Final Report Group 7

Project Team Members: Hao Heng, Junhe Zhang, Ze Gong

Introduction

In this project we will:

- We will download the dataset from Google Open Image dataset with certain categories we choose to train the model and generate all needed files for training yolov3 model on darknet framework.
- We will use the Darknet frame and pre-trained weight of the yolov3 model to train on our own dataset and evaluate the model.
- At last we will use our model on a video we recorded and a image we took to see the actual performance of the model

Keywords

- **Data Exploration**
  - Dataset creation
  - Detection
  - Evaluation

- **Model Application for Object Detection**
  - YOLO(early stage)
  - Pretrained Model(early stage)

- **Software Implementation Frameworks**
  - Numpy
  - Pandas
  - OpenCV
  - Darknet
  - OS

Data Exploration - Hao Heng

1. Download dataset

- In this project, we will download the dataset from google open image dataset. The categories we choose include “Car, Bicycle wheel, Bus, Jeans, Laptop, Person, Traffic light”. Each category contains 1000 images.
- We use the OIDv4_ToolKit to download directly to our cloud instance. The link of github of this toolkit: https://github.com/EscVM/OIDv4_ToolKit.git

  The command line is shown as below:

  python3 main.py downloader
  --classes Car Bicycle_wheel Bus Traffic_light Jeans Laptop Person
  --type_csv train
  -- multiclasses 1
  --limit 1000

  All this command should be in one line during typing. The “classes” should include all category names that you want to train on. If there are two words in this category such as “Bicycle wheel”, you need to add an underline between “Bicycle” and “wheel”. The “multiclasses” are set to 1 which means there is only one class in every image. The “limit” implies the number of instances of each category.

  Once the command is executed successfully, it will generate the folder which has two sub directories. One of these two contains all the images, the other contains two csv files that store corresponding classes information.

2. Prepare needed file for training

- Dataset folder

  First, in the folder storing the dataset, each image must have a txt file that has the same name of the image file. The txt file for each image should contain five parameters: class number, center coordinate x, center coordinate y, width of the bounding box, height of the bounding box.

- data.data file

  This file only contains five fixed lines. It should be looked like this:

  classes=the number of classes in the dataset

train=full path to train.txt file
valid=full path to test.txt file
names=full path to classes.names file
backup=backup

● classes.names file

This file contains all the names of categories in the dataset, one category for each line.

● train.txt

This file contains all the full path of images that are used for training the model. Like classes.names file, one line for one path.

● test.txt

Likewise, this file contains all the full path of images that are used for testing the model. Like classes.names file, one line for one path.

3. Preparation

● Converting the annotation

The important part during data preparation is how to convert the data format of Google Open Image into Yolov3 format. After downloading the dataset from Google Open Image dataset, besides the images, you will also get two csv files. 'class-descriptions-boxable.csv' contains the class names and corresponding encrypted string for each class.

'train-annotations-bbox.csv' contains all the annotation of each image, including image ID, class of the object in this image in encrypted form, and most important, the bounding box of that object. The bounding box is in "XMin, XMax, YMin, YMax" form.

First, we need to get all encrypted strings from 'class-descriptions-boxable.csv' for all classes we chose for our training.

Second, we need to filter out unnecessary rows in 'train-annotations-bbox.csv' based on the encrypted string we just got and calculate the bounding box of Yolov3 format. The algorithm is shown as below:

center x= (XMax+XMin)/2
center y=(YMax+YMin)/2
width=XMax-XMin
height=YMax-YMin

Third, we extract all the rows according to the image name that we downloaded, then we write the "class number, center x, center y, width, height" to txt file for each image

file in our own dataset. One line for each criteria in each txt file.

● Generating train.txt and test.txt

Now we have all the images followed by a .txt file that describes its class number and coordinate of the bounding box of the classes the image contains, then we need to read the full path of all images and split them for training and testing in 0.85 ratio and save them into train.txt and test.txt. Each path of images is in a new line.

● Generating data.data and classes.names

In this step, first we need to manually create a .txt file named as classes.txt to store all class names in our dataset. Each name of classes is in a new line.

Then we convert the classes.txt file into .names file by writing classes.names line by line.

At last, we write the data.data file line by line, and the content we already have listed in the previous description.

Now, we have our dataset ready for training.

Training YOLO in Darknet framework - Ze Gong

1. Setting up the configuration

- To train YOLO, the first step is setting up the configuration. We have 7 classes to identify and the filters for coco dataset is 36. The max_batches is the total number of iterations and when it comes to steps, the learning rate becomes 10 percent of before respectively. Batch is the number of samples that will be processed in one batch. Subdivisions is the number of mini batches in one batch.

  Classes = 7
  filters = (7 + 5) x 3 = 36
  max_batches = 7 x 2000 = 14000
  steps = 0.8 x max_batches / 0.9 x max_batches
  = 11200 / 12600
  batch = 32
  subdivisions = 8
  minibatch = batch/subdivisions = 4

- The minibatches here are determined by batch divided by subdivisions. GPU processes minibatch samples at once. The weights will be updated for batch samples, that is 1 iteration processes batch images.

2. Installing Darknet

- In most of the cases the general approach is as following or quite similar:

a. Install prerequisites (OpenCV, CUDA, cuDNN, etc.)
b. Clone repository

c. Adjust options in Makefile (for Linux & Mac)
d. Compile it (for Linux & Mac) or Build it (for Windows)

3. Preparing files for training

- In order to train on datasets, we need to copy data files (custom_data.data and ts_data.data) into the existing cfg folder inside the root Darknet directory.

4. Running training process

- To start training process in Darknet framework, navigate to the directory with executable file and type in Terminal or command line specific command as described below.

  /darknet detector train cfg/custom_data.data cfg/yolov3_custom_train.cfg weights/darknet53.conv.74

- Continue training with saved weights after 1000 iterations

It is possible to stop training, for example, after 1000 iteration and continue later by using already saved weights. In order to continue training just specify at the end of command location of needed weights to continue training from.

5. Find best weights to stop training

- Recommended number of iterations for training is defined by using following equation.

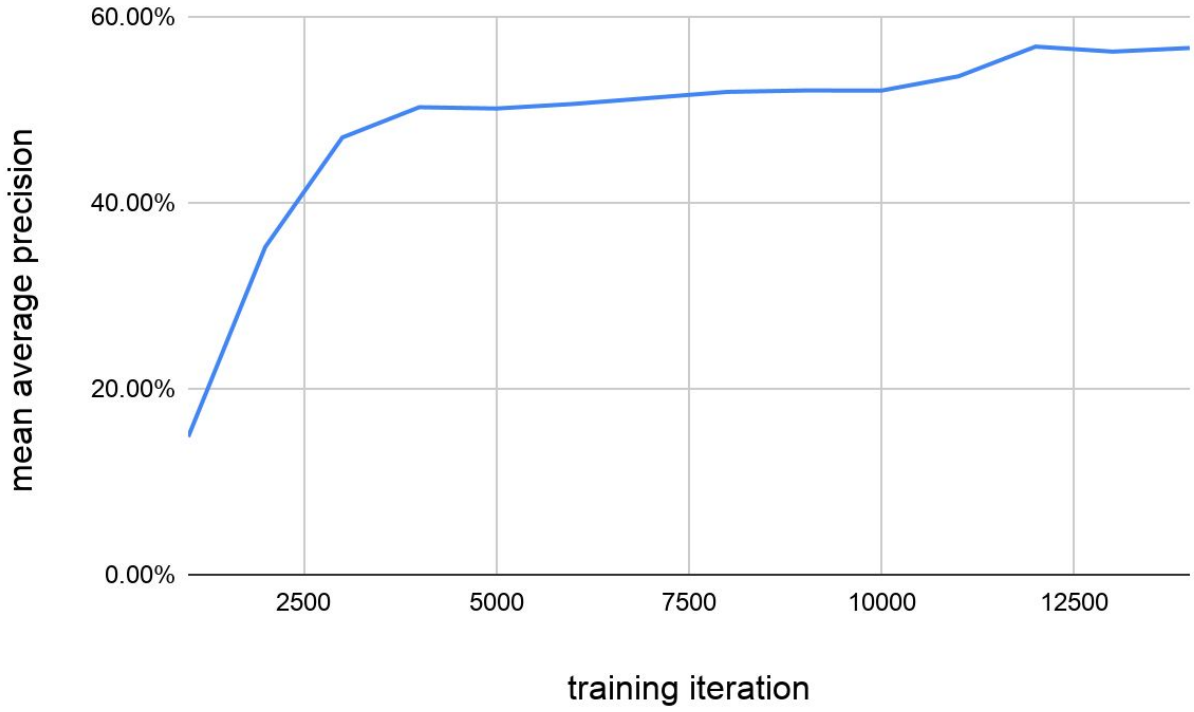
  max_batches = classes * 2000
  (but not less than 4000 in total)

  It means that every class should have around 2000 iterations. For example, in our custom dataset of 7 classes, the recommended total number of iterations for training on this particular dataset will be 14000.

- To define the best weights after training. Our task now is to define the best one to use for detection making sure that there is no overfitting. Weights will be saved every 1000 iterations.

- Algorithm is as following: start checking saved and trained weights from the end one by one calculating mean average precision (mAP). The goal is to find weights that have the biggest mAP. We can use these codes in the terminal to calculate the mAP.

  ./darknet detector map cfg/custom_data.data cfg/yolov3-custom_train.cfg backup/yolov3-custom_train_8000.weights



The best mAP is 56.82%, which appears at iteration 12000.

YOLO Model Testing - Junhe Zhang

- After the 14000 iteration has finished, it is time to test our model. Since YOLO can be applied to both images and videos, we decide to pass in a couple images and videos with our interested objects inside that we take from the life around us and check the results. The algorithm is as following:

    1. Read the input image and convert it into a blob object which is accepted by the YOLO framework.
    2. Load the best trained weights and the network structure file into the YOLO framework.
    3. Process blob image file into the model and generate output bounding boxes for each object inside the image.
    4. Using Non-Maximun_Suppression technique on the output bounding boxes to drop duplicated boxes on the same object with lower confidence.

    1. The first step is to read the testing image or video using openCV library. The following code will read the absolute path of an image and generate an openCV BGR image object. After this we have to convert the BGR image into an RGB blob object so that YOLO network can process it. It is also important to keep in mind that YOLO framework only accepts image shapes which can be by 32. So I reshaped the image to 416 x 416 for the best performance purpose. To speed up the object detection process it is also important to normalize the import by '1/255.0', since all pixel values are in the range between 0-255.

```
def read_image(img_dir, video = 0):
```

```
    # img should be absolute directory of image
    # cv2 image has format blue green red
    if not video:
        image_BGR = cv2.imread(img_dir)
    else:
        image_BGR = img_dir
    # getting height and width of image
    h, w = image_BGR.shape[:2] # first two value contains height and width
    blob = cv2.dnn.blobFromImage(image_BGR, 1 / 255.0, (416, 416),
                                 swapRB=True, crop=False)
    # reshape blob from (1, 3, 416, 416 ) to (416, 416, 3)
    blob_clean = blob[0, :, :, :].transpose(1, 2, 0)
    return (image_BGR, blob, w, h)
```

2. After cleaning the input data, it is time to load our best trained model. The best trained model was evaluated by Mean Average Precision (MAP), we choose the weight with the highest MAP as our best weight. With our network structure configuration file (.cfg) we can easily load our model using the openCV library.

```
# load and return trained YOLO model
def load_model(cfg_dir, weights_dir):
    return cv2.dnn.readNetFromDarknet(cfg_dir, weights_dir)
```

3. It is time to process the testing data into our model. The YOLO network structure we applied has standard 106 layers, which has layer 82, 94, 106 as its output layers. This indicates that there will be three outputs for each image during the forward processing. We need to first collect all the output bounding boxes and labels from all three output layers and then later filter out the less valuable ones.

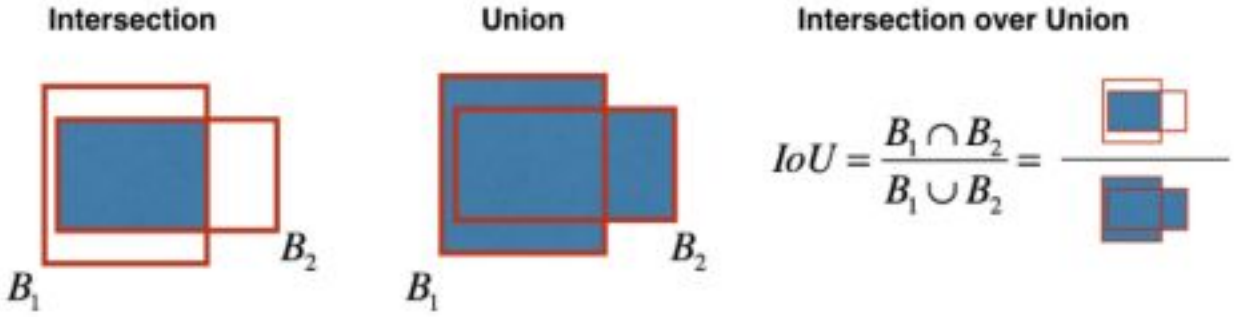
```
# process image on trained model
# return list of bounding box of detected objects
# img has to be converted to blob beforehand
def process_image(model, output_layers, img):
    print('Start process image')
    start = time.time()
    model.setInput(img)
    end = time.time()
    print('End process image, processing time {:.5f}'.format(end-start))
    return model.forward(output_layers)
```

4. After all the outputs were collected, we need to apply Non_Maximum_Suppression technique on these results, so that only the most important and non-duplicate bounding boxes will remain.

```
# remove multiple bounding boxes on the same object with lower confidence
def non_maximun_suppression(bounding_boxes, confidences, class_numbers,
probability_minimum, threshold):
    return cv2.dnn.NMSBoxes(bounding_boxes, confidences,\
                            probability_minimum, threshold)
```

Non_Maximum_Suppression Example:

Non_Maximum_Suppression Equation:



Display the result:

The images below demonstrate what the original image and its corresponding result image look like. The Jeans have a confidence rate of 62% and laptops have a confidence rate of 86%. To do object detection on videos, it is as simple as splitting input video into continuous frames and treating each frame as image. Then process each image with the steps that demonstrated in 1 - 4. The only difference is to create an openCV writer to save each modified frame (with bounding boxes and confidence rate drawed on the frame) back into video format (.avi).

Example Processed Image:

Links of Classified Videos:

- https://drive.google.com/file/d/1xc2RdESogHPag3VwwnbuNQs-RX--mZzF/view?usp=sharing
- https://drive.google.com/file/d/1luAp5ggEG8hvPBlzG8yjvvBXh1PIXClW/view?usp=sharing

Create openCV Video Writer:

```
220         if writer is None:
221             # Constructing code of the codec
222             # to be used in the function VideoWriter
223             fourcc = cv2.VideoWriter_fourcc(*'mp4v')
224 
225             # Writing current processed frame into the video file
226             writer = cv2.VideoWriter(video_dir+'result-'+video_name+'.avi', fourcc, 30,
227                                      (img.shape[1], img.shape[0]), True)
228         writer.write(img)
```

Final Result

The best model we got after the 14,000 iteration was acceptable, which could successfully detect all seven classes that we are interested in. However, it is not comparable to the official COCO dataset model provided by YOLO website, which can classify over 80 different classes with MAP above 90%. There are a couple classe that did not class well during the training process, for example: Person, Jeans, etc…. The reasons that cause this problem might be because the training dataset for Person and Jeans are overlapping. For instance, people wearing a pair of jeans only get labeled as either people or jeans. Another improvement could be increasing the size of the dataset and increasing the number of iterations.

```
class_id = 0, name = Car, ap = 51.01%           (TP = 308, FP = 165)
class_id = 1, name = Bicycle wheel, ap = 62.46%         (TP = 304, FP = 68)
class_id = 2, name = Bus, ap = 79.25%           (TP = 161, FP = 35)
class_id = 3, name = Traffic light, ap = 44.14%         (TP = 329, FP = 222)
class_id = 4, name = Jeans, ap = 55.24%         (TP = 183, FP = 64)
class_id = 5, name = Laptop, ap = 80.80%        (TP = 155, FP = 28)
class_id = 6, name = Person, ap = 23.76%        (TP = 230, FP = 308)

 for conf_thresh = 0.25, precision = 0.65, recall = 0.45, F1-score = 0.53
 for conf_thresh = 0.25, TP = 1670, FP = 890, FN = 2047, average IoU = 50.15 %

 IoU threshold = 50 %, used Area-Under-Curve for each unique Recall
 mean average precision (mAP@0.50) = 0.566669, or 56.67 %
```

Conclusion

Overall, our model still detects certain classes in images or videos, even though it performs bad at a couple of classes. But there is still a lot of space left to improve the model. In the future, we will plan to train on a bigger dataset and increase the number of iteration numbers.

References

[1]Joseph Redmon, Santosh Divvala, Ross Girshick, ALi Farhadi "You Only Look Once: Unified, Real-Time Object Detection", IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[2]Open Images Dataset V6 + Extensions. (n.d.). Retrieved December 08, 2020, from https://storage.googleapis.com/openimages/web/index.html

[3]Joseph R. (2013-2016). Darknet: Open Source Neural Networks in C. Retrieved from

http://pjreddie.com/darknet/