains a list of parameters that stores the following values, such as:

- nn.Conv2d() defines a convolutional layer, which has 3 parameters: in_channels, out_channels and kernelSize
- nn.Linear() defines the size of each input sample and the output sample for testing.
- nn.MaxPool2d() defines the 2D pooling operation over images.

It has a defined function named "forward", where it takes a parameter x, and it the variable x, which is a tensor and it undergoes the following changes: Convolutional Layers and Pooling:

Convolutional Layers and Pooling:

1. Applying the first convolution (conv1) then using a ReLU activation function and pooling layer (pool): x = self.pool(F.relu(self.conv1(x))).
2. Using a ReLU activation function on the outcomes of the second convolution(conv2) and applying pooling: x = self.pool(F.relu(self.conv2(x))).

Flattening:

1. Flattens tensor x starting from dimension 1 to convert multi-dimensional tensor to 2D tensor where each sample is represented by one row: x = torch.flatten(x, 1).

Fully Connected Layers:

1. Applies first fully connected layer (fc1) followed by ReLU activation: x = F.relu(self.fc1(x)).
2. Uses the second fully connected layer(fc2) alongside ReLU activations: x = F.relu(self.fc2(x)).
3. Applies final fully connected layer(fc3), which is usually utilized in producing output.

```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
```

```
        self.fc1 = nn.Linear(16 * 5 * 5, 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)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

6. Train the model, and discuss on the losses.

After training the model itself, there are some vital obeservations that I can take note:

1. The losses keep decreasing in some early iterations, but in some iterations, there are slight increases, but the loss values are somehow steady.

Sample output when epoch = 4

```
[Epoch 1, Batch 1000] loss: 2.268
[Epoch 1, Batch 2000] loss: 2.038
[Epoch 1, Batch 3000] loss: 1.896
[Epoch 1, Batch 4000] loss: 1.758
[Epoch 1, Batch 5000] loss: 1.669
[Epoch 1, Batch 6000] loss: 1.618
[Epoch 1, Batch 7000] loss: 1.558
[Epoch 1, Batch 8000] loss: 1.522
[Epoch 2, Batch 1000] loss: 1.448
[Epoch 2, Batch 2000] loss: 1.432
[Epoch 2, Batch 3000] loss: 1.398
[Epoch 2, Batch 4000] loss: 1.379
[Epoch 2, Batch 5000] loss: 1.369
[Epoch 2, Batch 6000] loss: 1.344
[Epoch 2, Batch 7000] loss: 1.300
[Epoch 2, Batch 8000] loss: 1.309
[Epoch 3, Batch 1000] loss: 1.246
[Epoch 3, Batch 2000] loss: 1.238
[Epoch 3, Batch 3000] loss: 1.222
[Epoch 3, Batch 4000] loss: 1.218
[Epoch 3, Batch 5000] loss: 1.207
[Epoch 3, Batch 6000] loss: 1.193
[Epoch 3, Batch 7000] loss: 1.207
[Epoch 3, Batch 8000] loss: 1.197
[Epoch 4, Batch 1000] loss: 1.134
[Epoch 4, Batch 2000] loss: 1.130
[Epoch 4, Batch 3000] loss: 1.130
[Epoch 4, Batch 4000] loss: 1.114
[Epoch 4, Batch 5000] loss: 1.134
[Epoch 4, Batch 6000] loss: 1.093
[Epoch 4, Batch 7000] loss: 1.112
[Epoch 4, Batch 8000] loss: 1.116
Finished Training
```

2. By increasing the epoch, it can be observed that the losses are always lower. Lower losses mean that image recognition is more accurate and lower probability of false positives.

Model is considered have a good performance if the loss is between 1 and 1.5

When the epoch is set to 6, there are loss values that have fallen down below 1.

But the initial iteration always have a loss amount of 2.

However, a major disadvantage is the increasing amount of time required to train the model itself. Also, this may effect the efficiency of the model required to detect and recognize multiple images at the same time.

7. What is the validation accuracy after you have trained the model?

After the model is trained, the accuracy of the network to recognize the 10000 images is around 61%.

However, there are accuracy differences in terms of classes of images.

For example, when the epoch is set to 4:

```
Accuracy for class: airplane is 66.4 %
Accuracy for class: automobile is 66.1 %
Accuracy for class: bird  is 34.5 %
Accuracy for class: cat   is 32.9 %
Accuracy for class: deer  is 50.8 %
Accuracy for class: dog   is 55.1 %
Accuracy for class: frog  is 62.2 %
Accuracy for class: horse is 71.1 %
Accuracy for class: ship  is 79.7 %
Accuracy for class: truck is 74.2 %
```

From the results itself, we can infer that the model have low accuracy in detecting in certain classes of images, such as bird and cat.