Homework 2: Convolutional Neural Network

Code:

```
#Author - Het Pathak

import argparse
import os
import random
import time
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from thop import profile
import cv2

# Check if CUDA is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


# Set random seed for reproducibility
def set_seed(seed):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.backends.cudnn.deterministic = True
        torch.backends.cudnn.benchmark = False


# Define the CNN model
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()
        # First CONV layer with specific requirements:
        # filter size: 5x5, stride: 1, no padding, 32 filters
        self.conv1 = nn.Conv2d(3, 32, kernel_size=5, stride=1, padding=0)

        # Additional CONV layers
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
```

```
        # Batch normalization layers
        self.bn1 = nn.BatchNorm2d(32)
        self.bn2 = nn.BatchNorm2d(64)
        self.bn3 = nn.BatchNorm2d(128)

        # Pooling layer
        self.pool = nn.MaxPool2d(2, 2)

        # Dropout for regularization
        self.dropout = nn.Dropout(0.25)

        # Calculate input size for first FC layer
        # Input: 32x32, after conv1: 28x28, after pool: 14x14
        # After conv2: 14x14, after pool: 7x7
        # After conv3: 7x7, after pool: 3x3
        # So 128 channels with 3x3 feature maps = 128*3*3 = 1152
        self.fc1 = nn.Linear(128 * 3 * 3, 512)
        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        # First convolutional layer with ReLU and batch norm
        x = self.pool(F.relu(self.bn1(self.conv1(x))))

        # Second convolutional layer with ReLU and batch norm
        x = self.pool(F.relu(self.bn2(self.conv2(x))))

        # Third convolutional layer with ReLU and batch norm
        x = self.pool(F.relu(self.bn3(self.conv3(x))))

        # Flatten the output for the fully connected layers
        x = x.view(-1, 128 * 3 * 3)

        # Fully connected layers with ReLU and dropout
        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 get_conv1_output(self, x):
        # Get the output of the first convolutional layer
        x = self.conv1(x)
        return x


# Load the CIFAR-10 dataset
def load_data():
    # Define data transformations
    transform_train = transforms.Compose([
```

```
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
  ])

  transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
  ])

  # Load training and test datasets
  trainset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_train)
  trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=128, shuffle=True, num_workers=2)

  testset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True, transform=transform_test)
  testloader = torch.utils.data.DataLoader(
    testset, batch_size=100, shuffle=False, num_workers=2)

  # CIFAR-10 classes
  classes = ('airplane', 'car', 'bird', 'cat', 'deer',
        'dog', 'frog', 'horse', 'ship', 'truck')

  return trainloader, testloader, classes


# Test the model on the full test set
def test_model(model, testloader, criterion=None):
  model.eval()
  correct = 0
  total = 0
  running_loss = 0.0
  with torch.no_grad():
    for data in testloader:
      images, labels = data
      images, labels = images.to(device), labels.to(device)
      outputs = model(images)
      _, predicted = torch.max(outputs.data, 1)
      total += labels.size(0)
      correct += (predicted == labels).sum().item()

      # Calculate test loss
      if criterion is not None:
        loss = criterion(outputs, labels)
        running_loss += loss.item()

  accuracy = 100 * correct / total
  test_loss = running_loss / len(testloader) if criterion is not None else 0
  return accuracy, test_loss
```

```
def train(args=None):
    if not os.path.exists('./model'):
        os.makedirs('./model')

    seeds = [42, 123, 1000]
    accuracies = []
    best_accuracy = 0
    best_model_state = None

    for seed_idx, seed in enumerate(seeds):
        set_seed(seed)
        trainloader, testloader, classes = load_data()
        model = CNNModel().to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
        scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)

        num_epochs = 10
        train_accuracies = []  # List to store training accuracies
        test_accuracies = []  # List to store testing accuracies
        print(f"\nSeed: {seed}")
        print("Epoch  TrainLoss  TrainAccuracy  TestAccuracy  TestLoss")

        for epoch in range(num_epochs):
            model.train()
            running_loss = 0.0
            correct = 0
            total = 0

            for i, data in enumerate(trainloader, 0):
                inputs, labels = data
                inputs, labels = inputs.to(device), labels.to(device)
                optimizer.zero_grad()
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()

                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
                running_loss += loss.item()

            train_accuracy = 100 * correct / total
            test_accuracy, test_loss = test_model(model, testloader, criterion)
            model.train()

            # Store accuracies for plotting
            train_accuracies.append(train_accuracy)
            test_accuracies.append(test_accuracy)
```

```
            avg_loss = running_loss / len(trainloader)
            print(
                f"{epoch + 1:<6} {avg_loss:.3f}        {train_accuracy:.2f}%          {test_accuracy:.2f}%
{test_loss:.3f}")

            # Track best model
            if test_accuracy > best_accuracy:
                best_accuracy = test_accuracy
                best_model_state = model.state_dict()

            scheduler.step()

        # Accuracy plot (only for the first seed)
        if seed_idx == 0:
            plt.figure(figsize=(10, 6))
            epochs = range(1, num_epochs + 1)
            plt.plot(epochs, train_accuracies, 'b-o', label='Training Accuracy')
            plt.plot(epochs, test_accuracies, 'r-s', label='Testing Accuracy')
            plt.title('Training and Testing Accuracy vs. Epochs')
            plt.xlabel('Epochs')
            plt.ylabel('Accuracy (%)')
            plt.grid(True)
            plt.legend()
            plt.savefig('./model/accuracy_plot.png')
            print(f"Training accuracy plot saved to './model/accuracy_plot.png'")

        # Save the final test accuracy
        final_accuracy = test_accuracies[-1]
        accuracies.append(final_accuracy)

    # Save only the best model
    torch.save(best_model_state, './model/cifar_model.pt')
    print(f"\nModel saved in file: ./model/cifar_model.pt")

    print("\n--- Training Summary ---")
    for i, (seed, accuracy) in enumerate(zip(seeds, accuracies)):
        print(f"Seed {seed}: Test Accuracy = {accuracy:.2f}%")

    mean_accuracy = np.mean(accuracies)
    std_accuracy = np.std(accuracies)
    print(f"Mean Accuracy: {mean_accuracy:.2f}%")
    print(f"Standard Deviation: {std_accuracy:.2f}%")


# Test function to predict a single image
def test(image_path):
    # Load the saved model
    model = CNNModel().to(device)
    model.load_state_dict(torch.load('./model/cifar_model.pt'))
    model.eval()
```

```
    # CIFAR-10 classes
    classes = ('airplane', 'car', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck')

    # Load and preprocess the image
    image = cv2.imread(image_path)
    if image is None:
        print(f"Error: Could not read image file: {image_path}")
        return

    # Resize to 32x32 and convert BGR to RGB
    image = cv2.resize(image, (32, 32))
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Convert to tensor and normalize
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
    ])

    image_tensor = transform(image).unsqueeze(0).to(device)

    # Predict
    with torch.no_grad():
        output = model(image_tensor)
        _, predicted = torch.max(output, 1)
        pred_class = classes[predicted.item()]

    print(f"Predicted class: {pred_class}")

    # Get the output of the first convolutional layer
    conv_output = model.get_conv1_output(image_tensor)
    conv_output = conv_output.detach().cpu().numpy()[0]

    # Visualize the output of each filter in the first CONV layer
    plt.figure(figsize=(8, 8))
    for i in range(32):
        plt.subplot(4, 8, i + 1)
        plt.imshow(conv_output[i], cmap='viridis')
        plt.axis('off')

    plt.tight_layout()
    plt.savefig('CONV_rslt.png')


# Function to load and test ResNet-20
def test_resnet20():
    # Import ResNet-20 model
    from resnet20_cifar import resnet20

    # Load pretrained model
    model = resnet20().to(device)
```

```
    model_path = "./model/resnet20_cifar10_pretrained.pt"
    model.load_state_dict(torch.load(model_path, map_location=device))
    model.eval()

    # Load test data with the same normalization as used in training
    transform_test = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
    ])

    testset = torchvision.datasets.CIFAR10(
        root='./data', train=False, download=True, transform=transform_test)
    testloader = torch.utils.data.DataLoader(
        testset, batch_size=100, shuffle=False, num_workers=2)

    # Test the model
    accuracy, test_loss = test_model(model, testloader)
    print(f"ResNet-20 Test Accuracy: {accuracy:.2f}%")


# Count parameters and MACs using THOP
def count_parameters_and_macs(model, model_name):
    # Create a dummy input
    dummy_input = torch.randn(1, 3, 32, 32).to(device)

    # Count MACs and parameters
    macs, params = profile(model, inputs=(dummy_input,))

    # print(f"{model_name} - Parameters: {params / 1e6:.2f}M, MACs: {macs / 1e6:.2f}M")
    print(f"{model_name} - Parameters: {int(params)}, MACs: {int(macs)}")


# Test inference speed
def inference_speed_test(args=None):
    # Load both models
    # 1. Custom CNN model
    cnn_model = CNNModel().to(device)
    cnn_model.load_state_dict(torch.load('./model/cifar_model.pt'))
    cnn_model.eval()

    # 2. ResNet-20 model
    from resnet20_cifar import resnet20
    resnet_model = resnet20().to(device)
    model_path = "./model/resnet20_cifar10_pretrained.pt"
    resnet_model.load_state_dict(torch.load(model_path,map_location=device))
    resnet_model.eval()

    # Create a dummy input
    dummy_input = torch.randn(1, 3, 32, 32).to(device)

    # Warm-up iterations
    for _ in range(10):
```

```
        _ = cnn_model(dummy_input)
        _ = resnet_model(dummy_input)

    # Test CNN model inference speed
    torch.cuda.synchronize() if torch.cuda.is_available() else None
    start_time = time.time()
    num_iterations = 1000

    with torch.no_grad():
        for _ in range(num_iterations):
            _ = cnn_model(dummy_input)

    torch.cuda.synchronize() if torch.cuda.is_available() else None
    end_time = time.time()
    cnn_inference_time = (end_time - start_time) / num_iterations * 1000  # in milliseconds

    # Test ResNet-20 model inference speed
    torch.cuda.synchronize() if torch.cuda.is_available() else None
    start_time = time.time()

    with torch.no_grad():
        for _ in range(num_iterations):
            _ = resnet_model(dummy_input)

    torch.cuda.synchronize() if torch.cuda.is_available() else None
    end_time = time.time()
    resnet_inference_time = (end_time - start_time) / num_iterations * 1000  # in milliseconds

    # Print results
    print(f"\nInference Speed Test Results (average over {num_iterations} iterations):")
    print(f"Custom CNN model: {cnn_inference_time:.4f} ms per inference")
    print(f"ResNet-20 model: {resnet_inference_time:.4f} ms per inference")

    # Also count parameters and MACs for both models
    print("\nModel Complexity Analysis:")
    count_parameters_and_macs(cnn_model, "Custom CNN")
    count_parameters_and_macs(resnet_model, "ResNet-20")


# Main function to parse arguments
def main():
    parser = argparse.ArgumentParser(description='CNN Classifier for CIFAR-10')
    parser.add_argument('command', type=str, help='train, test, or resnet20')
    parser.add_argument('image_path', nargs='?', help='Path to the image for testing')

    args = parser.parse_args()

    if args.command == 'train':
        train()
    elif args.command == 'test':
        if args.image_path is None:
            print("Error: Please provide an image path for testing")
```

```
            return
        test(args.image_path)
    elif args.command == 'resnet20':
        test_resnet20()
    elif args.command == 'inference':
        inference_speed_test()


if __name__ == '__main__':
    main()
```

Network Description:

- **Three Convolutional Layers**
  - conv1: 32 filters, 5×5 kernel, stride=1, no padding.
  - conv2: 64 filters, 3×3 kernel, stride=1, padding=1.
  - conv3: 128 filters, 3×3 kernel, stride=1, padding=1.
- **Batch Normalization** (bn1, bn2, bn3)
  - Normalizes activations to speed up training and stabilize learning.
- **Max Pooling** (pool)
  - **2×2 pooling after each convolutional layer** to reduce feature map size.
- **Dropout Regularization (0.25 probability)**
  - Applied before fully connected layers to prevent overfitting.
- **Fully Connected Layers (FC Layers)**
  - fc1: **512 neurons**
  - fc2: **128 neurons**
  - fc3: **10 neurons** (corresponding to CIFAR-10 classes).
- **Activation Function:**
  - Uses **ReLU** activation for non-linearity in all layers except the last one.
  - The last layer (fc3) outputs raw logits (before softmax).

Training Output:

```
Terminal    Local (2)

(.venv) PS C:\Users\hetpa\PycharmProjects\DeepLearningProjs> python CNNclassify.py train

Seed: 42
Epoch  TrainLoss  TrainAccuracy  TestAccuracy  TestLoss
1      1.597      40.65%         54.29%        1.330
2      1.230      55.56%         59.98%        1.130
3      1.086      61.36%         66.14%        0.960
4      0.985      65.16%         68.84%        0.887
5      0.917      67.80%         71.43%        0.814
6      0.859      70.05%         73.46%        0.751
7      0.811      71.80%         74.50%        0.715
8      0.774      72.93%         75.11%        0.704
9      0.746      74.10%         73.53%        0.774
10     0.715      75.21%         77.43%        0.653
11     0.696      75.74%         78.14%        0.629
12     0.667      76.96%         77.63%        0.640
13     0.647      77.49%         77.25%        0.652
14     0.630      78.27%         79.75%        0.583
15     0.617      78.79%         81.05%        0.563
16     0.604      79.18%         80.40%        0.568
17     0.588      79.86%         80.27%        0.584
18     0.573      80.18%         81.07%        0.554
19     0.558      80.95%         79.74%        0.592
20     0.557      81.01%         78.41%        0.640
Training accuracy plot saved to './model/accuracy_plot.png'
```

```
Terminal    Local (2)

Seed: 123
Epoch  TrainLoss  TrainAccuracy  TestAccuracy  TestLoss
1      1.629      39.20%         53.01%        1.268
2      1.248      54.93%         58.53%        1.130
3      1.091      60.94%         62.88%        1.057
4      0.990      64.68%         70.21%        0.858
5      0.917      67.49%         71.06%        0.833
6      0.869      69.51%         72.20%        0.782
7      0.817      71.56%         72.18%        0.797
8      0.777      72.85%         74.26%        0.749
9      0.741      74.20%         77.30%        0.651
10     0.717      74.90%         77.05%        0.671
11     0.688      75.99%         75.17%        0.709
12     0.669      76.83%         76.45%        0.688
13     0.650      77.51%         79.16%        0.597
14     0.635      77.88%         80.66%        0.558
15     0.615      78.76%         80.37%        0.579
16     0.598      79.39%         80.98%        0.545
17     0.595      79.51%         80.71%        0.564
18     0.570      80.34%         81.89%        0.529
19     0.563      80.49%         81.33%        0.525
20     0.545      81.10%         81.69%        0.530
```

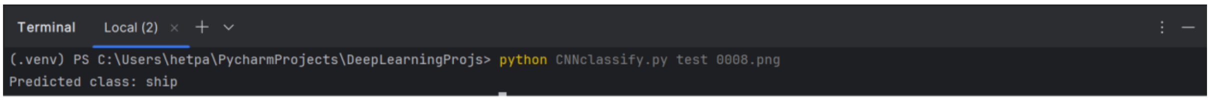
```
Terminal    Local (2)

Seed: 1000
Epoch  TrainLoss  TrainAccuracy  TestAccuracy  TestLoss
1      1.628      39.04%         53.42%        1.297
2      1.249      54.79%         61.30%        1.091
3      1.098      60.90%         67.25%        0.927
4      0.981      65.28%         69.82%        0.869
5      0.903      68.45%         72.57%        0.784
6      0.854      70.00%         73.87%        0.748
7      0.801      72.04%         75.73%        0.701
8      0.765      73.14%         76.36%        0.698
9      0.733      74.60%         75.89%        0.705
10     0.704      75.63%         77.97%        0.637
11     0.684      76.36%         77.24%        0.657
12     0.658      77.26%         79.05%        0.619
13     0.633      78.10%         78.77%        0.622
14     0.619      78.66%         79.76%        0.585
15     0.604      79.25%         81.17%        0.546
16     0.587      79.72%         81.53%        0.541
17     0.573      80.36%         81.09%        0.561
18     0.563      80.49%         81.30%        0.545
19     0.553      80.93%         82.60%        0.510
20     0.540      81.44%         81.01%        0.556

Model saved in file: ./model/cifar_model.pt
```

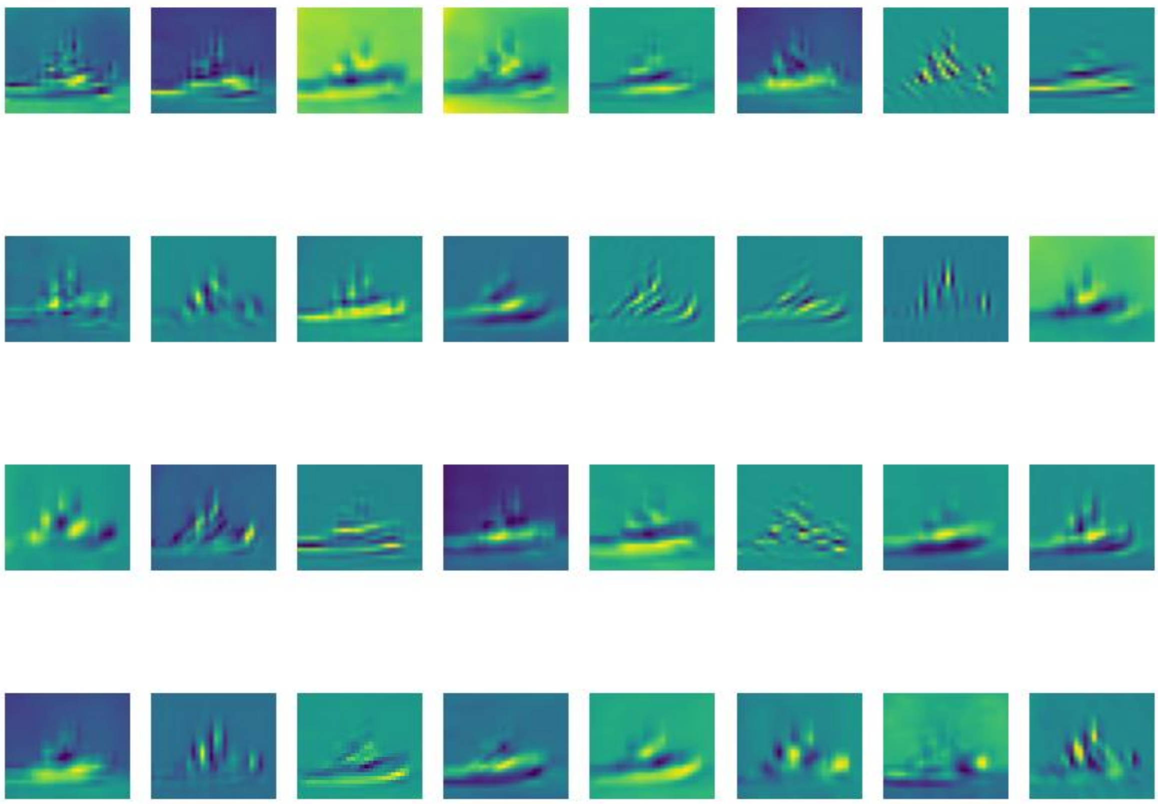
Highest Test Accuracy: 82.60%

Testing Output:



- 0008.png is test image from CIFAR-10 dataset.
- random_car.png is a random image of a car downloaded from google.

Visualization Result- First CONV layer:

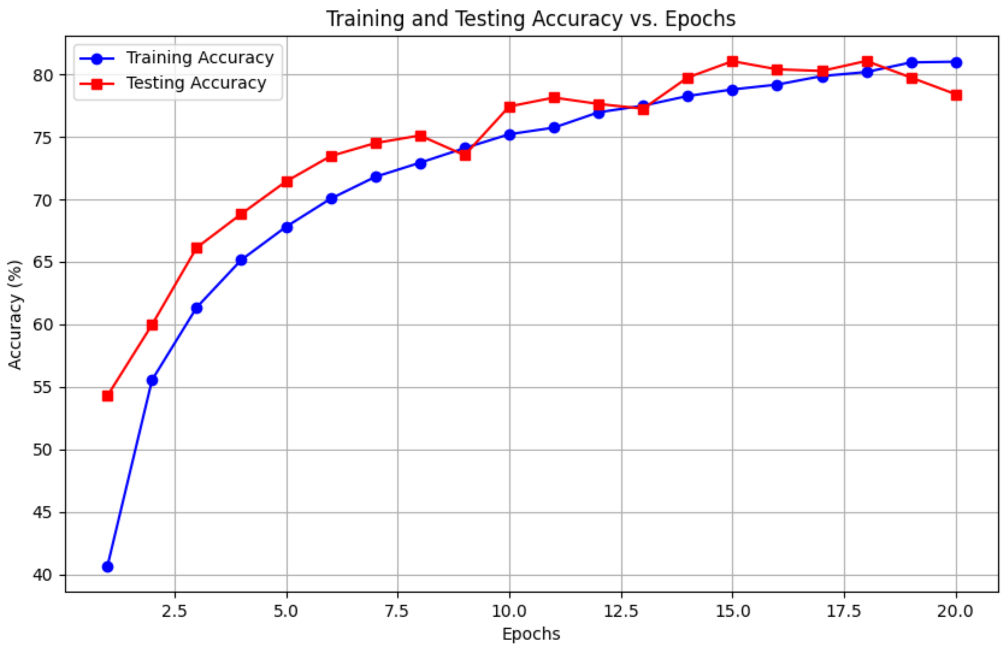


ResNet accuracy on CIFAR-10 dataset:

```
Terminal    Local (2)
Predicted class: ship
(.venv) PS C:\Users\hetpa\PycharmProjects\DeepLearningProjs> python CNNclassify.py resnet20
ResNet-20 Test Accuracy: 92.05%
```

Train/Test Accuracy vs Epochs plot:

- Plot for the 1st seed.



Inference Speed:

```
Inference Speed Test Results (average over 1000 iterations):
Custom CNN model: 2.7776 ms per inference
ResNet-20 model: 11.2904 ms per inference
```

Computation Costs and Number of Parameters:

```
Model Complexity Analysis:
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class 'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.pooling.MaxPool2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.dropout.Dropout'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
Custom CNN - Parameters: 752522, MACs: 9939200
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class 'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
[INFO] Register count_avgpool() for <class 'torch.nn.modules.pooling.AvgPool2d'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
ResNet-20 - Parameters: 272474, MACs: 41616064
```

Training Summary:

Seed 42: Final Test Accuracy = 78.41%

Seed 123: Final Test Accuracy = 81.69%

Seed 1000: Final Test Accuracy = 81.01%

Mean Accuracy: 80.37%

Standard Deviation: 1.41%

- To check Inference speed, Computation Cost, and Number of Parameters for my CNN model and pretrained resnet20 model run: "**python CNNclassify inference**".