Neural Networks Assignment 5

Design Process:

All inputs were resized to 128x128 and are loaded using a custom class which inherits PyTorch Dataset class to read the dataset from the folder without changing the folder and gives the class name. The data has been split into training set and testing set using scikit learn train_test_split() method to get the indices and these indices have been put in the torch.utils.data.Subset() method.

I initially started with a relatively simple neural network. It had one convolution layer with 3 input channels, 16 output channels and a kernel size of 3 with padding 1. This output then goes to another convolution layer which takes 16 channels as input and gives out 32 channels with kernel size 3. It also has a padding of 1. The output of each of these convolution layers is sent through a ReLu activation function. In between each of these convolution layers, there is a max pooling operation with a kernel size of 2x2 and stride 2. This final output is connected to a fully connected layer with 512 neurons. This FC layer is connected to an output layer which has 9 outputs. Using Stochastic Gradient Descent and Cross Entropy Loss as Loss function, I trained the model for 15 epochs. As training went on, the loss and accuracy on the training and testing set kept getting better but after a few epochs, the loss and accuracy on testing data was not getting any better. I tried other optimizer methods like RMSprop and Adam, and using these with a learning rate scheduler did not make any difference. Here are the graphs for this simple model:

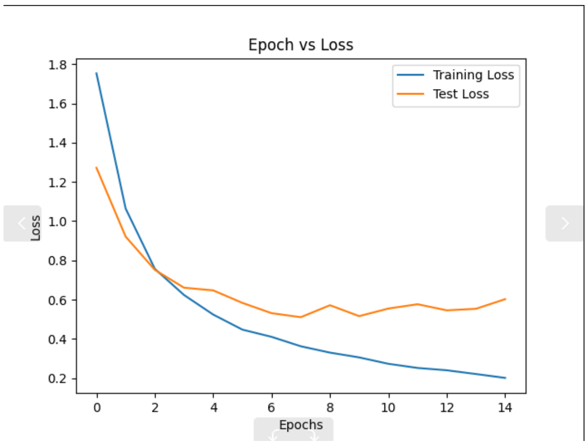

Figure 1: Epoch vs Loss for first model

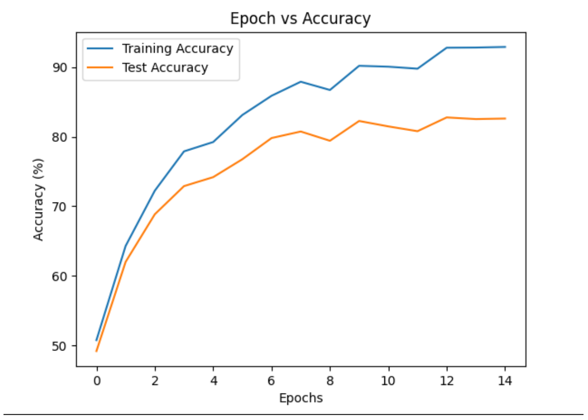

Figure 2: Epoch vs Accuracy for first model

I then decided to increase the number of layers and increase the number of parameters in the neural network. This network was much larger than the previous one. It had a convolution layer as input as the previous model but with 3 input channels, 32 output channels, kernel size 1 and padding 1. The next layer also had a convolution layer with double the output channels. This model has a total of 4 convolution layers each doubling the output channels. Each of these have the same maxpool operations in between as before and have ReLu activations in between. These convolution layers are also Batch Normalized. The final outputs are sent to a fully connected layer of 1024 neurons. There are 2 more fully connected layers with 512 and 256 neurons each and the final output layer has 9 outputs. The first three of the FC layers are also dropout layers.

The model took around 50 seconds to train per epoch on my Nvidia Geforce RTX 3070 laptop GPU and it ran for 15 epochs, using the same Cross Entropy Loss function and Adam optimizer with a scheduled learning rate. The accuracy and loss have kept getting better per epoch and accuracy was over 90 percent at the end of 15 epochs. The model was not overfitting and was performing quite well. But the disadvantage was that its size was almost 70 MB. Here are the graphs for this model:

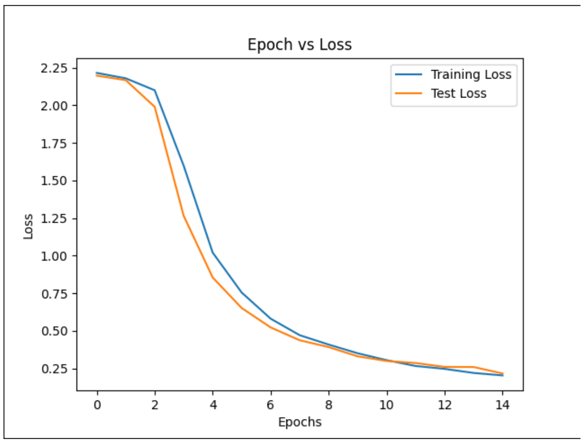

Figure 3: Epoch vs Loss for Large model

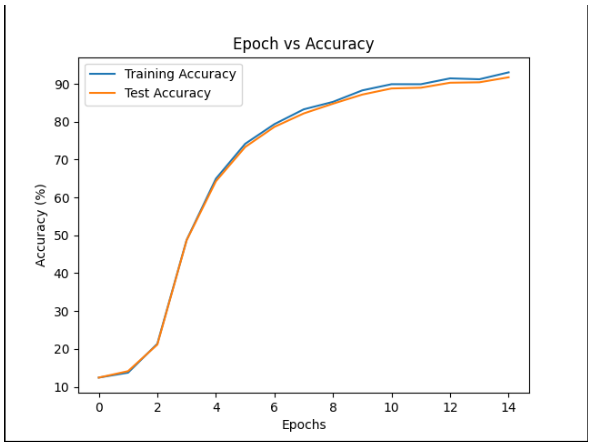

Figure 4: Epoch vs Accuracy for Large Model

Since the previous model was too large, I had to make a new model which was much smaller than this. This model has 3 convolution layers. The input convolution layer had 3 input channels, 16 output channels and a kernel size of 5. The other convolution layers double the channels and have the same kernel sizes. Between these convolution layers, there are maxpool layers with a kernel size of 2 and stride of 2. The output from these is connected to a fully connected layer with 256 neurons. This is connected to another layer with 128 neurons which is finally connected to the output layer with 9 outputs.

This model has exceptional performance and takes around 30 seconds which is much faster than the previous model. The saved model is also very small in size at around 9.8 MB. Using RMSprop and Cross Entropy Loss, training for 15 epochs gives a final accuracy of 98% over the entire dataset. Here are the final graphs:

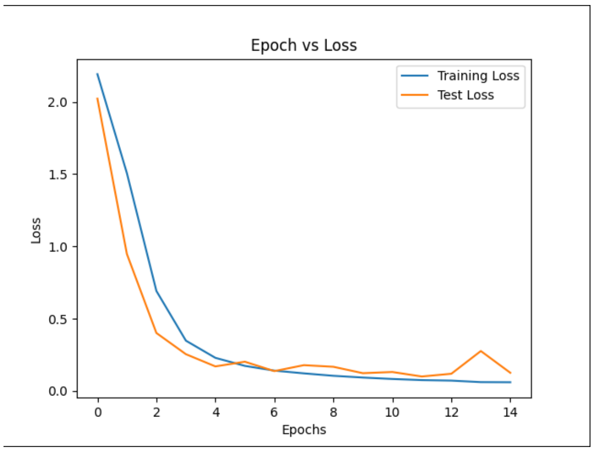

Figure 5: Epoch vs Loss of Final Model

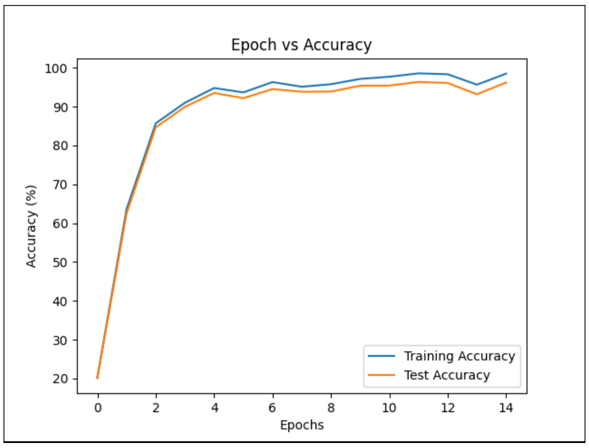

Figure 6: Epoch vs Accuracy of Final Model

Final Codes:

0601-671423599-VAKA.py –

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torch.nn.functional as F
import matplotlib.pyplot as plt
import time
from PIL import Image
import os
from PIL import Image
import torch
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms
from sklearn.model_selection import train_test_split

classes = []

class CustomImageDataset(Dataset):
    def __init__(self, directory, transform=None):
        self.directory = directory
        self.image_paths = [os.path.join(directory, fname) for fname in
os.listdir(directory) if os.path.isfile(os.path.join(directory, fname))]
        self.transform = transform

        # Assuming the filenames are of the format "classname_XXXX.jpg"
        # Extract unique class names and create a mapping to integer labels
        self.class_names = sorted({fname.split('_')[0] for fname in
os.listdir(directory)})
        self.class_to_idx = {cls_name: i for i, cls_name in
enumerate(self.class_names)}

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        img = Image.open(img_path).convert('RGB')  # Convert grayscale images
to RGB

        # Extract class name from the filename and convert to an integer label
        class_name = os.path.basename(img_path).split('_')[0]
        label = self.class_to_idx[class_name]

        if self.transform:
            img = self.transform(img)
```

```python
        return img, label

def show_images_with_labels(loader, num_images=5):
    dataiter = iter(loader)
    images, labels = next(dataiter)
    images = images.to(device)
    outputs = net(images)
    _, predicted = torch.max(outputs, 1)

    fig, axs = plt.subplots(1, num_images, figsize=(12, 2))
    for i in range(num_images):
        image = images[i].cpu() / 2 + 0.5  # Unnormalize the image
        image = image.numpy().transpose((1, 2, 0))
        label = classes[labels[i]]
        pred_label = classes[predicted[i]]

        axs[i].imshow(image)
        axs[i].set_title(f'True: {label}\nPredicted: {pred_label}')
        axs[i].axis('off')
    plt.show()

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
#device = 'cpu'
#Define the neural network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 5)
        self.conv3 = nn.Conv2d(32, 64, 5)
        self.fc1 = nn.Linear(64 * 12 * 12, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 9)
        # self.bn1 = nn.BatchNorm2d(16)
        # self.bn2 = nn.BatchNorm2d(32)
        # self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        # x = self.pool(F.relu(self.bn1(self.conv1(x))))
        # x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 12 * 12)
        x = F.relu(self.fc1(x))
        #x = self.dropout(x)
        x = F.relu(self.fc2(x))
        #x = self.dropout(x)
        x = self.fc3(x)
        return x

    # def __init__(self, num_classes=9):
    #     super(Net, self).__init__()
    #     self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
    #     self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
    #     self.pool = nn.MaxPool2d(2, 2)
    #     self.fc1 = nn.Linear(32 * 32 * 32, 512)  # After two poolings,
128x128 becomes 32x32
    #     self.fc2 = nn.Linear(512, num_classes)
    #     self.dropout = nn.Dropout(0.25)

    # def forward(self, x):
    #     x = self.pool(F.relu(self.conv1(x)))
    #     x = self.pool(F.relu(self.conv2(x)))
    #     x = x.view(-1, 32 * 32 * 32)
    #     x = F.relu(self.fc1(x))
    #     x = self.dropout(x)
    #     x = self.fc2(x)
    #     return x

    # def __init__(self):
    #     super(Net, self).__init__()
    #     self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
    #     self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
    #     self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
    #     #self.conv4 = nn.Conv2d(128, 256, 3, padding=1)
    #     self.pool = nn.MaxPool2d(2, 2)
    #     #self.fc1 = nn.Linear(256 * 8 * 8, 1024)
    #     self.fc1 = nn.Linear(128 * 16 * 16, 512)
    #     #self.fc2 = nn.Linear(1024, 512)
    #     self.fc2 = nn.Linear(512, 256)
    #     self.fc3 = nn.Linear(256, 9)
    #     self.dropout = nn.Dropout(0.5)
    #     self.batch_norm1 = nn.BatchNorm2d(32)
    #     self.batch_norm2 = nn.BatchNorm2d(64)
    #     self.batch_norm3 = nn.BatchNorm2d(128)
    #     #self.batch_norm4 = nn.BatchNorm2d(256)

    # def forward(self, x):
    #     x = self.pool(F.relu(self.batch_norm1(self.conv1(x))))
    #     x = self.pool(F.relu(self.batch_norm2(self.conv2(x))))
    #     x = self.pool(F.relu(self.batch_norm3(self.conv3(x))))
    #     #x = self.pool(F.relu(self.batch_norm4(self.conv4(x))))
    #     #x = x.view(-1, 256 * 8 * 8)
    #     x = x.view(-1, 128 * 16 * 16)
    #     x = F.relu(self.fc1(x))
    #     x = self.dropout(x)
    #     x = F.relu(self.fc2(x))
    #     x = self.dropout(x)
    #     # x = F.relu(self.fc3(x))
    #     # x = self.dropout(x)
    #     x = self.fc3(x)
    #     return x

def calculate_accuracy(loader):
    correct = 0
    total = 0
    with torch.no_grad():
        for data in loader:
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return (100 * correct / total)

if __name__=='__main__':
    net = Net()

    net.to(device)

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

    #
print(np.shape(np.array(Image.open('/home/vishalvaka/testing/geometry_dataset/
output/Circle_0a0b51ca-2a86-11ea-8123-8363a7ec19e6.png'))))
    # train_data = np.random.rand(8000, 200, 200, 3)

    # Define transformations (you can modify as needed)
    # data_transforms = transforms.Compose([
    # transforms.RandomHorizontalFlip(),
    # transforms.RandomRotation(10),
    # transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
    # transforms.Resize((128, 128)),
    # transforms.ToTensor()])

    data_transforms = transforms.Compose([
        transforms.Resize((128, 128)),
        transforms.ToTensor()
    ])
```

```python
    # Initialize the dataset and dataloader
    dataset = CustomImageDataset(os.path.dirname(os.path.abspath(__file__)) +
'/geometry_dataset/output', transform=data_transforms)
    #dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
    #print(dataloader)
    # Example: iterate through the DataLoader
    # for images, labels in dataloader:
    #     print(labels)
    #     class_names = [dataset.class_names[label] for label in labels]
    #     print(class_names)

    # train_len = int(0.8 * len(dataset))
    # test_len = len(dataset) - train_len

    # train_dataset, test_dataset = random_split(dataset, [train_len,
test_len])

    labels = [dataset[i][1] for i in range(len(dataset))]

    # Create a stratified split
    train_indices, test_indices, _, _ = train_test_split(
        list(range(len(dataset))),
        labels,
        test_size=0.2,
        stratify=labels
    )

    train_dataset = torch.utils.data.Subset(dataset, train_indices)
    test_dataset = torch.utils.data.Subset(dataset, test_indices)

    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True,
pin_memory=True, num_workers = 8)
    test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False,
pin_memory=True, num_workers = 8)

    criterion = nn.CrossEntropyLoss()
    #optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
    #optimizer = optim.Adam(net.parameters(), lr=0.001)
    optimizer = optim.RMSprop(net.parameters(), lr = 0.001, alpha=0.99,
eps=1e-08, weight_decay=0, momentum=0, centered=False)
    classes = dataset.class_names
    train_losses = []
    test_losses = []
    train_accuracies = []
    test_accuracies = []

    for epoch in range(15):  # Change the number of epochs as needed
```

```
        star_time = time.time()
        running_loss = 0.0
        for i, data in enumerate(train_loader, 0):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()

            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            # scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,
'min', patience=5, factor=0.1)
            # scheduler.step(loss)

            running_loss += loss.item()

        with torch.no_grad():
            test_loss = 0.0
            for data in test_loader:
                inputs, labels = data
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = net(inputs)
                loss_val = criterion(outputs, labels)
                test_loss += loss_val.item()


        train_accuracy = calculate_accuracy(train_loader)
        test_accuracy = calculate_accuracy(test_loader)
        end_time = time.time()
        print(f'Epoch {epoch + 1}, Loss: {running_loss / len(train_loader)},
Train Accuracy: {train_accuracy}%, Test Accuracy: {test_accuracy}%, Time
Taken: {end_time - star_time}')
        train_losses.append(running_loss / len(train_loader))
        test_losses.append(test_loss / len(test_loader))
        train_accuracies.append(train_accuracy)
        test_accuracies.append(test_accuracy)


    print('Finished Training')
    torch.save(net.state_dict(), '0602-671423599-VAKA.pth')

    # Plot training and test loss
    plt.figure()
    plt.plot(train_losses, label='Training Loss')
    plt.plot(test_losses, label='Test Loss')
    plt.title('Epoch vs Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

    # Plot training and test accuracy
    plt.figure()
    plt.plot(train_accuracies, label='Training Accuracy')
    plt.plot(test_accuracies, label='Test Accuracy')
    plt.title('Epoch vs Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.legend()
    plt.show()


    show_images_with_labels(test_loader, num_images=5)
```

0603-671423599-VAKA.py –

```
import torch
import torch.nn as nn
from torchvision import transforms
from PIL import Image
import os
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset

class InferenceDataset(Dataset):
    def __init__(self, directory, transform=None):
        self.directory = directory
        self.image_paths = [os.path.join(directory, fname) for fname in
os.listdir(directory) if os.path.isfile(os.path.join(directory, fname))]
        self.transform = transform

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        img = Image.open(img_path).convert('RGB')  # Convert grayscale images
to RGB

        if self.transform:
            img = self.transform(img)

        return img, img_path

# class CustomImageDataset(Dataset):
```

```
#     def __init__(self, directory, transform=None):
#         self.directory = directory
#         self.image_paths = [os.path.join(directory, fname) for fname in 
os.listdir(directory) if os.path.isfile(os.path.join(directory, fname))]
#         self.transform = transform

#         # Assuming the filenames are of the format "classname_XXXX.jpg"
#         # Extract unique class names and create a mapping to integer labels
#         self.class_names = sorted({fname.split('_')[0] for fname in 
os.listdir(directory)})
#         self.class_to_idx = {cls_name: i for i, cls_name in 
enumerate(self.class_names)}

#     def __len__(self):
#         return len(self.image_paths)

#     def __getitem__(self, idx):
#         img_path = self.image_paths[idx]
#         img = Image.open(img_path).convert('RGB')  # Convert grayscale 
images to RGB

#         # Extract class name from the filename and convert to an integer 
label
#         class_name = os.path.basename(img_path).split('_')[0]
#         label = self.class_to_idx[class_name]

#         if self.transform:
#             img = self.transform(img)

#         return img, label
# Define the Net class just like you did in your training script
class Net(nn.Module):
    # ... [same as your definition above]
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 5)
        self.conv3 = nn.Conv2d(32, 64, 5)
        self.fc1 = nn.Linear(64 * 12 * 12, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 9)
        # self.bn1 = nn.BatchNorm2d(16)
        # self.bn2 = nn.BatchNorm2d(32)
        # self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        # x = self.pool(F.relu(self.bn1(self.conv1(x))))
```

```python
        # x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 12 * 12)
        x = F.relu(self.fc1(x))
        #x = self.dropout(x)
        x = F.relu(self.fc2(x))
        #x = self.dropout(x)
        x = self.fc3(x)
        return x

# Load the saved model
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net = Net().to(device)
net.load_state_dict(torch.load('0602-671423599-VAKA.pth'))
net.eval()

# Transformation for inference (same as you used in training)
data_transforms = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor()
])

def predict_image(image_path):
    image = Image.open(image_path).convert('RGB')
    image = data_transforms(image).unsqueeze(0).to(device)
    outputs = net(image)
    _, predicted = torch.max(outputs, 1)
    return classes[predicted[0]]

def get_true_label(filename, classes):
    for cls in classes:
        if cls in filename:
            return cls
    return None

if __name__ == '__main__':
    directory = os.path.dirname(os.path.abspath(__file__)) +
'/geometry_dataset/output'  # same directory as training data
    dataset = InferenceDataset(directory, transform=data_transforms)
    dataloader = DataLoader(dataset, batch_size=32, shuffle=False,
num_workers=8)
    classes = sorted({fname.split('_')[0] for fname in os.listdir(directory)})
    int = 0
    results = {}
    correct = 0
    total = 0
```

```python
with torch.no_grad():
    for images, paths in dataloader:
        images = images.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        labels = [classes[idx] for idx in predicted]
        for p, label in zip(paths, labels):
            true_label = get_true_label(os.path.basename(p), classes)
            if true_label == label:
                correct += 1
            total += 1
            results[p] = label

# Print results
for img_path, label in results.items():
    print(f"{os.path.basename(img_path)}: {label}")
# for img_path, label in results.items():
#     print(f"{os.path.basename(img_path)}: {label}")
#         if filename.endswith(".png"):
#             image_path = os.path.join(directory, filename)
#             prediction = predict_image(image_path)
#             print(f"{filename}: {prediction}")
#             int+=1
#             print(int)

accuracy = 100 * correct / total
print(f"Accuracy: {accuracy:.2f}%")
```