Main Training

All models trained with:

batch_size = 4
epochs = 2
learning_rate = 0.001
momentum = 0.9
Mini-batch Gradient Descent
Cross Entropy Loss

Modifications made to models to fit 3x32x32 input size
Training accuracy given for all models

<u>LeNet()</u>

```
Model size:   parameters: 62006
Training time:  200.56246376037598
```

```
Accuracy of the network on the 10000 test images: 61 %
Accuracy for class: plane is 51.9 %
Accuracy for class: car   is 75.0 %
Accuracy for class: bird  is 55.6 %
Accuracy for class: cat   is 30.2 %
Accuracy for class: deer  is 47.7 %
Accuracy for class: dog   is 54.9 %
Accuracy for class: frog  is 67.3 %
Accuracy for class: horse is 74.3 %
Accuracy for class: ship  is 78.8 %
Accuracy for class: truck is 74.5 %
```

<u>AlexNet()</u>

```
Model size:   parameters: 36360074
Training time:  249.5857174396515
```

```
Accuracy of the network on the 10000 test images: 64 %
Accuracy for class: plane is 70.5 %
Accuracy for class: car   is 74.0 %
Accuracy for class: bird  is 38.6 %
Accuracy for class: cat   is 59.2 %
Accuracy for class: deer  is 45.5 %
Accuracy for class: dog   is 56.6 %
Accuracy for class: frog  is 74.7 %
Accuracy for class: horse is 67.9 %
Accuracy for class: ship  is 78.5 %
Accuracy for class: truck is 78.2 %
```

<u>Resnet-18()</u>

```
Model size:   parameters: 11184650
Training time:  259.30410146713257
```

```
Accuracy of the network on the 10000 test images: 52 %
Accuracy for class: plane is 40.7 %
Accuracy for class: car   is 59.9 %
Accuracy for class: bird  is 55.4 %
Accuracy for class: cat   is 51.6 %
Accuracy for class: deer  is 43.9 %
Accuracy for class: dog   is 39.6 %
Accuracy for class: frog  is 56.0 %
Accuracy for class: horse is 50.9 %
Accuracy for class: ship  is 68.7 %
Accuracy for class: truck is 61.4 %
```

K-fold Cross Validation

K-fold tests were performed with:

batch_size = 40

folds = 5

other factors same as regular training

LeNet()

```
Fold 1
-------
Test set: Average loss: 0.0098, Accuracy: 2931/(10000.0) (29%)

Fold 2
-------
Test set: Average loss: 0.0096, Accuracy: 2964/(10000.0) (30%)

Fold 3
-------
Test set: Average loss: 0.0095, Accuracy: 3086/(10000.0) (31%)

Fold 4
-------
Test set: Average loss: 0.0094, Accuracy: 3157/(10000.0) (32%)

Fold 5
-------
Test set: Average loss: 0.0093, Accuracy: 3249/(10000.0) (32%)
```

AlexNet()

```
Fold 1
-------
Test set: Average loss: 0.0115, Accuracy: 1678/(10000.0) (17%)

Fold 2
-------
Test set: Average loss: 0.0114, Accuracy: 1613/(10000.0) (16%)

Fold 3
-------
Test set: Average loss: 0.0115, Accuracy: 2077/(10000.0) (21%)

Fold 4
-------
Test set: Average loss: 0.0115, Accuracy: 1667/(10000.0) (17%)

Fold 5
-------
Test set: Average loss: 0.0114, Accuracy: 1991/(10000.0) (20%)
```

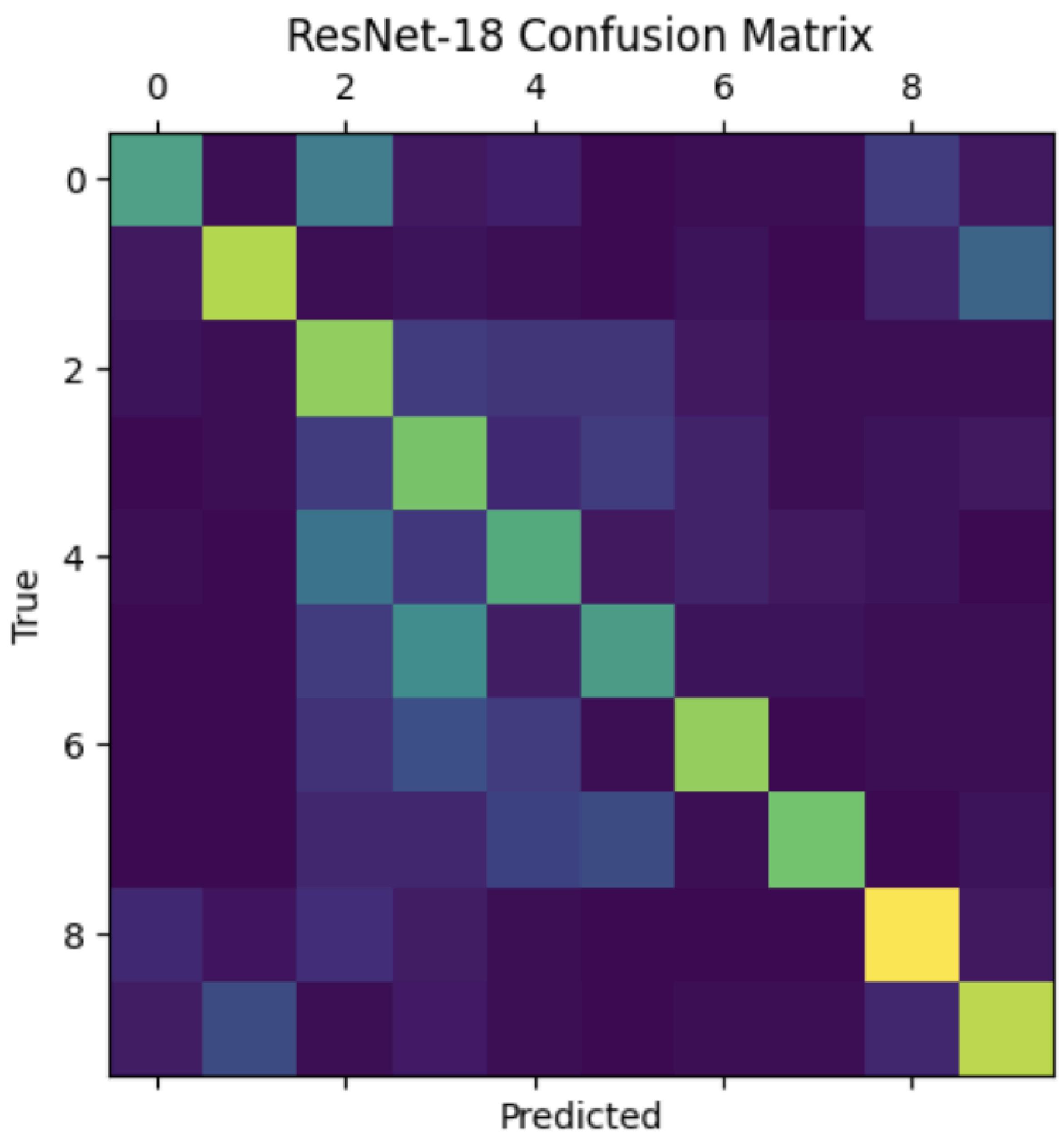
ResNet-18()

```
Fold 1
-------
Test set: Average loss: 0.0052, Accuracy: 6351/(10000.0) (64%)

Fold 2
-------
Test set: Average loss: 0.0051, Accuracy: 6416/(10000.0) (64%)

Fold 3
-------
Test set: Average loss: 0.0052, Accuracy: 6334/(10000.0) (63%)

Fold 4
-------
Test set: Average loss: 0.0054, Accuracy: 6176/(10000.0) (62%)

Fold 5
-------
Test set: Average loss: 0.0050, Accuracy: 6454/(10000.0) (65%)
```

ResNet()-18 was the best

Confusion matrix of ResNet-18:

```
[[407  11 300  34  52   5  12  14 129  36]
 [ 40 599  12  27  11   4  21   6  59 221]
 [ 22  11 554 120 103 106  41  14  17  12]
 [  4  15 128 516  80 120  60  19  26  32]
 [ 15   2 276 112 439  33  58  37  22   6]
 [  5   5 128 344  51 396  23  20  16  12]
 [  1   9  97 176 119  10 560   5  10  13]
 [  5   3  71  76 139 154  14 509   8  21]
 [ 81  30  90  44  18   2   7   1 687  40]
 [ 45 160  15  43  15   4  15  18  71 614]]
```



Three hypotheses

1. <u>Increasing batch size of LeNet()</u>
   This will decrease training time, with lower test accuracy as a result. There will be fewer changes to model parameters, preventing it from accurately fitting the data. However, the smaller amount of gradient calculations will decrease training time.

```
Model size:   parameters: 62006
Training time:  39.29397535324097
```

```
Accuracy of the network on the 10000 test images: 10 %
Accuracy for class: plane is 0.0 %
Accuracy for class: car   is 0.0 %
Accuracy for class: bird  is 0.0 %
Accuracy for class: cat   is 0.0 %
Accuracy for class: deer  is 13.0 %
Accuracy for class: dog   is 0.0 %
Accuracy for class: frog  is 0.0 %
Accuracy for class: horse is 1.7 %
Accuracy for class: ship  is 89.0 %
Accuracy for class: truck is 0.0 %
```

   An improvement in training time and decrease in accuracy is seen compared to the earlier result for LeNet(). The hypothesis is true.

2. <u>Decreasing training input of LeNet() to 30,000</u>
   This will decrease accuracy due to overfitting of the smaller training set.

```
Accuracy of the network on the 10000 test images: 51 %
Accuracy for class: plane is 67.1 %
Accuracy for class: car   is 63.7 %
Accuracy for class: bird  is 22.8 %
Accuracy for class: cat   is 30.3 %
Accuracy for class: deer  is 51.9 %
Accuracy for class: dog   is 40.3 %
Accuracy for class: frog  is 53.0 %
Accuracy for class: horse is 61.5 %
Accuracy for class: ship  is 58.5 %
Accuracy for class: truck is 64.2 %
```

Accuracy did decrease with a smaller training input size (compare to earlier result of 61% with 50,000 input size).

3. Modifying LeNet()

   Modifying LeNet() by adding two new fully connected layers as shown:

```
self.fc1 = nn.Linear(16*5*5, 120)
self.fcNew1 = nn.Linear(120, 100) # for hypothesis test (not in original model)
self.fc2 = nn.Linear(100, 84)
self.fcNew2 = nn.Linear(84, 32) # for hypothesis test
self.fc3 = nn.Linear(32, 10)
```

This will improve performance, since a larger number of parameters will allow the model to better fit the data.

```
Accuracy of the network on the 10000 test images: 49 %
Accuracy for class: plane is 60.5 %
Accuracy for class: car   is 71.5 %
Accuracy for class: bird  is 19.4 %
Accuracy for class: cat   is 42.6 %
Accuracy for class: deer  is 34.7 %
Accuracy for class: dog   is 31.7 %
Accuracy for class: frog  is 70.0 %
Accuracy for class: horse is 61.7 %
Accuracy for class: ship  is 55.2 %
Accuracy for class: truck is 50.9 %
```

Performance was actually worse with increased model complexity. Too much information may be discarded in the smaller layers.

Notebook code (also available at:
https://colab.research.google.com/drive/1b5e0wix6CxNkA0N08cvFSChJdfgo19jI?usp=sharing

```
# -*- coding: utf-8 -*-
"""Helhoski_Assignment_3

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1b5e0wix6CxNkA0N08cvFSChJdfgo19jI
"""

#@title imports
import torch
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
from sklearn.model_selection import KFold
import time
import torch.cuda as cuda

#@title Loading Dataset

# much of model setup and training from:
https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 40

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

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

#@title training
def train(model, train_loader, optimizer):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
```

```
        optimizer.zero_grad(set_to_none=True)

        data = data.to('cuda')
        target = target.to('cuda')
        output = model(data)
        output = output.to('cuda')
        criterion = nn.CrossEntropyLoss()
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

#@title Evaluation
def evalTotalSet(model):
  correct = 0
  total = 0
  model.eval()
  model.to('cuda')
  # since we're not training, we don't need to calculate the gradients for our outputs
  with torch.no_grad():
      for data in testloader:
          images, labels = data
          images = images.to('cuda')
          labels = labels.to('cuda')
          # calculate outputs by running images through the network
          outputs = model(images)
          # 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 10000 test images: {100 * correct // total} %')

def evalCategories(model):
  # prepare to count predictions for each class
  correct_pred = {classname: 0 for classname in classes}
  total_pred = {classname: 0 for classname in classes}
  model.eval()
  model.to('cuda')
  # again no gradients needed
  with torch.no_grad():
      for data in testloader:
          images, labels = data
          images = images.to('cuda')
          labels = labels.to('cuda')
          outputs = model(images)
          _, predictions = torch.max(outputs, 1)
          # collect the correct predictions for each class
```

```
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1
  # print accuracy for each class
  for classname, correct_count in correct_pred.items():
      accuracy = 100 * float(correct_count) / total_pred[classname]
      print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')

#@title Models

import torch.nn as nn
import torch.nn.functional as F

class LeNet(nn.Module): # variation of LeNet from
https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2,2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fcNew1 = nn.Linear(120, 100) # for hypothesis test (not in original model)
        self.fc2 = nn.Linear(100, 84)
        self.fcNew2 = nn.Linear(84, 32) # for hypothesis test
        self.fc3 = nn.Linear(32, 10)


    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x,1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fcNew1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fcNew2(x))
        x = self.fc3(x)
        return x

class AlexNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 96, 3)
        self.conv2 = nn.Conv2d(96, 256, 4, padding=1)
        self.conv3 = nn.Conv2d(256, 384, 3, padding=1)
        self.conv4 = nn.Conv2d(384, 384, 2, padding=1)
        self.conv5 = nn.Conv2d(384, 256, 3, padding=1)
```

```
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(256*4*4, 4096)
        self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(4096, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool(F.relu(self.conv5(x)))
        x = torch.flatten(x,1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

#@title Residual
class Residual(nn.Module):
    def __init__(self, input_channels, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.Conv2d(input_channels, num_channels, kernel_size=3, padding=1,
stride=strides)
        self.conv2 = nn.Conv2d(num_channels, num_channels, kernel_size=3, padding=1)

        if use_1x1conv:
            self.conv3 = nn.Conv2d(input_channels, num_channels, kernel_size=1, stride=strides)
        else:
            self.conv3 = None

        self.bn1 = nn.BatchNorm2d(num_channels)
        self.bn2 = nn.BatchNorm2d(num_channels)

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)

#@title ResNet-18
b1 = nn.Sequential(nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
                   nn.BatchNorm2d(64), nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

def resnet_block(input_channels, num_channels, num_residuals, first_block=False):
```

```
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(input_channels, num_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels, num_channels))
    return blk

b2 = nn.Sequential(*resnet_block(64, 64, 2, first_block=True))
b3 = nn.Sequential(*resnet_block(64, 128, 2))
b4 = nn.Sequential(*resnet_block(128, 256, 2))
b5 = nn.Sequential(*resnet_block(256, 512, 2))

resnet18Inst = nn.Sequential(b1, b2, b3, b4, b5,
                    nn.AdaptiveAvgPool2d((1,1)),
                    nn.Flatten(), nn.Linear(512, 10))

#@title training resnet

# training all 3 models and evaluating on test set
resnet18Inst.to('cuda')

batch_size = 4
epochs = 2
optimizer = optim.SGD(resnet18Inst.parameters(), lr=0.001, momentum=0.9)
#criterion = F.nll_loss()

trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True)
# method of counting parameters from:
https://discuss.pytorch.org/t/how-do-i-check-the-number-of-parameters-of-a-model/4325
print("Model size:   parameters:", sum(p.numel() for p in resnet18Inst.parameters() if
p.requires_grad))
start = time.time()
for epoch in range(epochs):
  train(resnet18Inst, trainloader, optimizer)
end = time.time()
print("Training time: ", end-start)

import numpy
import matplotlib.pyplot as plt


resnet18Inst = nn.Sequential(b1, b2, b3, b4, b5,
                    nn.AdaptiveAvgPool2d((1,1)),
                    nn.Flatten(), nn.Linear(512, 10))
PATH = './cifar_resNet.pth'
```

```
resnet18Inst.load_state_dict(torch.load(PATH, weights_only=True))

c_matrix = numpy.zeros((10,10), dtype=int)
evalCategories(resnet18Inst)
print(c_matrix)


#some code from:
https://stackoverflow.com/questions/20998083/show-the-values-in-the-grid-using-matplotlib


plt.matshow(c_matrix)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('ResNet-18 Confusion Matrix')
plt.show()

#@title training lenet
# training all 3 models and evaluating on test set
leNetInst = LeNet()
leNetInst.to('cuda')

batch_size = 4
epochs = 2
optimizer = optim.SGD(leNetInst.parameters(), lr=0.001, momentum=0.9)
#criterion = F.nll_loss()

trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True)

# method of counting parameters from:
https://discuss.pytorch.org/t/how-do-i-check-the-number-of-parameters-of-a-model/4325
print("Model size:   parameters:", sum(p.numel() for p in leNetInst.parameters() if
p.requires_grad))
start = time.time()
for epoch in range(epochs):
  train(leNetInst, trainloader, optimizer)
end = time.time()
print("Training time: ", end-start)

evalTotalSet(leNetInst)
evalCategories(leNetInst)

# training all 3 models and evaluating on test set
alexInst = AlexNet()
alexInst.to('cuda')
```

```
batch_size = 4
epochs = 2
optimizer = optim.SGD(alexInst.parameters(), lr=0.001, momentum=0.9)
#criterion = F.nll_loss()

trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)
# method of counting parameters from:
https://discuss.pytorch.org/t/how-do-i-check-the-number-of-parameters-of-a-model/4325
print("Model size:   parameters:", sum(p.numel() for p in alexInst.parameters() if
p.requires_grad))
start = time.time()
for epoch in range(epochs):
  train(alexInst, trainloader, optimizer)
end = time.time()
print("Training time: ", end-start)

PATH = './cifar_alexNet.pth'
torch.save(alexInst.state_dict(), PATH)

#@title k-fold
def run_k_fold(folds, batch_size, epochs):

  # kfold method/some code from:
https://saturncloud.io/blog/how-to-use-kfold-cross-validation-with-dataloaders-in-pytorch/#step-1
-importing-the-required-libraries
  kf = KFold(n_splits=folds)

  # Loop through each fold
  for fold, (train_idx, test_idx) in enumerate(kf.split(trainset)):
    print(f"Fold {fold + 1}")
    print("-------")

    # Define the data loaders for the current fold
    train_loader = torch.utils.data.DataLoader(dataset=trainset, batch_size=batch_size,
sampler=torch.utils.data.SubsetRandomSampler(train_idx),
                                               pin_memory=True, num_workers=2)
    test_loader = torch.utils.data.DataLoader(dataset=trainset, batch_size=batch_size,
sampler=torch.utils.data.SubsetRandomSampler(test_idx),
                                               pin_memory=True, num_workers=2)

    # have to redefined entire model inside block so that residual blocks are reset
    b1 = nn.Sequential(nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
             nn.BatchNorm2d(64), nn.ReLU(),
             nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

    def resnet_block(input_channels, num_channels, num_residuals, first_block=False):
```

```
    blk = []
    for i in range(num_residuals):
      if i == 0 and not first_block:
        blk.append(Residual(input_channels, num_channels, use_1x1conv=True, strides=2))
      else:
        blk.append(Residual(num_channels, num_channels))
    return blk

  b2 = nn.Sequential(*resnet_block(64, 64, 2, first_block=True))
  b3 = nn.Sequential(*resnet_block(64, 128, 2))
  b4 = nn.Sequential(*resnet_block(128, 256, 2))
  b5 = nn.Sequential(*resnet_block(256, 512, 2))

  model = nn.Sequential(b1, b2, b3, b4, b5,
              nn.AdaptiveAvgPool2d((1,1)),
              nn.Flatten(), nn.Linear(512, 10))
  model.to('cuda')
  optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

  for epoch in range(epochs):
    train(model, train_loader, optimizer)

  model.eval()
  test_loss = 0
  correct = 0
  with torch.no_grad():
    for data, target in test_loader:
      data = data.to('cuda')
      target = target.to('cuda')
      output = model(data)
      criterion = nn.CrossEntropyLoss()
      test_loss += criterion(output, target).item()
      pred = output.argmax(dim=1, keepdim=True)
      correct += pred.eq(target.view_as(pred)).sum().item()

  test_loss /= len(test_loader.dataset)
  accuracy = 100.0 * correct / (len(test_loader.dataset)/folds)
  print(f"Test set: Average loss: {test_loss:.4f}, Accuracy:
{correct}/({len(test_loader.dataset)/folds}) ({accuracy:.0f}%)")
  print()
```