Analyzing xView Dataset:

Group Project Report

Machine Learning II DATS 6203 - 11
Group 2: Jiarong Che, Diana Holcomb, Jiajun Wu

Introduction

In this project we built a deep convolutional neural network (CNN) based on PyTorch to detect objects in high-resolution satellite imagery. We also compared the difference between our model with the given Tensorflow baseline models (vanilla, multi-resolution, and multi-resolution augmented) that come with the xView dataset.

This topic is of interest to the team from a disaster relief perspective. We're also interested in using our new deep learning and PyTorch skills that we learned in class to analyze this dataset. Finally, the dataset is relatively recent, so there is a lot of unexplored territory and opportunity. These are the main motivation of our project.

Dataset

xView is the largest publicly available sets of overhead imagery to date, with over one million objects across 60 classes in over 1,400 square km of imagery. It contains images from complex scenes around the world, annotated with more than one million bounding boxes representing a diverse range of 60 object classes. Compared to other overhead imagery datasets, xView images are high-resolution, multi-spectral, and labeled with a greater variety of objects. The solution of xView data is 0.3 meters per pixel, that means this is the highest resolution that we can get from satellites currently.

One of the more interesting aspects of xView is that the dataset is so high quality that it may not extrapolate well onto real-world imagery of less quality. The RGB images provided have been ortho-rectified and processed for optimal color and contrast. The lack of cloud cover, the clear pictures, and angle of the photography are ideal for object detection and classification. However, models trained on this pristine dataset will therefore not handle significant cloud cover, blurry images, or images photographed at oblique angles well.

Sample Image 2355.tif

Data Analysis

Data Cleaning
As mentioned above, we lucked out with data cleaning. All of the xView images were selected because they are high quality images, and each image is guaranteed to have at least one tagged item in it. Unlike dealing with real-world imagery processing, we did not have to find images that were too cloudy, too blurry, photographed at extreme angles, or had no objects of interest within.

Data Preprocessing
Each 1 square km images is around 3,000 square pixels. Due to image size and object density, we encountered challenges when we tried to pass the entire image into a neural network. Of the 379 training images, only slightly more than 100 could be loaded before a memory error occurred. We also tried different sizes of chips. When we tried a smaller chip size, like in the MNIST or FashionMNIST, the amount of chipped images became enormous. Therefore, we decided to chip each image into 300x300 chips, which still resulted in 35858 chipped images. Some of the chips will be plotted. We can find which classes are present in the image and also visualize the chips with their labels. Additionally, we can shift and rotate the chips.

Sample 300 X 300 image chips of image 2355.tif

Loading the xView data into PyTorch's DataLoader was also an obstacle we had to overcome. Every PyTorch example we did in class had its own torchvision dataset for convenient use with a DataLoader. XView does not yet have a dataset. The xView website mentions that it has some PyTorch examples, but it seems those were taken down after the 2018 competition. Figuring out how to create a dataset object from scratch was challenging, the main challenge actually being in how PyTorch prefers its imagery formatting. NumPy and image-specific libraries such as PIL parse an image as (height, width, channels). PyTorch needs images to be formatted as (channels, height, width). Additionally, most online PyTorch examples used datasets that were already included in Torchvision, so examples of how to create our own were sparse and very problem-specific.

Another issue we came up against was converting our target array to a PyTorch tensor. All of the PyTorch examples we did in class had one annotation per image, i.e. this image is a shoe, or this image is the number 4. In the xView dataset, not only can many annotations occur in the same image (car, boat, building, etc.), there is also a variable amount of annotations per image - one image may contain 12 cars while another contains 3000 buildings. To solve this issue and get PyTorch to ingest our data, we had to create a custom collate function. We decided to pad every batch of targets to the length of the longest target array in the batch.

Evaluation Criteria
The interpolated mAP (mean average precision) is the primary metric for this project to measure the overlap between the true bounding boxes and our predicted bounding boxes. This method first compares the output bounding boxes with the groundtruth bounding boxes via the "Intersection over Union" method to find true positive, false positive, and false negative to calculate the precision/recall values. Then it finds the area under a smoothed precision/recall curve as the average precision per class (AP). Finally, it takes the mean of all the AP's to calculate the mAP. This method is often applied to computer vision problems because it takes both the bounding box *and* the label into account.

The xView dataset comes with three pretrained Tensorflow models, available here: https://github.com/DIUx-xView/baseline/releases

- Vanilla - the original out-of-the box model, all image chips 300x300
- MultiRes - model with variable image chip sizes (300x300, 400x400, 500x500)
- MultiRes_Aug - the MultiRes model, with additional augmentation in the form of

shifting, rotation, noise, and blurring

Vanilla Example MultiRes Aug Example

The xView team evaluated their pretrained Tensorflow models with a method known as mAP, or mean average precision, as mentioned above. The team was able to recreate these results with the help of xView's GitHub baseline and preprocessing projects.

MultiRes performed the best, with a mAP of 0.25. Vanilla performed the worst with a mAP of 0.14. MultiRes_Aug has a mAP of .15.

The best detected classes per model:

- Vanilla: passenger/cargo plane, small aircraft, building, passenger car, cargo/container car
- MultiRes: passenger/cargo plane, helicopter, shipping container lot, passenger car, building
- MultiRes_Aug: haul truck, passenger/cargo plane, small aircraft, building, tugboat

Key Findings

We learned several things about big data and the differences between PyTorch and TensorFlow on this project. First, we did not anticipate the big data being such a problem. Initially our dataset seemed small, with only ~1400 images included. The size of those images and the amount of chips we generated from them instead resulted in much more data than we anticipated. It was very tricky to avoid out-of-memory errors, and also our input and layer size calculations were much bigger than we had initially thought. It was very surprising to freeze up our GPC instances, since they can normally accommodate a lot of heavy processing.

We were initially drawn to PyTorch for three reasons: PyTorch is frequently faster than Tensorflow, PyTorch is easier to debug, and PyTorch's dynamic graph allows dynamic definitions that sounded conducive to a dataset with variable length targets. It seems that in the case of custom datasets with variable target array lengths per image, however, there are very few online resources to help with merely getting the data into a network. We were very surprised that we spent much more time on the data ingest pipeline than on the actual data training and testing.

Future Work

Given more time, the team has several ideas for future work. We'd like to create the equivalent of the MultiRes (varied image chip sizes) and MultiRes_Aug (shifting, rotation, noise, and blurring) in PyTorch and compare the results against the Tensorflow model. We'd also like to set up GridSearchCV and try various optimizers in PyTorch to see how accuracy and convergence time change. Currently we use Adam, but it would be interesting to compare with some of the more successful optimizers from our FashionMNIST project. Finding a better way to deal with the variable target array lengths than simple padding would hopefully improve model accuracy. Finally, we're interested in trying oversampling and undersampling. The distribution of the classes is not even, with many more building, cars, trucks, and buses than fixed-wing aircraft, tractors, and railway vehicles.

In class, Professor Jafari had mentioned that our issue is that our 60 classes have random numeric names instead of an ordered 0-59 list. This will require some heavy lifting of updating the ground-truth geoJSON target id's, but the team is excited to try to get past the loss backpropagation step.

Conclusion

Processing xView data in PyTorch proved to be much more challenging than our FashionMNIST assignment. There are a lot of multi-class per image and xView-specific examples in TensorFlow and Keras, but not a lot in PyTorch. Additionally, PyTorch doesn't easily accommodate variable length in the target arrays. We had hoped to do a comparison of Tensorflow and PyTorch over a large overhead imagery dataset, but got too bogged down in the engineering aspects. Our current challenge is properly calculating loss from a dataset with a variable length of target arrays - we get a "multi-target not supported" error, possibly based on how our class names are organized. The lesson we learned is that preprocessing and loading the data into a workable format for PyTorch takes much longer than we initially estimated.

References

xView Competition website: https://challenge.xviewdataset.org

xView dataset and ground truth geojson:
https://challenge.xviewdataset.org/download-links

Pretrained Tensorflow models: https://github.com/DIUx-xView/baseline/releases

xView Pre-Processing code: https://github.com/DIUx-xView/data_utilities

xView Baseline TensorFlow code: https://github.com/DIUx-xView/baseline

xView Baseline Docker container: https://hub.docker.com/r/xview2018/baseline/tags

Code

https://github.com/catsbergers/Final-Project-Group-2