CS 559 Neural Networks – Assignment 5

Zohair Hashmi | UIN: 668913771 | zhashm4@uic.edu

NOTE:

- Libraries used in python code: torch, numpy, torchvision, zipfle, tqdm, os
- Model Training Output is attached at the end of the file.

Design Procedure:

The given model runs for 20 epochs using and adams optimizer at a learning rate of 0.001. The model took significant time to train (approximately 7 hours) due to limitations of my GPU, which is why I have documented the results of the first successful instance of model training.

The code follows the following steps:

1. **Zip File Extraction:** Data is extracted from the zip file and stored in the same folder using zipfile library.
2. **Train Test Split:** I split the dataset into train & test by creating 2 respective folders and copying the data as instructed in the homework requirements (8000 images of each class to train, 2000 to test)
3. **Data Processing:** The data processing part has the following steps:
   - Defining data transformation: Images are downsampled to 32x32 and converted to tensors. Each image channel is normalized to 0.5.
   - Images are loaded as datasets using ImageFolder from torch.
   - Labels are extracted from file names and assigned to the respective dataset sample targets.
   - After processing train and test data loaders are created.
4. **Model Definition:**
   - **Input Layer:** An input of 3 channels (RGB) of size 32x32 is passed to the first convolutional layer.
   - **First Convolutional Layer:** After the first convolution, the output is passed through a Rectified Linear Unit (ReLU) activation and then max pooling is applied with a 2x2 window and a stride of 2.
   - **Second Convolutional Layer:** Similar to the first convolutional layer, it takes 32 input channels and produces 64 output channels. Again, the output is passed through a ReLU activation function and max pooling.
   - **First Fully Connected Layer:** The output from the second convolutional layer is flattened before passing the data through fully connected layers. The flattened output is connected to a fully connected layer with 128 neurons. ReLU activation is applied again.
   - **Second Fully Connected Layer:** The second fully connected layer produces an output with 9 neurons, which outputs 9 different classes in this network.
   - **Output:** 9 classes representing 9 different shapes.
5. **Parallel Processing:** To wrap the model and parallelize across multiple GPUs for faster training.
6. **Loss function:** Cross Entropy Loss is used for multi-class classification problems, which seems appropriate given that the network architecture ends with a fully connected layer of size 9, suggesting a classification task with 9 classes.
7. **Adam Optimizer:** Used with a learning rate of 0.001.
8. Default torch hyper-parameters are used for initializing the model, i.e. the weights and biases initialized for the convolutional neural network are the default ones provided by torch library.
9. Model is then trained using the above defined model for 20 Epochs. After each epoch loss is calculated and minimized.

Challenges:

Since I copied my files into two respective folders of train and test only, without creating further sub-folders for each output class, torch interpreted all the images to belong from a single class. Despite extracting labels from the file names and assigning to the respective the image labels the issue persisted. Hence, I achieved a 100% train and test accuracy in my first attempt of model training.

Therefore, to solve this problem I looked at the documentation for torch and the usage of ImageFolder. It turns out that creating datasets using ImageFolder leads to assignments of classes to sample labels based on the folder distributions and stores them as tuples. Since tuples are immutable, I had to convert *train_dataset.samples* to *test_dataset.samples* to lists first and then make the assignments of classes using the labels retrieved from the image file names.

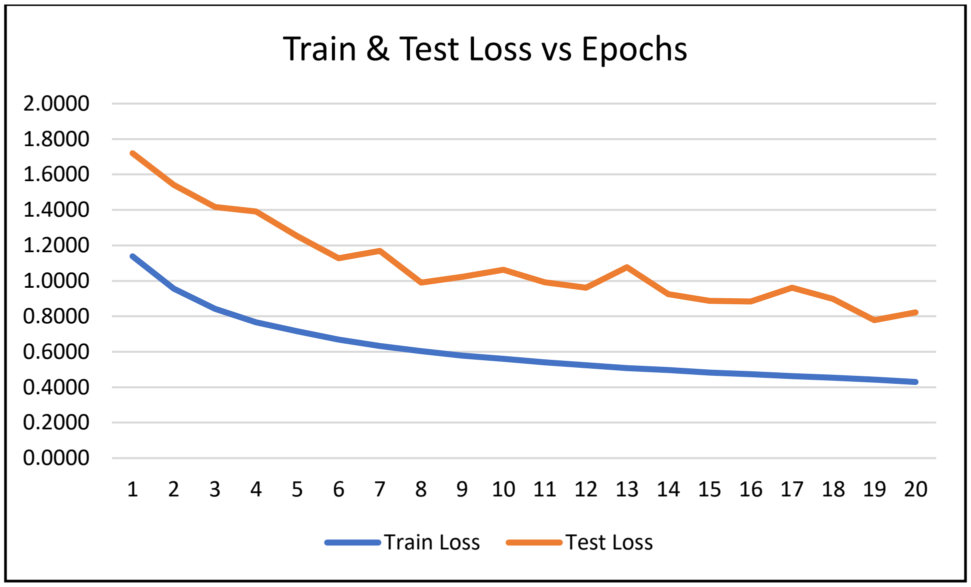
Results:

Following are the tabulated results of the model training and testing which are reflected in graphical manner in the next section.

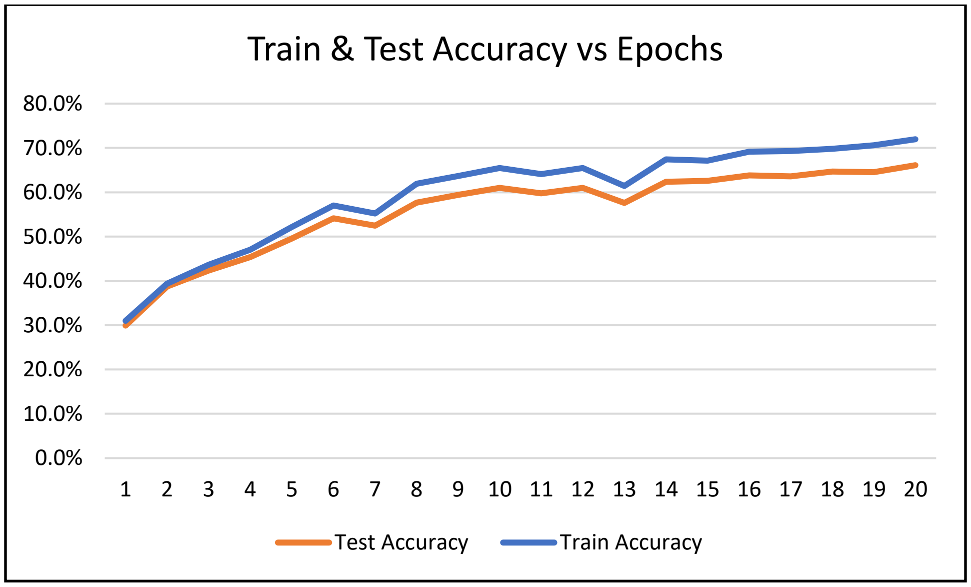
Epochs	Train Loss	Test Loss	Test Accuracy	Train Accuracy
1	1.1385	1.7201	29.9%	31.0%
2	0.9555	1.5416	38.7%	39.4%
3	0.8423	1.4167	42.3%	43.6%
4	0.7670	1.3921	45.4%	47.1%
5	0.7152	1.2516	49.5%	52.1%
6	0.6683	1.1272	54.1%	57.0%
7	0.6326	1.1699	52.5%	55.2%
8	0.6032	0.9905	57.6%	62.0%
9	0.5789	1.0230	59.4%	63.6%
10	0.5605	1.0632	61.0%	65.5%
11	0.5395	0.9917	59.8%	64.1%
12	0.5237	0.9608	61.0%	65.5%
13	0.5077	1.0764	57.6%	61.4%
14	0.4976	0.9257	62.3%	67.4%
15	0.4821	0.8867	62.6%	67.1%
16	0.4727	0.8841	63.8%	69.2%
17	0.4624	0.9611	63.6%	69.3%
18	0.4531	0.8988	64.7%	69.8%
19	0.4420	0.7782	64.5%	70.6%
20	0.4297	0.8215	66.1%	72.0%

Graphs (Training & Test):

The following graph shows that with each epoch there was a decrease in training and test loss observed. Both the losses tend to converge after a few epochs reflecting model's limitations to learn further.



The following graph shows that with each epoch there was an improvement in training and test accuracies observed. Training accuracy tends to improve better than the test accuracy as expected since the model happens to fit better on the given training set at each epoch.



Python Code – Model Design:

```python
############## ASSIGNMENT 5 ##############
## ZOHAIR HASHMI | 668913771 | zhashm4@uic.edu

### Importing Libraries
import os
import shutil
from tqdm import tqdm
import zipfile
import numpy as np

import torch
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim


### **ZIP FILE EXTRACTION**
zip_file_path = "geometry_dataset.zip"
dataset_path = "./geometry_dataset"

# Get the total number of files in the zip archive
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    file_list = zip_ref.namelist()
    total_files = len(file_list)

# Use tqdm to track progress during extraction
with tqdm(total=total_files, desc="Extracting files", unit="file") as pbar:
    with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
        for file in file_list:
            zip_ref.extract(file, dataset_path)
            pbar.update(1)


### **TRAIN TEST SPLIT**
# Sort the files in the dataset folder
files = os.listdir(os.path.join(dataset_path, 'output'))
files.sort()

# Create two folders for train and test
train_path = os.path.join(dataset_path, "train")
test_path = os.path.join(dataset_path, "test")
os.makedirs(train_path, exist_ok=True)
os.makedirs(test_path, exist_ok=True)
```

```python
# Copy 8000 .png images of each class to train & 2000 .png images to test folder
for i in range(9):

    # Train Folder
    for j in tqdm(range(8000), desc=f'Copying files to train (Class {i+1})'):
        source_path = os.path.join(dataset_path, "output", files[i*10000 + j])
        destination_path = os.path.join(train_path, "output", files[i*10000 + j])
        shutil.copy(source_path, destination_path)

    # Test Folder
    for j in tqdm(range(2000), desc=f'Copying files to test (Class {i+1})'):
        source_path = os.path.join(dataset_path, "output", files[i*10000 + j + 8000])
        destination_path = os.path.join(test_path, "output", files[i*10000 + j + 8000])
        shutil.copy(source_path, destination_path)

### **DATA PROCESSING**
# Define data transformations to apply to the images when loaded
transform = transforms.Compose([
    transforms.Resize((32, 32)),  # Resize images to a common size
    transforms.ToTensor(),  # Convert images to PyTorch tensors
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),  # Normalize pixel values
])

# Define the paths to your training and test data folders
train_path = os.path.join(dataset_path, "train")
test_path = os.path.join(dataset_path, "test")

train_dataset = ImageFolder(root=train_path, transform=transform)
test_dataset = ImageFolder(root=test_path, transform=transform)

# Get labels from file names and assign then to the respective datasets
train_labels = []
for file_name in train_dataset.samples:
    train_labels.append(file_name[0].split('/')[-1].split('\\')[3].split('_')[0])
train_labels = np.array(train_labels)

unique_labels, label_indices = np.unique(train_labels, return_inverse=True) # Convert labels to
unique indices
label_mapping = dict(zip(range(len(unique_labels)), unique_labels)) # Create a mapping from label
indices to labels
train_dataset.targets = torch.tensor(label_indices) # Add label indices to the dataset
train_dataset.classes = unique_labels # Change training classes to unique labels
train_dataset.class_to_idx = label_mapping # Change class_to_idx dictionary to match new labels
train_dataset.idx_to_class = {v: k for k, v in label_mapping.items()} # Change idx_to_class
dictionary to match new labels
train_dataset.samples = [list(elem) for elem in train_dataset.samples]
for i in range(len(train_dataset.samples)):
    train_dataset.samples[i][1] = label_indices[i]
```

```
test_labels = []
for file_name in test_dataset.samples:
    test_labels.append(file_name[0].split('/')[-1].split('\\')[3].split('_')[0])
test_labels = np.array(test_labels)

unique_labels, label_indices = np.unique(test_labels, return_inverse=True) # Convert labels to unique
indices
label_mapping = dict(zip(range(len(unique_labels)), unique_labels)) # Create a mapping from label
indices to labels
test_dataset.targets = torch.tensor(label_indices) # Add label indices to the dataset
test_dataset.classes = unique_labels # Change training classes to unique labels
test_dataset.class_to_idx = label_mapping # Change class_to_idx dictionary to match new labels
test_dataset.idx_to_class = {v: k for k, v in label_mapping.items()} # Change idx_to_class dictionary
to match new labels
test_dataset.samples = [list(elem) for elem in test_dataset.samples]
for i in range(len(test_dataset.samples)):
    test_dataset.samples[i][1] = label_indices[i]

# Create Train adn Test data loaders
train_loader = DataLoader(train_dataset, batch_size=250, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=250, shuffle=True)


## **MODEL DEFINITION**
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 8 * 8, 128)  # Adjust the input size based on your image dimensions
        self.fc2 = nn.Linear(128, 9)

    def forward(self, x):
        x = self.pool(nn.functional.relu(self.conv1(x)))
        x = self.pool(nn.functional.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8)  # Adjust the size based on your image dimensions
        x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = Net()
model = nn.DataParallel(model) # Wrap model for Parallel processing

loss_function = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```python
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)

model.to(device) # moving model to cuda

train_losses = []
test_losses = []
train_accuracies = []
test_accuracies = []

for epoch in range(20):
    print(f'Starting epoch {epoch+1}')

    current_loss = 0.0 # Reset current loss to zero for each epoch

    # Model Training
    model.train()
    for i, data in enumerate(tqdm(train_loader)): # batch wise training
        inputs, targets = data
        inputs = inputs.to(device)
        targets = targets.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_function(outputs, targets)
        loss.backward()
        optimizer.step()

        current_loss += loss.item()
        if i % 500 == 499:
            print('Loss after mini-batch %5d: %.3f' % (i + 1, current_loss / 500))
            current_loss = 0.0

    # Calculating train accuracy
    correct_train = 0
    total_train = 0
    with torch.no_grad():
        model.eval()
        for data in train_loader:
            inputs, targets = data
            inputs = inputs.to(device)
            targets = targets.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total_train += targets.size(0)
            correct_train += (predicted == targets).sum().item()

    train_accuracy = correct_train / total_train
    print(f'Training Accuracy after epoch {epoch+1}: {100 * train_accuracy:.2f}%')

    # Calculating test accuracy
    correct_test = 0
    total_test = 0
    with torch.no_grad():
        model.eval()
        for data in test_loader:
            inputs, targets = data
            inputs = inputs.to(device)
            targets = targets.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total_test += targets.size(0)
            correct_test += (predicted == targets).sum().item()

    test_accuracy = correct_test / total_test
    print(f'Test Accuracy after epoch {epoch+1}: {100 * test_accuracy:.2f}%')

    train_losses.append(current_loss / 500)
    test_losses.append(loss_function(outputs, targets).item())
    train_accuracies.append(train_accuracy)
    test_accuracies.append(test_accuracy)

# Save the model to disk
torch.save(model.state_dict(), 'model.pth')
# Store the loss & accuracy to txt file
with open('train_losses.txt', 'w') as f:
    for item in train_losses:
        f.write("%s\n" % item)

with open('test_losses.txt', 'w') as f:
    for item in test_losses:
        f.write("%s\n" % item)

with open('train_accuracies.txt', 'w') as f:
    for item in train_accuracies:
        f.write("%s\n" % item)

with open('test_accuracies.txt', 'w') as f:
    for item in test_accuracies:
        f.write("%s\n" % item)


exit()
```

Python Code – Inference Module:

```python
# import libraries
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
import numpy as np

# Define the neural network class
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 8 * 8, 128)  # Adjust the input size based on your image dimensions
        self.fc2 = nn.Linear(128, 9)

    def forward(self, x):
        x = self.pool(nn.functional.relu(self.conv1(x)))
        x = self.pool(nn.functional.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8)  # Adjust the size based on your image dimensions
        x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# ask user for test input
print("Enter the path of the image you want to test:")
test_image_path = input()

# Define data transformations to apply to the images when loaded
transform = transforms.Compose([
    transforms.Resize((32, 32)),  # Resize images to a common size
    transforms.ToTensor(),  # Convert images to PyTorch tensors
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),  # Normalize pixel values
])

# create image dataset
test_image_dataset = ImageFolder(root=test_image_path, transform=transform)

# get labels from file names
test_image_labels = []
for file_name in test_image_dataset.samples:
    test_image_labels.append(file_name[0].split('/')[-1].split('\\')[3].split('_')[0])
test_image_labels = np.array(test_image_labels)
```

```
unique_labels, label_indices = np.unique(test_image_labels, return_inverse=True) # Convert labels to
unique indices
label_mapping = dict(zip(range(len(unique_labels)), unique_labels)) # Create a mapping from label
indices to labels
test_image_dataset.targets = torch.tensor(label_indices) # Add label indices to the dataset
test_image_dataset.classes = unique_labels # Change training classes to unique labels
test_image_dataset.class_to_idx = label_mapping # Change class_to_idx dictionary to match new labels
test_image_dataset.idx_to_class = {v: k for k, v in label_mapping.items()} # Change idx_to_class
dictionary to match new labels
test_image_dataset.samples = [list(elem) for elem in test_image_dataset.samples]
for i in range(len(test_image_dataset.samples)):
    test_image_dataset.samples[i][1] = label_indices[i]

# reassigned test_image_loader
test_image_loader = DataLoader(test_image_dataset, batch_size=250, shuffle=True)

# load the model from disk
model = Net()
model.load_state_dict(torch.load('0602-668913771-Hashmi.pth'))

# Define the device for training
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)

# Move the model to the device specified above
model.to(device)

# test the model on the test image
correct_test_image = 0
total_test_image = 0

with torch.no_grad():
    model.eval()
    for data in test_image_loader:
        inputs, targets = data
        inputs = inputs.to(device)
        targets = targets.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total_test_image += targets.size(0)
        correct_test_image += (predicted == targets).sum().item()

test_image_accuracy = correct_test_image / total_test_image
print(f'Test Image Accuracy: {100 * test_image_accuracy:.2f}%')

exit()
```

Training Output:

```
cuda:0
Starting epoch 1
100%|██████████| 288/288 [28:56<00:00,  6.03s/it]
Training Accuracy after epoch 1: 30.98%
Test Accuracy after epoch 1: 29.91%
Starting epoch 2
100%|██████████| 288/288 [22:27<00:00,  4.68s/it]
Training Accuracy after epoch 2: 39.37%
Test Accuracy after epoch 2: 38.73%
Starting epoch 3
100%|██████████| 288/288 [01:25<00:00,  3.37it/s]
Training Accuracy after epoch 3: 43.64%
Test Accuracy after epoch 3: 42.32%
Starting epoch 4
100%|██████████| 288/288 [01:13<00:00,  3.91it/s]
Training Accuracy after epoch 4: 47.06%
Test Accuracy after epoch 4: 45.37%
Starting epoch 5
100%|██████████| 288/288 [01:11<00:00,  4.02it/s]
Training Accuracy after epoch 5: 52.14%
Test Accuracy after epoch 5: 49.53%
Starting epoch 6
100%|██████████| 288/288 [01:11<00:00,  4.03it/s]
Training Accuracy after epoch 6: 57.00%
Test Accuracy after epoch 6: 54.10%
Starting epoch 7
100%|██████████| 288/288 [01:13<00:00,  3.93it/s]
Training Accuracy after epoch 7: 55.21%
Test Accuracy after epoch 7: 52.47%
Starting epoch 8
100%|██████████| 288/288 [01:13<00:00,  3.89it/s]
Training Accuracy after epoch 8: 61.95%
Test Accuracy after epoch 8: 57.64%
Starting epoch 9
100%|██████████| 288/288 [01:13<00:00,  3.94it/s]
Training Accuracy after epoch 9: 63.63%
Test Accuracy after epoch 9: 59.39%
Starting epoch 10
100%|██████████| 288/288 [25:03<00:00,  5.22s/it]
Training Accuracy after epoch 10: 65.50%
Test Accuracy after epoch 10: 60.99%
Starting epoch 11
100%|██████████| 288/288 [24:21<00:00,  5.07s/it]
Training Accuracy after epoch 11: 64.12%
Test Accuracy after epoch 11: 59.77%
Starting epoch 12
100%|██████████| 288/288 [24:44<00:00,  5.16s/it]
Training Accuracy after epoch 12: 65.48%
Test Accuracy after epoch 12: 61.01%
Starting epoch 13
100%|██████████| 288/288 [20:13<00:00,  4.21s/it]
Training Accuracy after epoch 13: 61.39%
Test Accuracy after epoch 13: 57.58%
```

```
Starting epoch 14
100%|██████████| 288/288 [09:37<00:00,  2.01s/it]
Training Accuracy after epoch 14: 67.38%
Test Accuracy after epoch 14: 62.32%
Starting epoch 15
100%|██████████| 288/288 [24:20<00:00,  5.07s/it]
Training Accuracy after epoch 15: 67.10%
Test Accuracy after epoch 15: 62.57%
Starting epoch 16
100%|██████████| 288/288 [31:52<00:00,  6.64s/it]
Training Accuracy after epoch 16: 69.17%
Test Accuracy after epoch 16: 63.82%
Starting epoch 17
100%|██████████| 288/288 [23:44<00:00,  4.94s/it]
Training Accuracy after epoch 17: 69.27%
Test Accuracy after epoch 17: 63.62%
Starting epoch 18
100%|██████████| 288/288 [29:10<00:00,  6.08s/it]
Training Accuracy after epoch 18: 69.78%
Test Accuracy after epoch 18: 64.71%
Starting epoch 19
100%|██████████| 288/288 [34:55<00:00,  7.28s/it]
Training Accuracy after epoch 19: 70.60%
Test Accuracy after epoch 19: 64.52%
Starting epoch 20
100%|██████████| 288/288 [28:49<00:00,  6.00s/it]
Training Accuracy after epoch 20: 71.95%
Test Accuracy after epoch 20: 66.10%
```