Import packages to load data.

```
import pandas as pd # used for loading csv
import numpy as np # used to load and organize image data
```

Connect to drive.

```
from google.colab import drive
drive.mount("/content/drive")
```

```
Mounted at /content/drive
```

Train on Cuda if available:

```
import torch
train_on_gpu = torch.cuda.is_available()

if not train_on_gpu:
    print('CUDA is not available.  Training on CPU ...')
else:
    print('CUDA is available!  Training on GPU ...')
```

```
CUDA is not available.  Training on CPU ...
```

Import data files.

```
images = np.load("/content/drive/MyDrive/Colab Notebooks/Advanced Analytics/images.npy", allow_pickle=True)
labels = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/Advanced Analytics/labels.csv")
```

The next steps are used to explore the data. We start by checking the shape of the images data and the head of the labels to make sure they have imported correctly.

```
print(images.shape)
print(labels.shape)
labels.head()
```

```
(4750, 128, 128, 3)
(4750, 1)
```

	Label
0	Small-flowered Cranesbill
1	Small-flowered Cranesbill
2	Small-flowered Cranesbill
3	Small-flowered Cranesbill
4	Small-flowered Cranesbill

Next steps: Generate code with labels | View recommended plots | New interactive sheet

Determine mean and standard deviation of images data to use in normalization later:

```
import torchvision.transforms as transforms

transform = transforms.Compose([
    transforms.ToTensor()])

images_tr = torch.stack([transform(image) for image in images])

sumel = 0.0
countel = 0
for img in images_tr:
    sumel += img.sum([1, 2])
    countel += torch.numel(img[0])
```

```
mean = sumel/countel
std = torch.sqrt(sumel/countel)
print("mean: " + str(mean))
print("standard deviation: " + str(std))
```

```
mean: tensor([0.2068, 0.2889, 0.3283])
standard deviation: tensor([0.4548, 0.5375, 0.5730])
```

Get a list of plant labels being used.

```
labels['Label'].unique()
```

```
array(['Small-flowered Cranesbill', 'Fat Hen', 'Shepherds Purse',
       'Common wheat', 'Common Chickweed', 'Charlock', 'Cleavers',
       'Scentless Mayweed', 'Sugar beet', 'Maize', 'Black-grass',
       'Loose Silky-bent'], dtype=object)
```

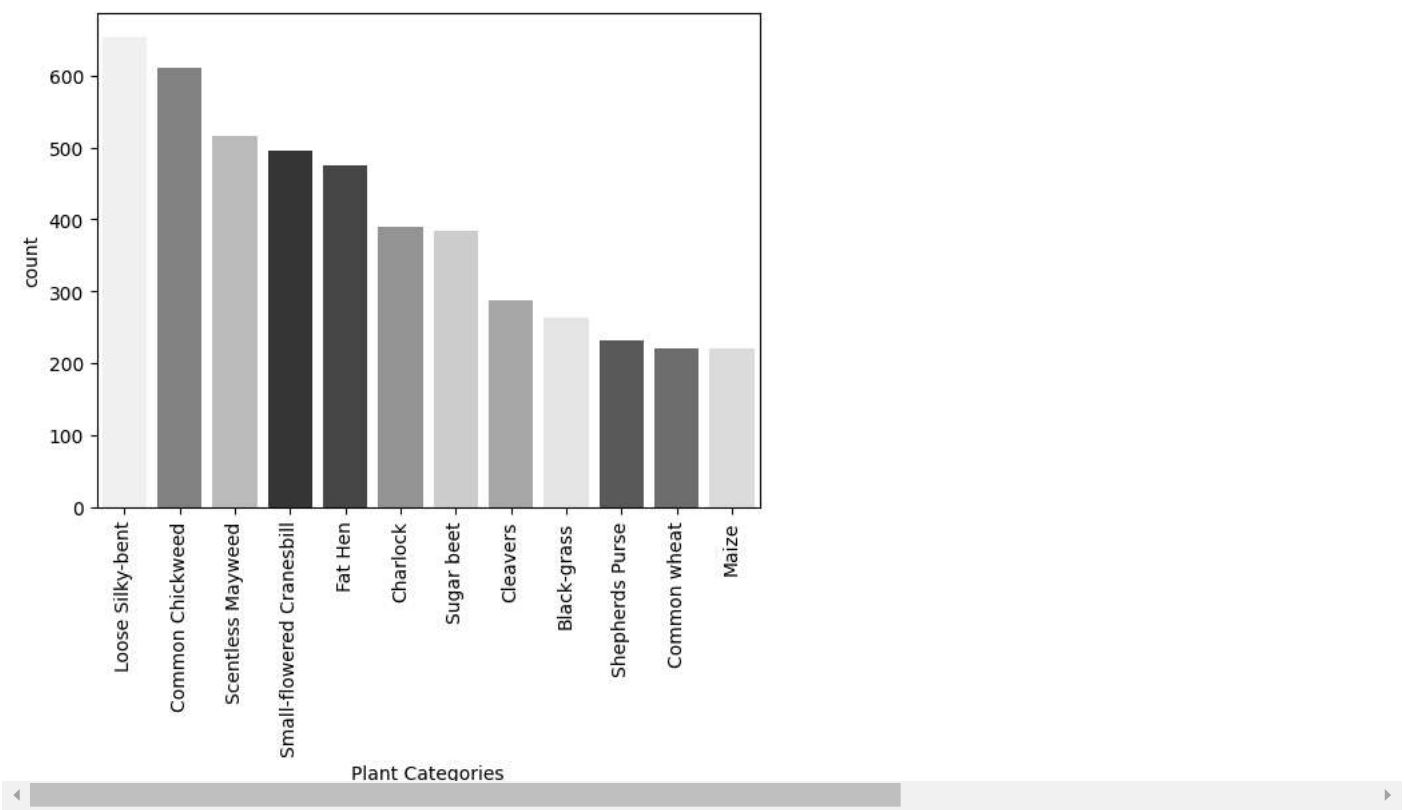
Create list from categories.

```
categories = ['Small-flowered Cranesbill', 'Fat Hen', 'Shepherds Purse',
       'Common wheat', 'Common Chickweed', 'Charlock', 'Cleavers',
       'Maize', 'Sugar beet', 'Scentless Mayweed', 'Black-grass',
       'Loose Silky-bent']
num_categories = len(categories) # will be used to create image grid
print(num_categories)
```

```
12
```

Visualize the distribution of the categories.

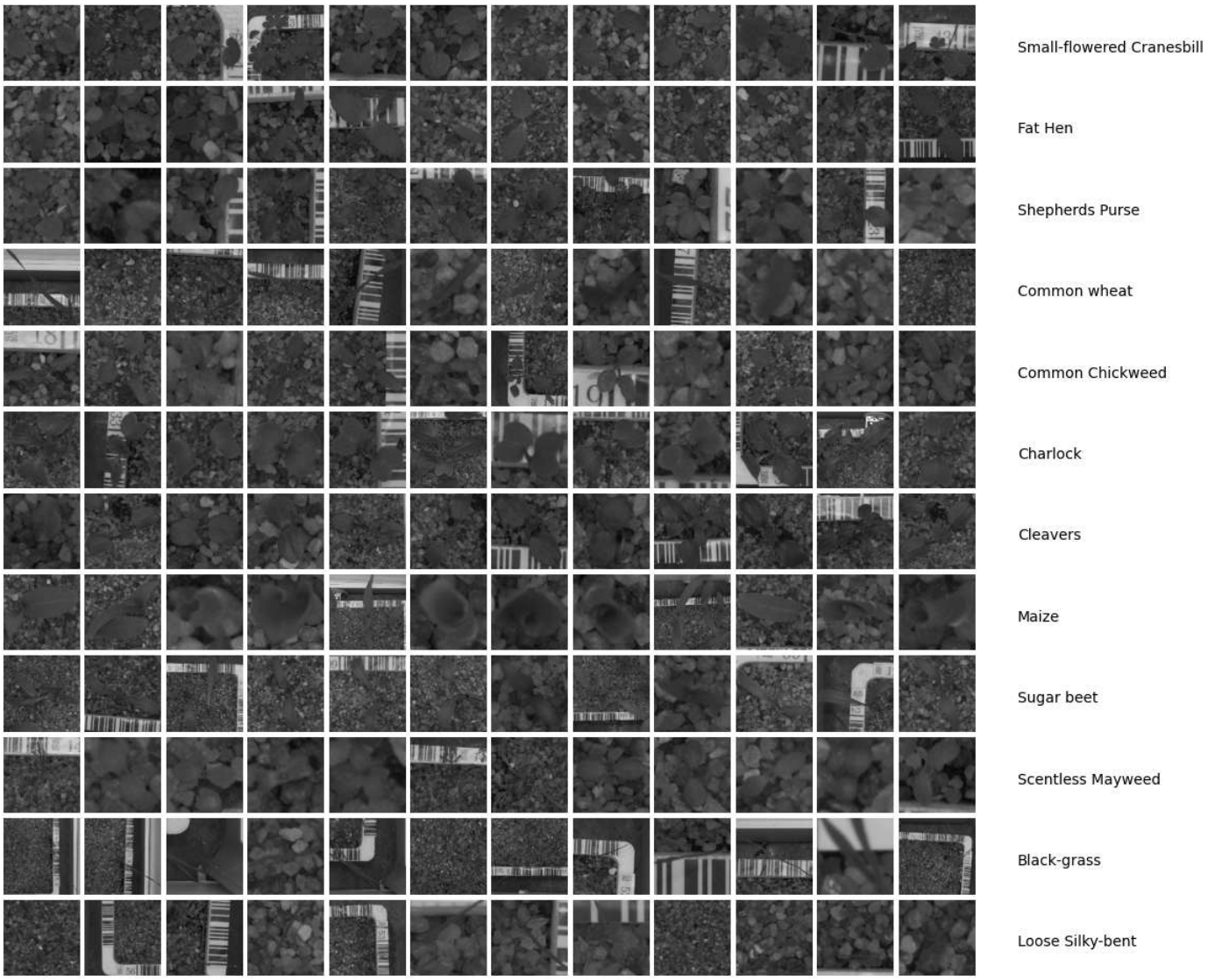
```
import matplotlib.pyplot as plt # used to show plot and add information
import seaborn as sns # used to create plot

sns.countplot(x='Label', data=labels, order=labels['Label'].value_counts().index, legend=False, hue='Label', palette='Greens_r')
plt.xlabel('Plant Categories')
plt.xticks(rotation=90)
plt.show()
```



We can see that the Loose Silky-bent category contains more images than the others, and that common wheat and maize have the fewest images.

Visualize samples with associated categories:

```
from mpl_toolkits.axes_grid1 import ImageGrid # used to create image grid
import random
# Set seed for reproducibility
random_state = 12
random.seed(random_state)

# Set grid parameters
fig = plt.figure(1, figsize=(num_categories, num_categories))
grid = ImageGrid(fig, 111, nrows_ncols=(num_categories, num_categories), axes_pad=0.05)
i = 0

# Create loop for plotting images in each category
for category_id, category in enumerate(categories):
  condition = labels["Label"] == category
  plant_label = labels[condition].index.tolist()
  for k in range(min(12, len(plant_label))):
      ax = grid[i]
      ax.imshow(images[plant_label[k]])
      ax.axis('off')
      if i % num_categories == num_categories - 1:
        # used to print the names for each category
        ax.text(200, 70, category, verticalalignment='center')
      i += 1

plt.show()
```



One-hot encoding of categories:

```
import pandas as pd

print(labels.head(5))
labels = pd.get_dummies(labels, dtype=int)
#labels.reset_index(inplace=False)
print(labels.head(5))
print(labels.shape)
```

```
                        Label
0  Small-flowered Cranesbill
1  Small-flowered Cranesbill
2  Small-flowered Cranesbill
3  Small-flowered Cranesbill
4  Small-flowered Cranesbill
   Label_Black-grass  Label_Charlock  Label_Cleavers  Label_Common Chickweed  \
0                  0               0               0                       0
1                  0               0               0                       0
2                  0               0               0                       0
3                  0               0               0                       0
4                  0               0               0                       0

   Label_Common wheat  Label_Fat Hen  Label_Loose Silky-bent  Label_Maize  \
0                   0              0                       0            0
1                   0              0                       0            0
2                   0              0                       0            0
3                   0              0                       0            0
```

```
4                  0             0                     0            0

   Label_Scentless Mayweed  Label_Shepherds Purse  \
0                        0                      0
1                        0                      0
2                        0                      0
3                        0                      0
4                        0                      0

   Label_Small-flowered Cranesbill  Label_Sugar beet
0                                1                 0
1                                1                 0
2                                1                 0
3                                1                 0
4                                1                 0
(4750, 12)
```

Train-validation-test split:

```
from sklearn.model_selection import train_test_split

# Split into train and test (80%/20%)
X_train, X_test1, y_train, y_test1 = train_test_split(
    images, labels, test_size=0.20, stratify=labels, random_state = random_state)
# Split into test and validation (50/50 = 10%/10% of total)
X_val, X_test, y_val, y_test = train_test_split(
    X_test1, y_test1, test_size=0.50, stratify=y_test1, random_state = random_state)

print(X_train.shape, X_test.shape, y_train.shape, y_test.shape, X_val.shape, y_val.shape)

# Save data sets
from numpy import asarray
from numpy import savetxt
X_train_array = asarray(X_train)
#np.save('/content/drive/MyDrive/Colab Notebooks/Advanced Analytics/X_train.npy', X_train_array)
y_train_array = asarray(y_train)
#np.save('/content/drive/MyDrive/Colab Notebooks/Advanced Analytics/y_train.npy', y_train_array)
X_val_array = asarray(X_val)
#np.save('/content/drive/MyDrive/Colab Notebooks/Advanced Analytics/X_val.npy', X_val_array)
y_val_array = asarray(y_val)
#np.save('/content/drive/MyDrive/Colab Notebooks/Advanced Analytics/y_val.npy', y_val_array)
X_test_array = asarray(X_test)
#np.save('/content/drive/MyDrive/Colab Notebooks/Advanced Analytics/X_test.npy', X_test_array)
y_test_array = asarray(y_test)
#np.save('/content/drive/MyDrive/Colab Notebooks/Advanced Analytics/y_test.npy', y_test_array)
```

```
(3800, 128, 128, 3) (475, 128, 128, 3) (3800, 12) (475, 12) (475, 128, 128, 3) (475, 12)
```

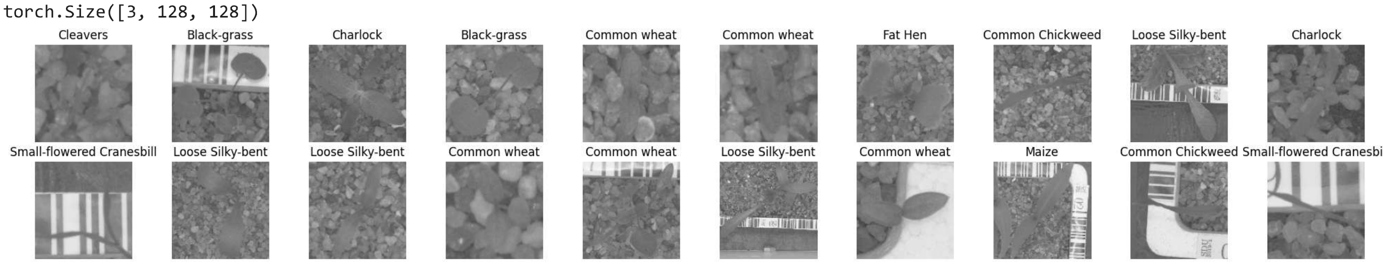
```
# Define a custom dataset class to apply transforms to individual images
class CustomImageDataset(torch.utils.data.Dataset):
    def __init__(self, images, labels, transform):
        self.images = images
        self.labels = labels
        self.transform = transform

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels.iloc[idx].values
        if self.transform:
            image = self.transform(image)

        return image, label

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

Create customized dataset, transform training images to tensor, normalize, and preview one batch:

```
from torchvision import datasets
import torchvision.transforms as transforms
import torch
from torch.utils.data import DataLoader, TensorDataset
import multiprocessing
import matplotlib.pyplot as plt
%matplotlib inline
```

```
from torchvision.datasets import ImageFolder
from PIL import Image

n_workers = multiprocessing.cpu_count()
print(n_workers)

mean = [0.2068, 0.2889, 0.3283]
std = [0.4548, 0.5375, 0.5730]

train_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])

# Create instance of the custom dataset
train_data = CustomImageDataset(images=X_train, labels=y_train, transform=train_transforms)

# Create DataLoader
train_loader = DataLoader(train_data, batch_size=20, num_workers=n_workers, shuffle=True)

# Obtain next batch
images, labels = next(iter(train_loader))
image = images[0]
label = labels[0]
print(image.shape)
print(label.shape)

# Function to display image
def imshow(img, sub):
  img = img / 2 + 0.5
  img = img.numpy()
  img = img.transpose((1, 2, 0))
  sub.imshow(img)
  sub.axis('off')

# Plot images
fig, subs = plt.subplots(2, 10, figsize=(25, 4))
for i, sub in enumerate(subs.flatten()):
  imshow(images[i], sub)
  # Get the index of the maximum value instead of the column name
  category_index = labels[i].argmax()
  sub.set_title(categories[category_index])

print(images.shape)
print(image.shape)
```

```
2
torch.Size([3, 128, 128])
torch.Size([12])
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). G
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). G
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). G
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). G
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). G
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). G
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). G
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). G
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). G
torch.Size([20, 3, 128, 128])
torch.Size([3, 128, 128])
```



Create function for transforms with augmentations and create custom dataset and dataloaders.

```
from torchvision import datasets
import torchvision.transforms as transforms
import torch
```

```
from torch.utils.data import DataLoader, TensorDataset
import multiprocessing
import matplotlib.pyplot as plt
%matplotlib inline
from torchvision.datasets import ImageFolder
from PIL import Image
import numpy as np

n_workers = multiprocessing.cpu_count()
print(n_workers)

# Create function for transformations for each dataset
# Function will allow for optimization of parameters
def get_transforms():
  mean = [0.2068, 0.2889, 0.3283]
  std = [0.4548, 0.5375, 0.5730]

  return{
    "train": transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
        transforms.RandomRotation(45),
        transforms.RandomAutocontrast(),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
    'validate': transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
    'test': transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ])
  }

# Define a custom dataset class to apply transforms to individual images
class CustomImageDataset(torch.utils.data.Dataset):
    def __init__(self, images, labels, transform=None):
        self.images = images
        self.labels = labels
        self.transform = transform

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels.iloc[idx].values  # Assuming labels is a pandas DataFrame

        # Convert the NumPy array to a PIL Image for augmentation
        image = Image.fromarray(image)

        if self.transform:
            image = self.transform(image)

        return image, label

#Create function for data loaders
def get_data_loaders(transforms, num_workers, random_seed=12):
  # Reseed random number generators for comparing experiments
  torch.manual_seed(random_seed)
  torch.cuda.manual_seed(random_seed)
  np.random.seed(random_seed)

  transforms = get_transforms()
  # Create each instance of the custom dataset
  train_data = CustomImageDataset(images=X_train, labels=y_train, transform=transforms['train'])
  val_data = CustomImageDataset(images=X_val, labels=y_val, transform=transforms['validate'])
  test_data = CustomImageDataset(images=X_test, labels=y_test, transform=transforms['test'])

  # Create data loaders
  train_loader = DataLoader(train_data, batch_size=100, num_workers=n_workers, shuffle=True)
  val_loader = DataLoader(val_data, batch_size=25, num_workers=n_workers, shuffle=True)
  test_loader = DataLoader(test_data, batch_size=25, num_workers=n_workers, shuffle=True)

  return {'train': train_loader, 'validate': val_loader, 'test': test_loader}
```

2

Define model structure:

```
import torch.nn as nn
from torchsummary import summary

# define CNN model
class Net(nn.Module):
    def __init__(self, num_classes=12):
        super(Net, self).__init__()

        self.model = nn.Sequential(
          # convolutional layer 1 (tensor 3x128x128 -> 16x64x64)
          nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1),
          nn.BatchNorm2d(16),
          nn.ELU(),
          nn.MaxPool2d(kernel_size=2, stride=2),
          # convolutional layer 2 - takes output of previous layer (16x64x64 -> 32x32x32)
          nn.Conv2d(16, 32, 3, padding=1),
          nn.BatchNorm2d(32),
          nn.ELU(),
          nn.MaxPool2d(2, 2),
          # convolutional layer 3 (32x32x32 -> 164x6x16)
          nn.Conv2d(32, 64, 3, padding=1),
          nn.BatchNorm2d(64),
          nn.ELU(),
          nn.MaxPool2d(2, 2),
          # linear layer (64x16x16 -> 500)
          nn.Flatten(),
          nn.Linear(64 * 16 * 16, 500),
          nn.Dropout(0.5),
          nn.BatchNorm1d(500),
          nn.ELU(),
          # linear layer (500 -> 12)
          nn.Linear(500, num_classes)
        )

    def forward(self, x):
      return self.model(x)

# create complete CNN
model = Net()
print(model)
summary(model, input_size=(3, 128, 128))
```

```
Net(
  (model): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ELU(alpha=1.0)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ELU(alpha=1.0)
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (12): Flatten(start_dim=1, end_dim=-1)
    (13): Linear(in_features=16384, out_features=500, bias=True)
    (14): Dropout(p=0.5, inplace=False)
    (15): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (16): ELU(alpha=1.0)
    (17): Linear(in_features=500, out_features=12, bias=True)
  )
)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 16, 128, 128]             448
       BatchNorm2d-2         [-1, 16, 128, 128]              32
              ELU-3         [-1, 16, 128, 128]               0
         MaxPool2d-4           [-1, 16, 64, 64]               0
            Conv2d-5           [-1, 32, 64, 64]           4,640
       BatchNorm2d-6           [-1, 32, 64, 64]              64
              ELU-7           [-1, 32, 64, 64]               0
```

```
         MaxPool2d-8          [-1, 32, 32, 32]               0
            Conv2d-9          [-1, 64, 32, 32]          18,496
       BatchNorm2d-10          [-1, 64, 32, 32]             128
              ELU-11          [-1, 64, 32, 32]               0
         MaxPool2d-12          [-1, 64, 16, 16]               0
           Flatten-13               [-1, 16384]               0
            Linear-14                 [-1, 500]       8,192,500
           Dropout-15                 [-1, 500]               0
       BatchNorm1d-16                 [-1, 500]           1,000
              ELU-17                 [-1, 500]               0
            Linear-18                  [-1, 12]           6,012
================================================================
Total params: 8,223,320
Trainable params: 8,223,320
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 11.52
Params size (MB): 31.37
Estimated Total Size (MB): 43.07
----------------------------------------------------------------
```

Create functions for training loop and validation/test loop:

```
!pip install livelossplot
```

Show hidden output

```
!pip install torchmetrics
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.6/24.6 MB 26.0 MB/s eta 0:00:00
Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (883 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 883.7/883.7 kB 32.1 MB/s eta 0:00:00
Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl (664.8 MB)
```

```
      Successfully uninstalled nvidia-cudnn-cu12 9.3.0.75
    Attempting uninstall: nvidia-cusolver-cu12
      Found existing installation: nvidia-cusolver-cu12 11.6.3.83
      Uninstalling nvidia-cusolver-cu12-11.6.3.83:
        Successfully uninstalled nvidia-cusolver-cu12-11.6.3.83
  Successfully installed lightning-utilities-0.12.0 nvidia-cublas-cu12-12.4.5.8 nvidia-cuda-cupti-cu12-12.4.127 nvidia-cuda-nvrtc-cu12-
```

```
import torch.optim as optim
import livelossplot
from livelossplot import PlotLosses
from torchmetrics import Precision, Recall

# train on cuda if available
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# Define the loss function
criterion = nn.CrossEntropyLoss()

def train_one_epoch(data_loaders, model, optimizer, criterion):
  liveloss = PlotLosses()
  for epoch in range(1):
    model.train()
    running_loss = 0.0
    running_corrects = 0
    logs = {}
    # Iterate over training batches
    for images, labels in data_loaders:
        optimizer.zero_grad()
        outputs = model(images)
        labels = torch.argmax(labels, dim=1)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    train_loss = running_loss / len(data_loaders)
    logs['train log loss'] = train_loss
    print(f"Training loss: {train_loss:.3f}")
    return train_loss
  liveloss.update(logs)
  liveloss.send()

def val_one_epoch(data_loaders, model, criterion):
  liveloss = PlotLosses()
  with torch.no_grad():
    for epoch in range(1):
      model.eval()
      running_loss = 0.0
      running_corrects = 0
      logs = {}
      # Initialize lists for storing predictions and labels
      preds = []
      actuals = []
      # Iterate over training batches
      for images, labels in data_loaders:
        outputs = model(images)
        labels = torch.argmax(labels, dim=1)
        loss = criterion(outputs, labels)
        running_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        running_corrects += torch.sum(predicted == labels.data)

      val_loss = running_loss / len(data_loaders)
      val_accuracy = float(running_corrects) / len(data_loaders.dataset)
      logs['val log loss'] = val_loss
      logs['val accuracy'] = val_accuracy
      print(f"Validation loss - validate: {val_loss:.3f}")
      print(f"Validation Accuracy: {val_accuracy:.3f}")
      return val_loss, val_accuracy
    liveloss.update(logs)
    liveloss.send()

def test_one_epoch(data_loaders, model, criterion):
  with torch.no_grad():
    for epoch in range(1):
      # Define metrics
      metric_precision = Precision(task = 'multiclass', num_classes = 12, average = None)
```

```
        metric_recall = Recall(task = 'multiclass', num_classes = 12, average = None)
        model.eval()
        running_loss = 0
        running_corrects = 0
        # Initialize lists for storing predictions and labels
        preds = []
        actuals = []
        # Iterate over training batches
        for images, labels in data_loaders:
          outputs = model(images)
          labels = torch.argmax(labels, dim=1)
          loss = criterion(outputs, labels)
          running_loss += loss.item()

          # Get and save predicted labels
          _, predicted = torch.max(outputs, 1)
          running_corrects += torch.sum(predicted == labels.data)
          preds.extend(outputs.cpu().numpy())
          actuals.extend(labels.cpu().numpy())

        # Convert predictions and actuals to numpy array
        preds = torch.tensor(np.array(preds))
        actuals = torch.tensor(np.array(actuals))

        # Calculate metrics
        metric_precision(preds, actuals)
        metric_recall(preds, actuals)


    val_loss = running_loss / len(data_loaders.dataset)
    accuracy = float(running_corrects) / len(data_loaders.dataset)
    print(f"Validation loss - test: {val_loss:.3f}")
    print(f"Test Accuracy: {accuracy:.3f}")
    precision = metric_precision.compute()
    recall = metric_recall.compute()
    print(f"Precision: {precision}")
    print(f"Recall: {recall}")
    return val_loss, preds, actuals
```

Create early stopping class:

```
class EarlyStopper:
    def __init__(self, patience=1, min_delta=0):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.min_val_loss = float('inf')

    def early_stop(self, val_loss):
        if val_loss < self.min_val_loss:
            self.min_val_loss = val_loss
            self.counter = 0
        elif val_loss > (self.min_val_loss + self.min_delta):
            self.counter += 1
            if self.counter >= self.patience:
                return True
        return False
```

Define train one model:

```
!pip install mlflow
```

Show hidden output

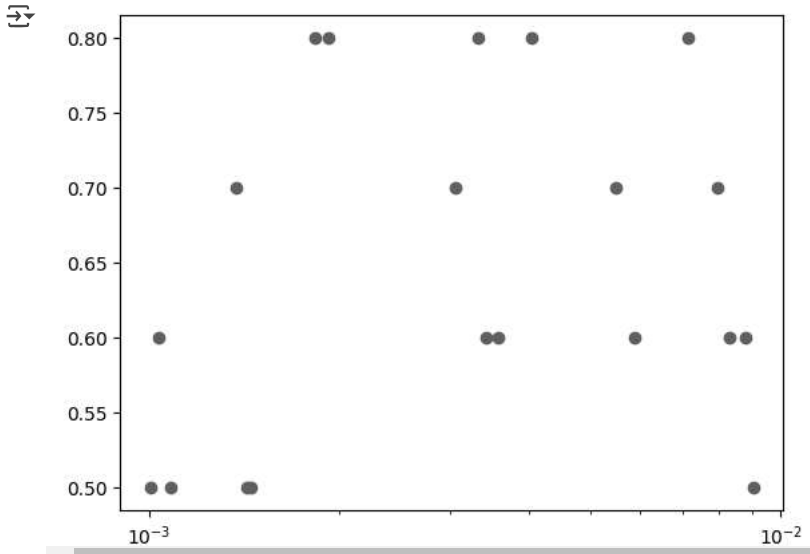
```
import mlflow
from torch.optim.lr_scheduler import ExponentialLR
import datetime # Used for experiment name
import os

# train on cuda if available
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
n_workers = multiprocessing.cpu_count()
save_path = "/content/drive/MyDrive/Colab Notebooks/Advanced Analytics"
```

```
def train_one_model(learning_rate, gamma, n_epochs):
  liveloss = PlotLosses()
  transforms = get_transforms()
  data_loaders = get_data_loaders(transforms, num_workers=n_workers)
  model = Net()
  val_loss_min = None

  if train_on_gpu:
    model.cuda()

  optimizer = optim.Adam(model.parameters(), lr = learning_rate)
  scheduler = ExponentialLR(optimizer, gamma=gamma)
  criterion = nn.CrossEntropyLoss()

  with mlflow.start_run():
    for epoch in range(n_epochs):
      logs= {}
      print("Epoch: ", epoch)
      train_loss = train_one_epoch(data_loaders['train'], model, optimizer, criterion)

      val_loss, val_accuracy = val_one_epoch(data_loaders['validate'], model, criterion)

      # Include early stop if necessary
      early_stopper = EarlyStopper(patience=3)
      if early_stopper.early_stop(val_loss):
        print("We are at epoch:", epoch)
        break

      # Save model if validation loss decreases by more than 1%
      if val_loss_min is None or(
          (val_loss_min - val_loss) / val_loss_min > 0.01
          ):

        # Save weights
        filename = os.path.join(save_path, 'best_val_loss.pt')
        torch.save(model.state_dict(), filename)

        val_loss_min = val_loss

      logs['validation loss'] = val_loss
      logs['training loss'] = train_loss
      logs["validation accuracy"] = val_accuracy
      liveloss.update(logs)
      liveloss.send()

      # Update learning rate
      scheduler.step(val_loss)

      # Test model on validation set (no optimization)
      val_loss, preds, actuals = test_one_epoch(data_loaders['validate'], model, criterion)

      # Restore best validation loss parameter
      model.load_state_dict(torch.load(filename))

      # Log hyperparameters
      mlflow.log_metric("val_loss", val_loss)
      mlflow.log_param("learning_rate", learning_rate)
      mlflow.log_param("n_epochs", n_epochs)
      mlflow.log_param("gamma", gamma)

      # Log metrics
      mlflow.log_metric("train_loss", train_loss)
      mlflow.log_metric("val_loss", val_loss)
      val_accuracy = (np.argmax(preds, axis=1) == np.array(actuals)).sum() / len(actuals)
      mlflow.log_metric("val_accuracy", val_accuracy)
      mlflow.log_artifact(filename)
    return model
  mlflow.end_run()
```

Use random search to explore hyperparameters:

```
# Create random grid of learning rate and gamma
min_lr = 0.001
```

```
max_lr = 0.01
n_grid = 20

# Sample learning rate log samples
lrs = 10**(np.random.uniform(np.log10(min_lr), np.log10(max_lr), n_grid))
# Uniformly sample gamma rate
gamma = np.random.randint(5, 9, n_grid) / 10

# Plot grid
_ = plt.scatter(lrs, gamma)
_ = plt.xscale("log")
```



Run experiment to determine best learning-rate and gamma values:

```
for lr, g in zip(lrs, gamma):
  train_one_model(lr, g, n_epochs=5)
```

```
Epoch:  0
Training loss: 1.779
Validation loss - validate: 1.661
Validation Accuracy: 0.453
Validation loss - test: 0.066
Test Accuracy: 0.453
Epoch:  1
<ipython-input-25-4c4b3fabe1a4>:63: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value),
  model.load_state_dict(torch.load(filename))
Training loss: 1.284
Validation loss - validate: 1.212
Validation Accuracy: 0.615
Validation loss - test: 0.048
Test Accuracy: 0.615
Epoch:  2
Training loss: 1.082
Validation loss - validate: 1.042
Validation Accuracy: 0.642
Validation loss - test: 0.042
Test Accuracy: 0.642
Epoch:  3
Training loss: 0.998
Validation loss - validate: 1.001
Validation Accuracy: 0.651
Validation loss - test: 0.040
Test Accuracy: 0.651
Epoch:  4
Training loss: 0.917
Validation loss - validate: 0.947
Validation Accuracy: 0.655
Validation loss - test: 0.038
Test Accuracy: 0.655
Epoch:  0
Training loss: 1.749
Validation loss - validate: 1.534
Validation Accuracy: 0.495
Validation loss - test: 0.061
Test Accuracy: 0.495
Epoch:  1
```

```
Training loss: 1.241
Validation loss - validate: 1.093
Validation Accuracy: 0.625
Validation loss - test: 0.044
Test Accuracy: 0.625
Epoch:  2
Training loss: 1.043
Validation loss - validate: 1.128
Validation Accuracy: 0.640
Validation loss - test: 0.045
Test Accuracy: 0.640
Epoch:  3
Training loss: 1.075
Validation loss - validate: 1.048
Validation Accuracy: 0.632
Validation loss - test: 0.042
Test Accuracy: 0.632
Epoch:  4
```

Retrieve mlflow data on experiment:

```
import mlflow

runs = mlflow.search_runs()
sorted_runs = runs[
    ["run_id",
     "params.learning_rate",
     "params.n_epochs",
     "params.gamma",
     "metrics.train_loss",
     "metrics.val_loss",
     "metrics.val_accuracy"
    ]
].sort_values(by="metrics.val_loss", ascending=True)
sorted_runs
```

	run_id	params.learning_rate	params.n_epochs	params.gamma	metrics.train_loss	metrics.val_loss	metric
14	a99b773d328a47d892da69bdea79e363	0.008293674908605475	5	0.6	0.929735	0.032949	
1	066614e2ad7b4c12be4fbead33459323	0.005863192512339714	5	0.6	0.884745	0.033293	
2	a86aeb3916f845e8bdcb24ab3d28ab75	0.003057577076255498	5	0.7	0.898287	0.033676	
7	c21ee1cd95924147accc3b554725c532	0.008794783157695697	5	0.6	0.912329	0.033954	
6	943325aba9fe4d498495ccecd8ab436e	0.007124190781828619	5	0.8	0.893025	0.034535	
16	59ad864641d64db9a0256d2ff0f9a13e	0.0034177429283790495	5	0.6	0.892751	0.035029	
18	b895821e6f4a438389374b937c4f2020	0.005496037616682325	5	0.7	0.950851	0.035216	
4	28c21eab661c4d7ca66cedb8dad2cb7d	0.0033206723595393755	5	0.8	0.895180	0.035251	
17	e2569cf3b43a4428b0b420a613251613	0.0018336439724345017	5	0.8	0.908110	0.035508	
8	e24e6c2bb94d4faba08583b9cc674a43	0.004037227142333932	5	0.8	0.884938	0.036171	
12	e68df2efea4e48fb8d69331c6e560805	0.0010799942111106168	5	0.5	0.931614	0.036356	
15	20538e0457f54cf7b66422b962c2c7ce	0.001034129584101865	5	0.6	0.932521	0.036451	
0	056fcf804e1a4c2e93ac9818bd71746c	0.0014478272725587603	5	0.5	0.916393	0.036469	
9	7097b88a839c4e69b39d9ba4bf18e607	0.0019223318118548767	5	0.8	0.906705	0.036512	
5	6f3c97eaaa9342f695bea08d93ea680d	0.0010052156317193804	5	0.5	0.935427	0.036734	
3	27e02fece26b408ebefd8bbfd584786f	0.003564820226076412	5	0.6	0.887444	0.036880	
10	f4527b932658420db3ad905570d5aa5d	0.0013715426633363898	5	0.7	0.916864	0.037160	
19	4052cdff09d64ea0ae9862ae6c5375d1	0.0014261422374240921	5	0.5	0.916973	0.037865	
11	e015346caffa414fb44137d40ce6704a	0.00905626946526	5	0.5	1.077178	0.039753	
13	69d740a70be24e09bd866b5f7e79cc09	0.00795636785524916	5	0.7	1.071292	0.041005	
20	fbb147191d944c978a9a56a81005f7ed	0.0014261422374240921	10	0.5	1.081689	0.041686	
21	1134b37b3f094aae9baeb0adbcc98128	None	None	None	NaN	NaN	

Next steps: Generate code with sorted_runs | View recommended plots | New interactive sheet

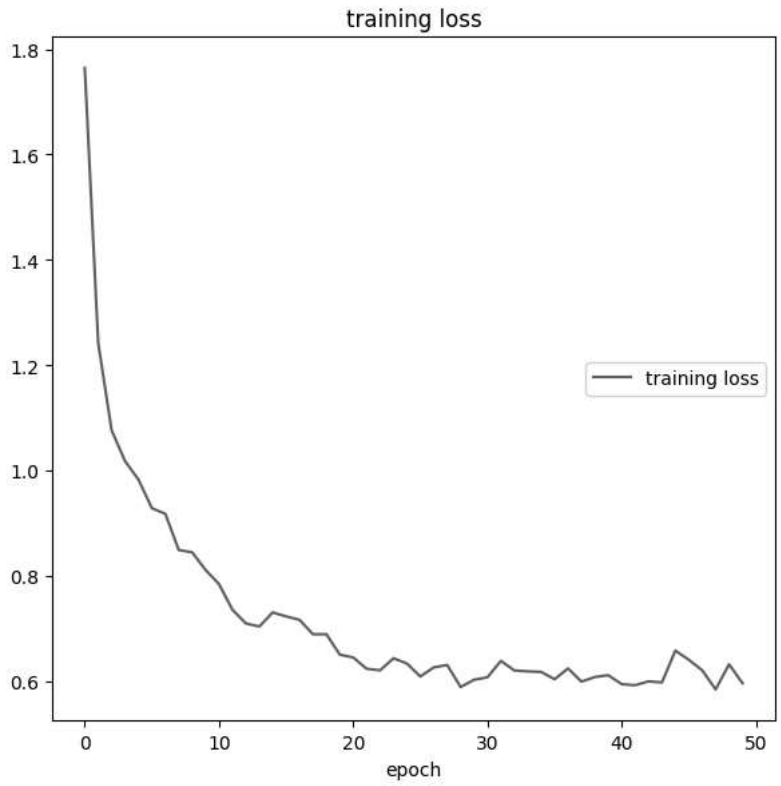
Save model with best parameters in case I need it later:

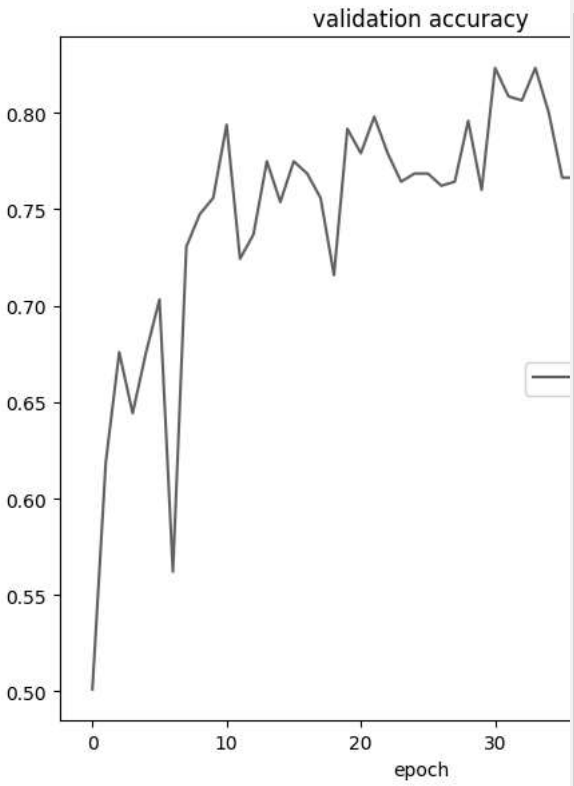
```
from mlflow.tracking import MlflowClient

lowest_loss_id = sorted_runs.iloc[0]['run_id']

# Fetch model for that run
client = MlflowClient()
local_path = client.download_artifacts(lowest_loss_id, "best_val_loss.pt", '.')
```

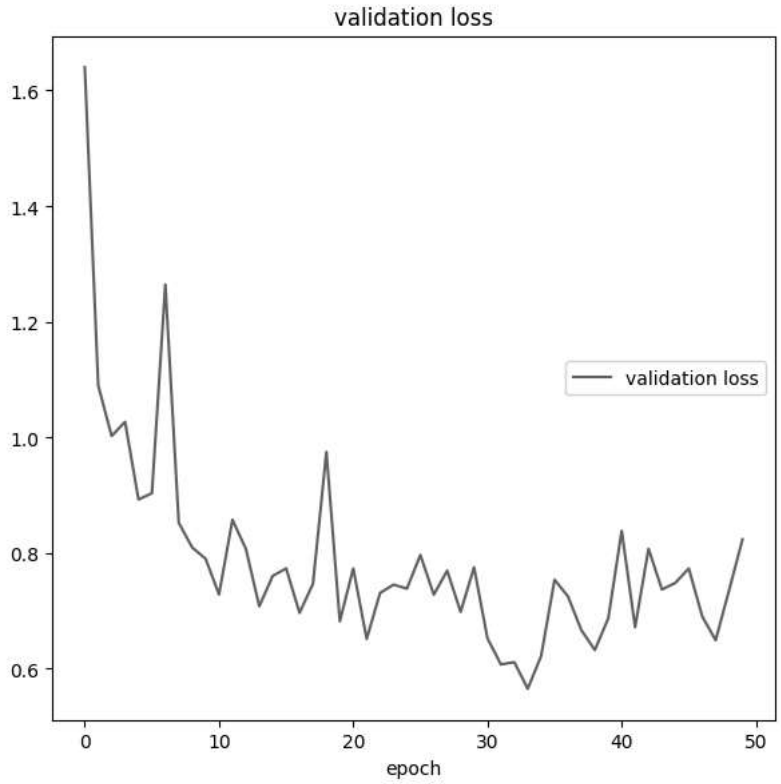
```
Downloading artifacts: 100%                                    1/1 [00:00<00:00, 13.15it/s]
```

Train model with selected parameters to determine optimal number of epochs to use:

```
model_train = train_one_model(learning_rate=0.008, gamma=0.6, n_epochs=50)
```



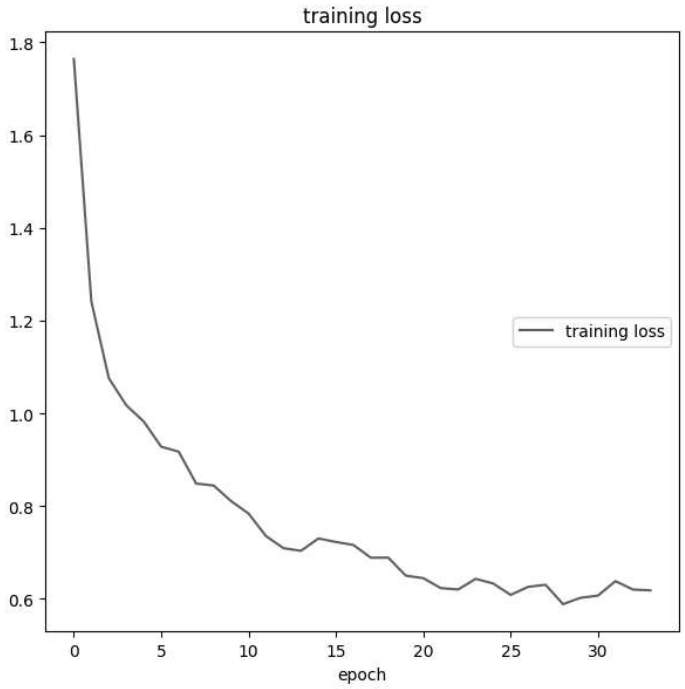


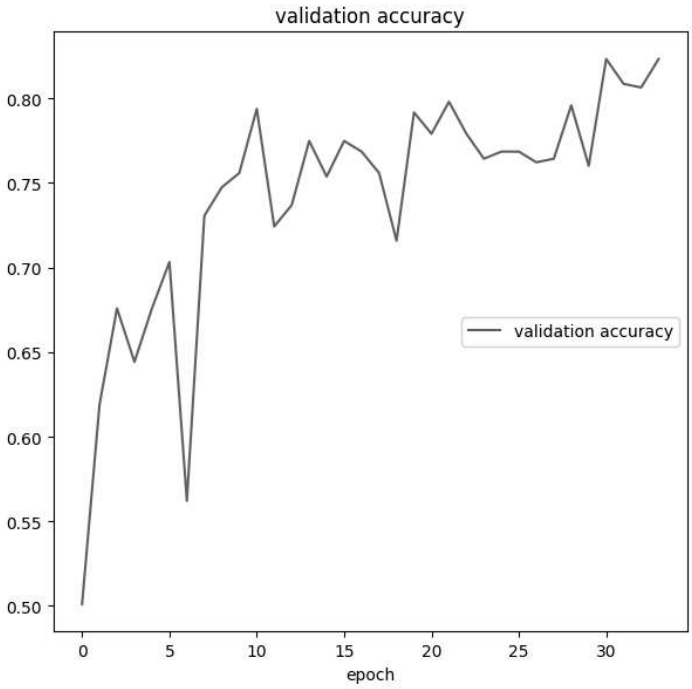


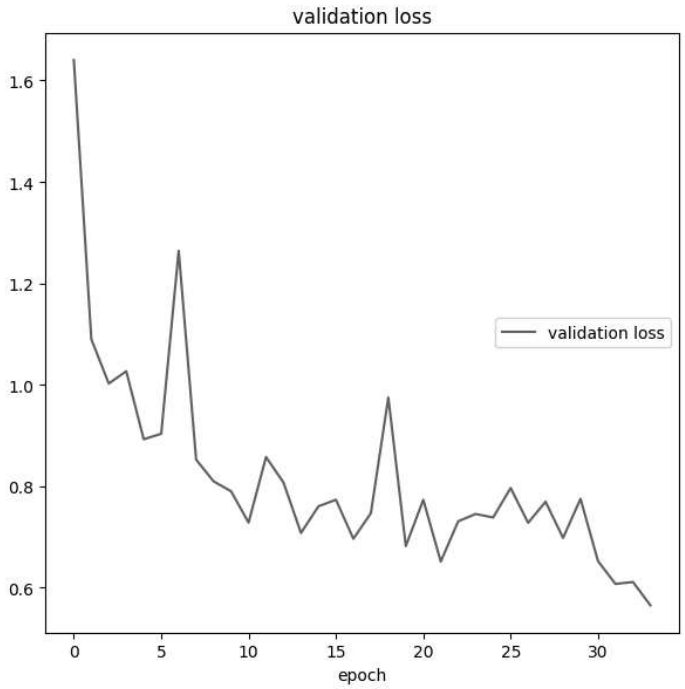

```
training loss
        training loss           (min:    0.584, max:    1.764, cur:    0.596)
validation accuracy
        validation accuracy     (min:    0.501, max:    0.823, cur:    0.758)
validation loss
        validation loss         (min:    0.565, max:    1.640, cur:    0.823)
Validation loss - test: 0.033
Test Accuracy: 0.758
```

Based on this experiment, using 34 epochs will be an ideal number. I will train a final model using these parameters.

```
final_model = train_one_model(learning_rate=0.008, gamma=0.6, n_epochs=34)
```








```
training loss
        training loss            (min:    0.589, max:    1.764, cur:    0.618)
validation accuracy
        validation accuracy      (min:    0.501, max:    0.823, cur:    0.823)
validation loss
        validation loss          (min:    0.565, max:    1.640, cur:    0.565)
Validation loss - test: 0.023
Test Accuracy: 0.823
Precision: tensor([0.4643, 0.8750, 0.9545, 0.9091, 0.7692, 0.8235, 0.7719, 0.9444, 0.7288,
        0.7083, 0.9574, 0.9189])
Recall: tensor([0.5000, 0.8974, 0.7241, 0.9836, 0.9091, 0.8750, 0.6769, 0.7727, 0.8431,
        0.7391, 0.9000, 0.8718])
```

Save final model:

```
model_scripted = torch.jit.script(final_model) # Export to TorchScript
model_scripted.save('model_scripted.pt') # Save
```