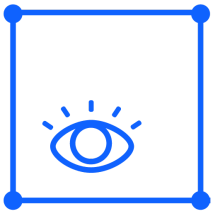


CV **Studio**

Transfer Learning with Convolutional Neural Networks For Classification with PyTorch and Computer Vision Learning Studio (CV Studio)

V 0.2

Project: Final_project_stop_signs

Training Run: Transfer learning for Stop Sign Classification

Estimated time needed: **40** minutes

In this lab, you will train a deep neural network for image classification using transfer learning, the image dataset will automatically be download from your CV Studio account. Experiment with different hyperparameters.

Objectives

In this lab you will train a state of the art image classifier using and CV Studio, CV Studio is a fast, easy and collaborative open source image annotation tool for teams and individuals. In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset in the lab, then use this Network to train your model. We will use the Convolutional Network as a feature generator, only training the output layer. In general, 100-200 images will give you a good starting point, and it only takes about half an hour. Usually, the more images you add, the better your results, but it takes longer and the rate of improvement will decrease.

- Import Libraries and Define Auxiliary Functions
- Create Dataset Object
- Load Model and Train

Import Libraries and Define Auxiliary Functions

In [2]:
```
#! conda install -c pytorch torchvision
#! pip install skillsnetwork tqdm
#!pip install  skillsnetwork
```

Libraries for OS and Cloud

In [3]:
```
import os
import uuid
import shutil
import json
from botocore.client import Config
import ibm_boto3
import copy
from datetime import datetime
from skillsnetwork import cvstudio
```

Libraries for Data Processing and Visualization

In [4]:
```
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import math
from matplotlib.pyplot import imshow
from tqdm import tqdm
from ipywidgets import IntProgress
import time
```

Deep Learning Libraries

In [5]:
```
import torch
import torchvision.models as models
from torch.utils.data import Dataset, DataLoader,random_split
from torch.optim import lr_scheduler
from torchvision import transforms
import torch.nn as nn
torch.manual_seed(0)
```

Out[5]:
```
<torch._C.Generator at 0x7fd5467e55d0>
```

Plot train cost and validation accuracy:

In [6]:
```
def plot_stuff(COST,ACC):
    fig, ax1 = plt.subplots()
    color = 'tab:red'
    ax1.plot(COST, color = color)
    ax1.set_xlabel('Iteration', color = color)
    ax1.set_ylabel('total loss', color = color)
    ax1.tick_params(axis = 'y', color = color)

    ax2 = ax1.twinx()
    color = 'tab:blue'
    ax2.set_ylabel('accuracy', color = color)  # we already handled the x-label
    ax2.plot(ACC, color = color)
    ax2.tick_params(axis = 'y', color = color)
    fig.tight_layout()  # otherwise the right y-label is slightly clipped
```

```
    plt.show()
```

Plot the transformed image:

In [7]:
```
def imshow_(inp, title=None):
    """Imshow for Tensor."""
    inp = inp .permute(1, 2, 0).numpy()
    print(inp.shape)
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)

    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)
    plt.show()
```

Compare the prediction and actual value:

In [8]:
```
def result(model,x,y):
    #x,y=sample
    z=model(x.unsqueeze_(0))
    _,yhat=torch.max(z.data, 1)

    if yhat.item()!=y:
        text="predicted: {} actual: {}".format(str(yhat.item()),y)
        print(text)
```

Define our device as the first visible cuda device if we have CUDA available:

In [9]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("the device type is", device)
```

```
the device type is cpu
```

Load Data

In this section we will preprocess our dataset by changing the shape of the image, converting to tensor and normalizing the image channels. These are the default preprocessing steps for image data. In addition, we will perform data augmentation on the training dataset. The preprocessing steps for the test dataset is the same, but W do not prform data augmentation on the test dataset.

```
mean = [0.485, 0.456, 0.406]
```

```
std = [0.229, 0.224, 0.225]
```

```
composed = transforms.Compose([transforms.Resize((224, 224)),
```

```
transforms.RandomHorizontalFlip(),transforms.RandomRotation(degrees=5)
                                  , transforms.ToTensor()
                                  , transforms.Normalize(mean, std)])
```

Download the data:

In [10]:
```
# Get the Dataset
# Initialize the CV Studio Client
cvstudioClient = cvstudio.CVStudio()
# # Download All Images
cvstudioClient.downloadAll()
```

```
100%|██████████| 197/197 [01:02<00:00,  3.16it/s]
```

We need to get our training and validation dataset. 90% of the data will be used for training.

In [11]:
```
percentage_train=0.9
train_set=cvstudioClient.getDataset(train_test='train',percentage_train=percenta
val_set=cvstudioClient.getDataset(train_test='test',percentage_train=percentage_
```

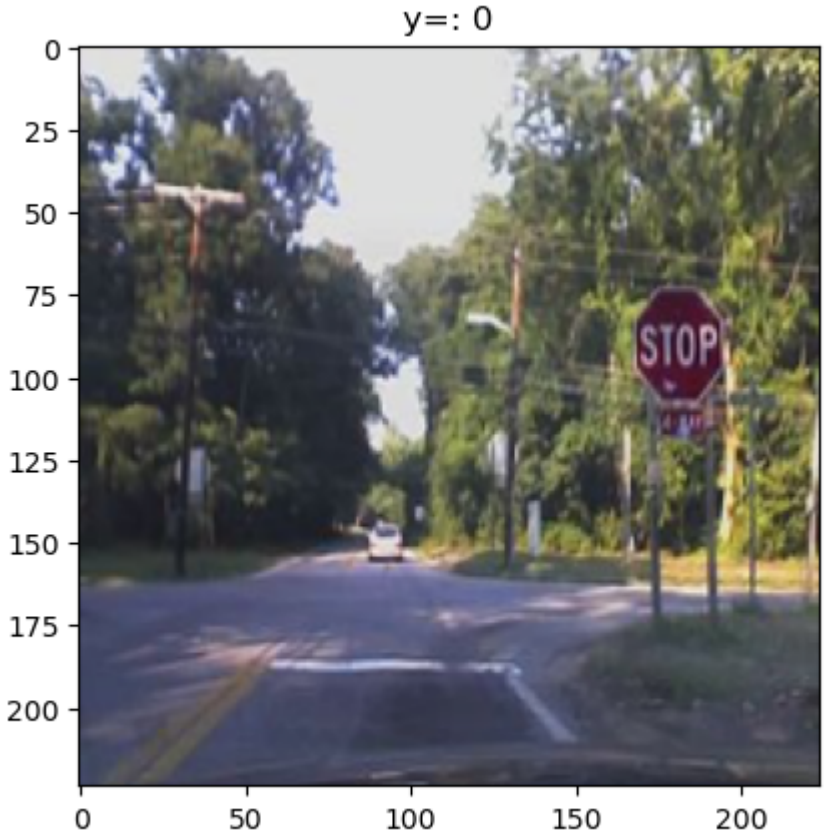
```
defult transform for pretrained model resnet18
this is the training set
defult transform for pretrained model resnet18
this is the test set
```

We can plot some of our dataset:

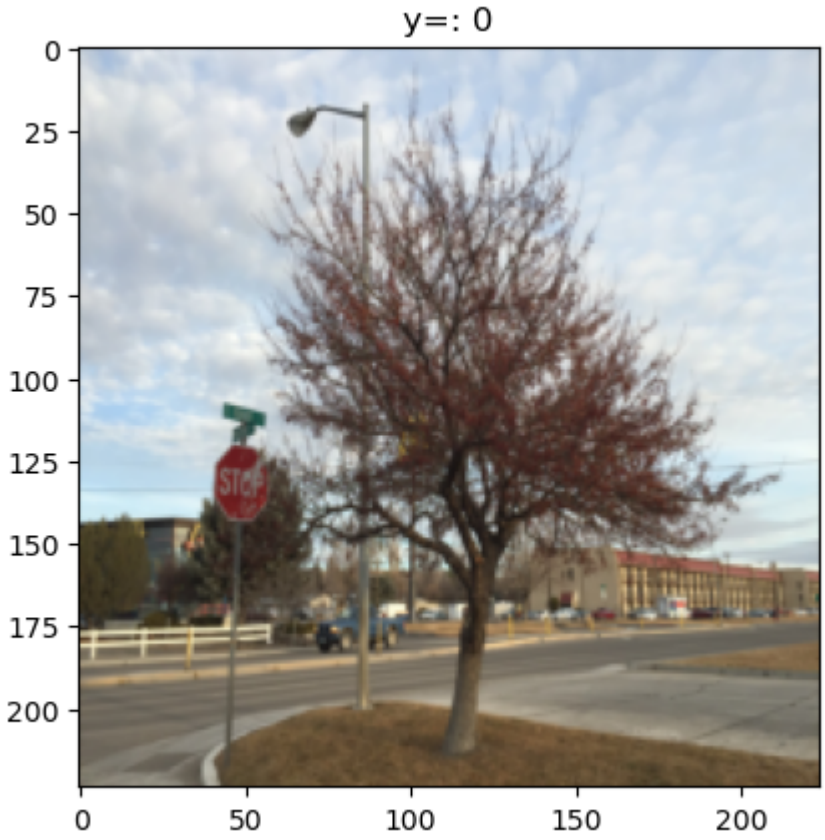
In [12]:
```
i=0

for x,y  in val_set:
    imshow_(x,"y=: {}".format(str(y.item())))
    i+=1
    if i==3:
        break
```

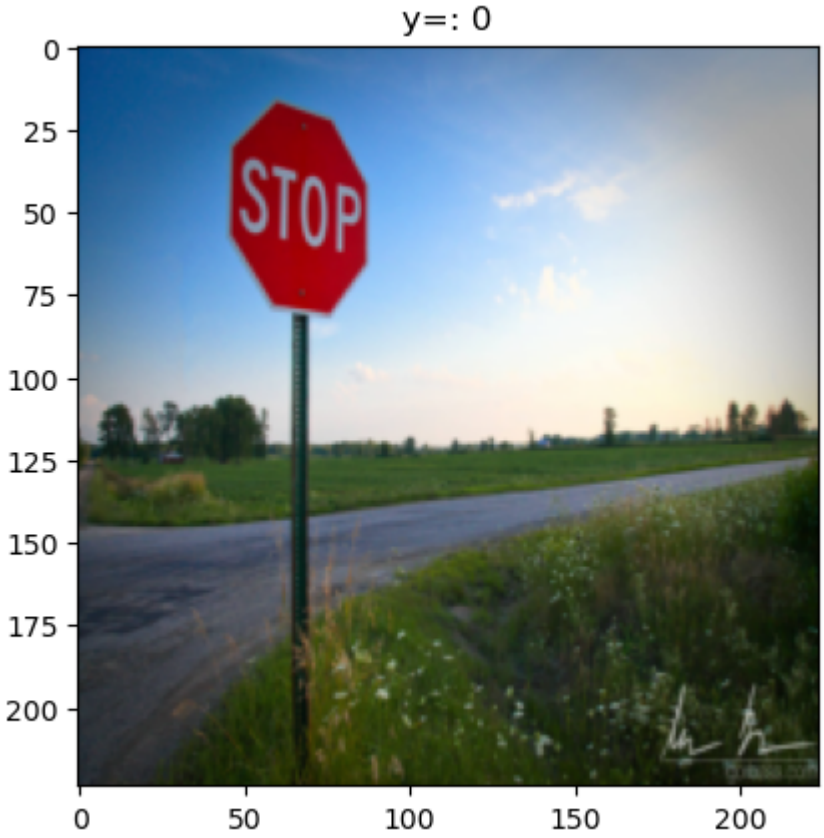
```
(224, 224, 3)
```




```
(224, 224, 3)
```




```
(224, 224, 3)
```



Hyperparameters

Experiment with different hyperparameters:

Epoch indicates the number of passes of the entire training dataset, here we will set the number of epochs to 10:

In [13]:
```
n_epochs=10
```

Batch size is the number of training samples utilized in one iteration. If the batch size is equal to the total number of samples in the training set, then every epoch has one iteration. In Stochastic Gradient Descent, the batch size is set to one. A batch size of 32--512 data points seems like a good value, for more information check out the following link.

In [14]:
```
batch_size=32
```

Learning rate is used in the training of neural networks. Learning rate is a hyperparameter with a small positive value, often in the range between 0.0 and 1.0.

In [15]:
```
lr=0.000001
```

Momentum is a term used in the gradient descent algorithm to improve training results:

In [16]:
```
momentum=0.9
```

If you set to `lr_scheduler=True` for every epoch use a learning rate scheduler changes the range of the learning rate from a maximum or minimum value. The learning rate usually decays over time.

In [17]:
```
lr_scheduler=True
base_lr=0.001
max_lr=0.01
```

Load Model and Train

This function will train the model

In [18]:
```
def train_model(model, train_loader,validation_loader, criterion, optimizer, n_e
    loss_list = []
    accuracy_list = []
    correct = 0
    #global:val_set
    n_test = len(val_set)
    accuracy_best=0
    best_model_wts = copy.deepcopy(model.state_dict())

    # Loop through epochs
        # Loop through the data in loader
    print("The first epoch should take several minutes")
    for epoch in tqdm(range(n_epochs)):

        loss_sublist = []
        # Loop through the data in loader

        for x, y in train_loader:
            x, y=x.to(device), y.to(device)
            model.train()

            z = model(x)
            loss = criterion(z, y)
            loss_sublist.append(loss.data.item())
            loss.backward()
            optimizer.step()

            optimizer.zero_grad()
        print("epoch {} done".format(epoch) )

        scheduler.step()
        loss_list.append(np.mean(loss_sublist))
        correct = 0


        for x_test, y_test in validation_loader:
            x_test, y_test=x_test.to(device), y_test.to(device)
            model.eval()
            z = model(x_test)
            _, yhat = torch.max(z.data, 1)
            correct += (yhat == y_test).sum().item()
        accuracy = correct / n_test
        accuracy_list.append(accuracy)
        if accuracy>accuracy_best:
```

```
            accuracy_best=accuracy
            best_model_wts = copy.deepcopy(model.state_dict())


        if print_:
            print('learning rate',optimizer.param_groups[0]['lr'])
            print("The validaion  Cost for each epoch " + str(epoch + 1) + ": "
            print("The validation accuracy for epoch " + str(epoch + 1) + ": " +
    model.load_state_dict(best_model_wts)
    return accuracy_list,loss_list, model
```

Load the pre-trained model resnet18. Set the parameter pretrained to true.

In [19]:
```
model = models.resnet18(pretrained=True)
```

```
0%|          | 0.00/44.7M [00:00<?, ?B/s]
```

We will only train the last layer of the network set the parameter `requires_grad` to `False`, the network is a fixed feature extractor.

In [20]:
```
for param in model.parameters():
        param.requires_grad = False
```

Number of classes

In [21]:
```
n_classes=train_set.n_classes
n_classes
```

Out[21]:
```
2
```

Replace the output layer model.fc of the neural network with a nn.Linear object, to classify `n_classes` different classes. For the parameters in_features remember the last hidden layer has 512 neurons.

In [22]:
```
# Type your code here
model.fc = nn.Linear(512, n_classes)
```

Set device type

In [23]:
```
model.to(device)
```

Out[23]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bia
s=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mod
e=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    )
```

```
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
```

```
  (fc): Linear(in_features=512, out_features=2, bias=True)
)
```

Cross-entropy loss, or log loss, measures the performance of a classification model combines LogSoftmax in one object class. It is useful when training a classification problem with C classes.

In [24]:
```
criterion = nn.CrossEntropyLoss()
```

Create a training loader and validation loader object.

In [25]:
```
train_loader = torch.utils.data.DataLoader(dataset=train_set , batch_size=batch_
validation_loader= torch.utils.data.DataLoader(dataset=val_set , batch_size=1)
```

Use the optim package to define an Optimizer that will update the weights of the model for us.

In [26]:
```
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum)
```

We use Cyclical Learning Rates

In [27]:
```
if lr_scheduler:
    scheduler = torch.optim.lr_scheduler.CyclicLR(optimizer, base_lr=0.001, max_
```

Now we are going to train model,for 500 images this take 25 minutes, depending on your dataset

In [28]:
```
start_datetime = datetime.now()
start_time=time.time()

accuracy_list,loss_list, model=train_model(model,train_loader , validation_loade

end_datetime = datetime.now()
current_time = time.time()
elapsed_time = current_time - start_time
print("elapsed time", elapsed_time )
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
The first epoch should take several minutes
epoch 0 done
 10%|▉         | 1/10 [00:23<03:29, 23.32s/it]
learning rate 0.002800000000000002
The validaion  Cost for each epoch 1: 0.7468355695406595
The validation accuracy for epoch 1: 0.55
epoch 1 done
 20%|██        | 2/10 [00:44<02:57, 22.25s/it]
learning rate 0.0046
The validaion  Cost for each epoch 2: 0.5763372083504995
The validation accuracy for epoch 2: 0.7
epoch 2 done
 30%|███       | 3/10 [01:05<02:31, 21.60s/it]
learning rate 0.006400000000000001
The validaion  Cost for each epoch 3: 0.39397523800532025
The validation accuracy for epoch 3: 0.65
epoch 3 done
 40%|████      | 4/10 [01:27<02:10, 21.79s/it]
learning rate 0.008199999999999999
The validaion  Cost for each epoch 4: 0.39018165071805316
The validation accuracy for epoch 4: 0.75
epoch 4 done
 50%|█████     | 5/10 [01:49<01:48, 21.72s/it]
learning rate 0.010000000000000002
The validaion  Cost for each epoch 5: 0.2566098670164744
The validation accuracy for epoch 5: 0.7
epoch 5 done
 60%|██████    | 6/10 [02:10<01:26, 21.64s/it]
learning rate 0.008199999999999999
The validaion  Cost for each epoch 6: 0.28704695651928586
The validation accuracy for epoch 6: 0.75
epoch 6 done
 70%|███████   | 7/10 [02:32<01:05, 21.75s/it]
learning rate 0.006400000000000001
The validaion  Cost for each epoch 7: 0.20405644426743189
The validation accuracy for epoch 7: 0.7
epoch 7 done
 80%|████████  | 8/10 [02:54<00:43, 21.85s/it]
learning rate 0.0046
The validaion  Cost for each epoch 8: 0.21286370481053987
The validation accuracy for epoch 8: 0.75
epoch 8 done
 90%|█████████ | 9/10 [03:16<00:21, 21.71s/it]
learning rate 0.002800000000000002
The validaion  Cost for each epoch 9: 0.17385217174887657
The validation accuracy for epoch 9: 0.8
epoch 9 done
100%|██████████| 10/10 [03:38<00:00, 21.87s/it]
learning rate 0.001
The validaion  Cost for each epoch 10: 0.14955409988760948
The validation accuracy for epoch 10: 0.8
elapsed time 218.87241005897522
```

Now run the following to report back the results of the training run to CV Studio

In [29]:
```
parameters = {
    'epochs': n_epochs,
    'learningRate': lr,
    'momentum':momentum,
    'percentage used training':percentage_train,
    "learningRatescheduler": {"lr_scheduler":lr_scheduler,"base_lr":base_lr, "ma


}
result = cvstudioClient.report(started=start_datetime, completed=end_datetime, p

if result.ok:
    print('Congratulations your results have been reported back to CV Studio!')
```

```
Congratulations your results have been reported back to CV Studio!
```

Save the model to model.pt

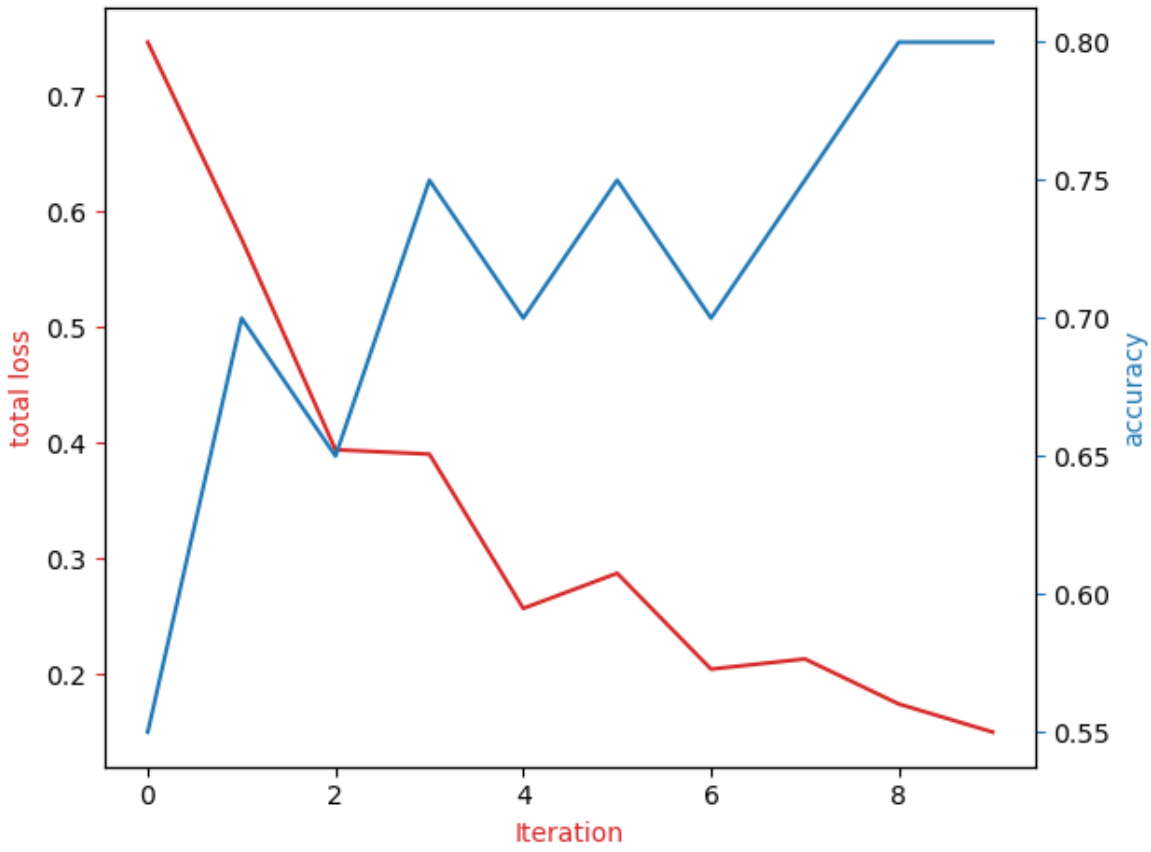
In [30]:
```
# Save the model to model.pt
torch.save(model.state_dict(), 'model.pt')

# Save the model and report back to CV Studio
result = cvstudioClient.uploadModel('model.pt', {'numClasses': n_classes})
```

```
File Uploaded
```

Plot train cost and validation accuracy, you can improve results by getting more data.

In [31]:
```
plot_stuff(loss_list,accuracy_list)
```



Load the model that performs best:

In [32]:
```
model = models.resnet18(pretrained=True)
model.fc = nn.Linear(512, n_classes)
model.load_state_dict(torch.load( "model.pt"))
model.eval()
```

Out[32]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bia
s=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mod
e=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    )
```

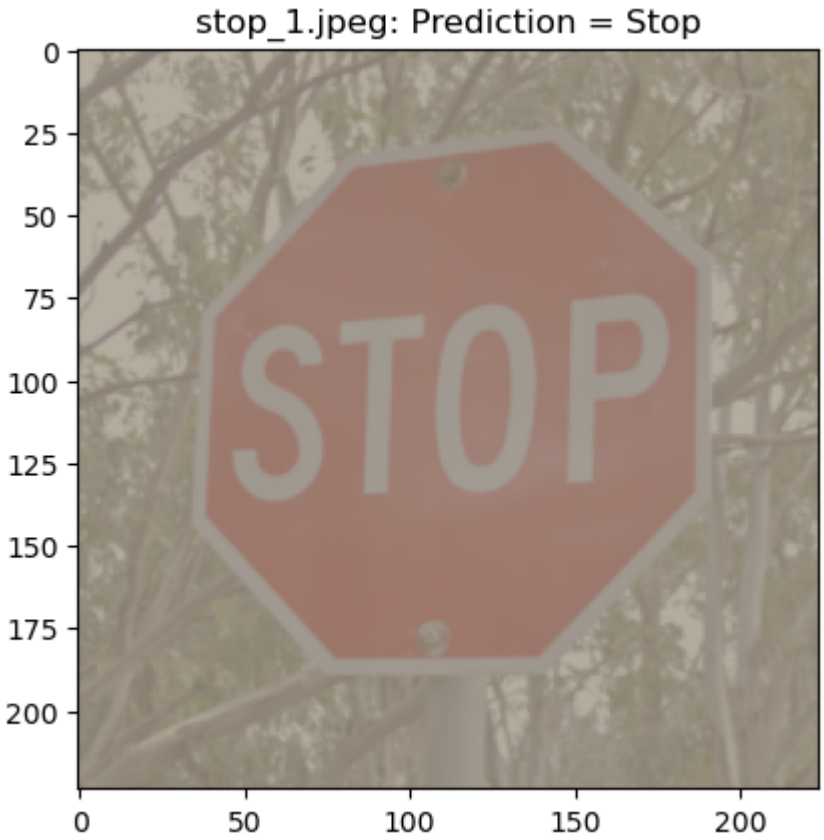
```
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
```

```
  (fc): Linear(in_features=512, out_features=2, bias=True)
)
```

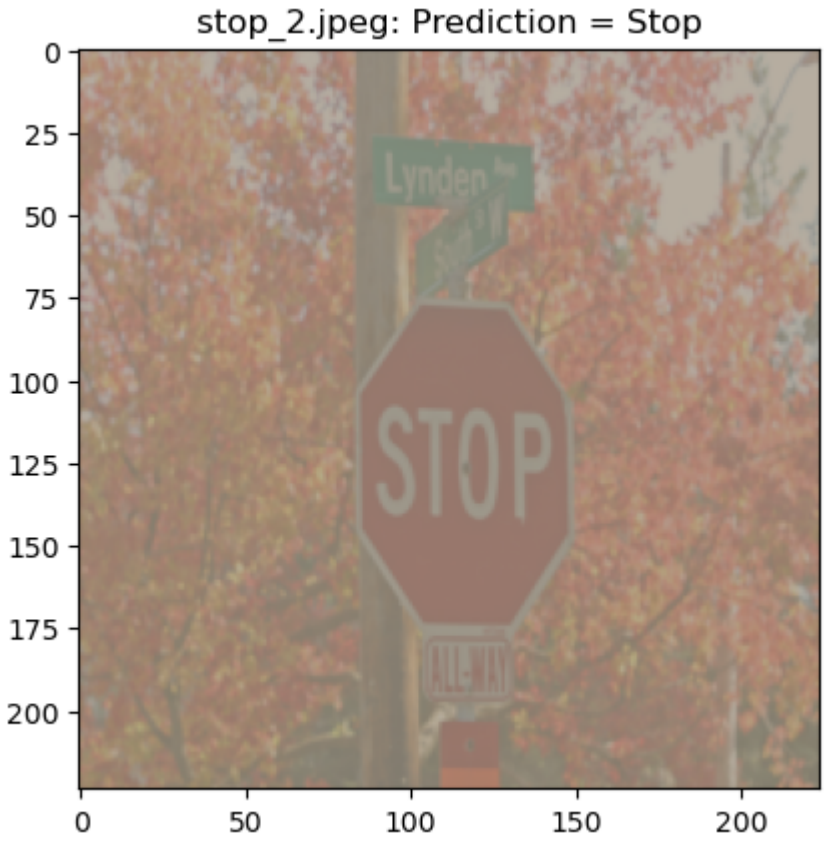
Testing

In [33]:
```
imageNames = ['stop_1.jpeg','stop_2.jpeg','stop_3.jpeg', 'stop_4.jpeg']
for imageName in imageNames:
    image = Image.open(imageName)
    transform = composed = transforms.Compose([transforms.Resize((224, 224)), tr
    x = transform(image)
    z=model(x.unsqueeze_(0))
    _,yhat=torch.max(z.data, 1)
    # print(yhat)
    prediction = "Stop"
    if yhat == 1:
        prediction ="Not Stop"
    imshow_(transform(image),imageName+": Prediction = "+prediction)
```

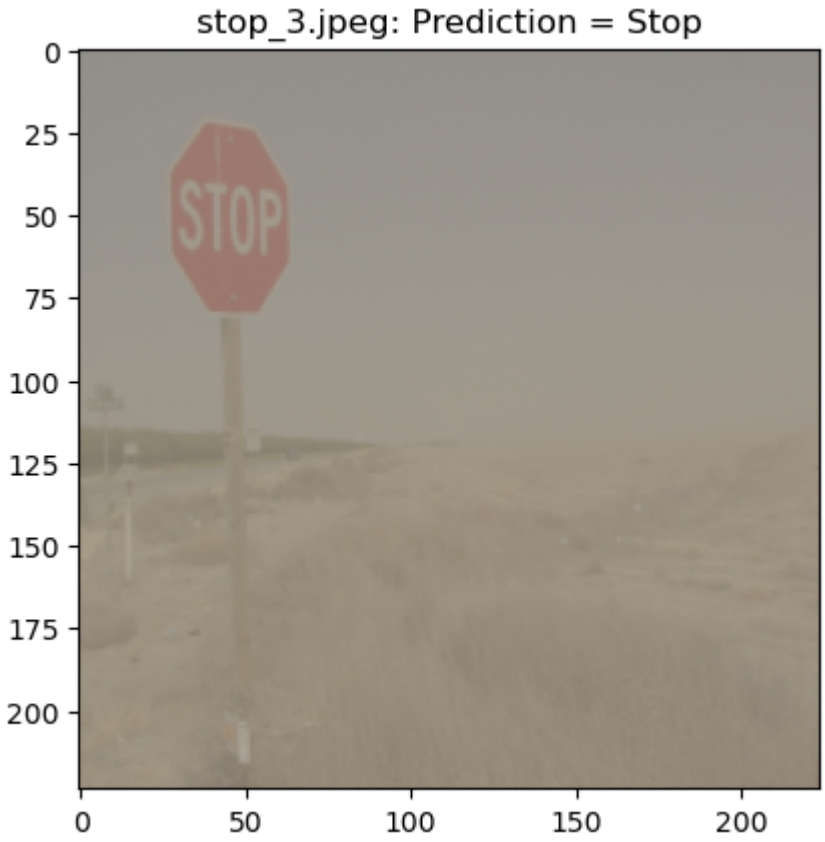
```
(224, 224, 3)
```



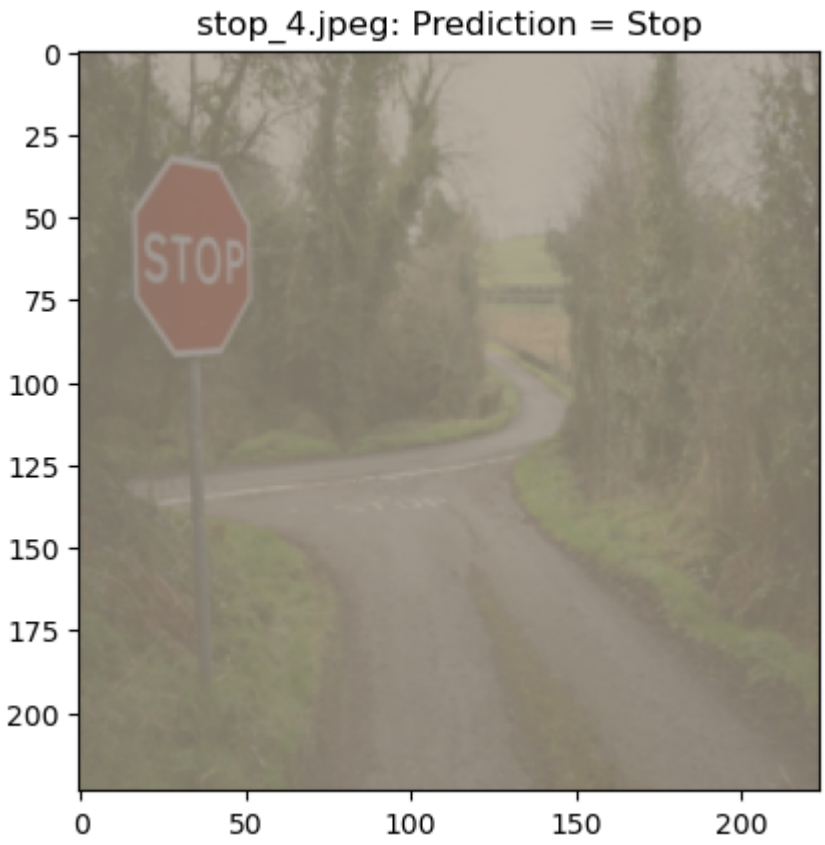

```
(224, 224, 3)
```




```
(224, 224, 3)
```

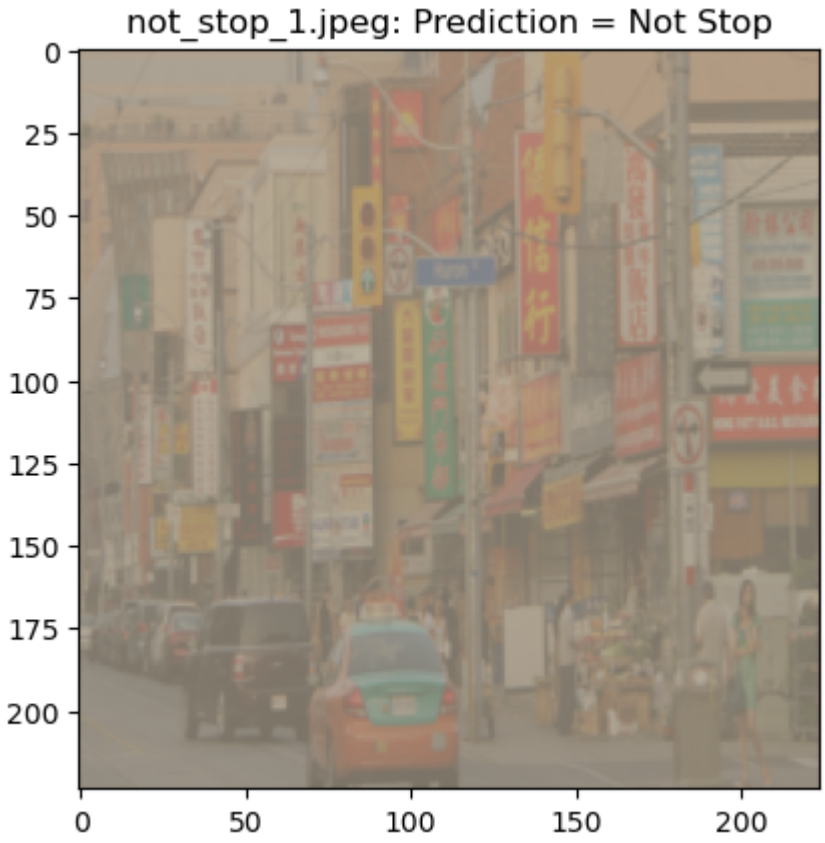



```
(224, 224, 3)
```

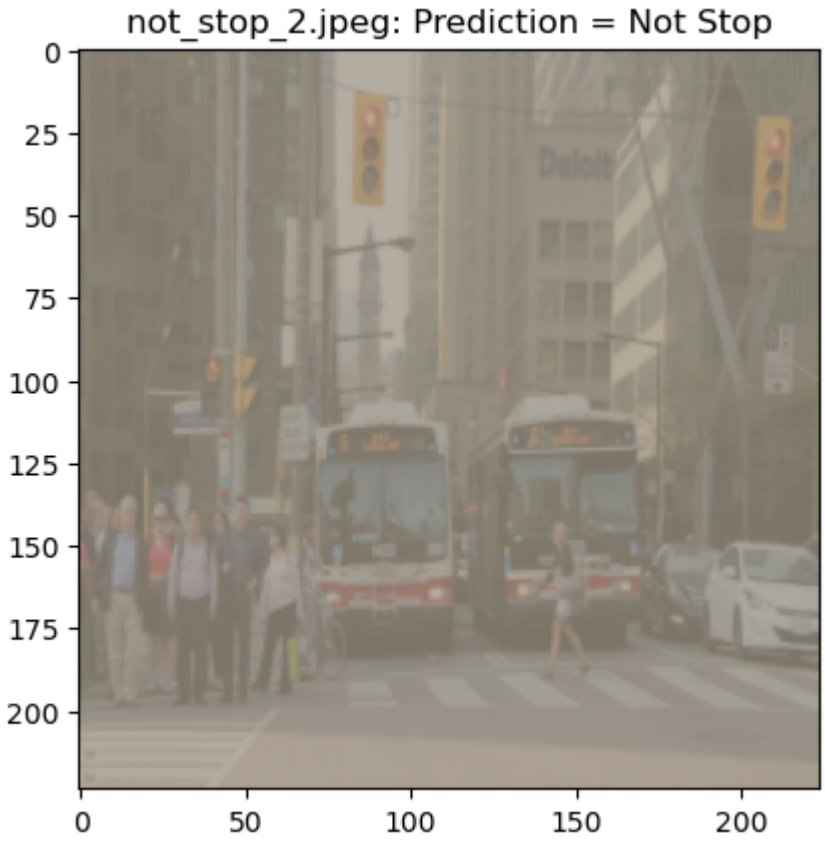


In [34]:
```
imageNames = ['not_stop_1.jpeg','not_stop_2.jpeg','not_stop_3.jpeg','not_stop_4.
for imageName in imageNames:
    image = Image.open(imageName)
    transform = composed = transforms.Compose([transforms.Resize((224, 224)), tr
    x = transform(image)
    z=model(x.unsqueeze_(0))
    _,yhat=torch.max(z.data, 1)
    # print(yhat)
    prediction = "Stop"
    if yhat == 1:
        prediction ="Not Stop"
    imshow_(transform(image),imageName+": Prediction = "+prediction)
```

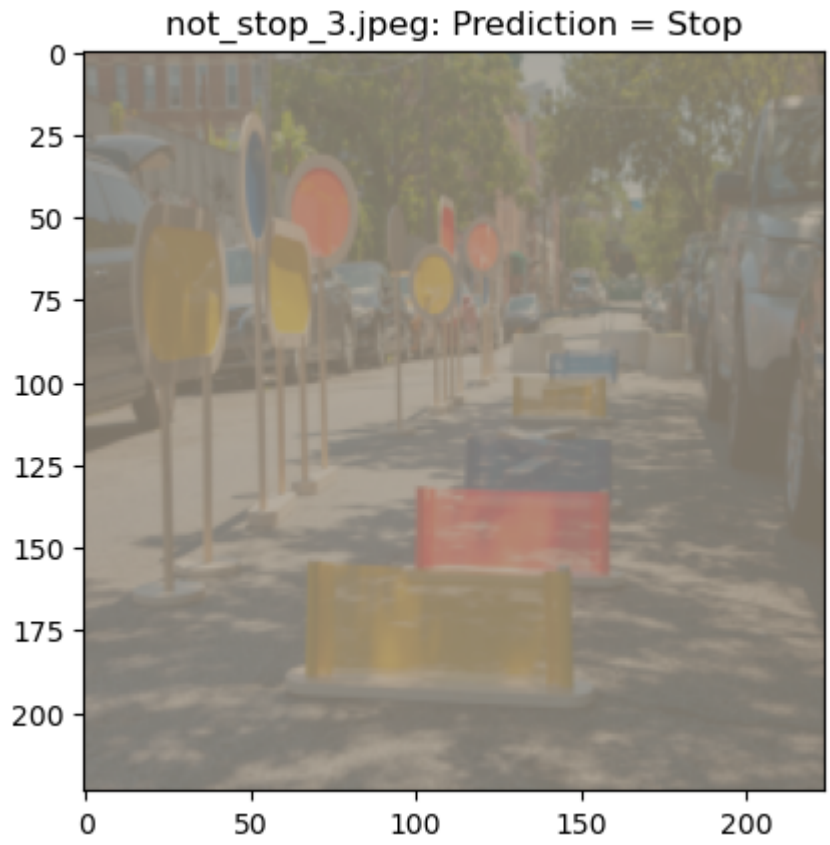
```
(224, 224, 3)
```



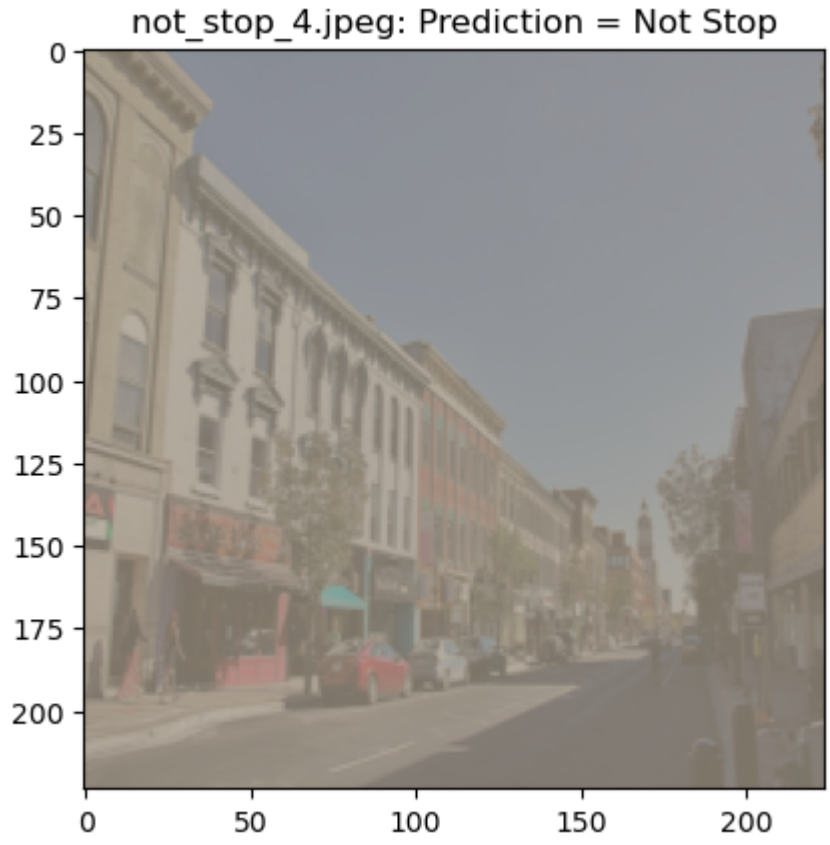

```
(224, 224, 3)
```




```
(224, 224, 3)
```




```
(224, 224, 3)
```



Authors

Joseph Santarcangelo,has a PhD in Electrical Engineering, his research focused on using machine learning, signal processing, and computer vision to determine how videos

impact human cognition. Joseph has been working for IBM since he completed his PhD.

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2021-05-25	0.3	Yasmine	Modifies Multiple Areas
2021-05-25	0.3	Kathy	Modified Multple Areas.
2021-03-08	0.2	Joseph	Modified Multiple Areas
2021-02-01	0.1	Joseph	Modified Multiple Areas