```
#!/bin/bash
!curl -L -o archive.zip\
https://www.kaggle.com/api/v1/datasets/download/kaustubhb999/tomatoleaf
```

```
!unzip archive.zip
```

```
Archive:  archive.zip
replace tomato/cnn_train.py? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

```
import torch
import torchvision

# Step 1) Dataset preparation
train_path = '/content/tomato/train'
test_path = '/content/tomato/val'
# 1) resize the image to 256x256
#torchvision.transforms.Resize(256,256),

# 2) convert input image to tensor
#torchvision.transforms.ToTensor(),

# 3) normalize the image
#torchvision.transforms.Normalize(mean=[0.5,0.5,0.5],std=[0.5,0.5,0.5])

# create an empty list
transform = [torchvision.transforms.Resize((256,256)),
 torchvision.transforms.ToTensor(),
 torchvision.transforms.Normalize(mean=[0.5,0.5,0.5],std=[0.5,0.5,0.5])]

transformation = torchvision.transforms.Compose(transform)

train_dataset = torchvision.datasets.ImageFolder(root=train_path,
                                                transform=transformation)
test_dataset = torchvision.datasets.ImageFolder(root=test_path,
                                                transform=transformation)

# split into training and testing dataset
# train_size = int(0.7*len(full_dataset))# 70% of data will be trained
# test_size = len(full_dataset) - train_size # 30% of data will be test
# train_dataset, test_dataset = torch.utils.data.random_split(full_dataset,[train_size,test_size])


print(len(train_dataset))
print(len(test_dataset))


# setting up your data loader
batch_size = 16
num_epochs = 30
num_classes = 3
learning_rate = 0.001
num_classes = 10

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Train loader
train_loader = torch.utils.data.DataLoader(train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True) # shuffle so AI learn

# Test loader
test_loader = torch.utils.data.DataLoader(test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)


# Step 3) Create CNN model
import torch

class CNN(torch.nn.Module):
    def __init__(self, num_classes=10):
        super(CNN, self).__init__()

        # First conv layer
        self.conv1 = torch.nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3, stride=1, padding=1)
        self.batch1 = torch.nn.BatchNorm2d(8)
        self.act1 = torch.nn.ReLU()
        self.pool1 = torch.nn.MaxPool2d(kernel_size=2)

        # Second conv layer
        self.conv2 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, stride=1, padding=1)
```

```
        self.batch2 = torch.nn.BatchNorm2d(16)
        self.act2 = torch.nn.ReLU()
        self.pool2 = torch.nn.MaxPool2d(kernel_size=2)

        # Third conv layer
        self.conv3 = torch.nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.batch3 = torch.nn.BatchNorm2d(32)
        self.act3 = torch.nn.ReLU()
        self.pool3 = torch.nn.MaxPool2d(kernel_size=2)

        # Fourth conv layer
        self.conv4 = torch.nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.batch4 = torch.nn.BatchNorm2d(64)
        self.act4 = torch.nn.ReLU()
        self.pool4 = torch.nn.MaxPool2d(kernel_size=2)

        # Fifth conv layer
        self.conv5 = torch.nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)
        self.batch5 = torch.nn.BatchNorm2d(128)
        self.act5 = torch.nn.ReLU()
        self.pool5 = torch.nn.MaxPool2d(kernel_size=2)

        # Flatten
        self.flatten = torch.nn.Flatten()

        # Fully connected layer
        self.fc = torch.nn.Linear(128 * 8 * 8, out_features=num_classes)

    def forward(self, x):
        # First conv layer
        out = self.conv1(x)
        out = self.batch1(out)
        out = self.act1(out)
        out = self.pool1(out)

        # Second conv layer
        out = self.conv2(out)
        out = self.batch2(out)
        out = self.act2(out)
        out = self.pool2(out)

        # Third conv layer
        out = self.conv3(out)
        out = self.batch3(out)
        out = self.act3(out)
        out = self.pool3(out)

        # Fourth conv layer
        out = self.conv4(out)
        out = self.batch4(out)
        out = self.act4(out)
        out = self.pool4(out)

        # Fifth conv layer
        out = self.conv5(out)
        out = self.batch5(out)
        out = self.act5(out)
        out = self.pool5(out)

        # Flatten
        out = self.flatten(out)

        # Fully connected layer
        out = self.fc(out)

        return torch.nn.functional.log_softmax(out, dim=1)
```

```
model = CNN(num_classes).to(device)
from torchsummary import summary

# Create the model instance
model = CNN(num_classes=10).to('cuda')  # Use 'cuda' if GPU is available, or 'cpu' otherwise

# Call the summary function
summary(model, input_size=(3, 256, 256))  # Adjust input_size based on your model
```

```
# Training the model

def test(model,test_loader,device):
```

```
    # set model to evaluation mode
    model.eval()

    with torch.no_grad():
      correct = 0
      total = 0
      for images, labels in test_loader:
        images = images.to(device) # x is already has the batch size of 100
        labels = labels.to(device)
        predicted_output = model(images)
        _, predicted = torch.max(predicted_output.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    acc = correct/total*100

    return acc

  # loss
  creiterion = torch.nn.CrossEntropyLoss()
  # optimizer
  optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

  #Train the model
  #Set the model into training mode
  epoch_loss = 0
  loss_list = [] #to store the losses in list iteration
  training_loss = [] # to store the epoch training loss
  training_acc = [] # to store training accuracy
  epoch_num = [] # to store num of epoch


  total_step = len(train_loader)
  for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

      # lets say i am using GPU, this will allow the gpu to process the data
      model.train()
      images = images.to(device)
      labels = labels.to(device)
      #label = torch.eye(num_classes)[labels].to(device)

      #forward pass
      outputs = model(images)
      loss = creiterion(outputs, labels)

      #backward pass
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

      # to calculate the loss
      epoch_loss = epoch_loss + loss.item()

      loss_list.append(epoch_loss)

      # to print out the loss for every step
      if (i+1) % 100 == 0:
        print('Epoch[{}/{}],Step [{}/{}],Loss{:.4f}'.format(epoch+1,num_epochs,i+1,len(train_loader),loss.item()))

    # training loss
    avg_loss = epoch_loss/(i+1)
    training_loss.append(avg_loss)


    #accuracy
    accuracy = test(model,test_loader,device)
    training_acc.append(accuracy)

    epoch_num.append(epoch)
    epoch_loss = 0


  # Step 3) Model evaluation

  # plot the graph
  import matplotlib.pyplot as plt
  plt.plot(epoch_num,training_acc)
  plt.show()

  # classification report
  from sklearn.metrics import confusion_matrix, classification_report
  import numpy as np
```

```
import seaborn as sns

y_pred = []
y_true = []

# iterate over test data
for images, labels in test_loader:
  images = images.to(device)  # Move images to the same device as the model
  predicted_output = model(images)# input into model becomes images
  _, predicted = torch.max(predicted_output.data, 1)
  y_pred.extend(predicted.data.cpu().numpy())

  labels = labels.data.cpu().numpy()
  y_true.extend(labels)

# confusion matrix
cf_matrix = confusion_matrix(y_true, y_pred)
print(cf_matrix)
print(classification_report(y_true, y_pred))
```

```
print(loss_list)

plt.plot(training_loss)
plt.show()
```

```
import torch
torch.save(model.state_dict(), 'Kali Turing_CNN.pt')
```