Deep Learning for Computer Vision - Final Project

Mapping Deforestation in the Amazon with Satellite Imagery

Robert Pendergast - rlp2153

For this project I am going to be identifying areas of deforestation in the Amazon Rainforest by training/fine-tuning several different models on a custom dataset that I assemble myself. It is split up into 4 parts:

1. Data Aquisition
2. Model Training
3. Model Comparison
4. Model Application

I hope you enjoy!

Part 1: Data Acquisition

In this step I acquire the data, annotate it, and curate it into training, validation, and test sets myself.

The data is from the ESA's Sentinel-2 Satellite. The ESA made all satellite images from Sentinel-2 publically available for any use, which is a great resource for this project. For this part, I have downloaded 4 scenes from the Sentinel-2 Data, displayed below:

In [ ]:
```
from google.colab import files
files.upload()
```

...

The origins of the data that I will use to train, test, and validate my model have been uploaded. Next I display them:

In [ ]:
```
#First install some necessary packages
import numpy as np
import matplotlib.pyplot as plt
import cv2
from google.colab.patches import cv2_imshow
import os
```

In [ ]:
```
df1 = 'Deforestation1.jpg'
df2 = 'Deforestation2.jpg'
f1 = 'Forest1.jpg'
f2 = 'Forest2.jpg'

sample_list = [df1,df2,f1,f2]
imgs = []

for image in sample_list:
  img = cv2.imread(image)
  print(image)
  cv2_imshow(img)
  imgs.append(img)
```

```
Deforestation1.jpg
```

From these images are then 'chopped-up' into smaller images, which I use as the data for this project. A visualization of this is given below:

In [ ]:
```
ex = imgs[0]
height, width, channels = ex.shape
print(height,width)
```

```
875 1215
```

Original Image dimensions are 875 x 1215, which is not very nice. To make the images nicer to work with, I resize them all:

In [ ]:
```
i = 0
for img in imgs:
  img = cv2.resize(img,(1800,1200))
  imgs[i] = img
  i+=1
```

In [ ]:
```
ex = imgs[0]
height, width, channels = ex.shape
print(height,width)
```

```
1200 1800
```

With the images resized, I can subdivide each image into an array of 50x50 pixel images! That way, each scene provides over 800 images for the dataset. This 'grid' is visualized below:

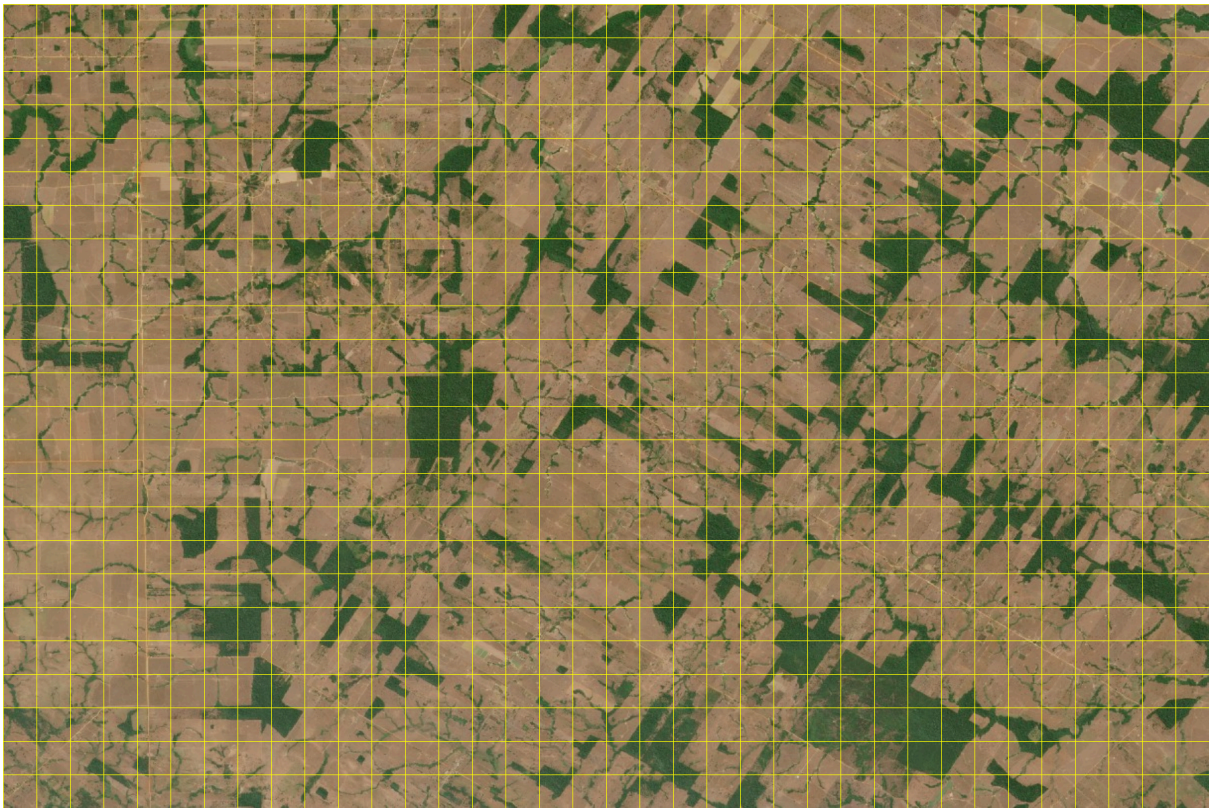
In [ ]:
```
ex_copy = ex.copy()

x = y = 0

while x < width:
    cv2.line(ex_copy,(x,0),(x,height),(0,255,255))
    x += 50

while y < height:
    cv2.line(ex_copy,(0,y),(width,y),(0,255,255))
    y += 50

cv2_imshow(ex_copy)
```



A brief visual inspection of this grid layout reveals that every cell has at least a little bit of deforestation visible. There are definetely some cells that do not include a lot, but such is the nature of creating your own dataset!

The next step is to take each image and crop them into smaller images in this grid pattern. However, before I do this, I should set up a directory for my data, that way everything is properly organized!

In [ ]:
```
os.mkdir('data')
os.chdir('data')
#Inside the data folder we want two more folders: deforestation and fore
os.mkdir('deforestation')
os.mkdir('forest')
```

First I cut all of the deforested images:

In [ ]:
```
os.chdir('./deforestation')

#Cut from Deforestation1.jpg
i = 0
x = y = 0

img = imgs[0]

#image cropping coordinates are flipped for some odd reason
while x <= 1150:
  while y <= 1750:
    cropped = img[x:x+49,y:y+49]
    cv2.imwrite(f"deforestation{i}.jpg",cropped)
    i+=1
    y +=50

  x+=50
  y = 0

#Cut from Deforestation2.jpg
x = y = 0

img = imgs[1]

while x <= 1150:
  while y <= 1750:
    cropped = img[x:x+49,y:y+49]
    cv2.imwrite(f"deforestation{i}.jpg",cropped)
    i+=1
    y +=50

  x+=50
  y = 0
```

Reset my directories...

In [ ]:
```
os.chdir('..')
os.chdir('forest')
```

Then I cut all of the forest images:

In [ ]:
```
#Cut from Forest1.jpg
i = 0
x = y = 0

img = imgs[2]

while x <= 1150:
  while y <= 1750:
    cropped = img[x:x+49,y:y+49]
    cv2.imwrite(f"forest{i}.jpg",cropped)
    i+=1
    y +=50

  x+=50
  y = 0

#Cut from Forest2.jpg
x = y = 0

img = imgs[3]

while x <= 1150:
  while y <= 1750:
    cropped = img[x:x+49,y:y+49]
    cv2.imwrite(f"forest{i}.jpg",cropped)
    i+=1
    y +=50

  x+=50
  y = 0
```

Great! Now I have exactly 1728 images for each class! This is more than enough data to train a model on!

Part 2: Model Training and Comparison

In this section, I train different classifying models on my data and compare how well each one does. Since transfer learning has proven to be very reliable, I will be training the following pre-trained models:

1. resnet18 - the same from homework5
2. resnet50 - a deeper resnet to see how network depth affects the outcome
3. Vision Transformer - checking out how transformers work!

After training each model, I will analyze how well each model performs on a test set. I will account for factors including training time, loss, and accuracy. These metrics will inform the model selection for part 3 of this project!

In [ ]:
```
#Importing pytorch modules
import torch
import torchvision
from torchvision import datasets
from torchvision import transforms
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader
from torch import nn
from torchvision.datasets import ImageFolder


os.chdir('../..')
```

The next step is to create the dataloaders that I will use to train each model. To do this, I first need to define a transform:

In [ ]:
```
transform = transforms.Compose([transforms.Resize((224,224)),
                                transforms.ToTensor()])
```

Then I create the dataset using the ImageFolder package:

In [ ]:
```
data_dir = './data'
dataset = ImageFolder(data_dir, transform)
```

To make sure that the images look good, I print out one from the forest class and the deforestation class:

In [ ]:
```
img, label = dataset[0]
print(label)
plt.imshow(img.T)
plt.axis("OFF")
plt.show()

img, label = dataset[3000]
print(label)
plt.imshow(img.T)
plt.axis("OFF")
```

```
0

<ipython-input-15-ebadd1699b32>:3: UserWarning: The use of `x.T` on ten
sors of dimension other than 2 to reverse their shape is deprecated and
it will throw an error in a future release. Consider `x.mT` to transpos
e batches of matrices or `x.permute(*torch.arange(x.ndim - 1, -1, -1))`
to reverse the dimensions of a tensor. (Triggered internally at ../ate
n/src/ATen/native/TensorShape.cpp:3683.)
  plt.imshow(img.T)
```

```
1
```

Out[15]: (-0.5, 223.5, 223.5, -0.5)

The dataset is looking good! Now I just have to split it up into training, validation, and test sets!

In [ ]:
```
from torch.utils.data import random_split

training, testing, validation = random_split(dataset,[2766,345,345]) #ad
```

Finally, I make my dataloaders:

In [ ]:
```
train_dataloader = DataLoader(training, batch_size = 256, shuffle = True
test_dataloader = DataLoader(testing, batch_size = 1, shuffle = True)
validation_dataloader = DataLoader(validation, batch_size = 1, shuffle =
```

With the dataloaders complete, I can go on to training my models!

Model 1 - Resnet18

Below I implement a resnet18 model fine-tuned on my data. The first step is to upload the models:

In [ ]:
```
from torchvision.models import resnet18, ResNet18_Weights

ft_resnet18 = resnet18(weights = ResNet18_Weights.DEFAULT)

#Print out model to understand its architecture
ft_resnet18
```

```
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pt
h" to /root/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth
100%|██████████| 44.7M/44.7M [00:00<00:00, 93.3MB/s]
```

Out[18]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), paddi
ng=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=Fa
lse)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padd
```

```
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padd
ing=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=F
alse)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padd
ing=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=F
alse)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
```

```
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

Since I am working with a binary classifier, the output layer should have one output logit followed by a sigmoid activation function:

In [ ]:
```
ft_resnet18.fc = nn.Sequential(
    nn.Linear(in_features = 512, out_features = 1, bias = True),
    nn.Sigmoid()
)
ft_resnet18
```

Out[19]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), paddi
ng=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=Fa
lse)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padd
```

```
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padd
ing=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=F
alse)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padd
ing=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=F
alse)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
```

```
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Sequential(
    (0): Linear(in_features=512, out_features=1, bias=True)
    (1): Sigmoid()
  )
)
```

Great! Now that the model surgery has been completed, all I have to do is freeze the weights before I go on to training the model:

In [ ]:
```
for param in ft_resnet18.parameters():
  param.requires_grad = False

for param in ft_resnet18.fc.parameters():
  param.requires_grad = True
```

Defining Training Function:

In [ ]:
```
def train_model(dataloader, model, loss_function, optimizer):
  model.train()

  for batch, (X,y) in enumerate(dataloader):
    X = X.to(device = 'cuda')
    y = y.to(device = 'cuda')

    X = X.float()
    y = y.float()

    prediction = model(X)
    y =y.unsqueeze(-1)
    loss = loss_function(prediction,y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

And the Testing Function:

In [ ]:
```
def test_model(dataloader, model, loss_function):
  accuracy = 0
  loss = 0

  model.eval()

  for X,y in dataloader:
    X = X.to(device = 'cuda')
    y = y.to(device = 'cuda')

    X = X.float()
    y = y.float()

    prediction = model(X)
    y =y.unsqueeze(-1)
    loss += loss_function(prediction,y)

    p = 0
    if prediction.item() > 0.5:
      p = 1


    if p == y:
      accuracy += 1

  accuracy /= len(dataloader)
  loss /= len(dataloader)
  return accuracy, loss
```

Finally, train the model!

In [ ]:
```
loss_fn = nn.BCELoss()
optimizer = torch.optim.SGD(ft_resnet18.parameters(), lr = 0.01)
epochs = 10

ft_resnet18 = ft_resnet18.to(device = 'cuda')

resnet18_achain = []
resnet18_lchain = []

for i in range(1,epochs+1):
  train_model(train_dataloader, ft_resnet18, loss_fn, optimizer)
  print(f"-----Epoch {i} Complete-----")
  accuracy, loss = test_model(validation_dataloader, ft_resnet18, loss_f
  resnet18_achain.append(accuracy)
  resnet18_lchain.append(loss.item())
  print(f"Accuracy: {accuracy}")
  print(f"Loss: {loss}\n")

print("-----Training Complete-----")
```

```
-----Epoch 1 Complete-----
Accuracy: 0.6579710144927536
Loss: 0.6544880270957947

-----Epoch 2 Complete-----
Accuracy: 0.8231884057971014
Loss: 0.5245287418365479

-----Epoch 3 Complete-----
Accuracy: 0.8840579710144928
Loss: 0.39600226283073425

-----Epoch 4 Complete-----
Accuracy: 0.9652173913043478
Loss: 0.27504822611808777

-----Epoch 5 Complete-----
Accuracy: 1.0
Loss: 0.1270439177751541

-----Epoch 6 Complete-----
Accuracy: 1.0
Loss: 0.0719383955001831

-----Epoch 7 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.05575154349207878

-----Epoch 8 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.04889107495546341

-----Epoch 9 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.0444011352956295

-----Epoch 10 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.04093603417277336

-----Training Complete-----
```

The model is training with near 100% accuracy... let's see how well it performs on the test set:

In [ ]:
```
accuracy, loss = test_model(test_dataloader, ft_resnet18, loss_fn)
print(f"Accuracy: {accuracy}")
print(f"Loss: {loss}\n")
```

```
Accuracy: 1.0
Loss: 0.03617455065250397
```

The test set trains with close to 100% accuracy! Let's print out a few examples to see how they are classified:

In [ ]:
```
i = 0
class_dict = {0:'Deforestation',1:'Forest'}

for X, y in test_dataloader:
  img = X
  X = X.to(device = 'cuda')

  X = X.float()

  prediction = ft_resnet18(X)

  p = 0
  if prediction.item() > 0.5:
    p = 1

  print(class_dict[p])
  plt.axis("OFF")
  plt.imshow(img.squeeze(0).T)
  plt.show()

  if (i>4):
    break
  i += 1
```

```
Deforestation
```

Looks like everything is working! This is very good news for the classifier!

Model 2: Resnet50

Even though the Resnet18 model performed extraordinarily well, I still want to experiment with a deeper network - Resnet50. This process follows the same steps as before:

In [ ]:
```
from torchvision.models import resnet50, ResNet50_Weights

ft_resnet50 = resnet50(weights = ResNet50_Weights.DEFAULT)

#Print out model to understand its architecture
ft_resnet50
```

```
Downloading: "https://download.pytorch.org/models/resnet50-11ad3fa6.pt
h" to /root/.cache/torch/hub/checkpoints/resnet50-11ad3fa6.pth
100%|██████████| 97.8M/97.8M [00:00<00:00, 157MB/s]
```

Out[26]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
```

I need to change the final fully connected layer:

In [ ]:
```
ft_resnet50.fc = nn.Sequential(
    nn.Linear(in_features = 2048, out_features = 1, bias = True),
    nn.Sigmoid()
)
```

Then freeze the parameters:

In [ ]:
```
for param in ft_resnet50.parameters():
  param.requires_grad = False

for param in ft_resnet50.fc.parameters():
  param.requires_grad = True
```

There's no need to redefine my training and testing functions, so I will go right into training the new model!

In [ ]:
```
loss_fn = nn.BCELoss()
optimizer = torch.optim.SGD(ft_resnet50.parameters(), lr = 0.01)
epochs = 10

ft_resnet50 = ft_resnet50.to(device = 'cuda')


resnet50_accuracy = []
resnet50_loss = []

for i in range(1,epochs+1):
  train_model(train_dataloader, ft_resnet50, loss_fn, optimizer)
  print(f"-----Epoch {i} Complete-----")
  accuracy, loss = test_model(validation_dataloader, ft_resnet50, loss_f
  resnet50_accuracy.append(accuracy)
  resnet50_loss.append(loss.item())
  print(f"Accuracy: {accuracy}")
  print(f"Loss: {loss}\n")

print("-----Training Complete-----")
```

```
-----Epoch 1 Complete-----
Accuracy: 0.7971014492753623
Loss: 0.6733976006507874

-----Epoch 2 Complete-----
Accuracy: 0.5739130434782609
Loss: 0.6552950143814087

-----Epoch 3 Complete-----
Accuracy: 0.5159420289855072
Loss: 0.6350894570350647

-----Epoch 4 Complete-----
Accuracy: 0.5333333333333333
Loss: 0.6355243921279907

-----Epoch 5 Complete-----
Accuracy: 0.6492753623188405
Loss: 0.5841630101203918

-----Epoch 6 Complete-----
Accuracy: 0.8985507246376812
Loss: 0.4806925058364868

-----Epoch 7 Complete-----
Accuracy: 1.0
Loss: 0.37778517603874207

-----Epoch 8 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.28121957182884216

-----Epoch 9 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.23448649048805237

-----Epoch 10 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.2088906466960907

-----Training Complete-----
```

This model trains incredibly well too, although it does have a worse loss than the 18 layer resnet. Another interesting feature to point out is that in Epoch 5 there was a mis-classified datapoint. Still, the training performed well. I now test the model on the test set:

In [ ]:
```
accuracy, loss = test_model(test_dataloader, ft_resnet50, loss_fn)
print(f"Accuracy: {accuracy}")
print(f"Loss: {loss}\n")
```

```
Accuracy: 1.0
Loss: 0.20090894401073456
```

The deeper model performs similarly to the fine-tuned resnet18! Depending on the run, it may misclassify images or it may not. The following block will display any misclassified images

In [ ]:
```
class_dict = {0:'Deforestation',1:'Forest'}

for X, y in test_dataloader:
  img = X
  X = X.to(device = 'cuda')
  y = y.to(device = 'cuda')

  X = X.float()
  y = y.float()

  prediction = ft_resnet50(X)

  p = 0
  if prediction.item() > 0.5:
    p = 1

  if not p == y:
    print(class_dict[p])
    plt.axis("OFF")
    plt.imshow(img.squeeze(0).T)
    plt.show()
```

Model 3: Vision Transformer

Finally, I want to experiment with the vision transformer to see how accurate that is in comparison to the convolution-based models. The bar is pretty high though, seeing as the fine-tuned resnet18 performed at 100% accuracy!

In [ ]:
```
from torchvision.models import vit_b_16, ViT_B_16_Weights

ft_vit = vit_b_16(weights = ViT_B_16_Weights.DEFAULT)

ft_vit
```

```
Downloading: "https://download.pytorch.org/models/vit_b_16-c867db91.pt
h" to /root/.cache/torch/hub/checkpoints/vit_b_16-c867db91.pth
100%|██████████| 330M/330M [00:06<00:00, 50.6MB/s]
```

Out[32]:
```
VisionTransformer(
  (conv_proj): Conv2d(3, 768, kernel_size=(16, 16), stride=(16, 16))
  (encoder): Encoder(
    (dropout): Dropout(p=0.0, inplace=False)
    (layers): Sequential(
      (encoder_layer_0): EncoderBlock(
        (ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
        (self_attention): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=768,
out_features=768, bias=True)
        )
        (dropout): Dropout(p=0.0, inplace=False)
        (ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
        (mlp): MLPBlock(
          (0): Linear(in_features=768, out_features=3072, bias=True)
          (1): GELU(approximate='none')
```

Perform the necessary network surgery:

In [ ]:
```
ft_vit.heads = nn.Sequential(
    nn.Linear(in_features = 768, out_features = 1, bias = True)
)
ft_vit
```

Out[33]:
```
VisionTransformer(
  (conv_proj): Conv2d(3, 768, kernel_size=(16, 16), stride=(16, 16))
  (encoder): Encoder(
    (dropout): Dropout(p=0.0, inplace=False)
    (layers): Sequential(
      (encoder_layer_0): EncoderBlock(
        (ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
        (self_attention): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=768,
out_features=768, bias=True)
        )
        (dropout): Dropout(p=0.0, inplace=False)
        (ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
        (mlp): MLPBlock(
          (0): Linear(in_features=768, out_features=3072, bias=True)
          (1): GELU(approximate='none')
          (2): Dropout(p=0.0, inplace=False)
          (3): Linear(in_features=3072, out_features=768, bias=True)
          (4): Dropout(p=0.0, inplace=False)
        )
```

Then freeze the parameters to allow for transfer learning:

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

for param in ft_vit.heads.parameters():
  param.requires_grad = True
```

Now, I just have to train the model! These function are already defined in the previous parts.

In [ ]:
```
loss_fn = nn.BCELoss()
optimizer = torch.optim.SGD(ft_vit.parameters(), lr = 0.01)
epochs = 10

ft_vit = ft_resnet50.to(device = 'cuda')

vit_accuracy = []
vit_loss = []

for i in range(1,epochs+1):
  train_model(train_dataloader, ft_vit, loss_fn, optimizer)
  print(f"-----Epoch {i} Complete-----")
  accuracy, loss = test_model(validation_dataloader, ft_vit, loss_fn)
  vit_accuracy.append(accuracy)
  vit_loss.append(loss.item())

  print(f"Accuracy: {accuracy}")
  print(f"Loss: {loss}\n")

print("-----Training Complete-----")
```

```
-----Epoch 1 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.20705947279930115

-----Epoch 2 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.2055712193250656

-----Epoch 3 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.20627114176750183

-----Epoch 4 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.20811839401721954

-----Epoch 5 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.2056569904088974

-----Epoch 6 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.20609314739704132

-----Epoch 7 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.20818911492824554

-----Epoch 8 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.2067616879940033

-----Epoch 9 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.206425741314888

-----Epoch 10 Complete-----
Accuracy: 0.9971014492753624
Loss: 0.20671357214450836

-----Training Complete-----
```

Training succeeded once again, and the model trains at near 100% accuracy! Interestingly, the vision transformer achieves a high accuracy much quicker than the previous two models. Let's see hwo it performs on the test set:

In [ ]:
```
accuracy, loss = test_model(test_dataloader, ft_vit, loss_fn)
print(f"Accuracy: {accuracy}")
print(f"Loss: {loss}\n")
```

```
Accuracy: 1.0
Loss: 0.20008213818073273
```

Once again, depending on the run, the vision transformer may or may not classify certain images in the test set incorrectly. The follow block outputs any of the misclassified images.

In [ ]:
```
class_dict = {0:'Deforestation',1:'Forest'}

for X, y in test_dataloader:
  img = X
  X = X.to(device = 'cuda')
  y = y.to(device = 'cuda')

  X = X.float()
  y = y.float()

  prediction = ft_vit(X)

  p = 0
  if prediction.item() > 0.5:
    p = 1

  if not p == y:
    print(class_dict[p])
    plt.axis("OFF")
    plt.imshow(img.squeeze(0).T)
    plt.show()
```

Part 3: Model Comparison

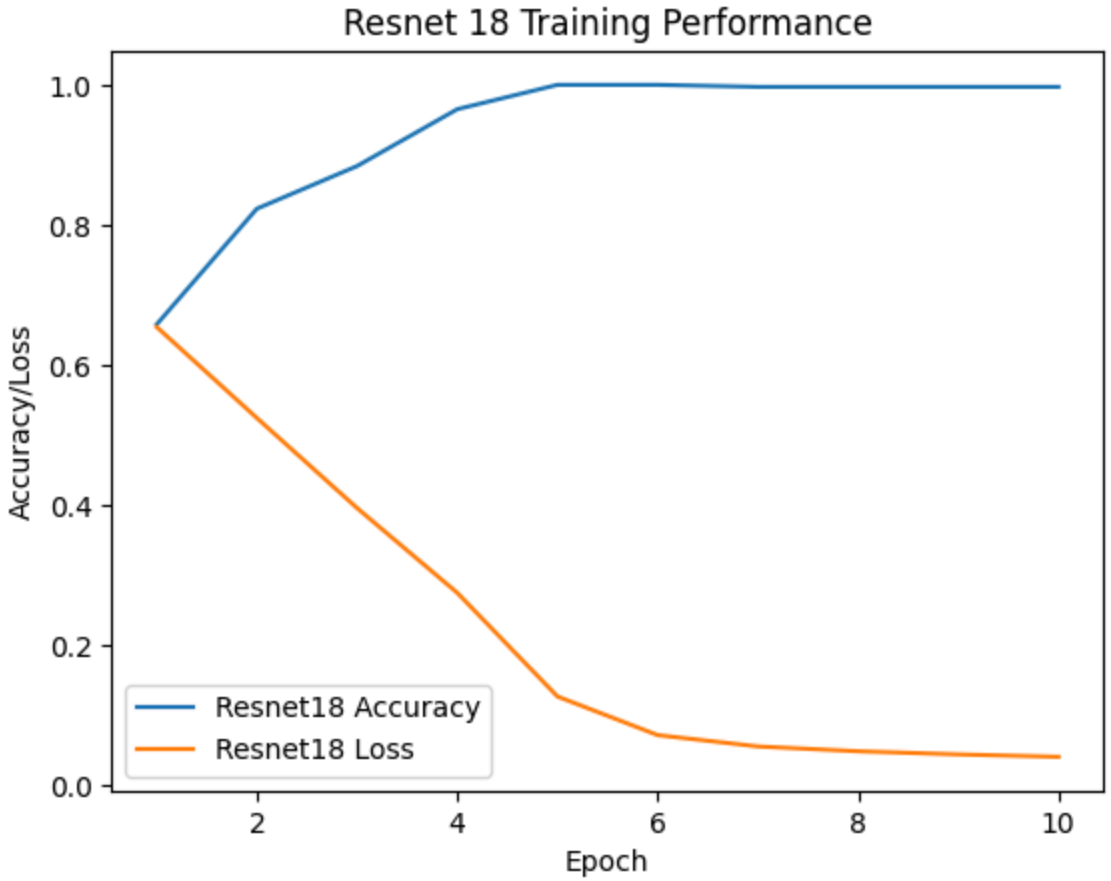
In this second I simply show the pros and cons of each model, displaying relevant graphs.

In [ ]:
```
epochs = [1,2,3,4,5,6,7,8,9,10] #ten epochs

plt.plot(epochs, resnet18_achain, label = "Resnet18 Accuracy")
plt.plot(epochs, resnet18_lchain, label = "Resnet18 Loss")

plt.xlabel("Epoch")
plt.ylabel("Accuracy/Loss")
plt.title("Resnet 18 Training Performance")
plt.legend()

plt.show()
```

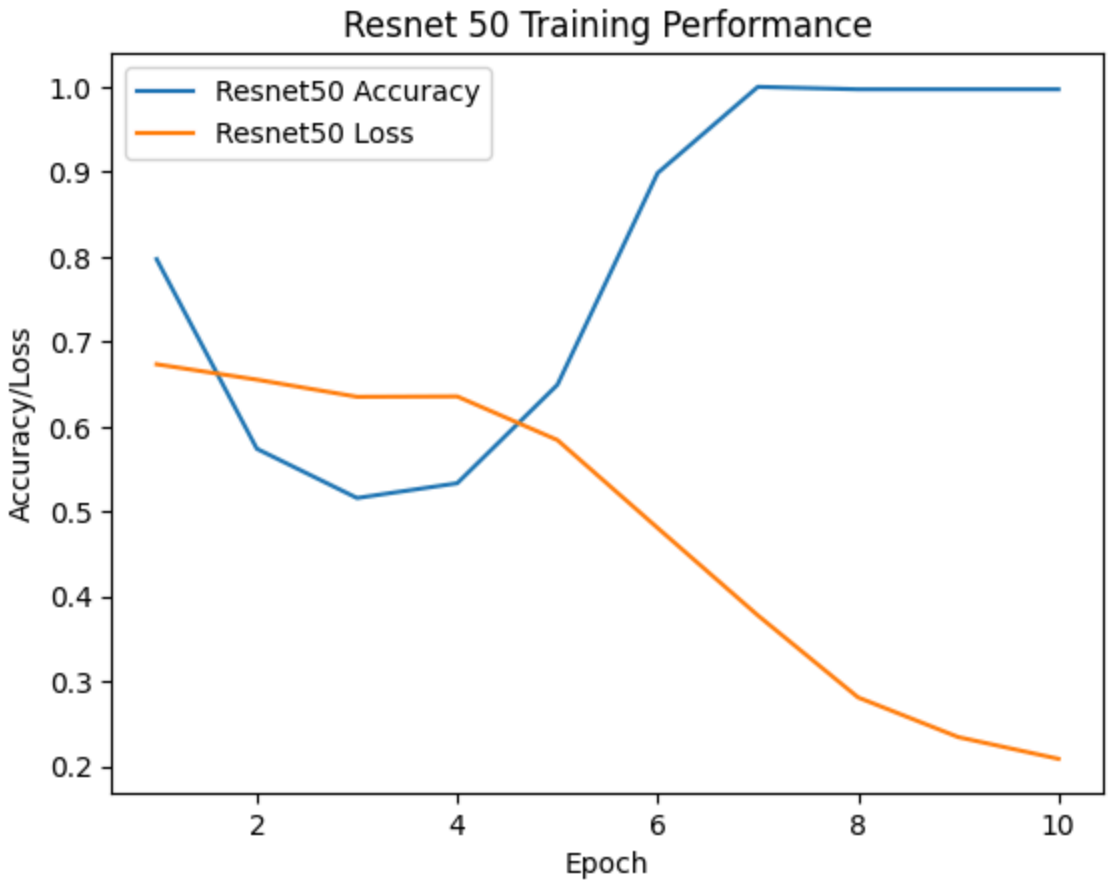


The fine-tuned resnet 18 model performs very well. The training accuracy increases to 99.7% and the loss is very low!

In [ ]:
```
plt.plot(epochs, resnet50_accuracy, label = "Resnet50 Accuracy")
plt.plot(epochs, resnet50_loss, label = "Resnet50 Loss")

plt.xlabel("Epoch")
plt.ylabel("Accuracy/Loss")
plt.title("Resnet 50 Training Performance")
plt.legend()

plt.show()
```



Just as with the fine-tuned resnet18, the fine-tuned resnet50 model performs equally as well, although the loss is higher.

In [ ]:
```
plt.plot(epochs, vit_accuracy, label = "ViT-16 Accuracy")
plt.plot(epochs, vit_loss, label = "ViT-16 Loss")

plt.xlabel("Epoch")
plt.ylabel("Accuracy/Loss")
plt.title("Vision Transformer Training Performance")
plt.legend()

plt.show()
```

The Vision Transformer's performance does not increase or decrease throughout the model's training period. This is very interesting, however also a little suspicious.

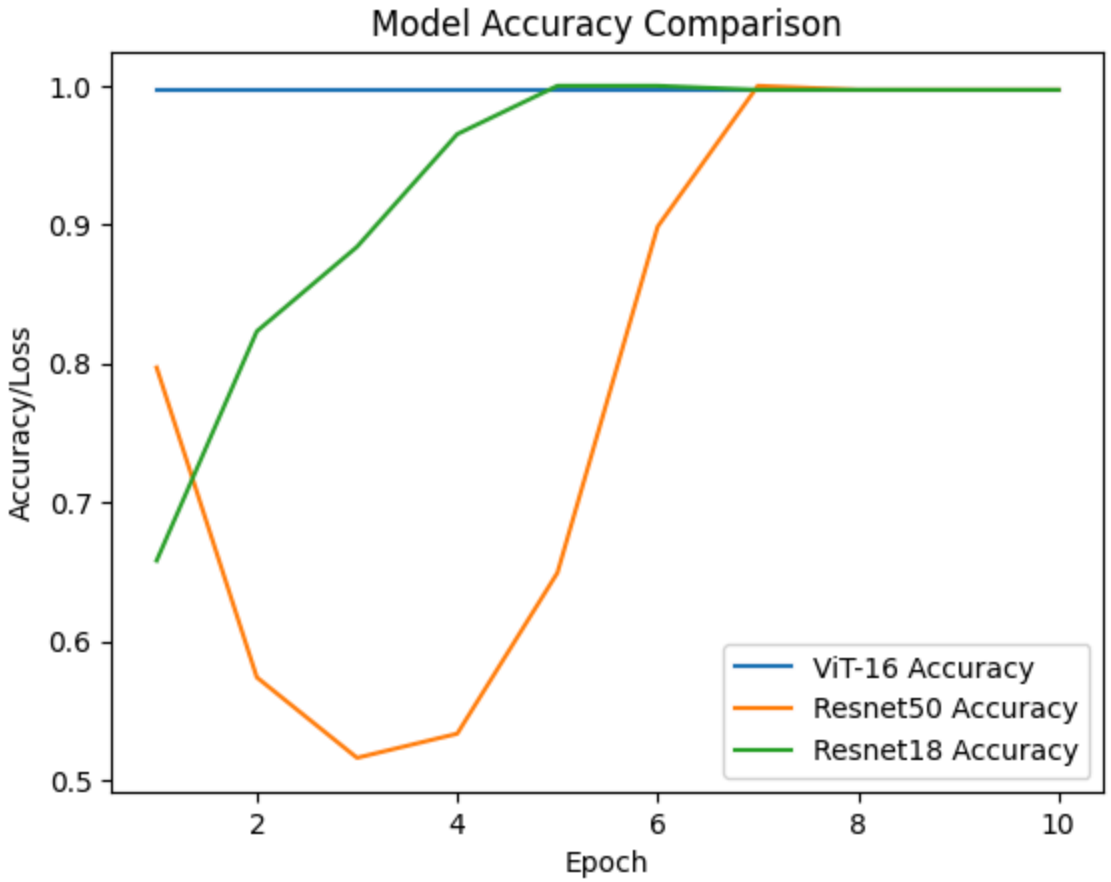
Fianlly, compare each of the models:

In [ ]:
```
plt.plot(epochs, vit_accuracy, label = "ViT-16 Accuracy")
plt.plot(epochs, resnet50_accuracy, label = "Resnet50 Accuracy")
plt.plot(epochs, resnet18_achain, label = "Resnet18 Accuracy")



plt.xlabel("Epoch")
plt.ylabel("Accuracy/Loss")
plt.title("Model Accuracy Comparison")
plt.legend()

plt.show()
```



Each of the models, at the end of their training, end up with relatively the same accuracy. Each model also has 100% accuracy on the test set (for this run) indicating that they are all good choices to move forward with. I end up choosing to move forward with the resnet 18 model, since it has the lowest loss.

Part 4: Model Application on Amazon Imagery

It was important for me in this project to develop a system that can integrate easily into some sort of usable platform. To that end, in this part I develop a method to localize areas of deforestation in a larger scene using the models above. Since all of the models above trained

incredibly well, I am going to use the fine-tuned resnet18 for this implementation, simply because it is the quickest model. (Although the time differences for this project were almost

The first step is to acquire some new scenes that are previously not-used. For this part, it should also be important that these scenes are not completely deforested or forested, but instead have a mix.

I decided to choose a scene from 2017, and another from the same location in 2024. This is to demonstrate just how severe the deforestation in the Amazon Rainforest is.

In [ ]:
```
files.upload()
```

...

Let's take a look at these images:

In [ ]:
```
a1 = cv2.imread('Analysis2017.jpg')
a2 = cv2.imread('Analysis2024.jpg')

print("Image from 2017:")
cv2_imshow(a1)

print("Image from 2024:")
cv2_imshow(a2)
```

```
Image from 2017:
```

```
Image from 2024:
```

These images are quite revealing by themselves of the deforestation in the Amazon. You can even see a wildfire burning in the 2017 image! I am interested in seeing how the model classifies the smoke...

In the next cell I develop a function to classify regions of these images as either forest or deforested.

In [ ]:
```
from PIL import Image

def analyze_image(image,save_dir):
  #First I must format the image into 1200x1800 pixels:
  image = cv2.resize(image,(1800,1200))
  overlay = image.copy()

  #Create a directory to save subimages in
  os.mkdir(save_dir)

  print(save_dir)

  #Then go through the image and segment 50x50 portions from it
  #This is basically the same as in the data processing step
  x = y = 0
  i = 0

  while x <= 1150:
    while y <= 1750:
      cropped = image[x:x+49,y:y+49]
      #I need to save the cropped image, and then load it again...
      #This is to turn it into jpg format
      cv2.imwrite(f"{save_dir}/test{i}.jpg",cropped)
      img = Image.open(f"{save_dir}/test{i}.jpg")

      #Format the image
      X = transform(img)
      X = X.unsqueeze(0)
      X = X.float()

      #print(X)

      X = X.to(device = 'cuda')

      #Image is properly formated, pass into the network
      pred = ft_resnet18(X)

      if pred.item() < 0.5:
        #print('Deforestation Detected')
        overlay = cv2.rectangle(overlay,(y,x),(y+50,x+50),(0,255*(pred.i


      y +=50
      i += 1

    x+=50
    y = 0
  image = cv2.addWeighted(overlay,0.3,image,0.7,0)
  return image
```

In [ ]:
```
test = cv2.imread('Analysis2017.jpg')
img = analyze_image(test,"test2017-8")

cv2_imshow(img)
```

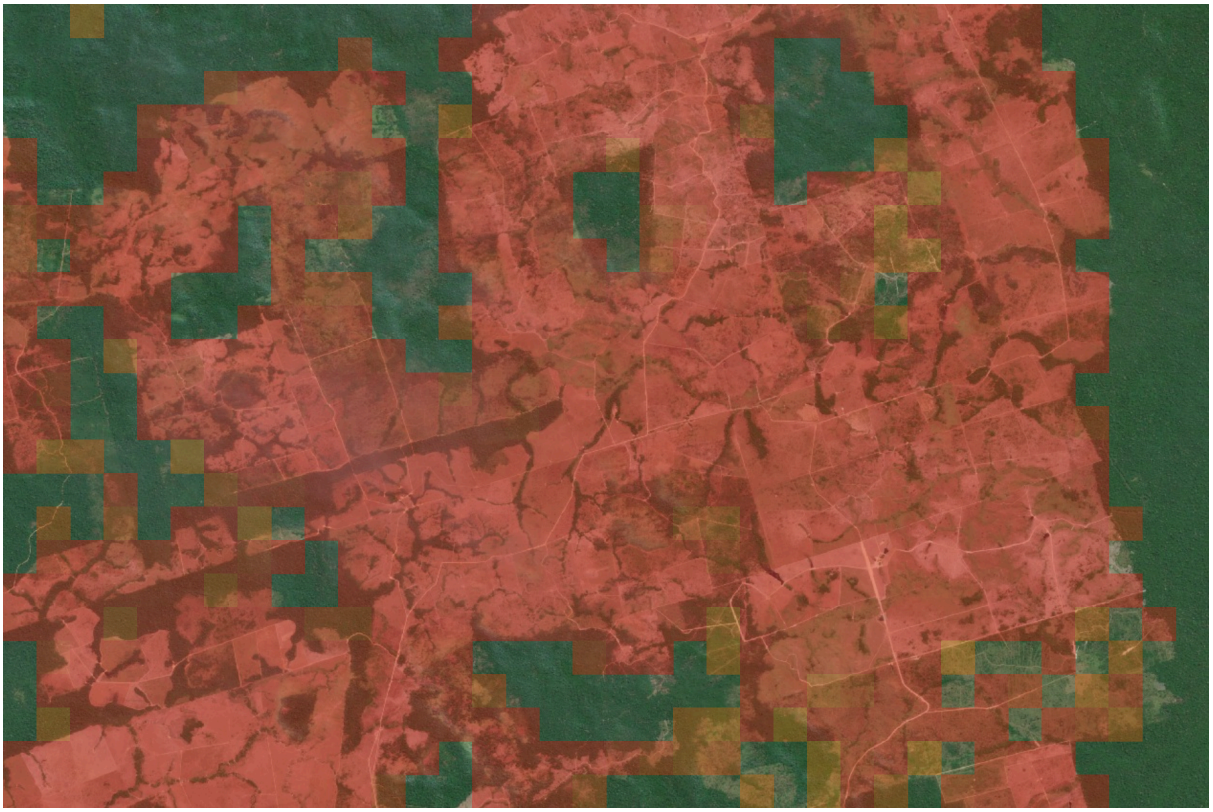
```
test2017-8
```

And the 2024 image:

In [ ]:
```
test = cv2.imread('Analysis2024.jpg')
img = analyze_image(test,"test2024-4")

cv2_imshow(img)
```

```
test2024-4
```



Looks like the regions were classified fairly well! The deeper red regions indicate the model is highly confident that those regions have some deforested parts, whereas the more yellow regions suggest that the model is less confident.

I hope you enjoyed! Thank you!

In [ ]: