```
!pip install ptflops
!pip install torchmetrics
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting ptflops
  Downloading ptflops-0.6.9.tar.gz (12 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: torch in /usr/local/lib/python3.9/dist-packages (from ptflops) (1.13.1+cu116)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.9/dist-packages (from torch->ptflops) (4.5.0)
Building wheels for collected packages: ptflops
  Building wheel for ptflops (setup.py) ... done
  Created wheel for ptflops: filename=ptflops-0.6.9-py3-none-any.whl size=11712 sha256=f648b1f4e1604dbb7407f6c832d39776434826d3b0081e3964adbbeb3992d54
  Stored in directory: /root/.cache/pip/wheels/86/07/9f/879035d99d7b639bbc564d23fed862a679aee7d1a2dced8c2e
Successfully built ptflops
Installing collected packages: ptflops
Successfully installed ptflops-0.6.9
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting torchmetrics
  Downloading torchmetrics-0.11.3-py3-none-any.whl (518 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 518.6/518.6 KB 4.1 MB/s eta 0:00:00
Requirement already satisfied: torch>=1.8.1 in /usr/local/lib/python3.9/dist-packages (from torchmetrics) (1.13.1+cu116)
Requirement already satisfied: numpy>=1.17.2 in /usr/local/lib/python3.9/dist-packages (from torchmetrics) (1.22.4)
Requirement already satisfied: packaging in /usr/local/lib/python3.9/dist-packages (from torchmetrics) (23.0)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.9/dist-packages (from torch>=1.8.1->torchmetrics) (4.5.0)
Installing collected packages: torchmetrics
Successfully installed torchmetrics-0.11.3
```

```
import torch
from torch import nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import datasets
import torchvision.transforms as transforms
from torchvision.transforms import ToTensor
import time
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch.nn.functional as F
import gc
from ptflops import get_model_complexity_info
from torchmetrics.classification import MulticlassConfusionMatrix
```

Problem 1

```
data_path = '../data-unversioned/ecgr4106/'
cifar10 = datasets.CIFAR10(data_path, train=True, download=True, transform=transforms.Compose([transforms.ToTensor(), transforms.Resize(size=(64, 64))]))
cifar10_val = datasets.CIFAR10(data_path, train=False, download=True, transform=transforms.Compose([transforms.ToTensor(), transforms.Resize(size=(64, 64))]
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ../data-unversioned/ecgr4106/ci
100%                                        170498071/170498071 [00:02<00:00, 81632319.27it/s]
Extracting ../data-unversioned/ecgr4106/cifar-10-python.tar.gz to ../data-unversioned/ecgr4106/
Files already downloaded and verified
```

```
def try_gpu(i=0):
    if torch.cuda.device_count() >= i+1:
        return torch.device(f'cuda:{i}')
    return torch.device('cpu')

def training_loop(n_epochs, optimizer, model, loss_fn, train_loader, val_loader, update_freq):
    train_loss_hist = []
    train_acc_hist = []
    val_acc_hist = []
    main_tic = time.perf_counter()

    for epoch in range(1, n_epochs + 1):
        tic = time.perf_counter()
        loss_train = 0.0
        correct_train = 0
        correct_val = 0
        model_argmax = []
        labels_argmax = []

        for imgs, lbls in train_loader:
            images = imgs.to(device=try_gpu())
            labels = lbls.to(device=try_gpu())
            outputs = model(images)
            del images
            loss = loss_fn(outputs, labels)
            del labels
            del outputs
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            loss_train += loss.item()
            gc.collect()
            torch.cuda.empty_cache()

        toc = time.perf_counter()

        with torch.no_grad():
            total = 0
            for imgs, lbls in train_loader:
                images = imgs.to(device=try_gpu())
                labels = lbls.to(device=try_gpu())
                outputs = model(images)
                del images
                _, predicted = torch.max(outputs, dim=1)
                del outputs
                total += labels.shape[0]
                correct_train += int((predicted == labels).sum())
                del labels
                del predicted
            train_acc = round(correct_train/total, 3)
            total = 0
            for imgs, lbls in val_loader:
                images = imgs.to(device=try_gpu())
                labels = lbls.to(device=try_gpu())
                outputs = model(images)
                del images
                _, predicted = torch.max(outputs, dim=1)
                del outputs
                if epoch == 1 or epoch == n_epochs or epoch % update_freq == 0:
                    model_argmax = model_argmax + predicted.tolist()
                    labels_argmax = labels_argmax + labels.tolist()
                total += labels.shape[0]
                correct_val += int((predicted == labels).sum())
                del labels
                del predicted
            val_acc = round(correct_val/total, 3)

        train_loss_hist.append(round(loss_train / len(train_loader), 5))
        train_acc_hist.append(train_acc)
        val_acc_hist.append(val_acc)
        label_set = set(labels_argmax)

        if epoch == 1 or epoch == n_epochs or epoch % update_freq == 0:
            print(f"Epoch {epoch}:\n\tDuration = {round(toc - tic, 3)} seconds\n\tTraining Loss: {train_loss_hist[-1]}\n\tTraining Accuracy: {train_acc_hist
            metric = MulticlassConfusionMatrix(num_classes=len(label_set))
            print(metric(torch.ByteTensor(model_argmax), torch.ByteTensor(labels_argmax)))

    main_toc = time.perf_counter()
    print(f"\nTotal Training Time = {round(main_toc - main_tic, 3)} seconds\nAverage Training Time per Epoch (including validation) = {round((main_toc - mai
    return train_loss_hist, train_acc_hist, val_acc_hist

def plot_model(title, loss_hist, train_hist, test_hist, leg_loc):
    fig, ax1 = plt.subplots()
    x = range(1, len(loss_hist)+1)
    ax1.plot(x, loss_hist, color='k')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Error')
    ax1.tick_params(axis='y')

    ax2 = ax1.twinx()
    ax2.set_ylabel('Accuracy')
    ax2.plot(x, train_hist)
    ax2.plot(x, test_hist)
    ax2.set_ylim([0, 1])
    ax1.tick_params(axis='y')

    fig.legend(["Training Loss", "Training Accuracy", "Testing Accuracy"], loc=leg_loc, bbox_to_anchor=(1, 1), bbox_transform=ax1.transAxes)
    plt.title(title)

def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.LazyBatchNorm2d())
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
    return nn.Sequential(*layers)
    return nn.Sequential(*layers)


train_loader_1 = DataLoader(cifar10, batch_size=64, shuffle=True)
val_loader_1 = DataLoader(cifar10_val, batch_size=64, shuffle=False)


class tinyVGG(nn.Module):
    def __init__(self, arch, num_classes=10):
        super(tinyVGG, self).__init__()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.conv_blks = nn.Sequential(*conv_blks, nn.Flatten())
        self.fc1 = nn.LazyLinear(128)
        self.fc2 = nn.LazyLinear(64)
        self.fc3 = nn.LazyLinear(num_classes)
        self.fc_drop = nn.Dropout(p=0.5)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.conv_blks(x)
        out = self.fc_drop(self.relu(self.fc1(out)))
        out = self.fc_drop(self.relu(self.fc2(out)))
        out = self.fc3(out)
        return out

model_0 = tinyVGG(arch=((1, 64), (1, 128))).to(device=try_gpu())
optimizer_0 = optim.SGD(model_0.parameters(), lr=0.08)
model_0.eval()
```

```
/usr/local/lib/python3.9/dist-packages/torch/nn/modules/lazy.py:180: UserWarning: Lazy modules are a new feature under heavy development so changes to
  warnings.warn('Lazy modules are a new feature under heavy development '
tinyVGG(
  (conv_blks): Sequential(
    (0): Sequential(
      (0): LazyConv2d(0, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
      (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (1): Sequential(
      (0): LazyConv2d(0, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
      (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (2): Flatten(start_dim=1, end_dim=-1)
  )
  (fc1): LazyLinear(in_features=0, out_features=128, bias=True)
  (fc2): LazyLinear(in_features=0, out_features=64, bias=True)
  (fc3): LazyLinear(in_features=0, out_features=10, bias=True)
  (fc_drop): Dropout(p=0.5, inplace=False)
  (relu): ReLU()
)
```

```
torch.cuda.empty_cache()
gc.collect()
t_loss_hist_0, t_acc_hist_0, v_acc_hist_0= training_loop(10,
                                                        optimizer_0,
                                                        model_0,
                                                        nn.CrossEntropyLoss(),
                                                        train_loader_1,
                                                        val_loader_1,
                                                        1)
title_0 = "Figure 0 - Loss and Accuracy per Epoch for tinyVGG"
plot_model(title_0, t_loss_hist_0, t_acc_hist_0, v_acc_hist_0, 'upper right')
```

```
Epoch 1:
        Duration = 62.585 seconds
        Training Loss: 2.07869
        Training Accuracy: 0.324
        Validation Accuracy: 0.33
tensor([[699,  94,   0,  33,   7,   7,  30,  26,   0, 104],
        [139, 536,   1,  49,   6,  11,  57,  14,   0, 187],
        [243, 122,  21, 135,  98,  38, 242,  55,   0,  46],
        [123, 109,   3, 318,  31,  89, 183,  38,   0, 106],
        [118,  65,   7, 133, 200,  21, 339,  70,   0,  47],
        [128,  88,   6, 251,  47, 236, 148,  33,   0,  63],
        [ 33,  78,   2, 159,  56,  24, 553,  13,   0,  82],
        [132, 117,   1, 118, 109,  30,  82, 236,   0, 175],
        [574, 160,   2,  54,   0,  24,   5,  11,   1, 169],
        [153, 232,   0,  35,   3,  11,  43,  27,   0, 496]])
Epoch 2:
        Duration = 55.115 seconds
        Training Loss: 1.74078
        Training Accuracy: 0.444
        Validation Accuracy: 0.441
tensor([[432,  66,   8,  38,   9,   6,  20,  43, 247, 131],
        [ 15, 588,   7,  28,   2,  11,  11,  24,  70, 244],
        [131,  68, 148, 133,  97,  43, 154, 112,  70,  44],
        [ 47,  53,  31, 376,  22, 100, 151,  66,  63,  91],
        [ 59,  28,  74, 104, 280,  32, 187, 129,  73,  34],
        [ 35,  43,  41, 250,  27, 252,  97, 118,  91,  46],
        [  7,  38,  38, 135,  60,  32, 563,  29,  39,  59],
        [ 48,  48,  16,  93,  47,  35,  34, 519,  35, 125],
        [ 55,  79,   0,  42,   3,   7,   8,  12, 657, 137],
        [ 14, 235,   2,  22,   5,   7,  16,  26,  78, 595]])
Epoch 3:
        Duration = 53.337 seconds
        Training Loss: 1.53197
        Training Accuracy: 0.495
        Validation Accuracy: 0.493
tensor([[586,  55,  56,  38,   6,   7,   4,  26, 177,  45],
        [ 44, 633,   6,  25,   2,  10,   3,  15,  61, 201],
        [ 80,  39, 390, 159,  63,  72,  49,  90,  31,  27],
        [ 45,  28,  82, 557,  19, 128,  24,  47,  31,  39],
        [ 62,  17, 167, 145, 292,  69,  57, 139,  37,  15],
        [ 32,  23,  78, 341,  26, 348,   9,  90,  30,  23],
        [ 12,  31,  85, 313,  91,  73, 322,  32,  12,  29],
        [ 56,  30,  49, 145,  26,  61,   2, 558,  21,  52],
        [107,  73,  11,  44,   6,   8,   1,  11, 685,  54],
        [ 55, 221,  14,  33,   4,   3,   2,  30,  80, 558]])
Epoch 4:
        Duration = 54.212 seconds
        Training Loss: 1.39665
        Training Accuracy: 0.549
        Validation Accuracy: 0.536
tensor([[548,   8,  83,  22,  55,  28,  34,  43, 138,  41],
        [ 47, 554,  30,  17,  13,  25,  24,  45,  54, 191],
        [ 42,   3, 352,  38, 253, 121, 109,  63,  12,   7],
        [ 11,   2,  75, 246, 112, 350, 137,  52,   6,   9],
        [ 28,   1,  93,  25, 573,  77, 125,  63,  14,   1],
        [  1,   2,  77,  81, 119, 590,  57,  59,   8,   6],
        [  3,   3,  58,  44, 146,  74, 650,   8,   6,   8],
        [ 12,   2,  50,  30, 134, 123,  28, 601,   6,  14],
        [114,  33,  18,  19,  36,  42,  22,  20, 637,  59],
        [ 41, 111,  20,  30,  17,  25,  35,  63,  52, 606]])
Epoch 5:
        Duration = 53.906 seconds
        Training Loss: 1.26966
        Training Accuracy: 0.592
        Validation Accuracy: 0.558
tensor([[534,  13, 134,  47,  25,  19,  50,  11, 123,  44],
        [ 34, 593,  15,  28,   6,   2,  35,  14,  52, 221],
        [ 38,   3, 533,  94,  84,  71, 123,  30,  14,  10],
        [ 11,   6,  96, 487,  54, 159, 148,  14,   9,  16],
        [ 23,   4, 236,  76, 384,  50, 151,  56,  14,   6],
        [  8,   1, 104, 248,  53, 447,  80,  35,  14,  10],
        [  7,   5,  71,  81,  53,  31, 737,   5,   4,   6],
        [ 12,   2,  74,  93,  84,  96,  65, 537,   9,  28],
        [ 90,  51,  26,  41,  14,  15,  23,   7, 670,  63],
        [ 33,  97,  24,  40,   5,   7,  50,  37,  49, 658]])
Epoch 6:
        Duration = 53.128 seconds
        Training Loss: 1.16226
        Training Accuracy: 0.635
        Validation Accuracy: 0.591
tensor([[660,  11,  63,  20,  17,   5,  24,  12, 167,  21],
        [ 36, 706,  16,  23,   7,   1,  10,  15,  99,  87],
        [ 68,   9, 470,  63, 177,  36,  91,  49,  31,   6],
        [ 22,   8, 104, 485,  97,  73, 127,  46,  25,  13],
        [ 33,   3, 115,  61, 580,  11,  99,  75,  22,   1],
        [ 15,   6, 104, 296,  94, 306,  72,  73,  26,   8],
        [  9,   4,  63,  73, 115,   9, 692,  14,  14,   7],
        [ 20,   3,  53,  75, 102,  34,  33, 654,  12,  14],
        [ 82,  33,  14,  14,  16,   3,   9,   4, 808,  17],
        [ 58, 155,  21,  27,  10,   0,  17,  41, 120, 551]])
Epoch 7:
```

```
            Duration = 53.04 seconds
            Training Loss: 1.07017
            Training Accuracy: 0.671
            Validation Accuracy: 0.602
    tensor([[766,  29,  48,  32,  14,   7,  24,  18,  31,  31],
            [ 50, 772,  19,  21,   4,   5,   7,  11,   8, 103],
            [ 86,   8, 565,  79,  72,  46,  92,  37,   5,  10],
            [ 30,   9, 119, 450,  64, 163, 110,  34,   3,  18],
            [ 43,   5, 208,  70, 450,  25, 115,  77,   5,   2],
            [ 13,   7,  99, 229,  55, 467,  58,  62,   3,   7],
            [ 12,   8,  91,  88,  53,  20, 707,  12,   2,   7],
            [ 27,   1,  64,  77,  53,  57,  24, 669,   1,  27],
            [215,  80,  28,  43,  17,  15,  13,   4, 546,  39],
            [ 66, 179,  18,  27,   8,  10,  12,  36,  19, 625]])
    Epoch 8:
            Duration = 54.059 seconds
            Training Loss: 0.97531
            Training Accuracy: 0.689
            Validation Accuracy: 0.608
    tensor([[741,  13,  47,  10,   1,   5,  18,   5, 135,  25],
            [ 52, 673,  15,  12,   2,   7,   3,   1,  82, 153],
            [ 93,   8, 618,  66,  34,  44,  65,  22,  35,  15],
            [ 41,  12, 139, 451,  30, 131, 100,  19,  50,  27],
            [ 57,   7, 266,  80, 360,  28, 118,  51,  29,   4],
            [ 34,   7, 114, 232,  34, 454,  58,  25,  29,  13],
            [ 13,  11, 111,  83,  21,  19, 712,   5,  13,  12],
            [ 46,   4,  88,  76,  62,  69,  29, 568,  16,  42],
            [ 92,  33,   9,   3,   2,   4,   6,   2, 819,  30],
            [ 76,  99,  15,   9,   2,   6,  14,  13,  87, 679]])
    Epoch 9:
            Duration = 53.973 seconds
            Training Loss: 0.89546
            Training Accuracy: 0.742
            Validation Accuracy: 0.635
    tensor([[685,  19,  47,  12,   9,   4,  22,   9, 164,  29],
            [ 39, 758,  11,  11,   3,   4,   6,   4,  71,  93],
            [ 79,  10, 523,  43, 127,  40,  97,  36,  35,  10],
            [ 26,  14,  99, 420,  87, 126, 138,  39,  36,  15],
            [ 39,   6, 108,  49, 588,  12, 111,  62,  22,   3],
            [ 26,   4,  92, 189,  89, 430,  77,  56,  26,  11],
            [  9,  13,  54,  47,  68,  10, 770,   8,  11,  10],
            [ 29,   4,  51,  41,  74,  43,  27, 693,  18,  20],
            [ 60,  37,   8,   7,   6,   5,  14,   3, 838,  22],
            [ 38, 150,  10,  15,   3,   5,  18,  26,  90, 645]])
    Epoch 10:
            Duration = 53.07 seconds
            Training Loss: 0.8161
            Training Accuracy: 0.774
            Validation Accuracy: 0.65
    tensor([[752,  15,  40,  19,  24,   8,  29,  22,  64,  27],
            [ 46, 738,  10,  16,  10,   7,  14,  10,  41, 108],
            [ 80,   5, 446,  66, 177,  59,  97,  53,  10,   7],
            [ 18,   5,  58, 484,  96, 134, 130,  47,  14,  14],
            [ 35,   2,  55,  55, 652,  21,  91,  82,   6,   1],
            [ 13,   2,  46, 202,  87, 500,  74,  66,   5,   5],
            [  6,   5,  27,  55, 105,  16, 761,  15,   6,   4],
            [ 20,   2,  26,  37,  77,  53,  24, 746,   4,  11],
            [102,  41,  13,  22,  14,  11,  23,   5, 742,  27],
            [ 59, 116,  12,  14,   8,   6,  25,  33,  46, 681]])

    Total Training Time = 703.778 seconds
```

```
class VGG_11(nn.Module):
    def __init__(self, arch, num_classes=10):
        super(VGG_11, self).__init__()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.conv_blks = nn.Sequential(*conv_blks, nn.Flatten())
        self.fc1 = nn.LazyLinear(4096)
        self.fc2 = nn.LazyLinear(4096)
        self.fc3 = nn.LazyLinear(num_classes)
        self.fc_drop = nn.Dropout(p=0.5)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.conv_blks(x)
        out = self.fc_drop(self.relu(self.fc1(out)))
        out = self.fc_drop(self.relu(self.fc2(out)))
        out = self.fc3(out)
        return out

model_1 = VGG_11(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).to(device=try_gpu())
optimizer_1 = optim.SGD(model_1.parameters(), lr=0.1)
model_1.eval()
```

```
    VGG_11(
      (conv_blks): Sequential(
        (0): Sequential(
          (0): LazyConv2d(0, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): ReLU()
          (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        )
        (1): Sequential(
          (0): LazyConv2d(0, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): ReLU()
          (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        )
        (2): Sequential(
          (0): LazyConv2d(0, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): ReLU()
          (3): LazyConv2d(0, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (4): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (5): ReLU()
          (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        )
        (3): Sequential(
          (0): LazyConv2d(0, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): ReLU()
          (3): LazyConv2d(0, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (4): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (5): ReLU()
          (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        )
        (4): Sequential(
          (0): LazyConv2d(0, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): ReLU()
          (3): LazyConv2d(0, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (4): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (5): ReLU()
          (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        )
        (5): Flatten(start_dim=1, end_dim=-1)
      )
      (fc1): LazyLinear(in_features=0, out_features=4096, bias=True)
      (fc2): LazyLinear(in_features=0, out_features=4096, bias=True)
      (fc3): LazyLinear(in_features=0, out_features=10, bias=True)
      (fc_drop): Dropout(p=0.5, inplace=False)
      (relu): ReLU()
    )
```

```
torch.cuda.empty_cache()
gc.collect()
t_loss_hist_1, t_acc_hist_1, v_acc_hist_1 = training_loop(10,
                                                        optimizer_1,
                                                        model_1,
                                                        nn.CrossEntropyLoss(),
                                                        train_loader_1,
                                                        val_loader_1,
                                                        2)
title_1 = "Figure 1 - Loss and Accuracy per Epoch for VGG-11"
plot_model(title_1, t_loss_hist_1, t_acc_hist_1, v_acc_hist_1, 'upper right')
```

```
    Epoch 1:
            Duration = 78.519 seconds
            Training Loss: 2.30272
            Training Accuracy: 0.1
            Validation Accuracy: 0.1
    tensor([[   0, 1000,    0,    0,    0,    0,    0,    0,    0,    0],
            [   0, 1000,    0,    0,    0,    0,    0,    0,    0,    0],
            [   0, 1000,    0,    0,    0,    0,    0,    0,    0,    0],
            [   0, 1000,    0,    0,    0,    0,    0,    0,    0,    0],
            [   0, 1000,    0,    0,    0,    0,    0,    0,    0,    0],
            [   0, 1000,    0,    0,    0,    0,    0,    0,    0,    0],
            [   0, 1000,    0,    0,    0,    0,    0,    0,    0,    0],
            [   0, 1000,    0,    0,    0,    0,    0,    0,    0,    0],
            [   0, 1000,    0,    0,    0,    0,    0,    0,    0,    0],
            [   0, 1000,    0,    0,    0,    0,    0,    0,    0,    0]])
    Epoch 2:
            Duration = 78.586 seconds
            Training Loss: 2.30284
            Training Accuracy: 0.1
            Validation Accuracy: 0.1
    tensor([[   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
            [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
            [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
            [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
            [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
            [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
            [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
            [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
            [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
            [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0]])
    Epoch 4:
            Duration = 78.412 seconds
            Training Loss: 2.30283
            Training Accuracy: 0.103
            Validation Accuracy: 0.103
    tensor([[   0,    0,    0,    0,    0,  984,    0,    0,    0,   16],
            [   0,    0,    0,    0,    0,  978,    0,    0,    0,   22],
            [   0,    0,    0,    0,    0, 1000,    0,    0,    0,    0],
            [   0,    0,    0,    0,    0,  998,    0,    0,    0,    2],
            [   0,    0,    0,    0,    0,  999,    0,    0,    0,    1],
            [   0,    0,    0,    0,    0, 1000,    0,    0,    0,    0],
            [   0,    0,    0,    0,    0,  999,    0,    0,    0,    1],
            [   0,    0,    0,    0,    0,  990,    0,    0,    0,   10],
            [   0,    0,    0,    0,    0,  963,    0,    0,    0,   37],
            [   0,    0,    0,    0,    0,  970,    0,    0,    0,   30]])
    Epoch 6:
            Duration = 76.961 seconds
            Training Loss: 2.30278
            Training Accuracy: 0.1
            Validation Accuracy: 0.1
    tensor([[   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0]])
    Epoch 8:
            Duration = 77.631 seconds
            Training Loss: 2.30272
            Training Accuracy: 0.1
            Validation Accuracy: 0.1
    tensor([[   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0],
            [   0,    0,    0, 1000,    0,    0,    0,    0,    0,    0]])
    Epoch 10:
            Duration = 77.327 seconds
            Training Loss: 2.30252
            Training Accuracy: 0.112
            Validation Accuracy: 0.111
```

```
class VGG_16(nn.Module):
    def __init__(self, arch, num_classes=10):
        super(VGG_16, self).__init__()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.conv_blks = nn.Sequential(*conv_blks, nn.Flatten())
        self.fc1 = nn.LazyLinear(4096)
        self.fc2 = nn.LazyLinear(4096)
        self.fc3 = nn.LazyLinear(num_classes)
        self.fc_drop = nn.Dropout(p=0.5)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.conv_blks(x)
        out = self.fc_drop(self.relu(self.fc1(out)))
        out = self.fc_drop(self.relu(self.fc2(out)))
        out = self.fc3(out)
        return out

model_2 = VGG_16(arch=((2, 64), (2, 128), (3, 256), (3, 512), (3, 512))).to(device=try_gpu())
optimizer_2 = optim.SGD(model_2.parameters(), lr=0.1)
model_2.eval()

torch.cuda.empty_cache()
gc.collect()
t_loss_hist_2, t_acc_hist_2, v_acc_hist_2 = training_loop(3,
                                                        optimizer_2,
                                                        model_2,
                                                        nn.CrossEntropyLoss(),
                                                        train_loader_1,
                                                        val_loader_1,
                                                        1)
title_2 = "Figure 2 - Loss and Accuracy per Epoch for VGG-16"
plot_model(title_2, t_loss_hist_2, t_acc_hist_2, v_acc_hist_2, 'upper right')


class VGG_19(nn.Module):
    def __init__(self, arch, num_classes=10):
        super(VGG_19, self).__init__()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.conv_blks = nn.Sequential(*conv_blks, nn.Flatten())
        self.fc1 = nn.LazyLinear(4096)
        self.fc2 = nn.LazyLinear(4096)
        self.fc3 = nn.LazyLinear(num_classes)
        self.fc_drop = nn.Dropout(p=0.5)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.conv_blks(x)
        out = self.fc_drop(self.relu(self.fc1(out)))
        out = self.fc_drop(self.relu(self.fc2(out)))
        out = self.fc3(out)
        return out

model_3 = VGG_19(arch=((2, 64), (2, 128), (4, 256), (4, 512), (4, 512))).to(device=try_gpu())
optimizer_3 = optim.SGD(model_3.parameters(), lr=0.1)
model_3.eval()


torch.cuda.empty_cache()
gc.collect()
t_loss_hist_3, t_acc_hist_3, v_acc_hist_3 = training_loop(3,
                                                        optimizer_3,
                                                        model_3,
                                                        nn.CrossEntropyLoss(),
                                                        train_loader_1,
                                                        val_loader_1,
                                                        1)
title_3 = "Figure 3 - Loss and Accuracy per Epoch for VGG-19"
plot_model(title_3, t_loss_hist_3, t_acc_hist_3, v_acc_hist_3, 'upper right')
```

Problem 2

```
class Inception(nn.Module):
    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        self.b1_1 = nn.LazyConv2d(c1, kernel_size=1)
        self.b2_1 = nn.LazyConv2d(c2[0], kernel_size=1)
        self.b2_2 = nn.LazyConv2d(c2[1], kernel_size=3, padding=1)
        self.b3_1 = nn.LazyConv2d(c3[0], kernel_size=1)
        self.b3_2 = nn.LazyConv2d(c3[1], kernel_size=5, padding=2)
        self.b4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.b4_2 = nn.LazyConv2d(c4, kernel_size=1)

    def forward(self, x):
        b1 = F.relu(self.b1_1(x))
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))
        b4 = F.relu(self.b4_2(self.b4_1(x)))
        return torch.cat((b1, b2, b3, b4), dim=1)


class GoogLeNet(nn.Module):
    def __init__(self, num_classes=10):
        super(GoogLeNet, self).__init__()
        self.stem = nn.Sequential(nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                                 nn.ReLU(),
                                 nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
                                 nn.LazyConv2d(64, kernel_size=1),
                                 nn.ReLU(),
                                 nn.LazyConv2d(192, kernel_size=3, padding=1),
                                 nn.ReLU(),
                                 nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.body1 = nn.Sequential(Inception(64, (96, 128), (16, 32), 32),
                                  Inception(128, (128, 192), (32, 96), 64),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.body2 = nn.Sequential(Inception(192, (96, 208), (16, 48), 64),
                                  Inception(160, (112, 124), (24, 64), 64),
                                  Inception(128, (128, 256), (24, 64), 64),
                                  Inception(112, (144, 288), (32, 64), 64),
                                  Inception(256, (160, 320), (32, 128), 128),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.body3 = nn.Sequential(Inception(256, (160, 320), (32, 128), 128),
                                  Inception(384, (192, 384), (48, 128), 128),
                                  nn.AdaptiveAvgPool2d((1,1)),
                                  nn.Flatten())
        self.fc = nn.LazyLinear(num_classes)


    def forward(self, x):
        out = self.stem(x)
        out = self.body1(out)
        out = self.body2(out)
        out = self.body3(out)
        out = self.fc(out)
        return out

model_4 = GoogLeNet().to(device=try_gpu())
optimizer_4 = optim.SGD(model_4.parameters(), lr=0.1)
model_4.eval()
```

```
          )
          (1): Inception(
            (b1_1): LazyConv2d(0, 384, kernel_size=(1, 1), stride=(1, 1))
            (b2_1): LazyConv2d(0, 192, kernel_size=(1, 1), stride=(1, 1))
            (b2_2): LazyConv2d(0, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (b3_1): LazyConv2d(0, 48, kernel_size=(1, 1), stride=(1, 1))
            (b3_2): LazyConv2d(0, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
            (b4_1): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
            (b4_2): LazyConv2d(0, 128, kernel_size=(1, 1), stride=(1, 1))
          )
          (2): AdaptiveAvgPool2d(output_size=(1, 1))
          (3): Flatten(start_dim=1, end_dim=-1)
        )
        (fc): LazyLinear(in_features=0, out_features=10, bias=True)
      )
```

```
torch.cuda.empty_cache()
gc.collect()
t_loss_hist_4, t_acc_hist_4, v_acc_hist_4 = training_loop(3,
                                                        optimizer_4,
                                                        model_4,
                                                        nn.CrossEntropyLoss(),
                                                        train_loader_1,
                                                        val_loader_1,
                                                        1)
title_4 = "Figure 4 - Loss and Accuracy per Epoch for GoogLeNet"
plot_model(title_4, t_loss_hist_4, t_acc_hist_4, v_acc_hist_4, 'upper right')
```

```
     Epoch 1:
             Duration = 60.74 seconds
             Training Loss: 2.30283
```

```
class AltGoogLeNet(nn.Module):
    def __init__(self, num_classes=10):
        super(GoogLeNet, self).__init__()
        self.stem = nn.Sequential(nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                                  nn.LazyBatchNorm2d(),
                                  nn.ReLU(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
                                  nn.LazyConv2d(64, kernel_size=1),
                                  nn.LazyBatchNorm2d(),
                                  nn.ReLU(),
                                  nn.LazyConv2d(192, kernel_size=3, padding=1),
                                  nn.LazyBatchNorm2d(),
                                  nn.ReLU(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.body1 = nn.Sequential(Inception(64, (96, 128), (16, 32), 32),
                                  Inception(128, (128, 192), (32, 96), 64),
                                  nn.LazyBatchNorm2d(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.body2 = nn.Sequential(Inception(192, (96, 208), (16, 48), 64),
                                  Inception(160, (112, 124), (24, 64), 64),
                                  Inception(128, (128, 256), (24, 64), 64),
                                  Inception(112, (144, 288), (32, 64), 64),
                                  Inception(256, (160, 320), (32, 128), 128),
                                  nn.LazyBatchNorm2d(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.body3 = nn.Sequential(Inception(256, (160, 320), (32, 128), 128),
                                  Inception(384, (192, 384), (48, 128), 128),
                                  nn.LazyBatchNorm2d(),
                                  nn.AdaptiveAvgPool2d((1,1)),
                                  nn.Flatten())
        self.fc = nn.LazyLinear(num_classes)


    def forward(self, x):
        out = self.stem(x)
        out = self.body1(out)
        out = self.body2(out)
        out = self.body3(out)
        out = self.fc(out)
        return out

model_5 = GoogLeNet().to(device=try_gpu())
optimizer_5 = optim.SGD(model_5.parameters(), lr=0.1)
model_5.eval()
```

```
        (0): Inception(
          (b1_1): LazyConv2d(0, 256, kernel_size=(1, 1), stride=(1, 1))
          (b2_1): LazyConv2d(0, 160, kernel_size=(1, 1), stride=(1, 1))
          (b2_2): LazyConv2d(0, 320, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (b3_1): LazyConv2d(0, 32, kernel_size=(1, 1), stride=(1, 1))
          (b3_2): LazyConv2d(0, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
          (b4_1): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
          (b4_2): LazyConv2d(0, 128, kernel_size=(1, 1), stride=(1, 1))
        )
        (1): Inception(
          (b1_1): LazyConv2d(0, 384, kernel_size=(1, 1), stride=(1, 1))
          (b2_1): LazyConv2d(0, 192, kernel_size=(1, 1), stride=(1, 1))
          (b2_2): LazyConv2d(0, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (b3_1): LazyConv2d(0, 48, kernel_size=(1, 1), stride=(1, 1))
          (b3_2): LazyConv2d(0, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
          (b4_1): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
          (b4_2): LazyConv2d(0, 128, kernel_size=(1, 1), stride=(1, 1))
        )
        (2): AdaptiveAvgPool2d(output_size=(1, 1))
        (3): Flatten(start_dim=1, end_dim=-1)
```

```
torch.cuda.empty_cache()
gc.collect()
t_loss_hist_5, t_acc_hist_5, v_acc_hist_5 = training_loop(10,
                                                        optimizer_5,
                                                        model_5,
                                                        nn.CrossEntropyLoss(),
                                                        train_loader_1,
                                                        val_loader_1,
                                                        2)
title_5 = "Figure 5 - Loss and Accuracy per Epoch for GoogLeNet with Batch Norm"
plot_model(title_5, t_loss_hist_5, t_acc_hist_5, v_acc_hist_5, 'upper right')
```

```
Epoch 1:
        Duration = 59.662 seconds
        Training Loss: 2.30278
        Training Accuracy: 0.1
        Validation Accuracy: 0.1
tensor([[   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
        [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
        [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
        [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
        [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
        [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
        [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
        [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
        [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0],
        [   0,    0, 1000,    0,    0,    0,    0,    0,    0,    0]])
Epoch 2:
        Duration = 59.788 seconds
        Training Loss: 2.3028
        Training Accuracy: 0.1
        Validation Accuracy: 0.1
tensor([[   0,    0,    0,    0,    0,    0,    0, 1000,    0,    0],
        [   0,    0,    0,    0,    0,    0,    0, 1000,    0,    0],
        [   0,    0,    0,    0,    0,    0,    0, 1000,    0,    0],
        [   0,    0,    0,    0,    0,    0,    0, 1000,    0,    0],
        [   0,    0,    0,    0,    0,    0,    0, 1000,    0,    0],
```

Problem 3

```
class Residual(nn.Module):
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1, stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1, stride=strides)
        else:
            self.conv3 = None
        self.bn = nn.LazyBatchNorm2d()

    def forward(self, x):
        y = F.relu(self.bn(self.conv1(x)))
        y = self.bn(self.conv2(y))
        if self.conv3:
            x = self.bn(self.conv3(x))
        y += x
        return F.relu(y)

def block(num_residuals, num_channels, first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels))
    return nn.Sequential(*blk)

     ---------------------------------------------------------------------------

class ResNet(nn.Module):
    def __init__(self, arch, num_classes=10):
        super(ResNet, self).__init__()
        self.stem = nn.Sequential(nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                                  nn.LazyBatchNorm2d(),
                                  nn.ReLU(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        blks = []
        for i, b in enumerate(arch):
            blks.append(block(*b, first_block=(i==0)))
        self.blks = nn.Sequential(*blks)
        self.head = nn.Sequential(nn.AdaptiveAvgPool2d((1, 1)),
                                     nn.Flatten(),
                                     nn.LazyLinear(num_classes))

    def forward(self, x):
        out = self.stem(x)
        out = self.blks(out)
        out = self.head(out)
        return out

model_6 = ResNet(arch=((2, 64), (2, 128), (2, 256), (2,512))).to(device=try_gpu())
optimizer_6 = optim.SGD(model_6.parameters(), lr=0.01)
model_6.eval()
```

```
        (conv2): LazyConv2d(0, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (bn): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): Residual(
        (conv1): LazyConv2d(0, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (conv2): LazyConv2d(0, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (bn): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Sequential(
      (0): Residual(
        (conv1): LazyConv2d(0, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (conv2): LazyConv2d(0, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (conv3): LazyConv2d(0, 128, kernel_size=(1, 1), stride=(2, 2))
        (bn): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): Residual(
        (conv1): LazyConv2d(0, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (conv2): LazyConv2d(0, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (bn): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (2): Sequential(
      (0): Residual(
        (conv1): LazyConv2d(0, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (conv2): LazyConv2d(0, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (conv3): LazyConv2d(0, 256, kernel_size=(1, 1), stride=(2, 2))
        (bn): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): Residual(
        (conv1): LazyConv2d(0, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (conv2): LazyConv2d(0, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (bn): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (3): Sequential(
      (0): Residual(
        (conv1): LazyConv2d(0, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (conv2): LazyConv2d(0, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (conv3): LazyConv2d(0, 512, kernel_size=(1, 1), stride=(2, 2))
        (bn): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): Residual(
        (conv1): LazyConv2d(0, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (conv2): LazyConv2d(0, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (bn): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
  (head): Sequential(
    (0): AdaptiveAvgPool2d(output_size=(1, 1))
    (1): Flatten(start_dim=1, end_dim=-1)
    (2): LazyLinear(in_features=0, out_features=10, bias=True)
  )
```

```
torch.cuda.empty_cache()
gc.collect()
t_loss_hist_6, t_acc_hist_6, v_acc_hist_6 = training_loop(10,
                                                        optimizer_6,
                                                        model_6,
                                                        nn.CrossEntropyLoss(),
                                                        train_loader_1,
                                                        val_loader_1,
                                                        2)
title_6 = "Figure 6 - Loss and Accuracy per Epoch for ResNet"
plot_model(title_6, t_loss_hist_6, t_acc_hist_6, v_acc_hist_6, 'upper right')
```

```
Epoch 1:
        Duration = 59.61 seconds
        Training Loss: 2.30173
        Training Accuracy: 0.149
        Validation Accuracy: 0.153
tensor([[637,   0,   0,   0, 363,   0,   0,   0,   0,   0],
        [325,   0,   0,   0, 675,   0,   0,   0,   0,   0],
        [232,   0,   0,   0, 768,   0,   0,   0,   0,   0],
        [177,   0,   0,   0, 823,   0,   0,   0,   0,   0],
        [109,   0,   0,   0, 891,   0,   0,   0,   0,   0],
        [164,   0,   0,   0, 836,   0,   0,   0,   0,   0],
        [112,   0,   0,   0, 888,   0,   0,   0,   0,   0],
        [268,   0,   0,   0, 732,   0,   0,   0,   0,   0],
        [597,   0,   0,   0, 403,   0,   0,   0,   0,   0],
        [522,   0,   0,   0, 478,   0,   0,   0,   0,   0]])
Epoch 2:
        Duration = 57.799 seconds
        Training Loss: 2.29885
        Training Accuracy: 0.126
        Validation Accuracy: 0.129
tensor([[972,   0,   0,   0,  28,   0,   0,   0,   0,   0],
        [877,   0,   0,   0, 121,   0,   0,   0,   0,   2],
        [797,   0,   0,   0, 203,   0,   0,   0,   0,   0],
        [762,   0,   0,   0, 237,   0,   0,   0,   0,   1],
        [684,   0,   0,   0, 315,   0,   0,   0,   0,   1],
        [831,   0,   0,   0, 167,   0,   0,   0,   0,   2],
        [659,   0,   0,   0, 341,   0,   0,   0,   0,   0],
        [884,   0,   0,   0, 116,   0,   0,   0,   0,   0],
        [962,   0,   0,   0,  38,   0,   0,   0,   0,   0],
        [955,   0,   0,   0,  45,   0,   0,   0,   0,   0]])
Epoch 4:
        Duration = 59.47 seconds
        Training Loss: 2.28171
        Training Accuracy: 0.154
        Validation Accuracy: 0.156
tensor([[924,   0,   0,   0,   7,   3,  38,   0,   1,  27],
        [734,   0,   0,   0,  33,  12, 145,   0,  12,  64],
        [555,   0,   0,   0,  85,  14, 237,   0,   2, 107],
        [555,   0,   0,   1,  64,  34, 241,   0,   5, 100],
        [416,   0,   0,   1, 104,  16, 344,   0,   0, 119],
        [620,   0,   0,   1,  50,  48, 186,   0,   2,  93],
        [359,   0,   0,   0, 100,  16, 404,   0,   1, 120],
        [687,   0,   0,   0,  68,  13, 132,   0,   2,  98],
        [939,   0,   0,   0,   5,   5,  36,   0,   5,  10],
        [842,   0,   0,   0,  27,   7,  46,   0,   4,  74]])
Epoch 6:
        Duration = 58.969 seconds
        Training Loss: 2.12357
        Training Accuracy: 0.208
        Validation Accuracy: 0.208
tensor([[748,  36,   1,   0,   0,  27,  39,  15,  90,  44],
        [363, 104,   3,   0,  16,  91, 115,  30, 191,  87],
        [239,  56,  13,   0,  11, 149, 380,  21,  56,  75],
        [167,  70,   8,   0,  35, 207, 300,  45,  54, 114],
        [117,  37,  10,   0,  18, 116, 560,  26,  41,  75],
        [153, 125,   9,   0,  18, 202, 288,  29,  97,  79],
        [ 63,  24,   1,   0,  26, 117, 635,  38,  17,  79],
        [142,  66,   4,   0,  22, 114, 219,  78, 119, 236],
        [729,  35,   2,   0,   3,  26,  16,  16, 116,  57],
        [415,  33,   4,   0,  13,  42,  71,  39, 214, 169]])
Epoch 8:
        Duration = 58.231 seconds
        Training Loss: 2.03023
        Training Accuracy: 0.25
        Validation Accuracy: 0.254
tensor([[ 49, 204,  14,  22,   0,  73,  56,  46, 332, 204],
        [  1, 377,   0,   6,   0, 169, 138, 106,  90, 113],
        [ 21,  67,  19,  37,   0, 148, 506,  92,  56,  54],
        [  3,  63,   5,  49,   0, 229, 455, 149,   7,  40],
        [  2,  36,   3,  23,   0,  94, 681,  90,  26,  45],
        [  4,  69,   3,  32,   0, 310, 443, 102,  10,  27],
        [  1,  10,   0,  33,   0,  67, 766, 101,   4,  18],
        [  2,  60,   3,  20,   0, 130, 312, 317,  12, 144],
        [  7, 238,   6,  13,   0,  85,  27,  81, 296, 247],
        [  0, 174,   0,  15,   0,  69, 104, 202,  84, 352]])
Epoch 10:
        Duration = 58.95 seconds
        Training Loss: 1.95409
        Training Accuracy: 0.289
        Validation Accuracy: 0.295
tensor([[400, 142,   6,  15,   1,  94,  44,  35, 127, 136],
        [ 32, 445,   3,   6,   0, 136,  94,  79, 103, 102],
        [103,  64,  13,  22,   0, 238, 426,  74,  16,  44],
        [ 26,  53,   8,  32,   1, 362, 351, 117,   8,  42],
        [ 28,  29,   4,  11,   0, 163, 625,  88,  18,  34],
        [ 19,  72,   6,  14,   0, 422, 347,  81,  11,  28],
        [  9,  19,   3,  21,   0, 149, 694,  88,   1,  16],
        [ 18,  89,   3,  13,   0, 174, 254, 322,  11, 116],
```

```
class Residual26(nn.Module):
    def __init__(self, in_channels, out_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(in_channels, kernel_size=1)
        self.conv2 = nn.LazyConv2d(in_channels, kernel_size=3, padding=1)
        self.conv3 = nn.LazyConv2d(out_channels, kernel_size=1)
        if use_1x1conv:
            self.conv4 = nn.LazyConv2d(out_channels, kernel_size=1, stride=strides)
        else:
            self.conv4 = None
        self.bn = nn.LazyBatchNorm2d()

    def forward(self, x):
        y = F.relu(self.bn(self.conv1(x)))
        y = F.relu(self.bn(self.conv2(y)))
        y = self.bn(self.conv3(y))
        if self.conv4:
            x = self.bn(self.conv4(x))
        y += x
        return F.relu(y)

def block26(num_residuals, in_channels, out_channels, first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual26(in_channels, out_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Residual26(in_channels, out_channels))
    return nn.Sequential(*blk)


class ResNet26(nn.Module):
    def __init__(self, num_classes=10):
        super(ResNet26, self).__init__()
        self.stem = nn.Sequential(nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                                nn.LazyBatchNorm2d(),
                                nn.ReLU(),
                                nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.b1_1 = nn.Sequential(nn.LazyConv2d(64, kernel_size=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(64, kernel_size=3, padding=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(256, kernel_size=1),
                                 nn.LazyBatchNorm2d())
        self.b1_2 = nn.Sequential(nn.LazyConv2d(256, kernel_size=1),
                                 nn.LazyBatchNorm2d())
        self.b2 = nn.Sequential(nn.LazyConv2d(64, kernel_size=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(64, kernel_size=3, padding=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(256, kernel_size=1),
                                 nn.LazyBatchNorm2d())
        self.b3_1 = nn.Sequential(nn.LazyConv2d(128, kernel_size=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(128, kernel_size=3, padding=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(512, kernel_size=1),
                                 nn.LazyBatchNorm2d())
        self.b3_2 = nn.Sequential(nn.LazyConv2d(512, kernel_size=1),
                                 nn.LazyBatchNorm2d())
        self.b4 = nn.Sequential(nn.LazyConv2d(128, kernel_size=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(128, kernel_size=3, padding=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(512, kernel_size=1),
                                 nn.LazyBatchNorm2d())
        self.b5_1 = nn.Sequential(nn.LazyConv2d(256, kernel_size=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(256, kernel_size=3, padding=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(1024, kernel_size=1),
                                 nn.LazyBatchNorm2d())
        self.b5_2 = nn.Sequential(nn.LazyConv2d(1024, kernel_size=1),
                                 nn.LazyBatchNorm2d())
        self.b6 = nn.Sequential(nn.LazyConv2d(256, kernel_size=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(256, kernel_size=3, padding=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(1024, kernel_size=1),
                                 nn.LazyBatchNorm2d())
        self.b7_1 = nn.Sequential(nn.LazyConv2d(512, kernel_size=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(512, kernel_size=3, padding=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(2048, kernel_size=1),
                                 nn.LazyBatchNorm2d())
        self.b7_2 = nn.Sequential(nn.LazyConv2d(2048, kernel_size=1),
                                 nn.LazyBatchNorm2d())
        self.b8 = nn.Sequential(nn.LazyConv2d(512, kernel_size=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(512, kernel_size=3, padding=1),
                                 nn.LazyBatchNorm2d(),
                                 nn.ReLU(),
                                 nn.LazyConv2d(2048, kernel_size=1),
                                 nn.LazyBatchNorm2d())
        self.relu = nn.ReLU()
        self.head = nn.Sequential(nn.AdaptiveAvgPool2d((1, 1)),
                                 nn.Flatten(),
                                 nn.LazyLinear(num_classes))

    def forward(self, x):
        out = self.stem(x)
        out = self.relu(self.b1_1(out) + self.b1_2(out))
        out = self.relu(out + self.b2(out))
        out = self.relu(self.b3_1(out) + self.b3_2(out))
        out = self.relu(out + self.b4(out))
        out = self.relu(self.b5_1(out) + self.b5_2(out))
        out = self.relu(out + self.b6(out))
        out = self.relu(self.b7_1(out) + self.b7_2(out))
        out = self.relu(out + self.b8(out))
        out = self.head(out)
        return out

model_7 = ResNet26().to(device=try_gpu())
optimizer_7 = optim.SGD(model_7.parameters(), lr=0.1)
model_7.eval()
```

```
        (3): LazyConv2d(0, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): ReLU()
        (6): LazyConv2d(0, 512, kernel_size=(1, 1), stride=(1, 1))
        (7): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
          (6): LazyConv2d(0, 1024, kernel_size=(1, 1), stride=(1, 1))
          (7): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (b7_1): Sequential(
          (0): LazyConv2d(0, 512, kernel_size=(1, 1), stride=(1, 1))
          (1): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): ReLU()
          (3): LazyConv2d(0, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (4): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (5): ReLU()
          (6): LazyConv2d(0, 2048, kernel_size=(1, 1), stride=(1, 1))
          (7): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (b7_2): Sequential(
          (0): LazyConv2d(0, 2048, kernel_size=(1, 1), stride=(1, 1))
          (1): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
```

```
torch.cuda.empty_cache()
gc.collect()
t_loss_hist_7, t_acc_hist_7, v_acc_hist_7 = training_loop(5,
                                                        optimizer_7,
                                                        model_7,
                                                        nn.CrossEntropyLoss(),
                                                        train_loader_1,
                                                        val_loader_1,
                                                        3)
title_7 = "Figure 7 - Loss and Accuracy per Epoch for ResNet26"
plot_model(title_7, t_loss_hist_7, t_acc_hist_7, v_acc_hist_7, 'upper right')
```

```
    Epoch 1:
            Duration = 317.779 seconds
            Training Loss: 2.15581
            Training Accuracy: 0.194
            Validation Accuracy: 0.194
```

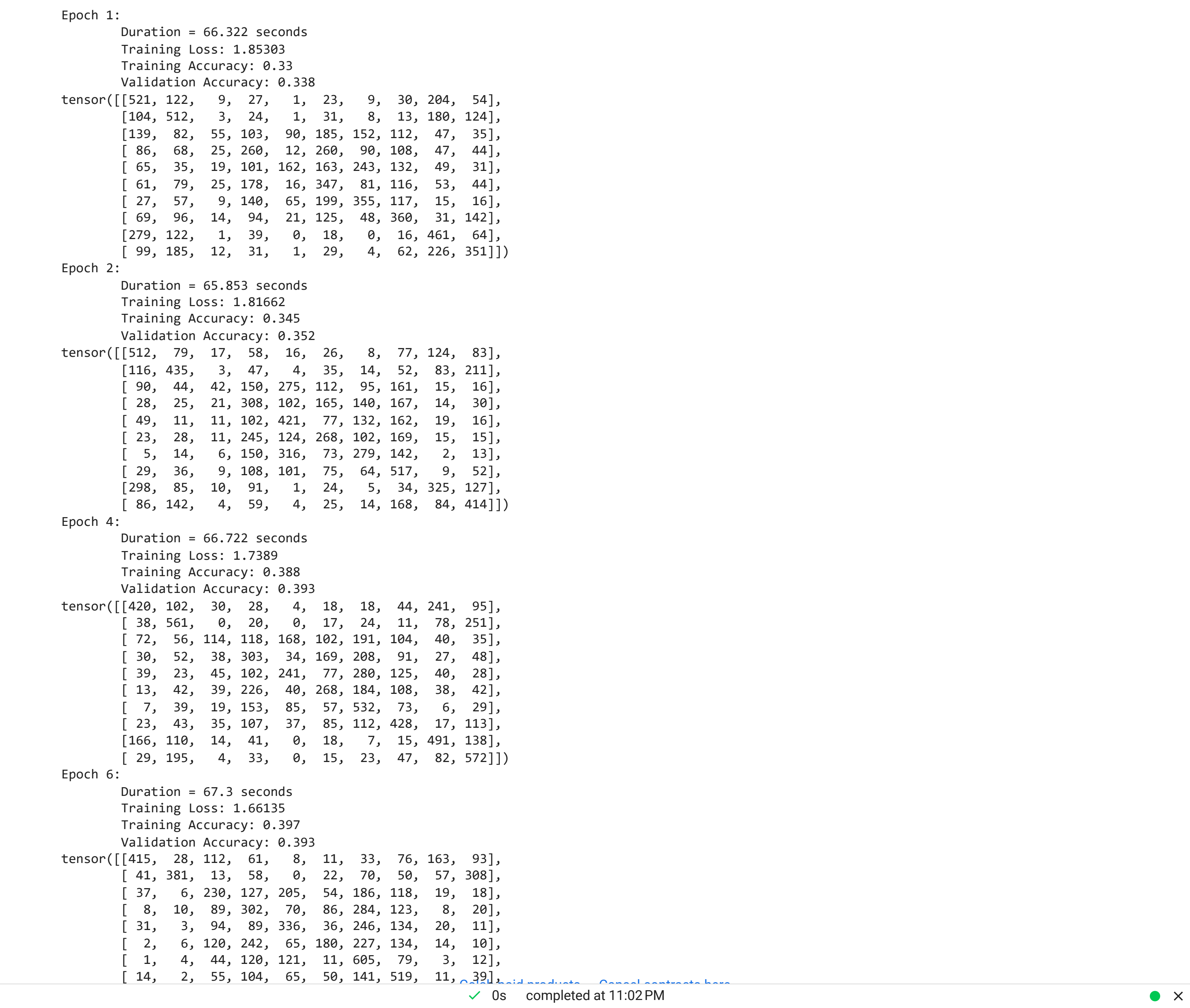
```
class ResNet34(nn.Module):
    def __init__(self, arch, num_classes=10):
        super(ResNet34, self).__init__()
        self.stem = nn.Sequential(nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                              nn.LazyBatchNorm2d(),
                              nn.ReLU(),
                              nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        blks = []
        for i, b in enumerate(arch):
            blks.append(block(*b, first_block=(i==0)))
        self.blks = nn.Sequential(*blks)
        self.head = nn.Sequential(nn.AdaptiveAvgPool2d((1, 1)),
                                 nn.Flatten(),
                                 nn.LazyLinear(num_classes))

    def forward(self, x):
        out = self.stem(x)
        out = self.blks(out)
        out = self.head(out)
        return out

model_8 = ResNet34(arch=((3, 64), (4, 128), (6, 256), (3,512))).to(device=try_gpu())
optimizer_8 = optim.SGD(model_8.parameters(), lr=0.01)
model_8.eval()
```

```
          (0): Residual(
            (conv1): LazyConv2d(0, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
            (conv2): LazyConv2d(0, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (conv3): LazyConv2d(0, 256, kernel_size=(1, 1), stride=(2, 2))
            (bn): LazyBatchNorm2d(0, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          )
```

```
torch.cuda.empty_cache()
gc.collect()
t_loss_hist_8, t_acc_hist_8, v_acc_hist_8 = training_loop(10,
                                                        optimizer_8,
                                                        model_8,
                                                        nn.CrossEntropyLoss(),
                                                        train_loader_1,
                                                        val_loader_1,
                                                        2)
title_8 = "Figure 8 - Loss and Accuracy per Epoch for ResNet34"
plot_model(title_8, t_loss_hist_8, t_acc_hist_8, v_acc_hist_8, 'upper right')
```



```
Epoch 1:
        Duration = 66.322 seconds
        Training Loss: 1.85303
        Training Accuracy: 0.33
        Validation Accuracy: 0.338
tensor([[521, 122,   9,  27,   1,  23,   9,  30, 204,  54],
        [104, 512,   3,  24,   1,  31,   8,  13, 180, 124],
        [139,  82,  55, 103,  90, 185, 152, 112,  47,  35],
        [ 86,  68,  25, 260,  12, 260,  90, 108,  47,  44],
        [ 65,  35,  19, 101, 162, 163, 243, 132,  49,  31],
        [ 61,  79,  25, 178,  16, 347,  81, 116,  53,  44],
        [ 27,  57,   9, 140,  65, 199, 355, 117,  15,  16],
        [ 69,  96,  14,  94,  21, 125,  48, 360,  31, 142],
        [279, 122,   1,  39,   0,  18,   0,  16, 461,  64],
        [ 99, 185,  12,  31,   1,  29,   4,  62, 226, 351]])
Epoch 2:
        Duration = 65.853 seconds
        Training Loss: 1.81662
        Training Accuracy: 0.345
        Validation Accuracy: 0.352
tensor([[512,  79,  17,  58,  16,  26,   8,  77, 124,  83],
        [116, 435,   3,  47,   4,  35,  14,  52,  83, 211],
        [ 90,  44,  42, 150, 275, 112,  95, 161,  15,  16],
        [ 28,  25,  21, 308, 102, 165, 140, 167,  14,  30],
        [ 49,  11,  11, 102, 421,  77, 132, 162,  19,  16],
        [ 23,  28,  11, 245, 124, 268, 102, 169,  15,  15],
        [  5,  14,   6, 150, 316,  73, 279, 142,   2,  13],
        [ 29,  36,   9, 108, 101,  75,  64, 517,   9,  52],
        [298,  85,  10,  91,   1,  24,   5,  34, 325, 127],
        [ 86, 142,   4,  59,   4,  25,  14, 168,  84, 414]])
Epoch 4:
        Duration = 66.722 seconds
        Training Loss: 1.7389
        Training Accuracy: 0.388
        Validation Accuracy: 0.393
tensor([[420, 102,  30,  28,   4,  18,  18,  44, 241,  95],
        [ 38, 561,   0,  20,   0,  17,  24,  11,  78, 251],
        [ 72,  56, 114, 118, 168, 102, 191, 104,  40,  35],
        [ 30,  52,  38, 303,  34, 169, 208,  91,  27,  48],
        [ 39,  23,  45, 102, 241,  77, 280, 125,  40,  28],
        [ 13,  42,  39, 226,  40, 268, 184, 108,  38,  42],
        [  7,  39,  19, 153,  85,  57, 532,  73,   6,  29],
        [ 23,  43,  35, 107,  37,  85, 112, 428,  17, 113],
        [166, 110,  14,  41,   0,  18,   7,  15, 491, 138],
        [ 29, 195,   4,  33,   0,  15,  23,  47,  82, 572]])
Epoch 6:
        Duration = 67.3 seconds
        Training Loss: 1.66135
        Training Accuracy: 0.397
        Validation Accuracy: 0.393
tensor([[415,  28, 112,  61,   8,  11,  33,  76, 163,  93],
        [ 41, 381,  13,  58,   0,  22,  70,  50,  57, 308],
        [ 37,   6, 230, 127, 205,  54, 186, 118,  19,  18],
        [  8,  10,  89, 302,  70,  86, 284, 123,   8,  20],
        [ 31,   3,  94,  89, 336,  36, 246, 134,  20,  11],
        [  2,   6, 120, 242,  65, 180, 227, 134,  14,  10],
        [  1,   4,  44, 120, 121,  11, 605,  79,   3,  12],
        [ 14,   2,  55, 104,  65,  50, 141, 519,  11,  39],
```