**PLANT DISEASE CLASSIFICATION USING RESNET-9 ☘️**

**Corresponding Kaggle notebook can be accessed**[**here**](https://www.kaggle.com/atharvaingle/plant-disease-classification-resnet-99-2)

**⚠️⚠️⚠️DISCLAIMER: This notebook is beginner friendly, so don't worry if you don't know much about CNNs and Pytorch. Even if you have used TensorFlow in the past and are new to PyTorch, hang in there, everything is explained clearly and concisely. You will get a good overview of how to use PyTorch for image classification problems.**

**Description of the dataset 📝**

This dataset is created using offline augmentation from the original dataset. The original PlantVillage Dataset can be found [here](https://github.com/spMohanty/PlantVillage-Dataset).This dataset consists of about 87K rgb images of healthy and diseased crop leaves which is categorized into 38 different classes. The total dataset is divided into 80/20 ratio of training and validation set preserving the directory structure. A new directory containing 33 test images is created later for prediction purpose.

Note: This description is given in the dataset itself

**Our goal 🎯**

Goal is clear and simple. We need to build a model, which can classify between healthy and diseased crop leaves and also if the crop have any disease, predict which disease is it.

**Let's get started....**

**Importing necessary libraries**

Let's import required modules

In [1]:

**!**pip install torchsummary

Collecting torchsummary

Downloading torchsummary-1.5.1-py3-none-any.whl (2.8 kB)

Installing collected packages: torchsummary

Successfully installed torchsummary-1.5.1

We would require torchsummary library to print the model's summary in keras style (nicely formatted and pretty to look) as Pytorch natively doesn't support that

In [2]:

**import** os *# for working with files*

**import** numpy **as** np *# for numerical computationss*

**import** pandas **as** pd *# for working with dataframes*

**import** torch *# Pytorch module*

**import** matplotlib.pyplot **as** plt *# for plotting informations on graph and images using tensors*

**import** torch.nn **as** nn *# for creating neural networks*

**from** torch.utils.data **import** DataLoader *# for dataloaders*

**from** PIL **import** Image *# for checking images*

**import** torch.nn.functional **as** F *# for functions for calculating loss*

**import** torchvision.transforms **as** transforms *# for transforming images into tensors*

**from** torchvision.utils **import** make\_grid *# for data checking*

**from** torchvision.datasets **import** ImageFolder *# for working with classes and images*

**from** torchsummary **import** summary *# for getting the summary of our model*

**%matplotlib** inline

**🧭 Exploring the data 🧭**

Loading the data

In [3]:

data\_dir **=** "../input/new-plant-diseases-dataset/New Plant Diseases Dataset(Augmented)/New Plant Diseases Dataset(Augmented)"

train\_dir **=** data\_dir **+** "/train"

valid\_dir **=** data\_dir **+** "/valid"

diseases **=** os**.**listdir(train\_dir)

In [4]:

*# printing the disease names*

print(diseases)

['Tomato\_\_\_Late\_blight', 'Tomato\_\_\_healthy', 'Grape\_\_\_healthy', 'Orange\_\_\_Haunglongbing\_(Citrus\_greening)', 'Soybean\_\_\_healthy', 'Squash\_\_\_Powdery\_mildew', 'Potato\_\_\_healthy', 'Corn\_(maize)\_\_\_Northern\_Leaf\_Blight', 'Tomato\_\_\_Early\_blight', 'Tomato\_\_\_Septoria\_leaf\_spot', 'Corn\_(maize)\_\_\_Cercospora\_leaf\_spot Gray\_leaf\_spot', 'Strawberry\_\_\_Leaf\_scorch', 'Peach\_\_\_healthy', 'Apple\_\_\_Apple\_scab', 'Tomato\_\_\_Tomato\_Yellow\_Leaf\_Curl\_Virus', 'Tomato\_\_\_Bacterial\_spot', 'Apple\_\_\_Black\_rot', 'Blueberry\_\_\_healthy', 'Cherry\_(including\_sour)\_\_\_Powdery\_mildew', 'Peach\_\_\_Bacterial\_spot', 'Apple\_\_\_Cedar\_apple\_rust', 'Tomato\_\_\_Target\_Spot', 'Pepper,\_bell\_\_\_healthy', 'Grape\_\_\_Leaf\_blight\_(Isariopsis\_Leaf\_Spot)', 'Potato\_\_\_Late\_blight', 'Tomato\_\_\_Tomato\_mosaic\_virus', 'Strawberry\_\_\_healthy', 'Apple\_\_\_healthy', 'Grape\_\_\_Black\_rot', 'Potato\_\_\_Early\_blight', 'Cherry\_(including\_sour)\_\_\_healthy', 'Corn\_(maize)\_\_\_Common\_rust\_', 'Grape\_\_\_Esca\_(Black\_Measles)', 'Raspberry\_\_\_healthy', 'Tomato\_\_\_Leaf\_Mold', 'Tomato\_\_\_Spider\_mites Two-spotted\_spider\_mite', 'Pepper,\_bell\_\_\_Bacterial\_spot', 'Corn\_(maize)\_\_\_healthy']

In [5]:

print("Total disease classes are: {}"**.**format(len(diseases)))

Total disease classes are: 38

In [6]:

plants **=** []

NumberOfDiseases **=** 0

**for** plant **in** diseases:

**if** plant**.**split('\_\_\_')[0] **not** **in** plants:

plants**.**append(plant**.**split('\_\_\_')[0])

**if** plant**.**split('\_\_\_')[1] **!=** 'healthy':

NumberOfDiseases **+=** 1

The above cell extract the number of unique plants and number of unique diseases

In [7]:

*# unique plants in the dataset*

print(f"Unique Plants are: \n{plants}")

Unique Plants are:

['Tomato', 'Grape', 'Orange', 'Soybean', 'Squash', 'Potato', 'Corn\_(maize)', 'Strawberry', 'Peach', 'Apple', 'Blueberry', 'Cherry\_(including\_sour)', 'Pepper,\_bell', 'Raspberry']

In [8]:

*# number of unique plants*

print("Number of plants: {}"**.**format(len(plants)))

Number of plants: 14

In [9]:

*# number of unique diseases*

print("Number of diseases: {}"**.**format(NumberOfDiseases))

Number of diseases: 26

So we have images of leaves of 14 plants and while excluding healthy leaves, we have 26 types of images that show a particular disease in a particular plant.

In [10]:

*# Number of images for each disease*

nums **=** {}

**for** disease **in** diseases:

nums[disease] **=** len(os**.**listdir(train\_dir **+** '/' **+** disease))

*# converting the nums dictionary to pandas dataframe passing index as plant name and number of images as column*

img\_per\_class **=** pd**.**DataFrame(nums**.**values(), index**=**nums**.**keys(), columns**=**["no. of images"])

img\_per\_class

Out[10]:

|  | **no. of images** |
| --- | --- |
| **Tomato\_\_\_Late\_blight** | 1851 |
| **Tomato\_\_\_healthy** | 1926 |
| **Grape\_\_\_healthy** | 1692 |
| **Orange\_\_\_Haunglongbing\_(Citrus\_greening)** | 2010 |
| **Soybean\_\_\_healthy** | 2022 |
| **Squash\_\_\_Powdery\_mildew** | 1736 |
| **Potato\_\_\_healthy** | 1824 |
| **Corn\_(maize)\_\_\_Northern\_Leaf\_Blight** | 1908 |
| **Tomato\_\_\_Early\_blight** | 1920 |
| **Tomato\_\_\_Septoria\_leaf\_spot** | 1745 |
| **Corn\_(maize)\_\_\_Cercospora\_leaf\_spot Gray\_leaf\_spot** | 1642 |
| **Strawberry\_\_\_Leaf\_scorch** | 1774 |
| **Peach\_\_\_healthy** | 1728 |
| **Apple\_\_\_Apple\_scab** | 2016 |
| **Tomato\_\_\_Tomato\_Yellow\_Leaf\_Curl\_Virus** | 1961 |
| **Tomato\_\_\_Bacterial\_spot** | 1702 |
| **Apple\_\_\_Black\_rot** | 1987 |
| **Blueberry\_\_\_healthy** | 1816 |
| **Cherry\_(including\_sour)\_\_\_Powdery\_mildew** | 1683 |
| **Peach\_\_\_Bacterial\_spot** | 1838 |
| **Apple\_\_\_Cedar\_apple\_rust** | 1760 |
| **Tomato\_\_\_Target\_Spot** | 1827 |
| **Pepper,\_bell\_\_\_healthy** | 1988 |
| **Grape\_\_\_Leaf\_blight\_(Isariopsis\_Leaf\_Spot)** | 1722 |
| **Potato\_\_\_Late\_blight** | 1939 |
| **Tomato\_\_\_Tomato\_mosaic\_virus** | 1790 |
| **Strawberry\_\_\_healthy** | 1824 |
| **Apple\_\_\_healthy** | 2008 |
| **Grape\_\_\_Black\_rot** | 1888 |
| **Potato\_\_\_Early\_blight** | 1939 |
| **Cherry\_(including\_sour)\_\_\_healthy** | 1826 |
| **Corn\_(maize)\_\_\_Common\_rust\_** | 1907 |
| **Grape\_\_\_Esca\_(Black\_Measles)** | 1920 |
| **Raspberry\_\_\_healthy** | 1781 |
| **Tomato\_\_\_Leaf\_Mold** | 1882 |
| **Tomato\_\_\_Spider\_mites Two-spotted\_spider\_mite** | 1741 |
| **Pepper,\_bell\_\_\_Bacterial\_spot** | 1913 |
| **Corn\_(maize)\_\_\_healthy** | 1859 |

**Visualizing the above information on a graph**

In [11]:

*# plotting number of images available for each disease*

index **=** [n **for** n **in** range(38)]

plt**.**figure(figsize**=**(20, 5))

plt**.**bar(index, [n **for** n **in** nums**.**values()], width**=**0.3)

plt**.**xlabel('Plants/Diseases', fontsize**=**10)

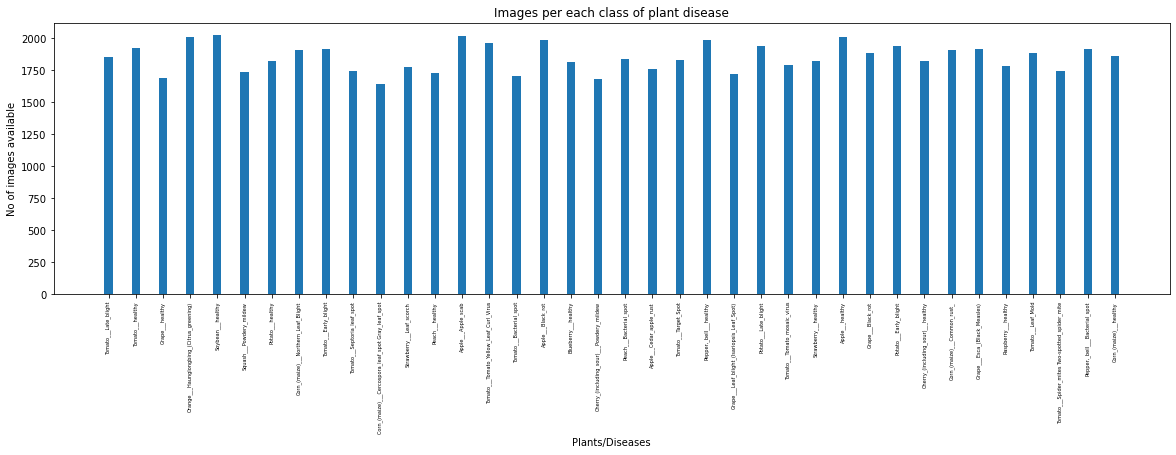
plt**.**ylabel('No of images available', fontsize**=**10)

plt**.**xticks(index, diseases, fontsize**=**5, rotation**=**90)

plt**.**title('Images per each class of plant disease')

Out[11]:

Text(0.5, 1.0, 'Images per each class of plant disease')



We can see that the dataset is almost balanced for all classes, so we are good to go forward

**Images available for training**

In [12]:

n\_train **=** 0

**for** value **in** nums**.**values():

n\_train **+=** value

print(f"There are {n\_train} images for training")

There are 70295 images for training

**🍳 Data Preparation for training 🍳**

In [13]:

*# datasets for validation and training*

train **=** ImageFolder(train\_dir, transform**=**transforms**.**ToTensor())

valid **=** ImageFolder(valid\_dir, transform**=**transforms**.**ToTensor())

torchvision.datasets is a class which helps in loading all common and famous datasets. It also helps in loading custom datasets. I have used subclass torchvision.datasets.ImageFolder which helps in loading the image data when the data is arranged in this way:

root/dog/xxx.png

root/dog/xxy.png

root/dog/xxz.png

root/cat/123.png

root/cat/nsdf3.png

root/cat/asd932\_.png

Next, after loading the data, we need to transform the pixel values of each image (0-255) to 0-1 as neural networks works quite good with normalized data. The entire array of pixel values is converted to torch [tensor](https://pytorch.org/tutorials/beginner/examples_tensor/two_layer_net_tensor.html#:~:text=A%20PyTorch%20Tensor%20is%20basically,used%20for%20arbitrary%20numeric%20computation.) and then divided by 255. If you are not familiar why normalizing inputs help neural network, read [this](https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d) post.

**Image shape**

In [14]:

img, label **=** train[0]

print(img**.**shape, label)

torch.Size([3, 256, 256]) 0

We can see the shape (3, 256 256) of the image. 3 is the number of channels (RGB) and 256 x 256 is the width and height of the image

In [15]:

*# total number of classes in train set*

len(train**.**classes)

Out[15]:

38

In [16]:

*# for checking some images from training dataset*

**def** show\_image(image, label):

print("Label :" **+** train**.**classes[label] **+** "(" **+** str(label) **+** ")")

plt**.**imshow(image**.**permute(1, 2, 0))

**🖼️ Some Images from training dataset 🖼️**

In [17]:

show\_image(**\***train[0])

Label :Apple\_\_\_Apple\_scab(0)



In [18]:

show\_image(**\***train[70000])

Label :Tomato\_\_\_healthy(37)



In [19]:

show\_image(**\***train[30000])

Label :Peach\_\_\_Bacterial\_spot(16)



In [20]:

*# Setting the seed value*

random\_seed **=** 7

torch**.**manual\_seed(random\_seed)

Out[20]:

In [21]:

*# setting the batch size*

batch\_size **=** 32

batch\_size is the total number of images given as input at once in forward propagation of the CNN. Basically, batch size defines the number of samples that will be propagated through the network.

For instance, let's say you have 1050 training samples and you want to set up a batch\_size equal to 100. The algorithm takes the first 100 samples (from 1st to 100th) from the training dataset and trains the network. Next, it takes the second 100 samples (from 101st to 200th) and trains the network again. We can keep doing this procedure until we have propagated all samples through of the network.

In [22]:

*# DataLoaders for training and validation*

train\_dl **=** DataLoader(train, batch\_size, shuffle**=True**, num\_workers**=**2, pin\_memory**=True**)

valid\_dl **=** DataLoader(valid, batch\_size, num\_workers**=**2, pin\_memory**=True**)

* DataLoader is a subclass which comes from torch.utils.data. It helps in loading large and memory consuming datasets. It takes in batch\_size which denotes the number of samples contained in each generated batch.
* Setting shuffle=True shuffles the dataset. It is heplful so that batches between epochs do not look alike. Doing so will eventually make our model more robust.
* num\_workers, denotes the number of processes that generate batches in parallel. If you have more cores in your CPU, you can set it to number of cores in your CPU. Since, Kaggle provides a 2 core CPU, I have set it to 2

In [23]:

*# helper function to show a batch of training instances*

**def** show\_batch(data):

**for** images, labels **in** data:

fig, ax **=** plt**.**subplots(figsize**=**(30, 30))

ax**.**set\_xticks([]); ax**.**set\_yticks([])

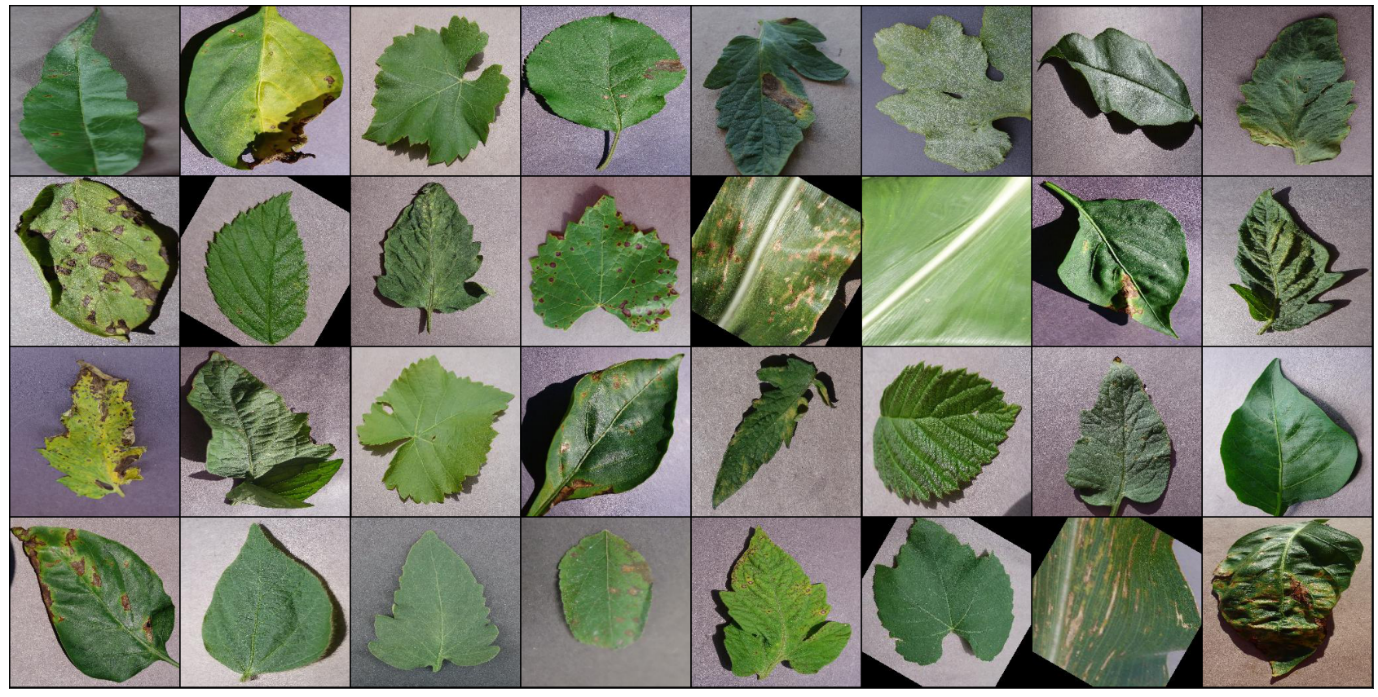
ax**.**imshow(make\_grid(images, nrow**=**8)**.**permute(1, 2, 0))

**break**

In [24]:

*# Images for first batch of training*

show\_batch(train\_dl)



**🏗️ Modelling 🏗️**

It is advisable to use GPU instead of CPU when dealing with images dataset because CPUs are generalized for general purpose and GPUs are optimized for training deep learning models as they can process multiple computations simultaneously. They have a large number of cores, which allows for better computation of multiple parallel processes. Additionally, computations in deep learning need to handle huge amounts of data — this makes a GPU’s memory bandwidth most suitable. To seamlessly use a GPU, if one is available, we define a couple of helper functions (get\_default\_device & to\_device) and a helper class DeviceDataLoader to move our model & data to the GPU as required

**Some helper functions**

In [25]:

*# for moving data into GPU (if available)*

**def** get\_default\_device():

"""Pick GPU if available, else CPU"""

**if** torch**.**cuda**.**is\_available:

**return** torch**.**device("cuda")

**else**:

**return** torch**.**device("cpu")

*# for moving data to device (CPU or GPU)*

**def** to\_device(data, device):

"""Move tensor(s) to chosen device"""

**if** isinstance(data, (list,tuple)):

**return** [to\_device(x, device) **for** x **in** data]

**return** data**.**to(device, non\_blocking**=True**)

*# for loading in the device (GPU if available else CPU)*

**class** DeviceDataLoader():

"""Wrap a dataloader to move data to a device"""

**def** \_\_init\_\_(self, dl, device):

self**.**dl **=** dl

self**.**device **=** device

**def** \_\_iter\_\_(self):

"""Yield a batch of data after moving it to device"""

**for** b **in** self**.**dl:

**yield** to\_device(b, self**.**device)

**def** \_\_len\_\_(self):

"""Number of batches"""

**return** len(self**.**dl)

Checking the device we are working with

In [26]:

device **=** get\_default\_device()

device

Out[26]:

device(type='cuda')

Wrap up our training and validation data loaders using DeviceDataLoader for automatically transferring batches of data to the GPU (if available)

In [27]:

*# Moving data into GPU*

train\_dl **=** DeviceDataLoader(train\_dl, device)

valid\_dl **=** DeviceDataLoader(valid\_dl, device)

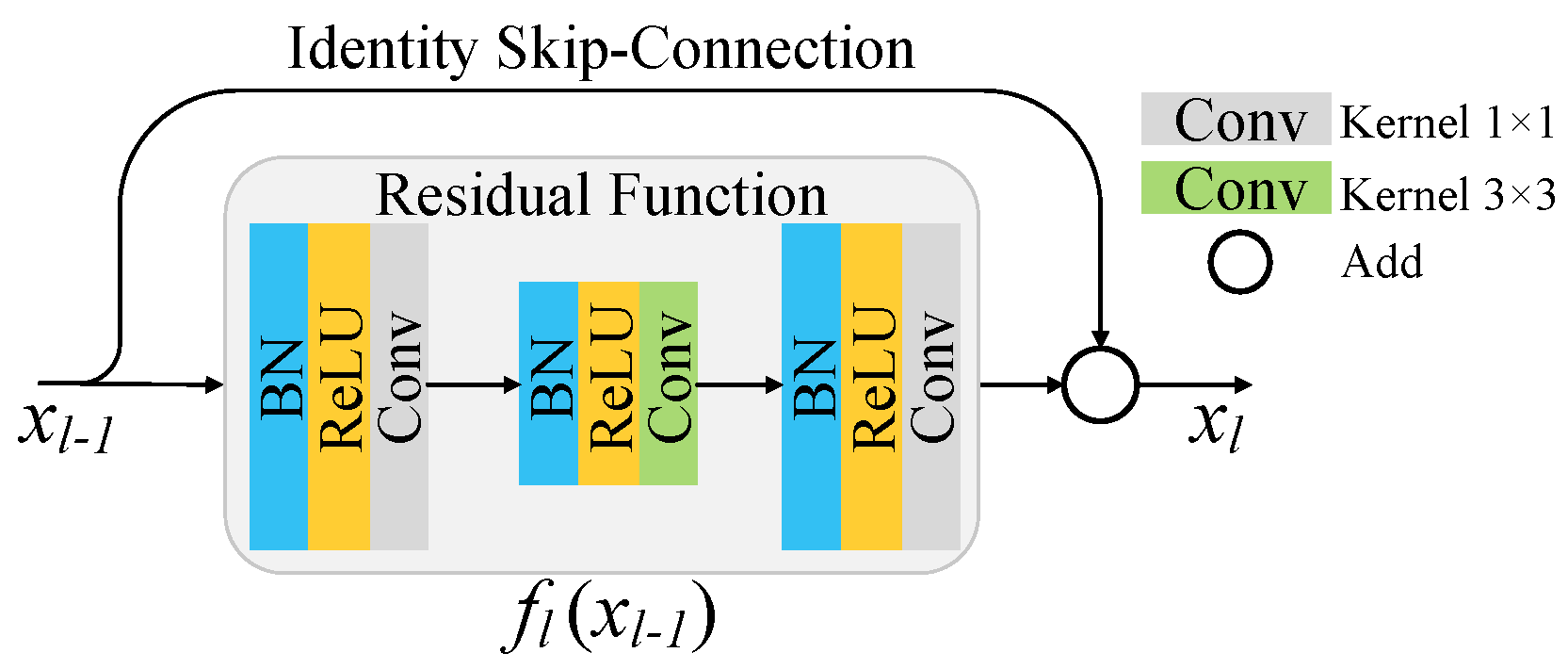
**👷 Building the model architecture 👷**

*We are going to use ResNet, which have been one of the major breakthrough in computer vision since they were introduced in 2015.*

If you want to learn more about ResNets read the following articles:

* [Understanding and Visualizing ResNets](https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8#:~:text=ResNet%20Layers,layers%20remains%20the%20same%20%E2%80%94%204.)
* [Overview of ResNet and its variants](https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035)
* [Paper with code implementation](https://paperswithcode.com/method/resnet)

In ResNets, unlike in traditional neural networks, each layer feeds into the next layer, we use a network with residual blocks, each layer feeds into the next layer and directly into the layers about 2–3 hops away, to avoid over-fitting (a situation when validation loss stop decreasing at a point and then keeps increasing while training loss still decreases). This also helps in preventing [vanishing gradient problem](https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484) and allow us to train deep neural networks. Here is a simple residual block:



**Residual Block code implementation**

In [28]:

**class** SimpleResidualBlock(nn**.**Module):

**def** \_\_init\_\_(self):

super()**.**\_\_init\_\_()

self**.**conv1 **=** nn**.**Conv2d(in\_channels**=**3, out\_channels**=**3, kernel\_size**=**3, stride**=**1, padding**=**1)

self**.**relu1 **=** nn**.**ReLU()

self**.**conv2 **=** nn**.**Conv2d(in\_channels**=**3, out\_channels**=**3, kernel\_size**=**3, stride**=**1, padding**=**1)

self**.**relu2 **=** nn**.**ReLU()

**def** forward(self, x):

out **=** self**.**conv1(x)

out **=** self**.**relu1(out)

out **=** self**.**conv2(out)

**return** self**.**relu2(out) **+** x *# ReLU can be applied before or after adding the input*

**Then we define our ImageClassificationBase class whose functions are:**

* training\_step - To figure out how “wrong” the model is going after training or validation step.We are using this function other than just an accuracy metric that is likely not going to be differentiable (this would mean that the gradient can’t be determined, which is necessary for the model to improve during training)

A quick look at the PyTorch docs that yields the cost function: [cross\_entropy](https://pytorch.org/docs/stable/nn.functional.html" \l "cross-entropy).

* validation\_step - Because an accuracy metric can’t be used while training the model, doesn’t mean it shouldn’t be implemented! Accuracy in this case would be measured by a threshold, and counted if the difference between the model’s prediction and the actual label is lower than that threshold.
* validation\_epoch\_end - We want to track the validation losses/accuracies and train losses after each epoch, and every time we do so we have to make sure the gradient is not being tracked.
* epoch\_end - We also want to print validation losses/accuracies, train losses and learning rate too because we are using learning rate scheduler (which will change the learning rate after every batch of training) after each epoch.

We also define an accuracy function which calculates the overall accuracy of the model on an entire batch of outputs, so that we can use it as a metric in fit\_one\_cycle

In [29]:

*# for calculating the accuracy*

**def** accuracy(outputs, labels):

\_, preds **=** torch**.**max(outputs, dim**=**1)

**return** torch**.**tensor(torch**.**sum(preds **==** labels)**.**item() **/** len(preds))

*# base class for the model*

**class** ImageClassificationBase(nn**.**Module):

**def** training\_step(self, batch):

images, labels **=** batch

out **=** self(images) *# Generate predictions*

loss **=** F**.**cross\_entropy(out, labels) *# Calculate loss*

**return** loss

**def** validation\_step(self, batch):

images, labels **=** batch

out **=** self(images) *# Generate prediction*

loss **=** F**.**cross\_entropy(out, labels) *# Calculate loss*

acc **=** accuracy(out, labels) *# Calculate accuracy*

**return** {"val\_loss": loss**.**detach(), "val\_accuracy": acc}

**def** validation\_epoch\_end(self, outputs):

batch\_losses **=** [x["val\_loss"] **for** x **in** outputs]

batch\_accuracy **=** [x["val\_accuracy"] **for** x **in** outputs]

epoch\_loss **=** torch**.**stack(batch\_losses)**.**mean() *# Combine loss*

epoch\_accuracy **=** torch**.**stack(batch\_accuracy)**.**mean()

**return** {"val\_loss": epoch\_loss, "val\_accuracy": epoch\_accuracy} *# Combine accuracies*

**def** epoch\_end(self, epoch, result):

print("Epoch [{}], last\_lr: {:.5f}, train\_loss: {:.4f}, val\_loss: {:.4f}, val\_acc: {:.4f}"**.**format(

epoch, result['lrs'][**-**1], result['train\_loss'], result['val\_loss'], result['val\_accuracy']))

**👷 Defining the final architecture of our model 👷**

In [30]:

*# Architecture for training*

*# convolution block with BatchNormalization*

**def** ConvBlock(in\_channels, out\_channels, pool**=False**):

layers **=** [nn**.**Conv2d(in\_channels, out\_channels, kernel\_size**=**3, padding**=**1),

nn**.**BatchNorm2d(out\_channels),

nn**.**ReLU(inplace**=True**)]

**if** pool:

layers**.**append(nn**.**MaxPool2d(4))

**return** nn**.**Sequential(**\***layers)

*# resnet architecture*

**class** ResNet9(ImageClassificationBase):

**def** \_\_init\_\_(self, in\_channels, num\_diseases):

super()**.**\_\_init\_\_()

self**.**conv1 **=** ConvBlock(in\_channels, 64)

self**.**conv2 **=** ConvBlock(64, 128, pool**=True**) *# out\_dim : 128 x 64 x 64*

self**.**res1 **=** nn**.**Sequential(ConvBlock(128, 128), ConvBlock(128, 128))

self**.**conv3 **=** ConvBlock(128, 256, pool**=True**) *# out\_dim : 256 x 16 x 16*

self**.**conv4 **=** ConvBlock(256, 512, pool**=True**) *# out\_dim : 512 x 4 x 44*

self**.**res2 **=** nn**.**Sequential(ConvBlock(512, 512), ConvBlock(512, 512))

self**.**classifier **=** nn**.**Sequential(nn**.**MaxPool2d(4),

nn**.**Flatten(),

nn**.**Linear(512, num\_diseases))

**def** forward(self, xb): *# xb is the loaded batch*

out **=** self**.**conv1(xb)

out **=** self**.**conv2(out)

out **=** self**.**res1(out) **+** out

out **=** self**.**conv3(out)

out **=** self**.**conv4(out)

out **=** self**.**res2(out) **+** out

out **=** self**.**classifier(out)

**return** out

Now, we define a model object and transfer it into the device with which we are working...

In [31]:

*# defining the model and moving it to the GPU*

model **=** to\_device(ResNet9(3, len(train**.**classes)), device)

model

Out[31]:

ResNet9(

(conv1): Sequential(

(0): Conv2d(3, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(2): ReLU(inplace=True)

)

(conv2): Sequential(

(0): Conv2d(64, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(2): ReLU(inplace=True)

(3): MaxPool2d(kernel\_size=4, stride=4, padding=0, dilation=1, ceil\_mode=False)

)

(res1): Sequential(

(0): Sequential(

(0): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(2): ReLU(inplace=True)

)

(1): Sequential(

(0): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(2): ReLU(inplace=True)

)

)

(conv3): Sequential(

(0): Conv2d(128, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(2): ReLU(inplace=True)

(3): MaxPool2d(kernel\_size=4, stride=4, padding=0, dilation=1, ceil\_mode=False)

)

(conv4): Sequential(

(0): Conv2d(256, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(2): ReLU(inplace=True)

(3): MaxPool2d(kernel\_size=4, stride=4, padding=0, dilation=1, ceil\_mode=False)

)

(res2): Sequential(

(0): Sequential(

(0): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(2): ReLU(inplace=True)

)

(1): Sequential(

(0): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(2): ReLU(inplace=True)

)

)

(classifier): Sequential(

(0): MaxPool2d(kernel\_size=4, stride=4, padding=0, dilation=1, ceil\_mode=False)

(1): Flatten(start\_dim=1, end\_dim=-1)

(2): Linear(in\_features=512, out\_features=38, bias=True)

)

)

*Getting a nicely formatted summary of our model (like in Keras). Pytorch doesn't support it natively. So, we need to install the torchsummary library (discussed earlier)*

In [32]:

*# getting summary of the model*

INPUT\_SHAPE **=** (3, 256, 256)

print(summary(model**.**cuda(), (INPUT\_SHAPE)))

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

Layer (type) Output Shape Param #

================================================================

Conv2d-1 [-1, 64, 256, 256] 1,792

BatchNorm2d-2 [-1, 64, 256, 256] 128

ReLU-3 [-1, 64, 256, 256] 0

Conv2d-4 [-1, 128, 256, 256] 73,856

BatchNorm2d-5 [-1, 128, 256, 256] 256

ReLU-6 [-1, 128, 256, 256] 0

MaxPool2d-7 [-1, 128, 64, 64] 0

Conv2d-8 [-1, 128, 64, 64] 147,584

BatchNorm2d-9 [-1, 128, 64, 64] 256

ReLU-10 [-1, 128, 64, 64] 0

Conv2d-11 [-1, 128, 64, 64] 147,584

BatchNorm2d-12 [-1, 128, 64, 64] 256

ReLU-13 [-1, 128, 64, 64] 0

Conv2d-14 [-1, 256, 64, 64] 295,168

BatchNorm2d-15 [-1, 256, 64, 64] 512

ReLU-16 [-1, 256, 64, 64] 0

MaxPool2d-17 [-1, 256, 16, 16] 0

Conv2d-18 [-1, 512, 16, 16] 1,180,160

BatchNorm2d-19 [-1, 512, 16, 16] 1,024

ReLU-20 [-1, 512, 16, 16] 0

MaxPool2d-21 [-1, 512, 4, 4] 0

Conv2d-22 [-1, 512, 4, 4] 2,359,808

BatchNorm2d-23 [-1, 512, 4, 4] 1,024

ReLU-24 [-1, 512, 4, 4] 0

Conv2d-25 [-1, 512, 4, 4] 2,359,808

BatchNorm2d-26 [-1, 512, 4, 4] 1,024

ReLU-27 [-1, 512, 4, 4] 0

MaxPool2d-28 [-1, 512, 1, 1] 0

Flatten-29 [-1, 512] 0

Linear-30 [-1, 38] 19,494

================================================================

Total params: 6,589,734

Trainable params: 6,589,734

Non-trainable params: 0

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

Input size (MB): 0.75

Forward/backward pass size (MB): 343.95

Params size (MB): 25.14

Estimated Total Size (MB): 369.83

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

None

**🏋️ Training the model 🏋️**

Before we train the model, Let’s define a utility functionan evaluate function, which will perform the validation phase, and a fit\_one\_cycle function which will perform the entire training process. In fit\_one\_cycle, we have use some techniques:

* **Learning Rate Scheduling**: Instead of using a fixed learning rate, we will use a learning rate scheduler, which will change the learning rate after every batch of training. There are many strategies for varying the learning rate during training, and the one we’ll use is called the *“One Cycle Learning Rate Policy”*, which involves starting with a low learning rate, gradually increasing it batch-by-batch to a high learning rate for about 30% of epochs, then gradually decreasing it to a very low value for the remaining epochs.
* **Weight Decay**: We also use weight decay, which is a regularization technique which prevents the weights from becoming too large by adding an additional term to the loss function.
* **Gradient Clipping**: Apart from the layer weights and outputs, it also helpful to limit the values of gradients to a small range to prevent undesirable changes in parameters due to large gradient values. This simple yet effective technique is called gradient clipping.

We'll also record the learning rate used for each batch.

In [33]:

*# for training*

@torch**.**no\_grad()

**def** evaluate(model, val\_loader):

model**.**eval()

outputs **=** [model**.**validation\_step(batch) **for** batch **in** val\_loader]

**return** model**.**validation\_epoch\_end(outputs)

**def** get\_lr(optimizer):

**for** param\_group **in** optimizer**.**param\_groups:

**return** param\_group['lr']

**def** fit\_OneCycle(epochs, max\_lr, model, train\_loader, val\_loader, weight\_decay**=**0,

grad\_clip**=None**, opt\_func**=**torch**.**optim**.**SGD):

torch**.**cuda**.**empty\_cache()

history **=** []

optimizer **=** opt\_func(model**.**parameters(), max\_lr, weight\_decay**=**weight\_decay)

*# scheduler for one cycle learniing rate*

sched **=** torch**.**optim**.**lr\_scheduler**.**OneCycleLR(optimizer, max\_lr, epochs**=**epochs, steps\_per\_epoch**=**len(train\_loader))

**for** epoch **in** range(epochs):

*# Training*

model**.**train()

train\_losses **=** []

lrs **=** []

**for** batch **in** train\_loader:

loss **=** model**.**training\_step(batch)

train\_losses**.**append(loss)

loss**.**backward()

*# gradient clipping*

**if** grad\_clip:

nn**.**utils**.**clip\_grad\_value\_(model**.**parameters(), grad\_clip)

optimizer**.**step()

optimizer**.**zero\_grad()

*# recording and updating learning rates*

lrs**.**append(get\_lr(optimizer))

sched**.**step()

*# validation*

result **=** evaluate(model, val\_loader)

result['train\_loss'] **=** torch**.**stack(train\_losses)**.**mean()**.**item()

result['lrs'] **=** lrs

model**.**epoch\_end(epoch, result)

history**.**append(result)

**return** history

Let's check our validation loss and accuracy

In [34]:

**%%time**

history **=** [evaluate(model, valid\_dl)]

history

CPU times: user 44 s, sys: 3.28 s, total: 47.3 s

Wall time: 1min 32s

Out[34]:

[{'val\_loss': tensor(3.6397, device='cuda:0'), 'val\_accuracy': tensor(0.0191)}]

Since there are randomly initialized weights, that is why accuracy come to near 0.019 (that is 1.9% chance of getting the right answer or you can say model randomly chooses a class). Now, declare some hyper parameters for the training of the model. We can change it if result is not satisfactory.

In [35]:

epochs **=** 2

max\_lr **=** 0.01

grad\_clip **=** 0.1

weight\_decay **=** 1e-4

opt\_func **=** torch**.**optim**.**Adam

Let's start training our model ....

Note: The following cell may take 15 mins to 45 mins to run depending on your GPU. In kaggle (P100 GPU) it took around 20 mins of Wall Time.

In [36]:

**%%time**

history **+=** fit\_OneCycle(epochs, max\_lr, model, train\_dl, valid\_dl,

grad\_clip**=**grad\_clip,

weight\_decay**=**1e-4,

opt\_func**=**opt\_func)

Epoch [0], last\_lr: 0.00812, train\_loss: 0.7466, val\_loss: 0.5865, val\_acc: 0.8319

Epoch [1], last\_lr: 0.00000, train\_loss: 0.1248, val\_loss: 0.0269, val\_acc: 0.9923

CPU times: user 11min 16s, sys: 7min 13s, total: 18min 30s

Wall time: 19min 53s

**We got an accuracy of 99.2 % 🙌🙌**

**📈 Plotting 📈**

**Helper functions for plotting**

In [37]:

**def** plot\_accuracies(history):

accuracies **=** [x['val\_accuracy'] **for** x **in** history]

plt**.**plot(accuracies, '-x')

plt**.**xlabel('epoch')

plt**.**ylabel('accuracy')

plt**.**title('Accuracy vs. No. of epochs');

**def** plot\_losses(history):

train\_losses **=** [x**.**get('train\_loss') **for** x **in** history]

val\_losses **=** [x['val\_loss'] **for** x **in** history]

plt**.**plot(train\_losses, '-bx')

plt**.**plot(val\_losses, '-rx')

plt**.**xlabel('epoch')

plt**.**ylabel('loss')

plt**.**legend(['Training', 'Validation'])

plt**.**title('Loss vs. No. of epochs');

**def** plot\_lrs(history):

lrs **=** np**.**concatenate([x**.**get('lrs', []) **for** x **in** history])

plt**.**plot(lrs)

plt**.**xlabel('Batch no.')

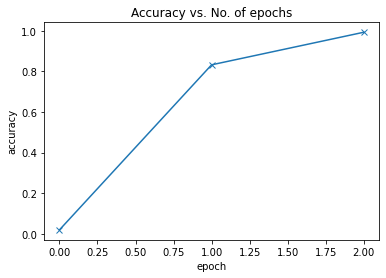
plt**.**ylabel('Learning rate')

plt**.**title('Learning Rate vs. Batch no.');

**Validation Accuracy**

In [38]:

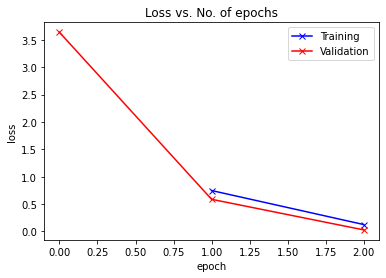
plot\_accuracies(history)



**Validation loss**

In [39]:

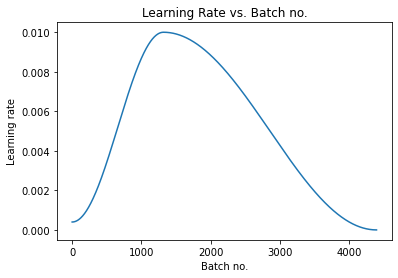
plot\_losses(history)



**Learning Rate overtime**

In [40]:

plot\_lrs(history)



**🧪 Testing model on test data 🧪**

**We only have 33 images in test data, so let's check the model on all images**

In [41]:

test\_dir **=** "../input/new-plant-diseases-dataset/test"

test **=** ImageFolder(test\_dir, transform**=**transforms**.**ToTensor())

In [42]:

test\_images **=** sorted(os**.**listdir(test\_dir **+** '/test')) *# since images in test folder are in alphabetical order*

test\_images

Out[42]:

['AppleCedarRust1.JPG',

'AppleCedarRust2.JPG',

'AppleCedarRust3.JPG',

'AppleCedarRust4.JPG',

'AppleScab1.JPG',

'AppleScab2.JPG',

'AppleScab3.JPG',

'CornCommonRust1.JPG',

'CornCommonRust2.JPG',

'CornCommonRust3.JPG',

'PotatoEarlyBlight1.JPG',

'PotatoEarlyBlight2.JPG',

'PotatoEarlyBlight3.JPG',

'PotatoEarlyBlight4.JPG',

'PotatoEarlyBlight5.JPG',

'PotatoHealthy1.JPG',

'PotatoHealthy2.JPG',

'TomatoEarlyBlight1.JPG',

'TomatoEarlyBlight2.JPG',

'TomatoEarlyBlight3.JPG',

'TomatoEarlyBlight4.JPG',

'TomatoEarlyBlight5.JPG',

'TomatoEarlyBlight6.JPG',

'TomatoHealthy1.JPG',

'TomatoHealthy2.JPG',

'TomatoHealthy3.JPG',

'TomatoHealthy4.JPG',

'TomatoYellowCurlVirus1.JPG',

'TomatoYellowCurlVirus2.JPG',

'TomatoYellowCurlVirus3.JPG',

'TomatoYellowCurlVirus4.JPG',

'TomatoYellowCurlVirus5.JPG',

'TomatoYellowCurlVirus6.JPG']

In [43]:

**def** predict\_image(img, model):

"""Converts image to array and return the predicted class

with highest probability"""

*# Convert to a batch of 1*

xb **=** to\_device(img**.**unsqueeze(0), device)

*# Get predictions from model*

yb **=** model(xb)

*# Pick index with highest probability*

\_, preds **=** torch**.**max(yb, dim**=**1)

*# Retrieve the class label*

**return** train**.**classes[preds[0]**.**item()]

In [44]:

*# predicting first image*

img, label **=** test[0]

plt**.**imshow(img**.**permute(1, 2, 0))

print('Label:', test\_images[0], ', Predicted:', predict\_image(img, model))

Label: AppleCedarRust1.JPG , Predicted: Apple\_\_\_Cedar\_apple\_rust



In [45]:

*# getting all predictions (actual label vs predicted)*

**for** i, (img, label) **in** enumerate(test):

print('Label:', test\_images[i], ', Predicted:', predict\_image(img, model))

Label: AppleCedarRust1.JPG , Predicted: Apple\_\_\_Cedar\_apple\_rust

Label: AppleCedarRust2.JPG , Predicted: Apple\_\_\_Cedar\_apple\_rust

Label: AppleCedarRust3.JPG , Predicted: Apple\_\_\_Cedar\_apple\_rust

Label: AppleCedarRust4.JPG , Predicted: Apple\_\_\_Cedar\_apple\_rust

Label: AppleScab1.JPG , Predicted: Apple\_\_\_Apple\_scab

Label: AppleScab2.JPG , Predicted: Apple\_\_\_Apple\_scab

Label: AppleScab3.JPG , Predicted: Apple\_\_\_Apple\_scab

Label: CornCommonRust1.JPG , Predicted: Corn\_(maize)\_\_\_Common\_rust\_

Label: CornCommonRust2.JPG , Predicted: Corn\_(maize)\_\_\_Common\_rust\_

Label: CornCommonRust3.JPG , Predicted: Corn\_(maize)\_\_\_Common\_rust\_

Label: PotatoEarlyBlight1.JPG , Predicted: Potato\_\_\_Early\_blight

Label: PotatoEarlyBlight2.JPG , Predicted: Potato\_\_\_Early\_blight

Label: PotatoEarlyBlight3.JPG , Predicted: Potato\_\_\_Early\_blight

Label: PotatoEarlyBlight4.JPG , Predicted: Potato\_\_\_Early\_blight

Label: PotatoEarlyBlight5.JPG , Predicted: Potato\_\_\_Early\_blight

Label: PotatoHealthy1.JPG , Predicted: Potato\_\_\_healthy

Label: PotatoHealthy2.JPG , Predicted: Potato\_\_\_healthy

Label: TomatoEarlyBlight1.JPG , Predicted: Tomato\_\_\_Early\_blight

Label: TomatoEarlyBlight2.JPG , Predicted: Tomato\_\_\_Early\_blight

Label: TomatoEarlyBlight3.JPG , Predicted: Tomato\_\_\_Early\_blight

Label: TomatoEarlyBlight4.JPG , Predicted: Tomato\_\_\_Early\_blight

Label: TomatoEarlyBlight5.JPG , Predicted: Tomato\_\_\_Early\_blight

Label: TomatoEarlyBlight6.JPG , Predicted: Tomato\_\_\_Early\_blight

Label: TomatoHealthy1.JPG , Predicted: Tomato\_\_\_healthy

Label: TomatoHealthy2.JPG , Predicted: Tomato\_\_\_healthy

Label: TomatoHealthy3.JPG , Predicted: Tomato\_\_\_healthy

Label: TomatoHealthy4.JPG , Predicted: Tomato\_\_\_healthy

Label: TomatoYellowCurlVirus1.JPG , Predicted: Tomato\_\_\_Tomato\_Yellow\_Leaf\_Curl\_Virus

Label: TomatoYellowCurlVirus2.JPG , Predicted: Tomato\_\_\_Tomato\_Yellow\_Leaf\_Curl\_Virus

Label: TomatoYellowCurlVirus3.JPG , Predicted: Tomato\_\_\_Tomato\_Yellow\_Leaf\_Curl\_Virus

Label: TomatoYellowCurlVirus4.JPG , Predicted: Tomato\_\_\_Tomato\_Yellow\_Leaf\_Curl\_Virus

Label: TomatoYellowCurlVirus5.JPG , Predicted: Tomato\_\_\_Tomato\_Yellow\_Leaf\_Curl\_Virus

Label: TomatoYellowCurlVirus6.JPG , Predicted: Tomato\_\_\_Tomato\_Yellow\_Leaf\_Curl\_Virus

**We can see that the model predicted all the test images perfectly!!!!**

**Saving the model**

**There are several ways to save the model in Pytorch, following are the two most common ways**

1. **Save/Load state\_dict (Recommended)**

When saving a model for inference, it is only necessary to save the trained model’s learned parameters. Saving the model’s state\_dict with the torch.save() function will give you the most flexibility for restoring the model later, which is why it is the recommended method for saving models.

A common PyTorch convention is to save models using either a .pt or .pth file extension.

Remember that you must call model.eval() to set dropout and batch normalization layers to evaluation mode before running inference. Failing to do this will yield inconsistent inference results.

In [46]:

*# saving to the kaggle working directory*

PATH **=** './plant-disease-model.pth'

torch**.**save(model**.**state\_dict(), PATH)

1. **Save/Load Entire Model**

This save/load process uses the most intuitive syntax and involves the least amount of code. Saving a model in this way will save the entire module using Python’s [pickle](https://docs.python.org/3/library/pickle.html) module. The disadvantage of this approach is that the serialized data is bound to the specific classes and the exact directory structure used when the model is saved. The reason for this is because pickle does not save the model class itself. Rather, it saves a path to the file containing the class, which is used during load time. Because of this, your code can break in various ways when used in other projects or after refactors.

In [47]:

*# saving the entire model to working directory*

PATH **=** './plant-disease-model-complete.pth'

torch**.**save(model, PATH)

**Conclusion**

ResNets perform significantly well for image classification when some of the parameters are tweaked and techniques like scheduling learning rate, gradient clipping and weight decay are applied. The model is able to predict every image in test set perfectly without any errors !!!!

**References**

* [CIFAR10 ResNet Implementation](https://jovian.ai/aakashns/05b-cifar10-resnet)
* [PyTorch docs](https://pytorch.org/)