FIRE IN THE AMAZON

Understanding the Amazon with Pytorch

Schloeter, Daniela

The George Washington University - December 2019

INTRODUCTION

The objective of this project is to classify satellite images from the Amazon Rainforest. Forests are crucial for the planet, they absorb CO2 and return oxygen. Rainforests also contribute to maintain the world's water cycle and serve as habitat for an extensive number of animals and plants. We could keep counting the reasons why rainforests are so important. The Amazon is the biggest rainforest in the world, covering almost 2 million miles of land. Thus, we must protect it. Therefore, it is crucial to understand it and track its changes. Being able to label images from the Amazon could help local governments and stakeholders find faster and more efficient solutions to problems in the area.

The dataset for this project was obtained from the Kaggle competition "Planet: Understanding the Amazon from Space". The dataset is provided by Planet and its Brazilian partner, SCCON. The dataset contains 40479 images with their corresponding labels. For this project, we are going to use 30763(76%) images for training, 8096(20%) images for testing, and 1620(4%) images for validating.

The dataset has 17 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. Each image can have one or more labels assigned to it. This is a multilabel classification problem.

My main contribution to this project was developing the base of the code and debugging when necessary. I tested and tracked the accuracy for twelve models and created the main code files to perform and test the model. Through the project, we encountered several coding errors, which we solved by researching similar problems on the internet and by testing the shapes, sizes, and types of our variables in the debugger.

BACKGROUND

For the second exam in our Machine Learning 2 class, we had to perform a multi-label classification problem in a blood cell database. The exam had to be completed in PyTorch. The exam served as an introduction to multi-label classification and the PyTorch framework. PyTorch is a relatively new framework and learning to use it adequately is of great use for solving deep learning problems. Thus, we decided to do the project in PyTorch.

One of the most popular techniques for image classification is deep neural networks. Learning from existing tagged images to label unseen images. One of the methods learned in class is Convolution Neural Network. This technique uses a kernel to learn the features from the images creating feature maps. In the final layers of the network, the image is classified into the predicted target. Convolutional Neural Networks are a powerful resource for image classification. Below we can see a network diagram for CNN. (Figure 1)

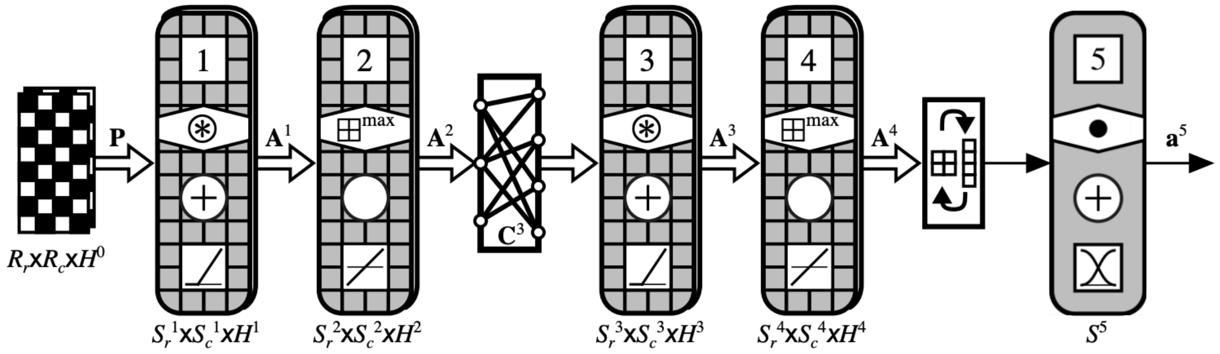

Figure 1- CNN diagram showing connections and the transformation vector. Retrieved from Prof. Amir Jafari's course presentations.

EXPERIMENTAL SETUP

Nevertheless, before entering into the details of how the networks were created for this project, it is essential to discuss how the loading of the images for a multi-label image classification problem should be performed in PyTorch.

First, I uploaded the CSV file, which contained a list of the images with their corresponding labels. I inspected the dataset to see all the possible combinations. I found that the most common combination is 'clear' and 'primary' which occurs 13 636, followed by 'partly_cloudy' and 'primary' occurring 3 630 times and 'cloudy,' 2 089 times. There are about 150 combinations that occur only once.

Once the CSV file was uploaded, I proceeded to split the data in a training, validation, and test set and save each set as a different CSV file. The division was performed using the train test split function from Sklearn.

Once the new files were created, I created a Class to upload each image and its encoded target with an index to enable the iteration method of the DataLoader. The class was named CustomDatasetFromImages, and it receives as argument the CSV path from the images to be uploaded. The class is initialized with a set of transformations to apply to the images. Then the length of the dataset is specified, and the final initialization step is the selection of the image to be processed.

The get item section of the class uploads the image and changes it from RGBA (4 channels) to RGB (3 channels) and applies the corresponding transformations from the initialization. After preparing the image, the get item reads the text labels and using the Label Binarizer from Sklearn returns an array of zeros and ones where one represents the observed label. The Label Binarizer returns different array sizes for each image, depending on the number of labels corresponding to that image. Each element of the array is a list containing 16 zeros and 1 one in the corresponding position of the label. I created a for loop to add the lists to have arrays of the same shape with zeros and ones in the corresponding labels of the image. The label is later converted to a torch object and a float number.

The transformations used in the image class are created by calling the class compose with each transformation inside. The first transformation consisted of resizing the image to 224 as it is the size used for the pre-trained models. I then proceeded to flip horizontally and vertically the images randomly. The next transformation was color jitter, which allows to adjust the brightness, saturation, contrast, and hue of the image. This transformation helped define and enhance the details in the images. Next, I transformed the image into a tensor object. Finally, I normalized the image as specified in the PyTorch documentation for pre-trained models.

For the next step I created an object that uses the DataLoader from PyTorch. The first argument of the data loader is an iterable dataset obtained from the class described above. The batch size must be specified for the data loader to create the variables to train or validate the model. Additionally, I defined the shuffle option as True and set the number of workers to 4. The number of workers enables the number of batches that are processed at the same time, which helps to reduce the running time.

I created a class of the model to be trained to predict the corresponding labels. I started by using the model I created for my machine learning exam two as a baseline. After changing hyper-parameters and the network architecture seven times, we realized it was more convenient to use a pre-trained network and I changed to the VGG19 network. I also tried the VGG19_bn network because the documentation suggested better results than the VGG19 network. For the VGG19 and VGG19_bn models, the final output must be changed from 1000 categories to the number of categories from the dataset. In this case, the output was changed to 17.

After preparing the model and hyperparameters, I created two for loops. The first for loop is for the desired number of epochs for the model to be trained, and the second for loop uses de data loader to load the data in batches for training purposes. This loop also includes the validation set, which helps us measure the loss changes for the model. I tracked the validation score with the BCEwithlogitloss criterion.

Once the model was trained and saved in the cloud, I proceeded to test the results in a predict file that uploaded the test set and predicted using the model created previously. I evaluated the results with the F-beta score from Sklearn to compare it with the Kaggle competition results. The F-beta score receives arrays of integer numbers, so the output of the network was transformed to zeros and ones depending on the probability (0 for $x<0.5$, 1 for $x>=0.5$). Finally, I saved the results for each run in a table.

RESULTS

Using the CustomDatasetFromImages, I successfully uploaded the images with shape (3,224,224) and the labels of shape (1,17) (an array of zeros and ones). The data loader divided the data into batches to train the model. I started using my model from the exam two results and registered all the performances. Below, you can find the description of each run I did for the CNN network built 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

After the seventh run we decided it was not useful to keep running the built from scratch Convolution Neural Network because no important changes in the accuracy score were obtained. Thus, I started implementing pretrained networks.

The first network I tested was the VGG19 network, which consists of a CNN of 19 layers provided by PyTorch. The VGG19 is proved to have an excellent performance in image classification. VGG19 architecture is shown below.

```
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) )
```

For the VGG19 run, I used a learning rate of 0.1, two epochs, a batch size of 90, a dropout of 0.1, and SGD as the optimizer. The training was for 120 min and the accuracy improved by 10%. The F-beta score obtained was 0.75.

The next run I did was with the VGG19 network using the same parameters as before but I included more epochs. It ran half epoch more before crashing, and the F-beta score for the testing set was 0.798.

I then proceeded to reduce the learning rate to 0.01 and run the model with four epochs. The F-beta score decreased to 0.759.

The documentation of pre-trained models from PyTorch showed better performance for the VGG19_bn (with Batch Normalization), so I proceeded to test it with the same parameters specified before but only one epoch and obtained a F-beta score of 0.8042 in 43 minutes.

I made a second run of VGG19_bn with the same hyperparameters except for the number of epochs that was changed to 2. The F-beta score improved to 0.836. Thus, the network selected to classify the Amazon satellite images was the VGG19 with batch normalization.

SUMMARY AND CONCLUSIONS

Deep learning problems are time-consuming and computationally expensive. With the advances of technology, image classification is no longer a problem. There are multiple methods and resources to improve the performance of a classification problem. Through this project, we faced the challenges of preparing data with the specific parameters of a network and a framework. I also learned how even when knowing how hyper-parameters and network architecture work, it is sometimes tough to improve the performance of a built from scratch network.

The existence of pre-trained models helps save time and computational power by accomplishing better results in a faster and more efficient way. Therefore, it is essential in deep learning to be continually updating your knowledge and to contribute towards the deep learning community to make work more efficient. There is no recipe on how a model architecture should be for a specific problem, but experience could help us improve in a more efficient way.

For this multilabel classification problem, I created a table to track the performance for each change I made to our network. It helped me understand what the next steps should be. Below is the summary table of results.

Run #	**Change**	**Fbeta score**
1	Initial run from Exam 2 results	0.650
2	LR	0.610
3	LR, Batch size and Dropout	0.680
4	Dropout and Added RandomHorizontalFlip	0.650
5	Network architecture	0.643
6	Network architecture and number of epochs	0.650
7	Pretrained network	0.752
8	Number of epochs	0.798
9	LR and Number of epochs	0.759
10	LR, Number of epochs, batch size and pretrained model	0.804
11	Number of epochs	0.836

The best f-beta score we obtained was 0.836 after using the pre-trained model VGG19_bn (with Batch normalization). Improvements could be made by adding more preprocessing, adjusting hyperparameters, trying different optimizers or even combinations of optimizers, using combinations of models, or training for a longer time the model.

From 185 lines of code, around 58 (31%) were copied exactly from the internet without any modification, and 127 (69%) were either modified by me or created from scratch. This calculation is hard because sometimes I used code from previous classes or projects, which I'm not sure if initially were taken from the internet. When I am coding, I intend not to copy any chunk of code without deeply understanding what it is doing.

REFERENCES

Jafari, A. (n.d.). Github Amir Jafari. Retrieved from https://github.com/amir-jafari/Deep-Learning

Leman, J. (n.d.). 4 Reasons Why We Need the Amazon Rainforest. Retrieved from https://www.popularmechanics.com/science/environment/a28910396/amazon-rainforest-importance/

Marcelino, P. (2018). Transfer learning from pre-trained models. Retrieved from https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751

Nooney, K. (2018). Deep dive into multi-label classification..! (With detailed Case Study). Retrieved from https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11

Prakash, J. (2017). Almost any Image Classification Problem using PyTorch. Retrieved from https://medium.com/@14prakash/almost-any-image-classification-problem-using-pytorch-i-am-in-love-with-pytorch-26c7aa979ec4

PyTorch. (2019). TORCHVISION.MODELS. Retrieved from https://pytorch.org/docs/stable/torchvision/models.html

Rosebrock, A. (2017). ImageNet: VGGNet, ResNet, Inception, and Xception with Keras. Retrieved from https://www.pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/

Shrimali, V. (2019). PyTorch for Beginners: Image Classification using Pre-trained models. Retrieved from https://www.learnopencv.com/pytorch-for-beginners-image-classification-using-pre-trained-models/

Tsang, S.-H. (2018). Review: VGGNet — 1st Runner-Up (Image Classification), Winner (Localization) in ILSVRC 2014. Retrieved from https://www.rainforestconcern.org/forest-facts/why-are-rainforests-important