FIRE IN THE AMAZON

Understanding the Amazon with Pytorch

Schloeter, Daniela

Sutter Schneider, Guillermina

December, 2019 - George Washington University

Introduction

Understanding the Amazon with Pytorch. During the last couple of years, the focus of international-development-oriented policies has slowly moved towards the environment. The increasing loss of area forest imposes a threat to biodiversity, contributes to climate change, habitat loss, and has many other devastating effects. Therefore, it is of crucial importance to analyze and understand these areas to be able to take action faster and more effectively.

The Amazon rainforest is located in South America and accounts for the largest share of deforestation. Most of the forest is contained within Brazil (60%), Peru (13%), Colombia (10%), and minor amounts in Venezuela, Ecuador, Bolivia, Guyana, Suriname, and French Guiana. Over the last couple of months, the media attention was focused on the Amazon rainforest. More than 80,000 fires were reported from August through September, accounting for more than an 80 percent increase in the number of fires compared to 2018.

As native-born Latin Americans, we believe and understand the importance of studying the region so as to provide precise information about the conditions of the Amazon rainforest. Being able to classify satellite images of the Amazon rainforest would help policy-makers in the region in the design of effective measures to fight deforestation.

Since the dataset we will work with contains labels for slash-and-burn areas[1], it will allow to identify such areas in the Amazon and be of great importance within the next years to evaluate and keep track of how this phenomenon develops.

[1] Slash-and-burn agriculture can be considered to be a subset of the shifting cultivation label and is used for areas that demonstrate recent burn events. This is to say that the shifting cultivation patches appear to have dark brown or black areas consistent with recent burning. This NASA Earth Observatory article gives a good primer on the practice as does this wikipedia article.

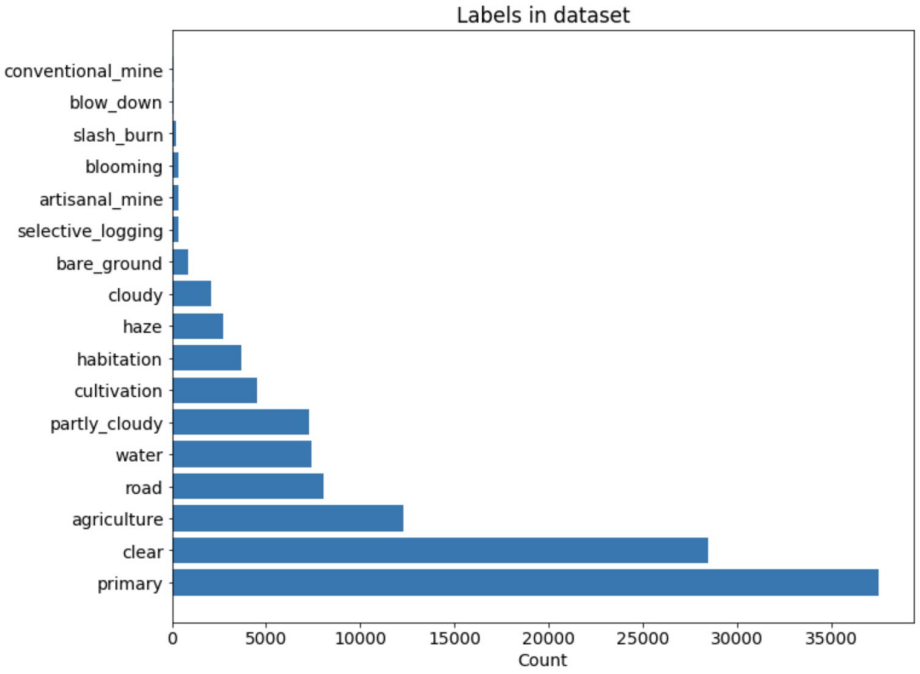
Dataset

The dataset belongs to the Kaggle competition "Planet: Understanding the Amazon from Space." Satellite image chips with atmospheric conditions and various classes of land cover/land use are provided by Planet Labs and its Brazilian partner SCCON.

The data is already divided into a train and test set, and contains 40,479 jpg files and 20,522 jpg files of size 256x256, respectively. In order to access the data, we downloaded Homebrew to unzip .7z files and followed these instructions to unzip files. Since we did not have a validation set and the test set did not come with labels to evaluate the accuracy of our model, we decided to take 20% of the train set and create a test set and 5% to create a validation set. The final structure of the data looks as follows:

```
Train set: 30,763
Test set: 8,096
Validation set: 1,620
```

Each image chip has at least one and potentially a combination of more than one of the following labels: agriculture, artisanal mining, bare ground, blooming, blow down, clear, cloudy, conventional mining, cultivation, habitation, haze, partly cloudy, primary, road, selective logging, slash and burn, and water. The most common labels in this data set are rainforest, agriculture, rivers, towns/cities, and roads, whereas the least common labels are selective logging, slash and burn, blooming, conventional mining, artisanal mining, and blow down. A breakdown of the label count is shown below:



The labels can also be broken into three broad groups: atmospheric conditions, common land cover and land use phenomena, and rare land cover and land use phenomena. Images under the atmospheric conditions closely mirror what one would see in a local weather forecast: clear, partly cloudy, cloudy, and haze. Labels such as rainforest, agriculture, rivers, towns/cities, and roads, are contained in the common land cover and land use phenomena group. Lastly, in the rare land cover and land use phenomena group one can find images with the following labels: selective logging, slash and burn, blooming, conventional mining, artisanal mining, and blow down.

Network structure

In this section we will lay down the basic principles and architecture of the different networks used along the project. Four different types of networks were tested for this project: a CNN built from scratch, and three pretrained models from pytorch (VGG19, VGG19_bn, and ResNet50)

CNN - Convolutional Neural Network built from scratch

A Convolutional Neural Network is a deep learning algorithm that takes images as inputs, assign importance to certain objects in the images, and differentiate one image from another. It is a mapping process.

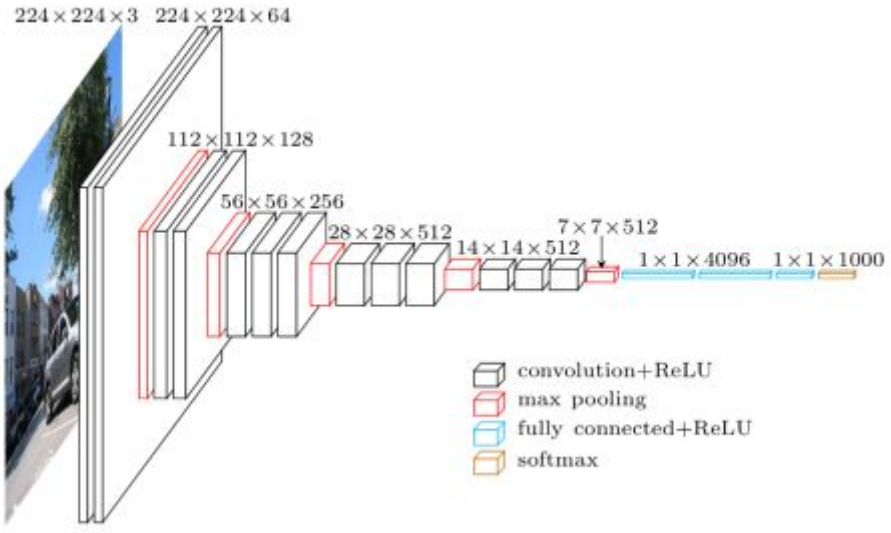
We should give special attention to five elements when starting to build a CNN: Max or Avg pooling, the kernel size, stride, batch normalization and number of layers. Max or

Avg pooling is used for reducing the spatial size of the image. It is often used for saving computational power. The kernel size and stride serve to calculate the size of the feature maps obtained in each layer. Batch normalization refers to the scaling applied to the inputs within each layer. The first layers create the main feature maps of the images. Additional layers create more detailed feature maps.

VGG -Very Deep Convolutional Networks for Large-Scale Image Recognition

The VGG -19 model is a CNN trained from the Image Large Scale Visual Recognition Challenge (ILSVRC) dataset. The ILSVRC uses a subset of the ImageNet database of around 1000 images in each of 1000 categories. The model has 19 layers and it can classify up to one thousand categories. It was invented by the Visual Geometry Group from the University of Oxford. The VGG-19_bn has the same structure (19 layers) as the VGG19 but it has Batch Normalization included.

From Pytorch documentation we know that all pretrained models expect mini-batches of input images of size (3,H,W). The 3, makes reference to images of 3-channel RGB. The height and weight should be of at least 224 pixels. Then we need to normalize the images using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225]. The following image shows the visual representation of the VGG19 network and it is followed by a table with the details about the different VGG structures.



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

VGG19 ARCHITECTURE

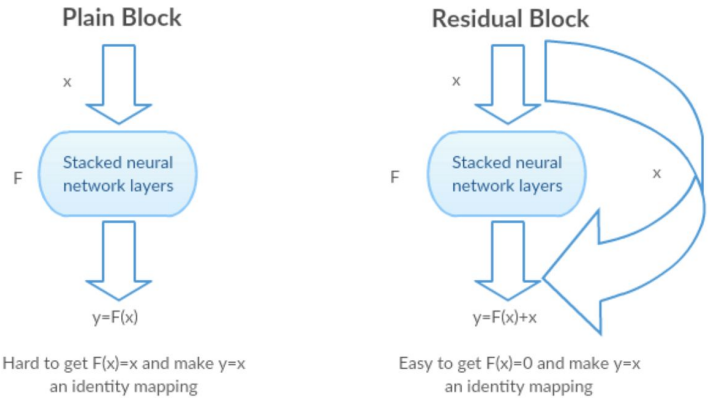
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
```

```
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0,dilation=1,ceil_mode=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0,dilation=1,ceil_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2,padding=0,dilation=1,ceil_mode=False))
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=17, bias=True))
```

ResNet50 - Residual Network

ResNet-50 is a convolutional neural network that is trained on more than a million images. The network is 50 layers deep and can classify images into 1000 object categories. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of at least 224x224.

As the image below shows, this model skips the training of few layers using skip-connections or residual connections.



As shown below, we present the network architecture of the four-layer ResNet50.

```
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_running_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=False)
     (bn1):BatchNorm2d(64,eps=1e-05,momentum=0.1,affine=True, track_running_stats=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_running_stats=True)
     (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
     (bn3):BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (relu): ReLU(inplace=True)
     (downsample): Sequential(
       (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
       (1):BatchNorm2d(256,eps=1e-05,momentum=0.1, affine=True, track_running_stats=True)))
   (1): Bottleneck(
     (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
     (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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_running_stats=True)
     (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
     (bn3): BatchNorm2d(256,eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (relu): ReLU(inplace=True))
   (2): Bottleneck(
     (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
     (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256,eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)))
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05,momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3,3), stride=(2,2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05,momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05,momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=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_running_stats=True)))
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128,eps=1e-05, momentum=0.1, affine=True, track_running_stats=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_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512,eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True))
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128,eps=1e-05, momentum=0.1, affine=True, track_running_stats=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_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512,eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True))
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128,eps=1e-05, momentum=0.1, affine=True, track_running_stats=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_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512,eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)))
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256,eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
  (conv2):Conv2d(256,256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256,eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024,eps=1e-05,momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (downsample): Sequential(
    (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
    (1): BatchNorm2d(1024,eps=1e-05,momentum=0.1,affine=True,track_running_stats=True)))
(1): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256,eps=1e-05, momentum=0.1, affine=True, track_running_stats=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_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3):BatchNorm2d(1024,eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True))
(2): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256,eps=1e-05, momentum=0.1, affine=True, track_running_stats=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_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024,eps=1e-05,momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True))
(3): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256,eps=1e-05, momentum=0.1, affine=True, track_running_stats=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_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024,eps=1e-05,momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True))
(4): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256,eps=1e-05, momentum=0.1, affine=True, track_running_stats=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_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024,eps=1e-05,momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True))
(5): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05,momentum=0.1, affine=True, track_running_stats=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_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024,eps=1e-05,momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)))
(layer4): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512,eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(512,512, kernel_size=(3,3), stride=(2,2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512,eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048,eps=1e-05,momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(2048,eps=1e-05,momentum=0.1,affine=True,track_running_stats=True)))
  (1): Bottleneck(
    (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512,eps=1e-05, momentum=0.1, affine=True, track_running_stats=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_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048,eps=1e-05,momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True))
  (2): Bottleneck(
    (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512,eps=1e-05, momentum=0.1, affine=True, track_running_stats=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_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048,eps=1e-05,momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)))
(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(fc): Linear(in_features=2048, out_features=17, bias=True))
```

Experimental setup

Image Preprocessing

In order to prepare the data for a pretrained model we performed a series of transformations to the images with `transforms.Compose( )`. It resizes the images to 224x224, flips them randomly both horizontally and vertically, changes its brightness (0.1), saturation (10), contrast (30), and hue (0.1) with ColorJitter. Feeding the network with different-looking inputs as is the case for randomly flipping the images. Lastly,

the class turns the images into tensors and normalizes them -a requirement for inputs when using pytorch pretrained models.

We incorporated the previous transformations into a `CustomDatasetFromImages( )` class. This class was created to iterate over the dataset by assigning indexes to the images. The class receives a path to a csv file. This file contains the image names and its labels in the first and second columns, respectively. The class extracts an image name from the csv file, look for it in a previously assigned folder, and uploads the image. After uploading, the images are changed to RGB (3 channels) to match the specified parameters for pretrained models in Pytorch. Following this, the images are transformed as previously described. For the label preprocessing, since the networks require one-hot-encoded labels, we used a label binarizer plus a for loop as a form of encoding. The label binarizer returns an array of size (number of classes contained for each image x 17). 17 is the number of the categories. As we are dealing with multi-label classification, an image label can have one or more than one label attached to it (which is most of the cases for this dataset). Thus, our one-hot-encoded labels show a 1 for each label attached to the images. On the contrary, when the class is not included in the image label, the array shows a 0. The target is transformed into a torch object. The class returns an array for the image and an array for the label.

The class is used to create a dataset which is the input of the DataLoader. The DataLoader receives a dataset, batch size, the shuffle option (which we set to *True*) and the number of workers (4). The DataLoader is used for both the training set and the validation set and returns the X and y to train and validate our model.

The parameters for the model were selected based on the Cuda memory limitations and by testing accuracy of each model run. The number of epochs was limited due to time restrictions. The models tested initially ran in 20 min approximately but the accuracy was low. Therefore, when the model was changed to a pretrained one it took around 2 hours and 30 minutes to run. Increasing the number of epochs increased the running time and reduced the possibilities of improving the model by limiting the number of tests to perform.

The final parameters are listed below.

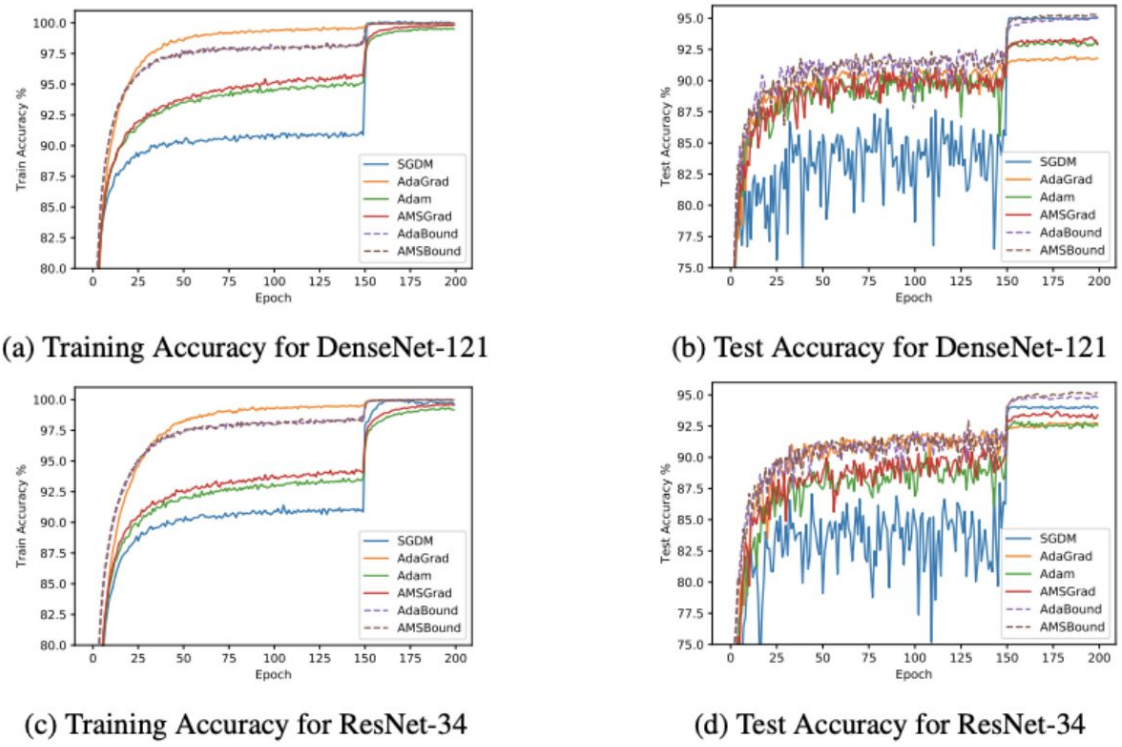
LR = 0.1
N_EPOCHS = 5
BATCH_SIZE = 70

Finally, LabelBinarizer( ) shuffles the output label. Ours result in:

```
['agriculture' 'artisinal_mine' 'bare_ground' 'blooming' 'blow_down'
 'clear' 'cloudy' 'conventional_mine' 'cultivation' 'habitation' 'haze'
'partly_cloudy' 'primary' 'road' 'selective_logging' 'slash_burn' 'water']
```

Optimizers and Loss Functions

`optim.SGD( )`: This article explains this optimizer performs computations on a small subset or random selection of data examples instead of on the whole dataset --which is inefficient. SGD produces the same performance as regular gradient descent when the learning rate is low. We did not choose Adam because it can sometimes fail to converge to an optimal solution under specific settings, as this paper states. See image below for optimizer comparison:



(a) Training Accuracy for DenseNet-121

(b) Test Accuracy for DenseNet-121

(c) Training Accuracy for ResNet-34

(d) Test Accuracy for ResNet-34

`BCEWithLogitLoss( )`: Since we had a multi-binary classification problem, we decided to move forward with this loss function for the validation set. BCEWithLogitsLoss = One Sigmoid Layer + BCELoss

`F-beta Score( )`: Based on the sklearn documentation, the F-beta score is the weighted harmonic mean of precision and recall, reaching its optimal value at 1 and its worst value at 0. It takes into consideration both the recall and precision metrics. In order for us to compare with the models in the Kaggle competition, we chose this metric with a β equal to 2 to favor recall over precision.

Results

Using the scripts from Exam II, we started building the CNN by adapting the structure as we saw fit. We also changed the dropout, learning rate, batch size, and number of epochs. To evaluate the changes on the model's performance, we changed the previous parameters one at a time (fine-tuning). After seven runs of fine-tuning and modifying the network architecture, the model's performance was not significantly improving. Therefore, we decided to try pretrained networks. We tested ResNet50 and VGG19, and compared each network's performance.

A more in-detail structure of the process described is shown below:

CNN from scratch

Run #1

Activation function= relu, Dropout = 0.25, LR = 0.1, Optimizer = SGD, Batch size = 500, Epochs = 10

- 1st layer: Kernel size (3, 3), Max pool (2, 2)
- 2nd layer:Kernel size (3, 3), BatchNorm2d, Max pool (2, 2)
- 3rd layer: Kernel size (3, 3)
- 4th layer: Kernel size (3, 3), BatchNorm2d, Max pool (2, 2)
- 5th layer: Kernel size (3, 3), BatchNorm2d, Max pool (2, 2)
- 6th layer: Kernel size (2, 2), BatchNorm2d, Max pool (2, 2)
- 7th layer: Linear, BatchNorm1d, Dropout
- 8th layer: Linear

Fbeta score = 0.65

Run #2

Activation function= relu, Dropout=0.25, LR=0.01,Optimizer=SGD, Batch size=500, Epochs=10

- 1st layer: Kernel size (3,3), Max pool (2,2)
- 2nd layer:Kernel size (3,3), BatchNorm2d, Max pool (2,2)
- 3rd layer: Kernel size (3,3)
- 4th layer: Kernel size (3,3), BatchNorm2d, Max pool (2,2)
- 5th layer: Kernel size (3,3), BatchNorm2d, Max pool (2,2)
- 6th layer: Kernel size (2,2), BatchNorm2d, Max pool (2,2)
- 7th layer: Linear, BatchNorm1d, Dropout
- 8th layer: Linear

BCEwithlogitloss(train)=0.9, time=19min,Fbeta score=0.61

Run #3

Activation function= relu, Dropout=0.2, LR=0.1,Optimizer=SGD, Batch size=506, Epochs=10

- 1st layer: Kernel size (3,3), Max pool (2,2)
- 2nd layer:Kernel size (3,3), BatchNorm2d, Max pool (2,2)
- 3rd layer: Kernel size (3,3)
- 4th layer: Kernel size (3,3), BatchNorm2d, Max pool (2,2)
- 5th layer: Kernel size (3,3), BatchNorm2d, Max pool (2,2)
- 6th layer: Kernel size (2,2), BatchNorm2d, Max pool (2,2)
- 7th layer: Linear, BatchNorm1d, Dropout
- 8th layer: Linear

BCEwithlogitloss(train)=0.2, time=19min,Fbeta score=0.68

Run #4

Activation function= relu, Dropout=0.1, LR=0.1,Optimizer=SGD, Batch size=506, Epochs=10 (Added horizontal flip in pretraining)

- 1st layer: Kernel size (3,3), Max pool (2,2)
- 2nd layer:Kernel size (3,3), BatchNorm2d, Max pool (2,2)
- 3rd layer: Kernel size (3,3)
- 4th layer: Kernel size (3,3), BatchNorm2d, Max pool (2,2)
- 5th layer: Kernel size (3,3), BatchNorm2d, Max pool (2,2)
- 6th layer: Kernel size (2,2), BatchNorm2d, Max pool (2,2)
- 7th layer: Linear, BatchNorm1d, Dropout
- 8th layer: Linear

BCEwithlogitloss(train)=0.25, time=17min,Fbeta score=0.65

Run #5

Activation function= relu, Dropout=0.1, LR=0.1,Optimizer=SGD, Batch size=506, Epochs=10

- 1st layer: Kernel size (3,3), Max pool (2,2)
- 2nd layer:Kernel size (3,3), BatchNorm2d, Max pool (2,2)
- 3rd layer: Kernel size (3,3)

- 4th layer: Kernel size (3,3), BatchNorm2d, Max pool (2,2)
- 5th layer: Kernel size (3,3), BatchNorm2d, Max pool (2,2)
- 6th layer: Linear, BatchNorm1d, Dropout
- 7th layer: Linear

BCEwithlogitloss(train)=1.04, Fbeta score=0.64

Run #6

Activation function = relu, Dropout = 0.1, LR = 0.1,Optimizer = SGD, Batch size = 506, Epochs = 25

- 1st layer: Kernel size (3, 3), Max pool (2, 2)
- 2nd layer: Kernel size (3, 3), BatchNorm2d, Max pool (2, 2)
- 3rd layer: Kernel size (3, 3)
- 4th layer: Kernel size (3, 3), BatchNorm2d, Max pool (2, 2)
- 5th layer: Kernel size (3, 3), BatchNorm2d, Max pool (2, 2)
- 6th layer: Kernel size (2, 2), BatchNorm2d, Max pool (2, 2)
- 7th layer: Kernel size (2, 2), BatchNorm2d, Max pool (2, 2)
- 8th layer: Linear, BatchNorm1d, Dropout
- 9th layer: Linear

BCEwithlogitloss(train) = 0.26, time = 43min, Fbeta score = 0.65

Run #7

Activation function = relu, Dropout = 0.1, LR = 0.1,Optimizer = SGD, Batch size = 506, Epochs = 10

- 1st layer: Kernel size (3, 3), Max pool (2, 2)
- 2nd layer: Kernel size (3, 3), BatchNorm2d, Max pool (2, 2)
- 3rd layer: Kernel size (3, 3)
- 4th layer: Kernel size (3, 3), BatchNorm2d, Max pool (2, 2)
- 8th layer: Linear, BatchNorm1d, Dropout
- 9th layer: Linear

BCEwithlogitloss(train) = 0.25, time = 43min, Fbeta score = 0.65

The following table displays the accuracy scores and parameters measures

VGG19

Run #1

LR = 0.1,Optimizer = SGD, Batch size = 90, Epochs = 2

time = 120 min, F-beta score = 0.752

Run #2

LR = 0.1,Optimizer = SGD, Batch size = 90, Epochs = 2 1/2

time = 150 min, F-beta score = 0.798

Run #3

LR = 0.01,Optimizer = SGD, Batch size = 90, Epochs = 4

time = >180 min, F-beta score = 0.759

VGG19_bn

Run #1

LR = 0.1, Optimizer = SGD, Batch size = 70, Epochs = 1

time = 42 min, F-beta score = 0.804

Run #2

LR = 0.1,Optimizer = SGD, Batch size = 70, Epochs = 2

time = 200 min, F-beta score = 0.836

Run #3

LR = 0.1,Optimizer = SGD, Batch size = 70, Epochs = 5

time = 510 min, F-beta score = 0.840

ResNet50

Run #1

LR = 0.03, Optimizer = SGD, Batch size = 90, Epochs = 3

time = 90 min, F-beta score = 0.73

Run #2

(1-2 neurons shut down)

LR = 0.03,Optimizer = SGD, Batch size = 90, Epochs = 3

time = 150 min, F-beta score =0.781

Run #3

(1-2 neurons shut down)

LR = 0.03,Optimizer = SGD, Batch size = 90, Epochs = 2

time = 45 min, F-beta score = 0.767

From the results displayed above, the worst-performing model was the CNN built from scratch. Although by building a CNN network from scratch we can carefully fine-tune the hyperparameters and the network architecture, it takes more time to build and more runs to improve performance.

ResNet50 performed better compared to the accuracy scores of the previous model. Particularly for this model, shutting down half of the layers significantly improved the F-beta score.

VGG-19 performed similar to ResNet50. The F-beta score was between 0.78 and 0.75 depending on the learning rate and epochs used.

Finally, we can conclude that the best-performing model is the VGG-19_bn. While this model took longer to train it was able to produce an F-beta score of 0.84.

Conclusion

During this project we faced two types of restrictions: one associated with time and another one with the computational power. Complex networks can take longer to train. There is sometimes a trade-off between runtime and model accuracy. Something similar can be said about computational cost. More complex models demand more computational power. For example, most of the times when we increased the size of the batches, CUDA ran out of memory and it would have taken fewer minutes to train the model. But if we decreased the size of the batches, the model ran but took longer to train.

Finding wide varieties of multi-label classification examples as guides to follow, it sometimes became difficult. Pytorch was initially released in October, 2016 with an

updated and more stable version released in October, 2019. It is a relatively new framework and most issues have not been covered yet by Pytorch forums. Thus, there are not as many examples as other frameworks that have been in the marketplace for a longer time.

Within this project's scope, and given the time and computational power restrictions, we can also conclude pretrained networks performed better on this dataset than the CNN we built from scratch.