sugarcane

February 19, 2024

[29]:
```
#pip install --force-reinstall --no-deps torch==2.1.2
```

[1]:
```
#pip install torch --upgrade.
# Comment: earlier versions of my program did not work because there was an␣
 ↪error in Pytorch and this wasted days of my time
# So make sure the most updated version!

import torch                                          # imports the core PyTorch␣
 ↪library
import torch.nn as nn                                 # torch.nn library is a␣
 ↪high-level interface for building and training neural networks
import torch.optim as optim                           # Needed for optimizer
from torch.utils.data import DataLoader               # Needed for loading data␣
 ↪from Sugarcane png files
from torchvision import datasets, transforms          # Needed for making sure␣
 ↪data from  Sugarcane png files is properly formatted
import torch.nn.functional as F                       # Needed for functional␣
 ↪equation in forward loop
import numpy as np                                    # Needed for outputting␣
 ↪weights and biases
from torchvision.datasets import VisionDataset        # VisionDataset is is␣
 ↪designed to be a base class for datasets in computer vision tasks
from PIL import Image                                 # Needed for manipulating ␣
 ↪Sugarcane png image files
import os                                             # Needed for operating␣
 ↪system for dealing with dir of training and test png files
from torchvision import transforms
from skimage import exposure
import matplotlib.pyplot as plt

# Set the device
device = "mps" if torch.backends.mps.is_available() else "cpu"
# Check PyTorch has access to MPS (Metal Performance Shader, Apple's GPU␣
 ↪architecture)
print(f"Is MPS (Metal Performance Shader) built? {torch.backends.mps.
 ↪is_built()}")
```

```
Is MPS (Metal Performance Shader) built? True
```

[2]:
```python
class Sugarcane(VisionDataset):
    def __init__(self, root, train=True, transform=None, target_transform=None):
        super(Sugarcane, self).__init__(root, transform=transform,␣
 ↪target_transform=target_transform)

        self.train = train
        self.data_folder = "training" if train else "testing"
        self.images_folder = os.path.join(self.root, self.data_folder)

        self.image_paths = self._get_image_paths()

    def _get_image_paths(self):
        image_paths = []
        for digit_folder in os.listdir(self.images_folder):
            digit_folder_path = os.path.join(self.images_folder, digit_folder)

            # Check if it's a directory before listing its contents
            if os.path.isdir(digit_folder_path):
                for image_name in os.listdir(digit_folder_path):
                    image_path = os.path.join(digit_folder_path, image_name)
                    image_paths.append((image_path, int(digit_folder)))

        return image_paths

    def __getitem__(self, index):
        image_path, label = self.image_paths[index]

        # Skip files with the '.DS_Store' extension
        if image_path.endswith('.DS_Store'):
            return self.__getitem__(index + 1)

        image = Image.open(image_path)

        # Check if the image has an alpha channel (4 channels)
        if image.mode == 'RGBA':
            image = image.convert('RGB')

        if self.transform is not None:
            image = self.transform(image)

        return image, label

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

[3]:
```
transform = transforms.Compose([
    transforms.Resize((244, 244)),
    transforms.ToTensor()])
# this considers the colors of the image
# Assuming 'image' is your RGB image tensor (H x W x C)
#transformed_image = transform(image)
```

[ ]:
```
```

[4]:
```
# Split the dataset into training and validation sets
# Download the Sugarcane PNG files to your Desktop from this site (kaggle)

Sugarcane_dataset = Sugarcane(root='/Users/peternoble/Downloads/
 ↪Sugarcane_true', train=True, transform=transform)
#Sugarcane_dataset = Sugarcane(root='/Users/peternoble/Desktop/Sugarcane',␣
 ↪train=True, transform=transform)
train_size = int(0.7 * len(Sugarcane_dataset))
val_size = len(Sugarcane_dataset) - train_size
train_dataset, val_dataset = torch.utils.data.random_split(Sugarcane_dataset,␣
 ↪[train_size, val_size])
```

[5]:
```
# Create data loaders
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32,␣
 ↪shuffle=True)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=32,␣
 ↪shuffle=False)
```

[6]:
```
inputs, labels = next(iter(train_loader))
print(inputs.shape)
```

```
torch.Size([32, 3, 244, 244])
```

[7]:
```
# Print the size of the training and validation datasets
print(f'Training dataset size: {len(train_loader.dataset)}')
print(f'Validation dataset size: {len(val_loader.dataset)}')
```

```
Training dataset size: 1765
Validation dataset size: 757
```

[8]:
```
# Set the path where you want to save the images. Useful for visualizing the␣
 ↪actual images used for validation
output_path = '/Users/peternoble/Desktop/results'
# Ensure the output directory exists
os.makedirs(output_path, exist_ok=True)
# Variables to store all images and labels
all_images = []
all_labels = []
```

```
# Iterate through the validation loader and concatenate images and labels
for batch_idx, (images, labels) in enumerate(val_loader):
# Apply histogram equalization to each image in the batch
#     equalized_images = torch.stack([torch.tensor(exposure.equalize_hist(img.
 ↪numpy())) for img in images])

#     all_images.append(equalized_images)
    all_images.append(images)
    all_labels.append(labels)

# Concatenate the lists to get all images and labels
all_images = torch.cat(all_images, dim=0)
all_labels = torch.cat(all_labels, dim=0)

# Define a function to save images
def save_images(images, labels, output_path):
    for i in range(len(images)):
        image, label = images[i], labels[i]
        # Assuming the images are in the range [0, 1], and using torchvision to␣
 ↪convert to PIL
        image_pil = transforms.ToPILImage()(image)
        image_pil.save(os.path.join(output_path, f"image_{i}_label_{label}.
 ↪png"))

# Save all equalized images from the validation dataset
save_images(all_images, all_labels, output_path)
```

[9]:
```
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=2,␣
 ↪stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten = nn.Flatten()
        # Adjust the input size of the fully connected layer based on the␣
 ↪output size of the previous layers
        self.fc1 = nn.Linear(32 * 122 * 122, 5)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.flatten(x)
        x = self.fc1(x)
        return x
```

[10]:
```
# Initialize the model
model = SimpleModel()
```

```
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

[11]:
```
# Assuming dataset is your DataLoader
for inputs, labels in Sugarcane_dataset:
    print("Input shape:", inputs.shape)
    break
```

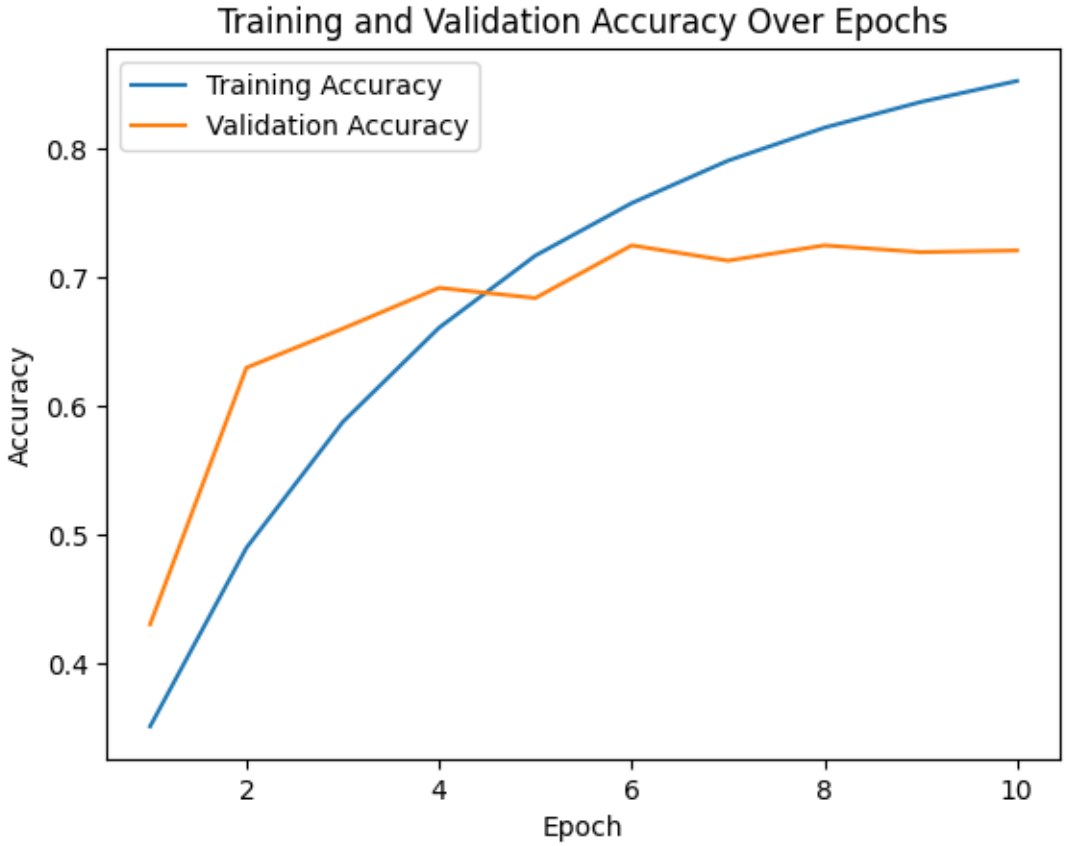
```
Input shape: torch.Size([3, 244, 244])
```

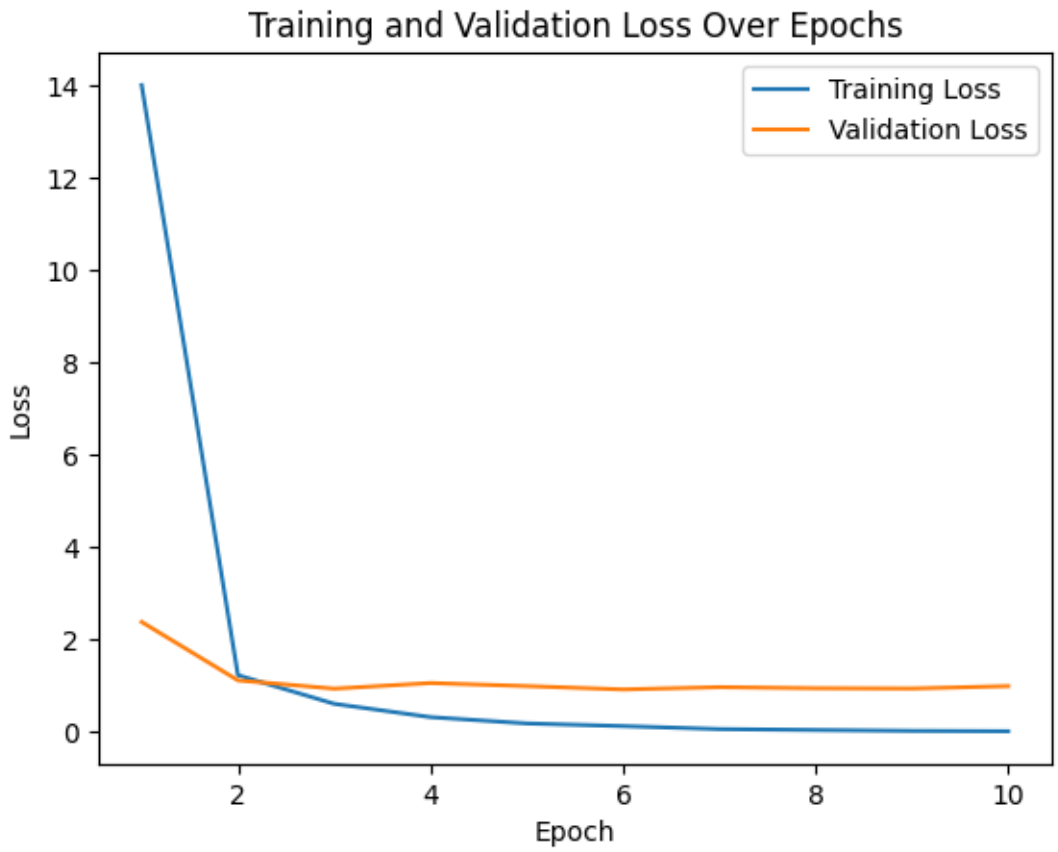
[12]:
```
num_epochs = 10
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []
correct_train = 0
total_train = 0
correct_val = 0
total_val = 0

for epoch in range(num_epochs):
    model.train()
    running_train_loss = 0.0

    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()

        _, predicted_train = outputs.max(1)
        total_train += labels.size(0)
        correct_train += predicted_train.eq(labels).sum().item()

    average_train_loss = running_train_loss / len(train_loader)
    train_losses.append(average_train_loss)
    train_accuracy = correct_train / total_train
    train_accuracies.append(train_accuracy)

    model.eval()
    running_val_loss = 0.0

    # Reset variables for accuracy calculation
    correct_val = 0
    total_val = 0
```

```
    with torch.no_grad():
        for val_inputs, val_labels in val_loader:
            val_outputs = model(val_inputs)
            val_loss = criterion(val_outputs, val_labels)
            running_val_loss += val_loss.item()

            _, predicted_val = val_outputs.max(1)
            total_val += val_labels.size(0)
            correct_val += predicted_val.eq(val_labels).sum().item()

    average_val_loss = running_val_loss / len(val_loader)
    val_losses.append(average_val_loss)
    val_accuracy = correct_val / total_val
    val_accuracies.append(val_accuracy)

    print(f"Epoch {epoch + 1}/{num_epochs}, Training Loss: {average_train_loss:.
↪4f}, Training Accuracy: {train_accuracy:.4f}, Validation Loss:␣
↪{average_val_loss:.4f}, Validation Accuracy: {val_accuracy:.4f}")

# Plotting the accuracy curve
epochs = range(1, len(train_accuracies) + 1)
plt.plot(epochs, train_accuracies, label='Training Accuracy')
plt.plot(epochs, val_accuracies, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy Over Epochs')
plt.legend()
plt.show()

# Plotting the loss curve
epochs = range(1, len(train_losses) + 1)
plt.plot(epochs, train_losses, label='Training Loss')
plt.plot(epochs, val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss Over Epochs')
plt.legend()
plt.show()
```

```
Epoch 1/10, Training Loss: 14.0037, Training Accuracy: 0.3513, Validation Loss:
2.3837, Validation Accuracy: 0.4306
Epoch 2/10, Training Loss: 1.2313, Training Accuracy: 0.4901, Validation Loss:
1.1191, Validation Accuracy: 0.6301
Epoch 3/10, Training Loss: 0.6111, Training Accuracy: 0.5879, Validation Loss:
0.9394, Validation Accuracy: 0.6605
Epoch 4/10, Training Loss: 0.3231, Training Accuracy: 0.6612, Validation Loss:
1.0609, Validation Accuracy: 0.6922
```

```
Epoch 5/10, Training Loss: 0.1865, Training Accuracy: 0.7173, Validation Loss:
0.9948, Validation Accuracy: 0.6843
Epoch 6/10, Training Loss: 0.1290, Training Accuracy: 0.7581, Validation Loss:
0.9225, Validation Accuracy: 0.7252
Epoch 7/10, Training Loss: 0.0642, Training Accuracy: 0.7911, Validation Loss:
0.9722, Validation Accuracy: 0.7133
Epoch 8/10, Training Loss: 0.0448, Training Accuracy: 0.8167, Validation Loss:
0.9488, Validation Accuracy: 0.7252
Epoch 9/10, Training Loss: 0.0262, Training Accuracy: 0.8367, Validation Loss:
0.9423, Validation Accuracy: 0.7199
Epoch 10/10, Training Loss: 0.0180, Training Accuracy: 0.8530, Validation Loss:
0.9942, Validation Accuracy: 0.7213
```





[13]:
```
# Save the model state_dict
torch.save(model.state_dict(), '244_10_epochs.pth')
```

[14]:
```
# Assuming you have a validation DataLoader named val_loader
loaded_model = SimpleModel()
loaded_model.load_state_dict(torch.load('244_10_epochs.pth'))
#loaded_model.eval()

model.eval()                                          # Set the model to␣
 ↪evaluation mode
all_predictions = []
all_labels = []                                       # This list was missing␣
 ↪in the original code

with torch.no_grad():
    for inputs, labels in val_loader:
        try:
            # Ensure the input size is correct
            inputs = inputs.view(inputs.size(0), 3, 244, 244)
```

```
            #inputs = inputs.view(inputs.size(0), 3, 488, 488)

            outputs = model(inputs)
            predictions = torch.argmax(outputs, dim=1)
            all_predictions.extend(predictions.tolist())
            all_labels.extend(labels.tolist())  # Populate the true labels
        except Exception as e:
            print(f"An error occurred: {e}")

# Now you have lists of all predicted labels and true labels
# You can compare them to compute accuracy or analyze the classification␣
 ↪breakdown

# Compute accuracy
if len(all_labels) > 0:
    accuracy = sum(p == l for p, l in zip(all_predictions, all_labels)) /␣
 ↪len(all_labels)
    print(f"Accuracy: {accuracy:.2%}")
else:
    print("No labels available for computing accuracy.")
# Your custom labels
custom_labels = ['Healthy', 'Yellow', 'Rust', 'RedRot', 'Mosaic']

# Analyze classification breakdown
for class_label in range(len(custom_labels)):
    correct = sum(p == class_label and l == class_label for p, l in␣
 ↪zip(all_predictions, all_labels))
    total = all_labels.count(class_label)

    if total == 0:
        print(f"Class {custom_labels[class_label]}: No examples in the␣
 ↪validation set")
    else:
        percentage = correct / total if total != 0 else 0.0
        print(f"{custom_labels[class_label]}: {correct}/{total} ({percentage:.
 ↪2%})")
```

```
Accuracy: 72.13%
Healthy: 115/152 (75.66%)
Yellow: 105/138 (76.09%)
Rust: 118/163 (72.39%)
RedRot: 116/164 (70.73%)
Mosaic: 92/140 (65.71%)
```

[15]:
```
# Assuming you have a validation DataLoader named val_loader
loaded_model = SimpleModel()
loaded_model.load_state_dict(torch.load('244_10_epochs.pth'))
```

```
model.eval()  # Set the model to evaluation mode
params = list(loaded_model.parameters())
print(params[0].shape)
print(params[1].shape)
print(params[2].shape)
print(params[3].shape)

flattened_weights = params[0].data.flatten()
np.savetxt('weights_0.txt', flattened_weights)

flattened_weights1 = params[1].data.flatten()
np.savetxt('biases_0.txt', flattened_weights1)

print (flattened_weights1)
flattened_weights2 = params[2].data.flatten()
np.savetxt('weights_1.txt', flattened_weights2)

flattened_weights3 = params[3].data.flatten()
np.savetxt('biases_1.txt', flattened_weights3)

# Print the weights and biases
for name, param in loaded_model.named_parameters():
    if 'weight' in name or 'bias' in name:
        print(f"Parameter: {name}, Shape: {param.shape}")
        print("Values:")
        print(param.data)
        print("\n")
```

```
torch.Size([32, 3, 2, 2])
torch.Size([32])
torch.Size([5, 476288])
torch.Size([5])
tensor([ 0.1399,  0.2444,  0.0356,  0.0575, -0.0547,  0.0273,  0.0954, -0.1022,
        -0.0263,  0.2364, -0.2725,  0.2721, -0.0651, -0.0576, -0.1025,  0.1305,
         0.0803, -0.1817, -0.1563, -0.2643, -0.1689,  0.0618, -0.1050, -0.0124,
        -0.2040, -0.1170, -0.0611, -0.2142,  0.1345, -0.0018, -0.2511, -0.1311])
Parameter: conv1.weight, Shape: torch.Size([32, 3, 2, 2])
Values:
tensor([[[[ 7.8986e-02,  2.3883e-01],
          [-1.3255e-01, -1.6590e-01]],

         [[-3.3597e-02, -6.2142e-02],
          [ 1.5637e-01, -2.2117e-01]],

         [[-3.0068e-01, -1.2209e-01],
          [-8.3980e-03,  1.6060e-01]]],
```

```
[[[ 7.5073e-02, -4.7525e-02],
  [-6.6986e-02, -2.2751e-02]],

 [[ 2.3166e-01, -2.3633e-01],
  [-3.0123e-01,  5.5085e-02]],

 [[ 2.2719e-02, -3.8310e-02],
  [-2.0552e-01, -1.7857e-01]]],


[[[ 1.9131e-01,  2.0604e-01],
  [ 8.9179e-02, -1.4576e-01]],

 [[-1.6341e-01, -2.1567e-01],
  [ 8.8184e-03, -1.1986e-01]],

 [[ 2.7937e-01, -5.6992e-02],
  [-3.3118e-02, -8.2434e-02]]],


[[[ 2.3211e-01,  5.5662e-02],
  [ 5.1870e-02, -3.0315e-01]],

 [[ 9.9252e-02, -2.6878e-01],
  [-6.3919e-02,  1.2674e-01]],

 [[ 4.8208e-02, -2.6665e-02],
  [-3.1725e-02, -9.3331e-02]]],


[[[-2.5400e-01,  3.5618e-02],
  [ 1.9778e-02,  7.9772e-02]],

 [[-8.9680e-02,  1.4974e-01],
  [ 2.5708e-01,  3.4747e-02]],

 [[-2.6265e-01,  4.9889e-02],
  [-6.2271e-02,  8.6957e-02]]],


[[[ 4.2167e-02, -3.1409e-01],
  [ 2.5066e-01, -1.1310e-01]],

 [[-3.1736e-01,  2.3452e-01],
  [ 2.5181e-01,  1.1306e-01]],

 [[-5.0839e-02, -1.2732e-01],
```

```
  [ 1.3343e-01, -1.4736e-01]]],


[[[-2.7620e-01,  2.7560e-03],
  [-2.0050e-01, -1.1650e-01]],

 [[-8.7776e-02, -2.1965e-01],
  [ 1.6625e-01,  1.1006e-01]],

 [[ 2.6803e-01,  1.7136e-01],
  [-1.7687e-01,  2.2120e-01]]],


[[[ 2.4328e-01,  2.5548e-01],
  [ 3.6142e-02, -1.9289e-01]],

 [[ 7.2556e-02, -2.3201e-01],
  [-2.1673e-01, -1.2144e-01]],

 [[ 7.8316e-02,  1.9299e-01],
  [ 3.2892e-03,  7.0821e-02]]],


[[[ 1.4652e-02,  1.4205e-01],
  [-2.4312e-01,  2.4071e-01]],

 [[ 1.0928e-01, -1.6793e-01],
  [ 9.6177e-02, -1.1580e-01]],

 [[ 1.5772e-02, -4.9692e-02],
  [-2.6447e-01,  2.6529e-01]]],


[[[-2.5835e-01, -2.4275e-01],
  [-9.7544e-02,  2.0356e-01]],

 [[ 2.0638e-01,  1.2590e-01],
  [ 4.0820e-02, -1.4484e-02]],

 [[-2.3842e-01, -7.8008e-02],
  [-1.4430e-01, -2.6282e-01]]],


[[[ 4.4423e-02,  1.6097e-01],
  [-1.6815e-01, -7.6319e-02]],

 [[ 1.8699e-01, -1.0789e-01],
  [ 1.6740e-01, -2.8284e-01]],
```

```
 [[-2.8552e-01,  5.0739e-02],
  [-2.2074e-01, -2.2025e-01]]],


[[[ 1.2319e-01, -2.5377e-01],
  [-1.8798e-01, -2.6239e-01]],

 [[-2.8980e-01,  2.9460e-02],
  [ 1.5190e-01, -2.3637e-01]],

 [[-1.2093e-01,  3.0310e-02],
  [-1.6424e-01, -3.8178e-02]]],


[[[ 8.5294e-02, -3.1377e-02],
  [ 1.7474e-01,  2.5499e-01]],

 [[ 4.0669e-02,  6.5525e-02],
  [-2.0906e-01, -5.5761e-02]],

 [[-1.5496e-01,  8.4396e-02],
  [-5.6261e-02, -2.1889e-01]]],


[[[-4.0927e-02, -1.4479e-01],
  [-3.7058e-02, -2.1759e-01]],

 [[ 8.3368e-03, -2.1073e-02],
  [ 1.0844e-01,  5.9493e-02]],

 [[ 4.6416e-03, -2.3173e-01],
  [-1.8637e-01,  7.9020e-02]]],


[[[-3.9642e-02,  2.4054e-01],
  [ 2.9586e-02,  5.8906e-02]],

 [[-2.8276e-01,  1.3793e-01],
  [-1.0423e-02, -1.5392e-01]],

 [[ 9.9896e-02, -1.4172e-01],
  [ 1.1613e-01,  1.9855e-01]]],


[[[-2.1269e-01, -2.1339e-01],
  [ 1.6719e-01,  3.1175e-02]],
```

```
 [[ 9.0501e-02,  2.4754e-01],
  [-1.4321e-01, -1.5066e-01]],

 [[ 2.3172e-01, -2.0722e-01],
  [-1.1304e-01,  1.0308e-01]]],


[[[ 8.1298e-02, -8.8694e-02],
  [ 1.4400e-01, -2.5443e-01]],

 [[ 1.8891e-02, -9.0315e-02],
  [-3.2405e-02, -1.4030e-01]],

 [[ 1.8860e-01,  1.3856e-01],
  [-7.3678e-02,  6.7442e-02]]],


[[[-1.1887e-01, -6.8235e-02],
  [-2.4846e-01, -1.5271e-01]],

 [[-1.4610e-01, -1.7744e-02],
  [ 2.2685e-01, -2.3127e-01]],

 [[-5.2694e-03, -1.2230e-01],
  [ 1.0904e-01,  2.5996e-01]]],


[[[-1.3876e-01,  1.9928e-01],
  [ 1.7912e-01, -6.9105e-02]],

 [[-1.4627e-01,  2.5588e-01],
  [-1.5578e-01, -5.9382e-02]],

 [[ 1.7291e-01, -1.5519e-01],
  [-1.0590e-01,  1.4567e-01]]],


[[[-1.2792e-01, -1.1896e-01],
  [ 1.5863e-01, -1.9588e-02]],

 [[ 1.7453e-01, -1.0583e-01],
  [-1.7354e-01,  2.2518e-01]],

 [[-1.7193e-02, -9.1657e-02],
  [-1.6113e-02,  2.7271e-01]]],


[[[ 1.6366e-01,  5.3007e-03],
```

```
  [ 2.2057e-01,  2.4066e-01]],

 [[-1.8122e-01,  1.0790e-01],
  [ 1.1741e-02,  1.2695e-01]],

 [[-2.6638e-01, -2.6038e-01],
  [ 9.6589e-03,  1.9461e-01]]],


[[[-2.2093e-01, -2.5777e-01],
  [ 2.3584e-01, -1.6950e-01]],

 [[ 2.0100e-01, -1.2422e-02],
  [ 2.8912e-01, -2.1728e-01]],

 [[ 1.7489e-02, -3.0809e-02],
  [ 2.1927e-01, -2.8228e-01]]],


[[[-8.8299e-02,  2.1157e-01],
  [-1.0475e-01, -1.0923e-01]],

 [[ 1.2818e-01, -2.3540e-01],
  [-2.0534e-01,  1.3105e-01]],

 [[-3.2646e-02, -1.3261e-02],
  [ 2.7359e-02,  2.1562e-01]]],


[[[-1.3708e-01,  1.3017e-01],
  [-1.6821e-01, -1.3442e-01]],

 [[-4.0991e-02,  1.4495e-01],
  [ 1.3254e-01,  2.1204e-01]],

 [[-7.6667e-02, -2.6998e-01],
  [-1.6011e-01,  7.7882e-02]]],


[[[-2.6236e-01, -2.2162e-01],
  [-4.7680e-02, -7.3963e-02]],

 [[ 6.2191e-02,  9.4130e-03],
  [-2.3601e-01, -2.7209e-01]],

 [[-1.1110e-01,  2.4138e-01],
  [ 2.6106e-01, -1.7909e-01]]],
```

```
[[[-2.8816e-01,  2.1914e-01],
  [-1.9646e-01,  1.4791e-01]],

 [[-1.0505e-02,  1.9403e-01],
  [ 1.6743e-01,  2.0148e-01]],

 [[ 9.9769e-02, -2.2484e-01],
  [-1.1972e-01, -1.8142e-02]]],


[[[ 1.9340e-01, -1.7529e-01],
  [ 1.5903e-01,  1.5692e-01]],

 [[ 4.2436e-02, -7.6184e-02],
  [-2.1678e-01,  9.0169e-02]],

 [[-2.5756e-01, -3.6800e-03],
  [ 1.7198e-01, -2.2125e-04]]],


[[[-1.4449e-01, -1.7400e-02],
  [ 2.6375e-01,  2.1723e-01]],

 [[ 1.1240e-01, -2.4891e-01],
  [ 1.5268e-01,  2.4937e-02]],

 [[ 1.0551e-01, -7.1981e-02],
  [ 6.8832e-03, -4.6968e-02]]],


[[[ 3.1631e-01, -2.1076e-01],
  [-1.5741e-01, -1.8806e-01]],

 [[ 5.2859e-02, -1.1277e-04],
  [-4.2876e-02, -2.7454e-01]],

 [[-1.3396e-01,  1.9535e-01],
  [-2.2739e-01,  2.7519e-01]]],


[[[ 9.4511e-02, -1.3435e-01],
  [ 8.6138e-02,  1.7312e-01]],

 [[-1.9299e-01,  2.0838e-01],
  [-1.6330e-02,  1.2540e-02]],

 [[-1.2776e-01, -1.4695e-01],
```

```
          [ 2.8272e-02, -9.1614e-02]]],


        [[[ 2.1134e-01, -1.3662e-01],
          [-3.7783e-02,  8.4289e-02]],

         [[ 2.0309e-02, -2.0300e-01],
          [ 7.9256e-02,  2.1043e-01]],

         [[-2.1008e-01,  1.7128e-01],
          [ 2.2354e-01,  8.1586e-02]]],


        [[[ 1.4957e-01, -1.2159e-01],
          [ 2.3394e-01,  8.0101e-02]],

         [[-2.7624e-01,  2.2013e-01],
          [ 2.2635e-01, -4.0984e-02]],

         [[-2.1353e-01, -8.2074e-02],
          [ 9.9692e-02, -1.3651e-01]]]])


Parameter: conv1.bias, Shape: torch.Size([32])
Values:
tensor([ 0.1399,  0.2444,  0.0356,  0.0575, -0.0547,  0.0273,  0.0954, -0.1022,
        -0.0263,  0.2364, -0.2725,  0.2721, -0.0651, -0.0576, -0.1025,  0.1305,
         0.0803, -0.1817, -0.1563, -0.2643, -0.1689,  0.0618, -0.1050, -0.0124,
        -0.2040, -0.1170, -0.0611, -0.2142,  0.1345, -0.0018, -0.2511, -0.1311])


Parameter: fc1.weight, Shape: torch.Size([5, 476288])
Values:
tensor([[ 8.9815e-03,  1.1242e-02,  8.4387e-03,  …, -2.1847e-02,
         -2.8235e-02, -2.9261e-02],
        [ 3.8277e-03,  7.2185e-03,  7.1915e-03,  …,  6.1499e-03,
          3.2571e-03, -7.2333e-03],
        [ 6.9256e-04, -6.5939e-04, -9.6211e-04,  …,  8.6416e-03,
          1.4516e-02,  2.3487e-02],
        [-6.4180e-03, -7.8350e-03, -3.5977e-03,  …,  1.2723e-03,
          1.8146e-03,  6.0735e-03],
        [ 5.6331e-05, -4.0572e-03, -7.3207e-03,  …,  1.0599e-02,
          1.4022e-02,  1.5712e-02]])


Parameter: fc1.bias, Shape: torch.Size([5])
Values:
tensor([ 0.0003,  0.0043, -0.0003, -0.0017,  0.0072])
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]: