CS 419 PROJECT REPORT: Skribble Buddy

This project has been undertaken as a part of the CS-419 M course taught by Prof. Abir De (Spring Semester ‘23).

Objective:

To train a CNN in order to determine the class of a user drawn image out of the pre-trained N classes

Model Specifications:

Number of Classes = 40

Images per class = 10,000

Testing Accuracy = 85.1457% on 80,000 test images

Libraries Used:

The following libraries have been used:

torch

pygame

numpy

os

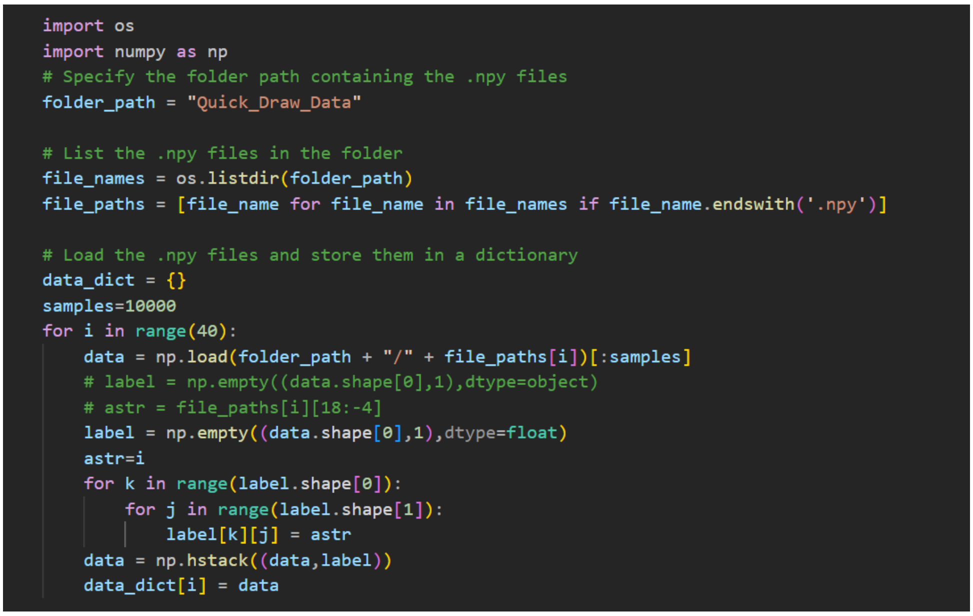
matplotlib

PIL

Skimage

Methodology:

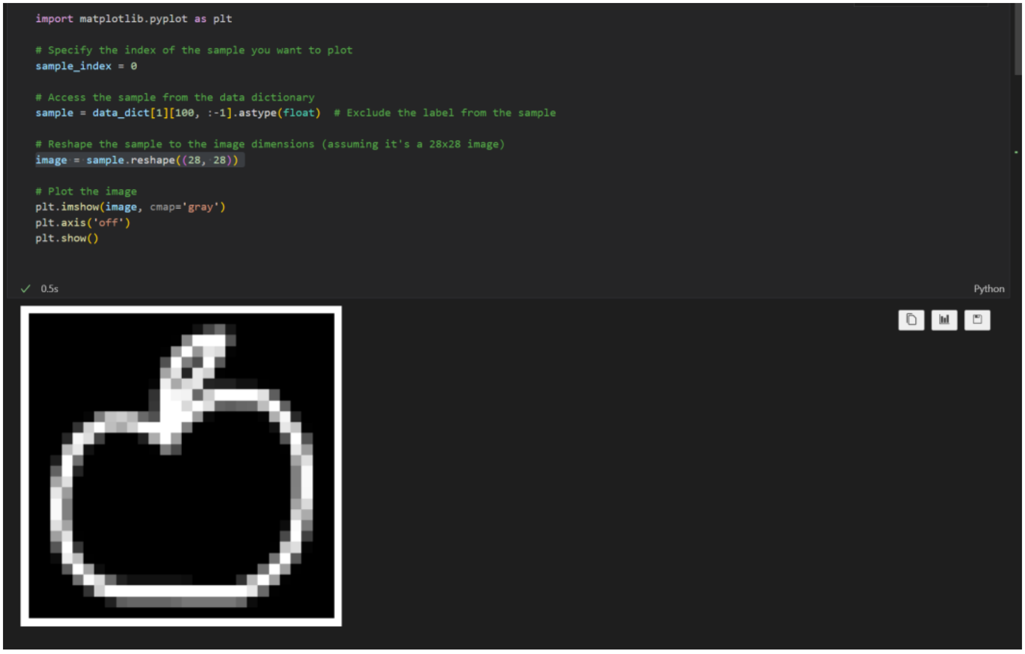
1) The data is loaded using os and numpy into a dictionary termed data_dict



```
import os
import numpy as np
# Specify the folder path containing the .npy files
folder_path = "Quick_Draw_Data"

# List the .npy files in the folder
file_names = os.listdir(folder_path)
file_paths = [file_name for file_name in file_names if file_name.endswith('.npy')]

# Load the .npy files and store them in a dictionary
data_dict = {}
samples=10000
for i in range(40):
    data = np.load(folder_path + "/" + file_paths[i])[:samples]
    # label = np.empty((data.shape[0],1),dtype=object)
    # astr = file_paths[i][18:-4]
    label = np.empty((data.shape[0],1),dtype=float)
    astr=i
    for k in range(label.shape[0]):
        for j in range(label.shape[1]):
            label[k][j] = astr
    data = np.hstack((data,label))
    data_dict[i] = data
```

2) A look up table is created since the labels are numeric in nature for simpler training

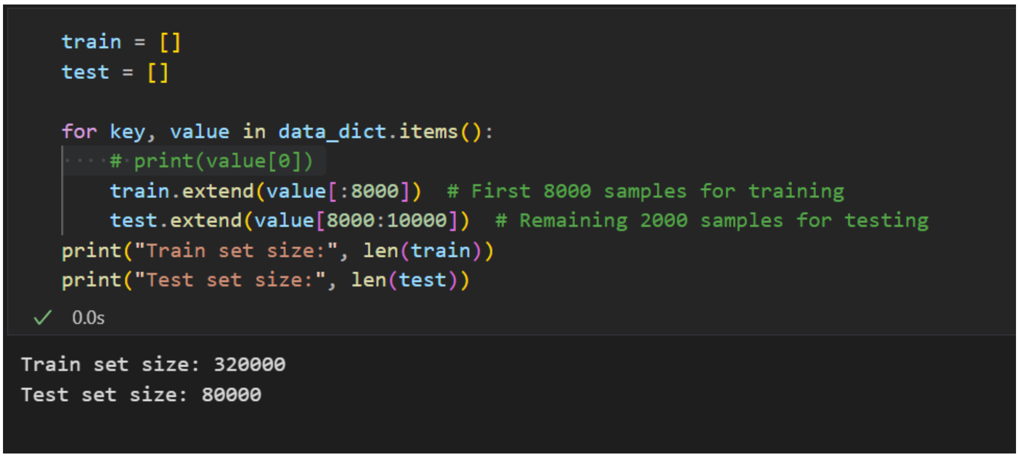
3) A sample of the data is shown



```
import matplotlib.pyplot as plt

# Specify the index of the sample you want to plot
sample_index = 0

# Access the sample from the data dictionary
sample = data_dict[1][100, :-1].astype(float)  # Exclude the label from the sample

# Reshape the sample to the image dimensions (assuming it's a 28x28 image)
image = sample.reshape((28, 28))

# Plot the image
plt.imshow(image, cmap='gray')
plt.axis('off')
plt.show()
```

✓ 0.5s

4) The data is split into training and testing in the ratio 4:1

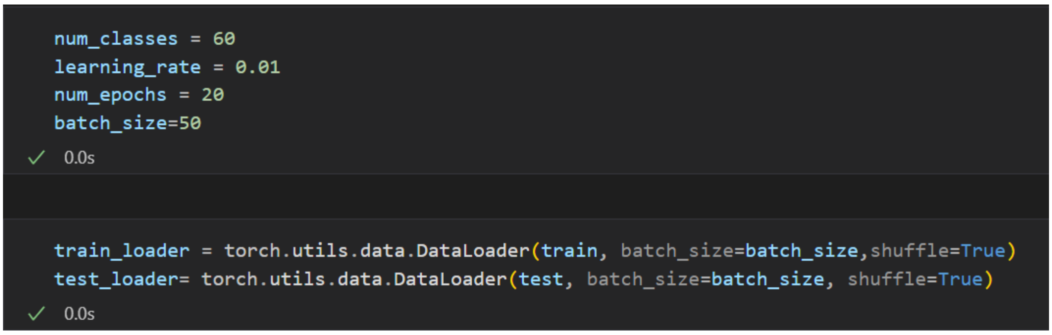


```
train = []
test = []

for key, value in data_dict.items():
    # print(value[0])
    train.extend(value[:8000])  # First 8000 samples for training
    test.extend(value[8000:10000])  # Remaining 2000 samples for testing
print("Train set size:", len(train))
print("Test set size:", len(test))
```

✓ 0.0s

```
Train set size: 320000
Test set size: 80000
```

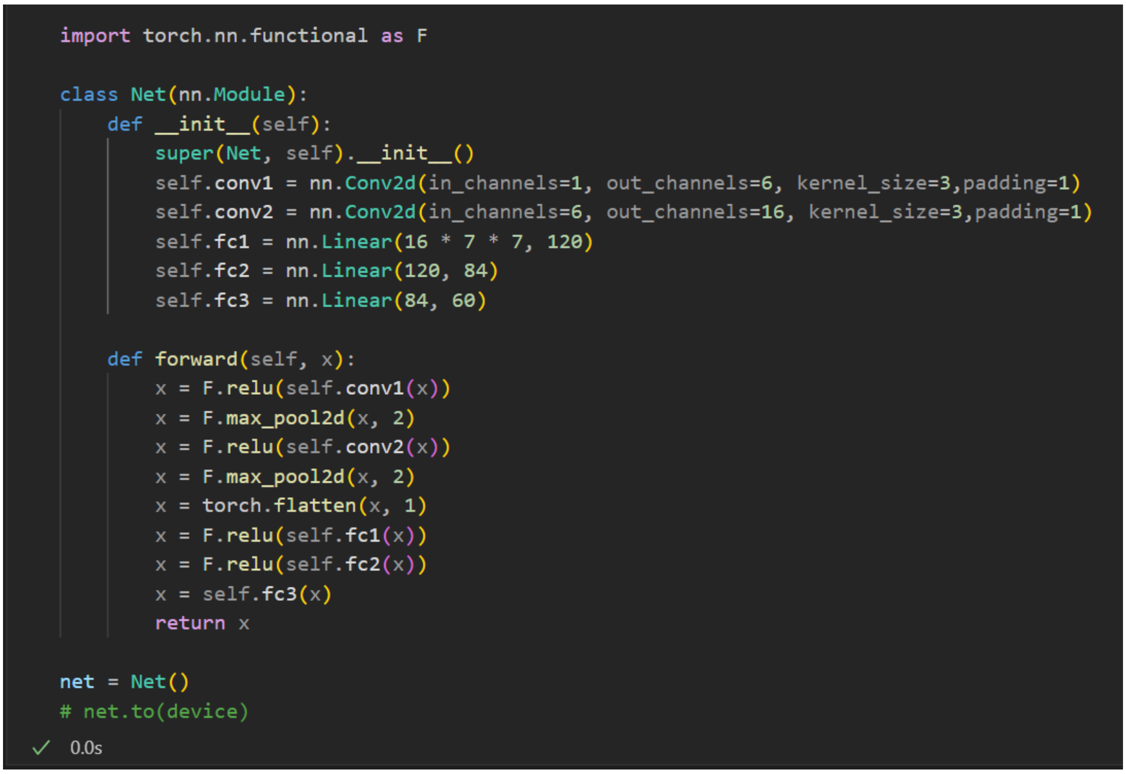
5) The Learning parameters are set and the train and test data is loaded into DataLoader type containers



```
num_classes = 60
learning_rate = 0.01
num_epochs = 20
batch_size=50
✓ 0.0s
```

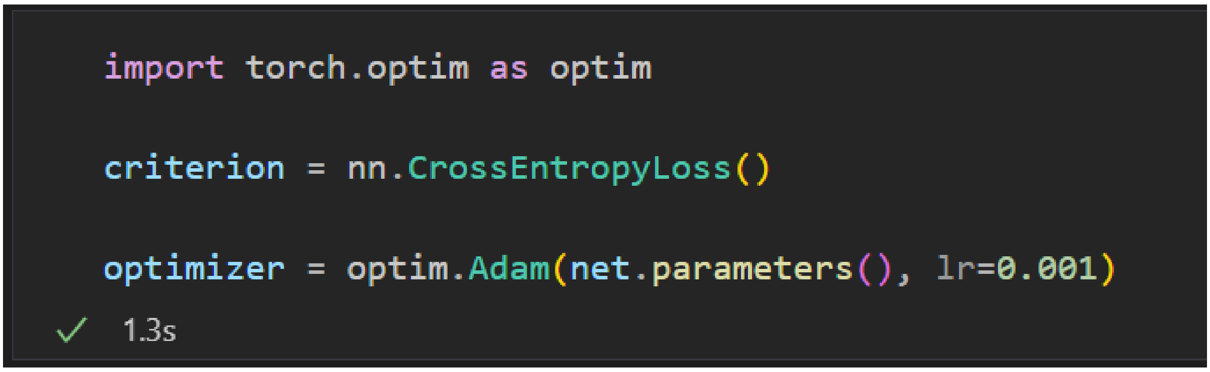
```
train_loader = torch.utils.data.DataLoader(train, batch_size=batch_size,shuffle=True)
test_loader= torch.utils.data.DataLoader(test, batch_size=batch_size, shuffle=True)
✓ 0.0s
```

6) The neural network architecture is specified



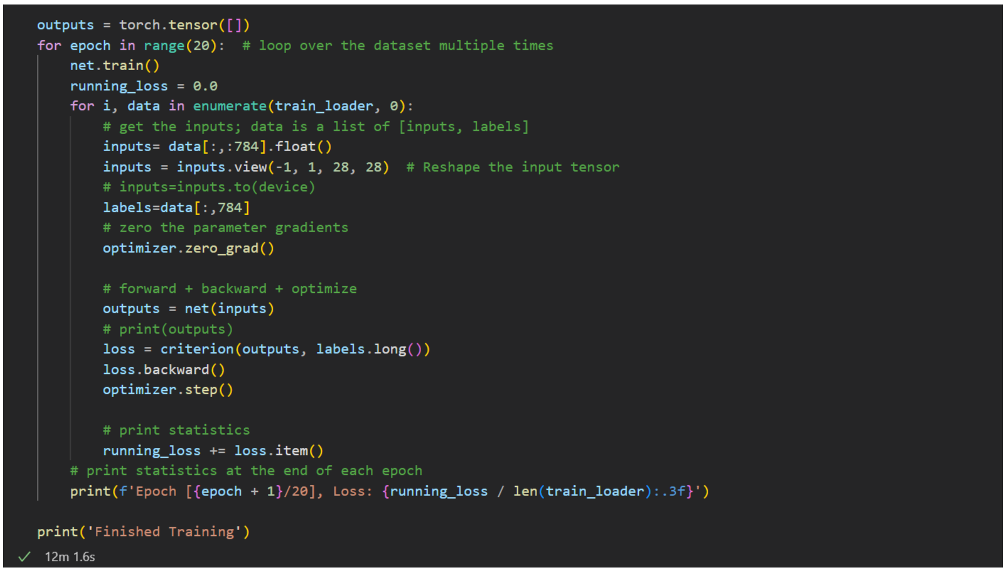
```
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=3,padding=1)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=3,padding=1)
        self.fc1 = nn.Linear(16 * 7 * 7, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 60)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()
# net.to(device)
✓ 0.0s
```

7) The loss function (cross entropy loss) and optimizer(Adam) are fixed



```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(net.parameters(), lr=0.001)
✓ 1.3s
```

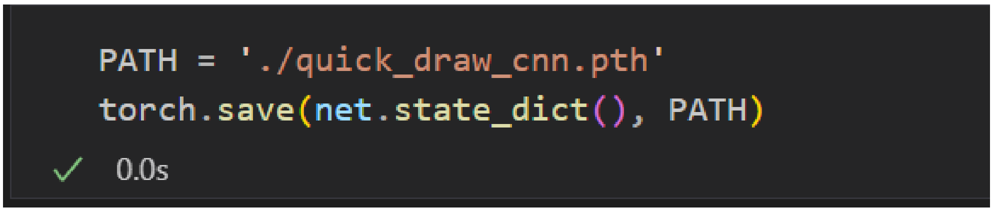
8) The data is trained for 20 epochs



```
outputs = torch.tensor([])
for epoch in range(20):  # loop over the dataset multiple times
    net.train()
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs= data[:,:784].float()
        inputs = inputs.view(-1, 1, 28, 28)  # Reshape the input tensor
        # inputs=inputs.to(device)
        labels=data[:,784]
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        # print(outputs)
        loss = criterion(outputs, labels.long())
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
    # print statistics at the end of each epoch
    print(f'Epoch [{epoch + 1}/20], Loss: {running_loss / len(train_loader):.3f}')

print('Finished Training')
```

✓ 12m 1.6s

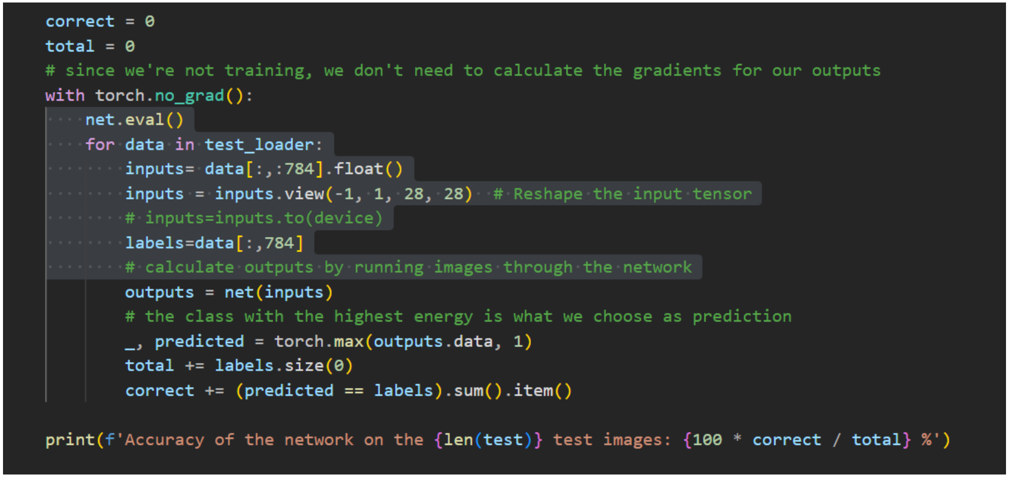
9) The trained model is saved for later use



```
PATH = './quick_draw_cnn.pth'
torch.save(net.state_dict(), PATH)
```

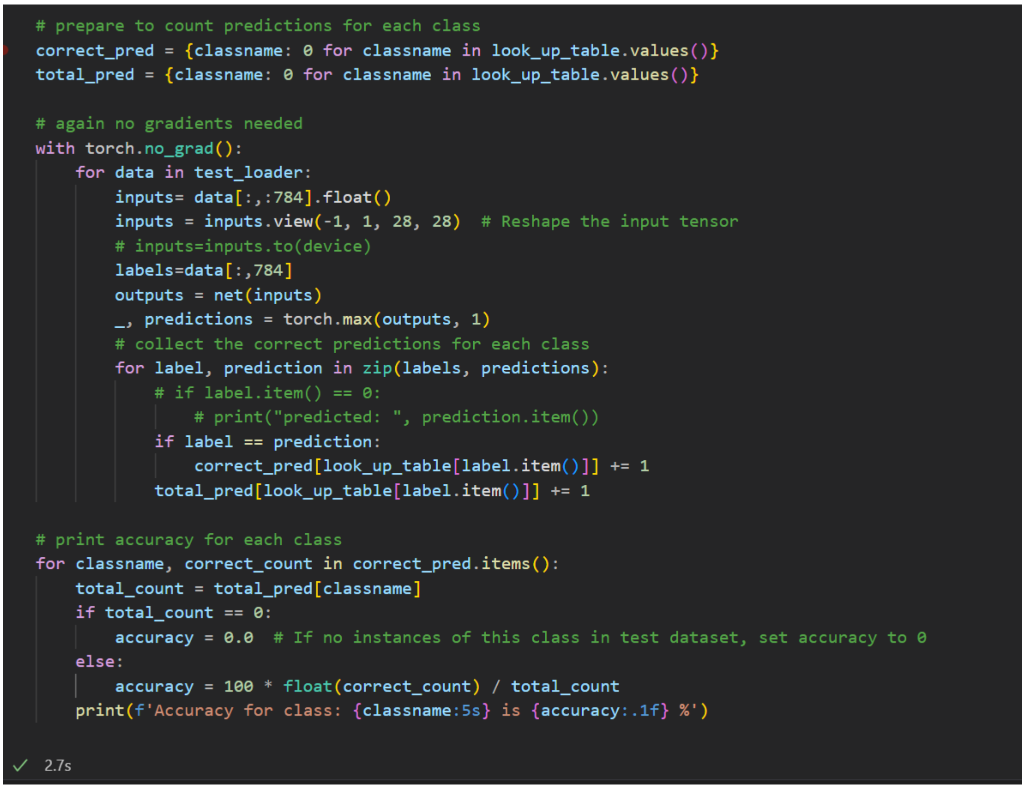
✓ 0.0s

10) Accuracy on the training data is determined



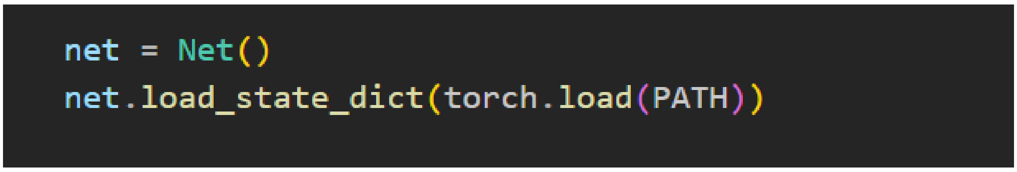
```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    net.eval()
    for data in test_loader:
        inputs= data[:,:784].float()
        inputs = inputs.view(-1, 1, 28, 28)  # Reshape the input tensor
        # inputs=inputs.to(device)
        labels=data[:,784]
        # calculate outputs by running images through the network
        outputs = net(inputs)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

11) Class-wise accuracy is determined



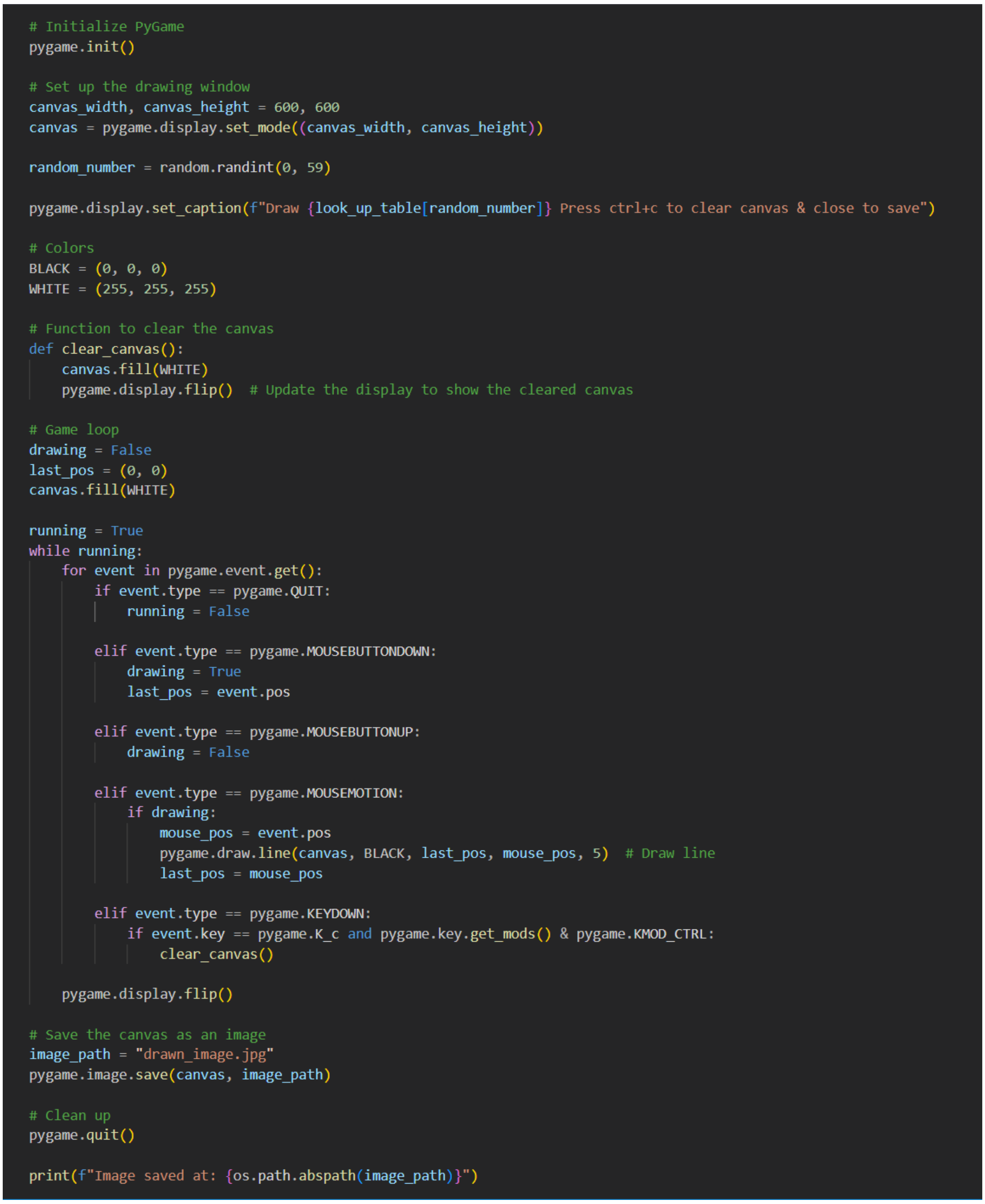
```python
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in look_up_table.values()}
total_pred = {classname: 0 for classname in look_up_table.values()}

# again no gradients needed
with torch.no_grad():
    for data in test_loader:
        inputs= data[:,:784].float()
        inputs = inputs.view(-1, 1, 28, 28)  # Reshape the input tensor
        # inputs=inputs.to(device)
        labels=data[:,784]
        outputs = net(inputs)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            # if label.item() == 0:
                # print("predicted: ", prediction.item())
            if label == prediction:
                correct_pred[look_up_table[label.item()]] += 1
            total_pred[look_up_table[label.item()]] += 1

# print accuracy for each class
for classname, correct_count in correct_pred.items():
    total_count = total_pred[classname]
    if total_count == 0:
        accuracy = 0.0  # If no instances of this class in test dataset, set accuracy to 0
    else:
        accuracy = 100 * float(correct_count) / total_count
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

12) CNN is loaded



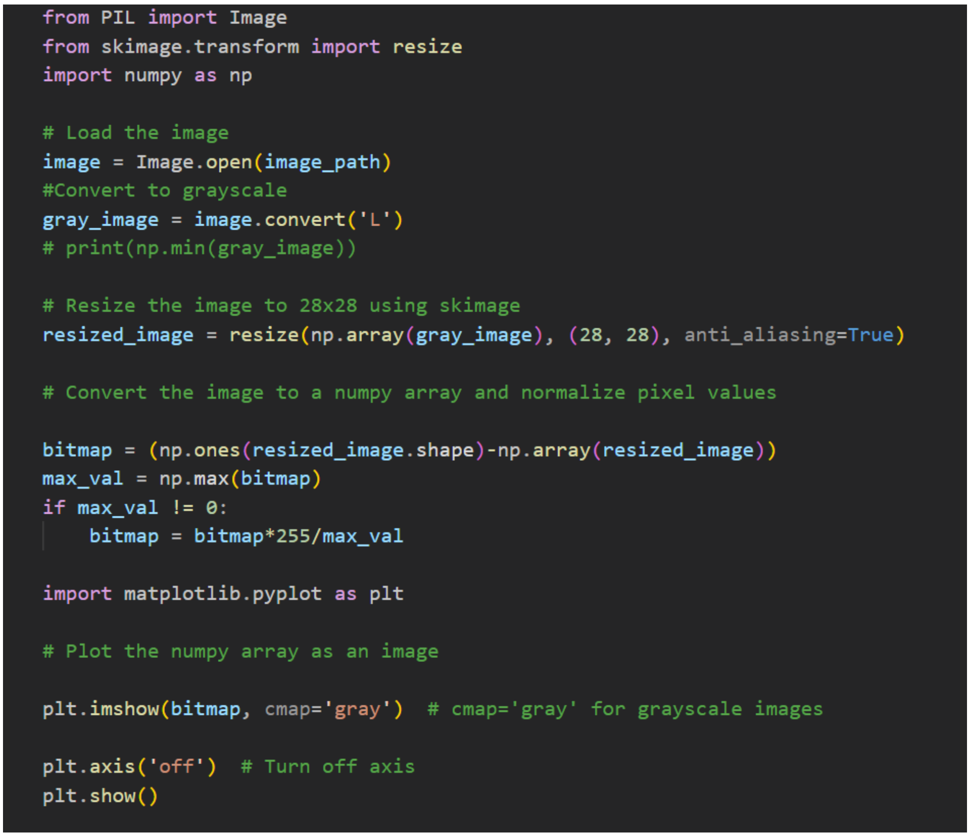
```python
net = Net()
net.load_state_dict(torch.load(PATH))
```

13) User is asked to give an input using pygame and the image is saved on closing the tab



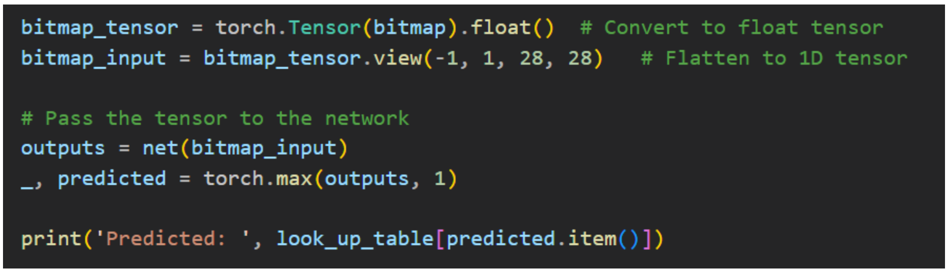
```
# Initialize PyGame
pygame.init()

# Set up the drawing window
canvas_width, canvas_height = 600, 600
canvas = pygame.display.set_mode((canvas_width, canvas_height))

random_number = random.randint(0, 59)

pygame.display.set_caption(f"Draw {look_up_table[random_number]} Press ctrl+c to clear canvas & close to save")

# Colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)

# Function to clear the canvas
def clear_canvas():
    canvas.fill(WHITE)
    pygame.display.flip()  # Update the display to show the cleared canvas

# Game loop
drawing = False
last_pos = (0, 0)
canvas.fill(WHITE)

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        elif event.type == pygame.MOUSEBUTTONDOWN:
            drawing = True
            last_pos = event.pos

        elif event.type == pygame.MOUSEBUTTONUP:
            drawing = False

        elif event.type == pygame.MOUSEMOTION:
            if drawing:
                mouse_pos = event.pos
                pygame.draw.line(canvas, BLACK, last_pos, mouse_pos, 5)  # Draw line
                last_pos = mouse_pos

        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_c and pygame.key.get_mods() & pygame.KMOD_CTRL:
                clear_canvas()

    pygame.display.flip()

# Save the canvas as an image
image_path = "drawn_image.jpg"
pygame.image.save(canvas, image_path)

# Clean up
pygame.quit()

print(f"Image saved at: {os.path.abspath(image_path)}")
```

14) The image is resized and brought to a similar format as that of the training data



```python
from PIL import Image
from skimage.transform import resize
import numpy as np

# Load the image
image = Image.open(image_path)
#Convert to grayscale
gray_image = image.convert('L')
# print(np.min(gray_image))

# Resize the image to 28x28 using skimage
resized_image = resize(np.array(gray_image), (28, 28), anti_aliasing=True)

# Convert the image to a numpy array and normalize pixel values

bitmap = (np.ones(resized_image.shape)-np.array(resized_image))
max_val = np.max(bitmap)
if max_val != 0:
    bitmap = bitmap*255/max_val

import matplotlib.pyplot as plt

# Plot the numpy array as an image

plt.imshow(bitmap, cmap='gray')  # cmap='gray' for grayscale images

plt.axis('off')  # Turn off axis
plt.show()
```

15) Final prediction is made



```python
bitmap_tensor = torch.Tensor(bitmap).float()  # Convert to float tensor
bitmap_input = bitmap_tensor.view(-1, 1, 28, 28)   # Flatten to 1D tensor

# Pass the tensor to the network
outputs = net(bitmap_input)
_, predicted = torch.max(outputs, 1)

print('Predicted: ', look_up_table[predicted.item()])
```

Extent of completion

The project is entirely complete