

## **Advanced Machine Learning Final Project**

### **Region Convolutional Neural Network**

*Faster RCNN Implementation with Proposal Network Backed by Inception V3,  
Inception V4, VGG16, Resnet50*

Team Members:	Zehao Dong	zd2188
	Zishuo Li	zl2528
	Hongyang Yang	hy2500
	Yanbing Liu	yl3519

## Abstract

In this project, we explored the performance of faster RCNN, construct faster RCNN with proposal network backed by a pre-trained inception classifier *Inception V4*, *Inception V3* on Keras, and simplified faster RCNN with *VGG16*, *Resnet 50* based on Keras, and applied the network models on Pascal VOC 2007 and 2012 dataset. Given the limitation of time, we utilized the approximate joint training method and trained VGG16 with 800~1000 epoch, Resnet 50 with 20~30 epoch, Inception V4 with 20 epoch on Tensorflow with GPU. We will continue training on GPU with several days, and will update the result (pickle file, more images labels, etc) on our github: [https://github.com/YvonnaCU/AML\\_Final\\_Project-5.git](https://github.com/YvonnaCU/AML_Final_Project-5.git). Please refer to this link with the most recent result.

### Keywords:

Models: Inception-V4, Resenet-50, VGG16, RPN, Faster-RCNN, Fast-RCNN

Environment: Google Cloud, GPU (K80, P100), Spark, Hadoop Cluster, MongoDB

## Introduction

### 👑 Existing Methods:

There are many useful methodologies of object detection, one of the breakthrough is RCNN. Based on the Region Proposal, RCNN fulfills the object detection using selective search for the feature extraction from CNN and SVM classification. However, RCNN request a large memory from the disk for the pre-extraction of multiple region proposal. Since the traditional CNN ask the fixed input image size, the image crop or warp process can cause the anamorphose and loss some information. What's more, the CNN compute every proposal region which has thousands of overlap, increasing computation time.

SPP-Net solve the problem of time consumption in CNN on overlapped proposal region by replacing the last fully connected layer in RCNN with the Spatial Pyramid Pooling. Unfortunately, all the training processes are still isolated, leading to the demand of the temporary memory for those intermediate processes. Furthermore, SPP-Net can not tune the two side convolution layers beside SPP-Layer and the fully connected layer simultaneously. This will limited the performance of deep CNN.

Fast RCNN accelerate the RCNN and solve the problem of the isolated training process in SPP-Net. According to the SPP-Net, fast RCNN propose a simplified ROI pooling layer instead of the Pyramid pooling, and add a mapping from the proposal region, enabling the whole network propagate backwards. What we should notice is that fast RCNN combine the classifier and bounding box regression with multi-task loss layer, that means replacing the SVM in the

classifier layer and Bounding Box with Softmax Loss and Smooth L1 Loss separately. Nevertheless, the region proposal process in fast RCNN is still time consuming.

### **Our Improvement:**

#### **Code simplicity & explainability:**

Simplified and reconstructed Faster Rcn with VGG16 and Resnet 50: We used brief structure to construct RPN and Classifier in Faster RCNN model, and provided detailed structure and explanation for method to construct VGG16 and Resnet 50 in Faster RCNN. Our original code make it easier to understand & replicate in keras.

#### **Accessibility & reproducibility:**

We provided the links to download our data (original & processed), code (easy to understand & replicate) and the link to our Google Cloud GPU instance.

#### **Proposed feature extraction net of Inception-V4:**

We constructed new raw feature extraction net: Inception V4 and compared performance of Inception V4 with Vgg16 & resnet50. Inception network has awesome performance in computer vision field and the features generated by Inception network could have good effect in Faster RCNN, so we proposed Inception-V4 as new feature extraction net.

#### **Computation power:**

The project is based on Google Cloud Linux VM. We deployed Spark Hadoop Cluster to improve the data process speed, Mongoddb for data storage & interaction with Python and GPU to increase computational power. Our project is highly-scalable industry level project with strong computation power.

## **Data Preparation**

### **Dataset Information:**

We have two datasets: Pascal VOC & MS COCO

#### **1. Pascal VOC 2007 and VOC 2012:**

The reason why we use both VOC 2007 and 2012 is that VOC 2007 include the test dataset while 2012 doesn't, and the test dataset will help improve the accuracy of the training result. The data pool has 24000 data, and we shuffle the data to have 22,000 training samples, 2,000 test samples. The total size of the data is around 3 GB.

Original Link: <http://host.robots.ox.ac.uk/pascal/VOC/>

## 2. MS COCO:

MS COCO has larger & more object classes, we use the MS 2017 dataset here, the training dataset has around 120,000 has test dataset has around 40,000 samples. The total size of data is around 24 GB

Original Link: <http://cocodataset.org/#home>

## 👑 Data Preprocess:

For data preprocess, we mainly use three methods to preprocess data:

a. **Python iterator**, b. **Spark Cluster Preprocess** & c. **Mongodb Storage**

### 1. Pascal VOC:

Since this data is around 3 GB, considering computational efficiency & memory limitation, we preprocess all data to generate three files: list of hash tables (Provide detail information including file path, bbox size, width & length), hash tables of object classes & object statistics. Later, we use python iterator to yield samples during each epoch.


Preprocessed data link:

<https://storage.googleapis.com/amlgpu/keras-frcnn-master.zip>

### 2. MS COCO:

Since the data is extremely large, python memory couldn't handle the same way as Pascal VOC, so we deployed Spark cluster on Google Cloud (1 master node & 5 worker node). We upload the file to HDFS & use pyspark to calculate the necessary information the same as above. Later for convenience of calling data & interaction with python, we use mongodb to store the data processed from Spark.

Spark Cluster screenshot:

 Compute Engine

VM instances

Instance groups

Instance templates

Disks

Snapshots

Images

Committed use discounts

Metadata

Health checks

VM instances








CREATE INSTANCE

IMPORT VM

REFRESH

START

Filter VM instances

<input type="checkbox"/> Name ^	Zone	Recommendation	Internal IP	External IP	Connect
<input type="checkbox"/>  rcnn	us-west1-b		10.138.0.2	35.227.148.57 	SSH ^ ⋮
<input type="checkbox"/>  spark-m	us-east1-c		10.142.0.4	104.196.45.103	SSH ^ ⋮
<input type="checkbox"/>  spark-w-0	us-east1-c		10.142.0.2	None	SSH ^ ⋮
<input type="checkbox"/>  spark-w-1	us-east1-c		10.142.0.3	None	SSH ^ ⋮
<input type="checkbox"/>  spark-w-2	us-east1-c		10.142.0.6	None	SSH ^ ⋮
<input type="checkbox"/>  spark-w-3	us-east1-c		10.142.0.5	None	SSH ^ ⋮

```
[I 22:47:11.900 NotebookApp] Kernel started: 05ec296f-952e-44a8-8471-8cebd3e8846f
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
[Stage 0:=> (1 + 9) / 50] [I 22:49:12.884 NotebookApp] Saving
file at /r_cnn_data_process.ipynb
[Stage 0:=====> (16 + 9) / 50] [I 22:51:11.935 NotebookApp] Saving
file at /r_cnn_data_process.ipynb
[Stage 0:=====> (27 + 9) / 50] [I 22:53:11.963 NotebookApp] Saving
file at /r_cnn_data_process.ipynb
[Stage 0:=====> (42 + 8) / 50]
```

HDFS screenshot:

```
liz003@spark-m:~/Notebooks$ hadoop fs -ls
17/12/18 20:23:31 INFO gcs.GoogleHadoopFileSystemBase: GHFS version: 1.6.1-hadoop2
Found 6 items
drwxr-xr-x - liz003 hadoop 0 2017-12-15 00:15 .sparkStaging
drwxr-xr-x - liz003 hadoop 0 2017-12-09 00:51 eg
-rw-r--r-- 2 liz003 hadoop 58921352 2017-12-15 00:10 model.h5
drwxr-xr-x - liz003 hadoop 0 2017-12-08 23:45 test2017
drwxr-xr-x - liz003 hadoop 0 2017-12-15 00:35 test_set_output
drwxr-xr-x - liz003 hadoop 0 2017-12-15 02:47 train2017
```

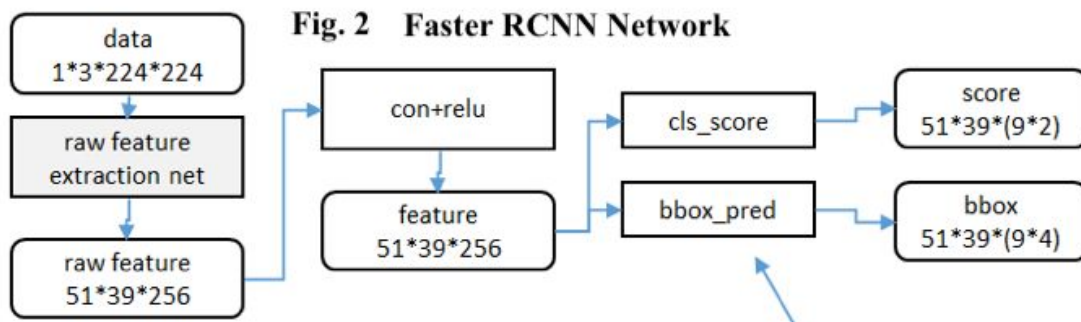
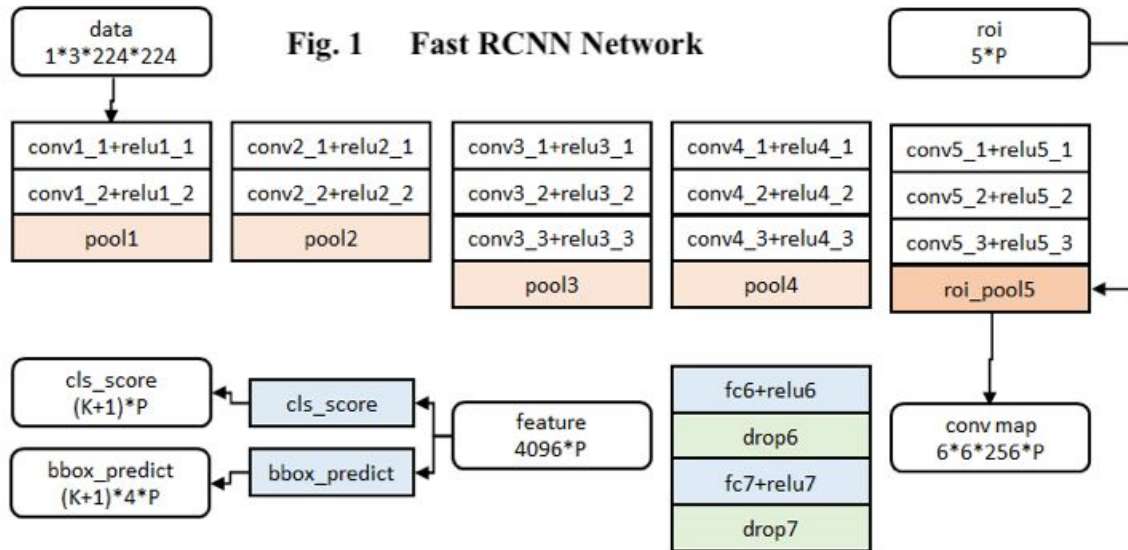
Spark processed data link:

[https://storage.googleapis.com/amlgpu/test\\_set\\_output/part-00000](https://storage.googleapis.com/amlgpu/test_set_output/part-00000)

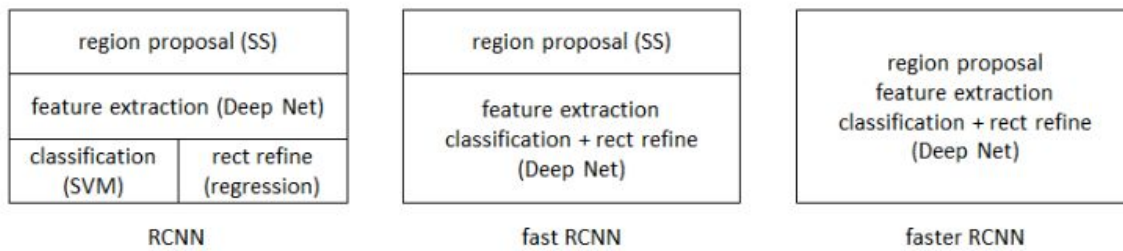
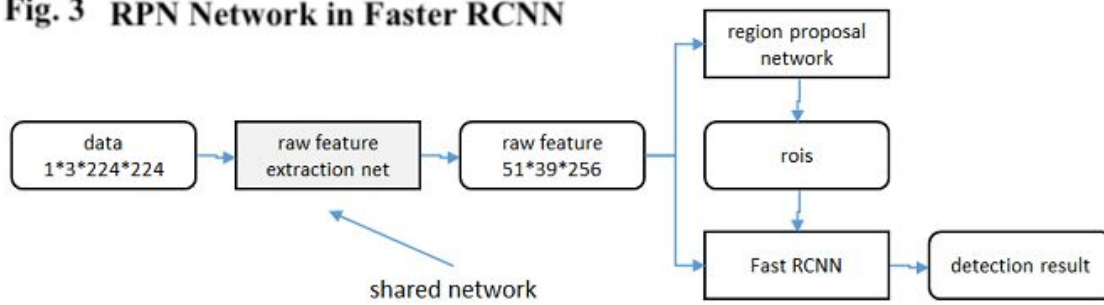
(Totally 50 parts, plz change the last part-00000 to part-00050 to get all parts)

## Model Exploration

### 👑 Fast RCNN vs Faster RCNN:



Consider 9 'anchors' on each of the  $51 \times 39$  positions

**Fig. 3 RPN Network in Faster RCNN****Fig. 4 The Difference between RCNN, Fast RCNN and Faster RCNN**

The above figures show the difference between fast RCNN and faster RCNN whole network. The following table describe the difference in detail:

Procedures	Fast RCNN	Faster RCNN
original image --> feature map	Conv + pooling	shared network (VGG, Resnet, etc)
feature map --> Roi	SPP-Net get external proposal region	RPN
Roi --> fc	combine feature and external proposal --> Roi pooling --> fc	combine feature and internal proposal --> Roi pooling --> fc
fc --> softmax & bbox regression	output	output

## 👑 Faster RCNN Detail:

### 1. Convolutional layers:

In this feature extract layer, we employ the VGG16, Resnet 50, Inception V3, Inception V4 as the basic convolutional + resolutional + pooling process. The feature map generated from these processes is used as the shared feature for RPN layer and the fully connected layer.

#### (1) VGG16:

In VGG16, what we should pay attention to is that all the convolutional layers is 3x3 kernel size with 1 padding, and all the pooling layers is 2x2 kernel size with 2 stride. Since we have four conv+relu+pooling processes, the output size from every convolutional layer is the same with its input size, and only the pooling layer turn the height and width of the input image into half of its original size in the output. After four such processes, the height and width become 1/16 of its original size (note:  $1/2^4$ ). We have totally five conv+relu process, but it does not change the image size. Thus, the convolutional layer will give a size fixed output, and this will help us connect the feature map with original image. Details can be found in [vgg16\\_model\\_present.html](#) (cell 11)

#### (2) Resnet 50:

Replacing VGG16 with resnet50 in raw feature extraction net has been proved with high potential to improve performance in the field of detection, segmentation, video analysis and recognition. As such, we find it a wise practice to make this replacement. To make this faster rcnn work, output shape of Resnet50 should be figured out. Based on the structure in the Resnet50, (2,2) strides and “valid” padding method, we give details about how to get the right output length and output width in [Resnet50\\_model\\_present.html](#) (cell 9).

#### (3) Inception V4:

Before the breakthrough with advent of GoogLeNet, networks with deeper structure continuously attracted attention. Thus, we believe it is reasonable to use inception net in the faster rcnn structure. We choose inception v4 and inception v3 to make experiments. As before, the dimension of output from this inception net has a great influence on the success of faster rcnn model. We present details about this question and corresponding classifier function in [InceptionV4\\_model\\_present.html](#) (cell 15 and cell 13)

#### (4) Inception V3:

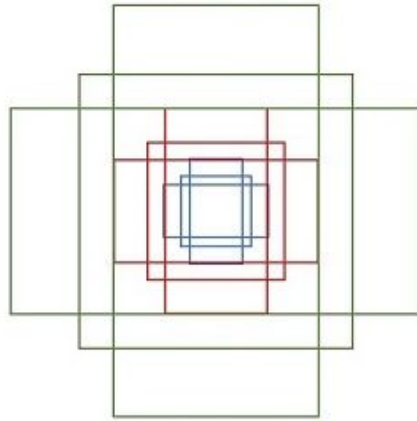
Compared with inception v4, inception v3 has a much deeper structure, thus we want to explore the performance. Based on the experience we have when we construct VGG16, Resnet50 and Inception V4, we made a implementation of this net based on the analysis as before. Details are in [InceptionV3\\_model\\_present.html](#)



## 2. Anchor:

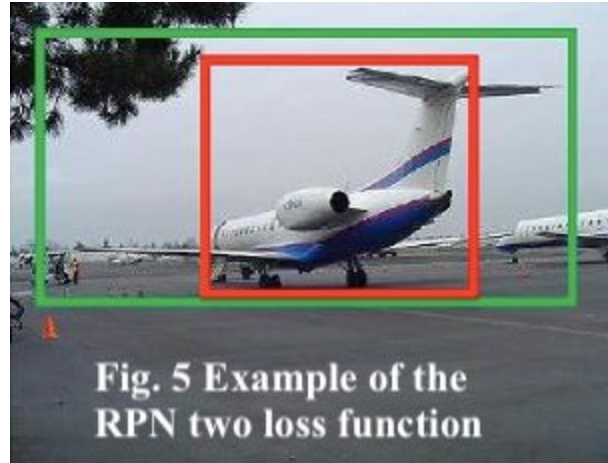
Faster RCNN choose 9 different size anchors applied on each sliding window in feature map. They are composed with 3 different areas with  $128^2$ ,  $256^2$ ,  $512^2$  and 3 different width and height ratio 1:1, 1:2, 2:1. Shown in Fig. 6. With those anchors, when traversing all the feature maps, each point will return 9 different original proposal region. Although such proposal is not accurate, there are two correction process in RPN with loss function, that means the original proposal region will update twice before ending the proposal selection process.

**Fig. 6 The Anchors with 9 different size**



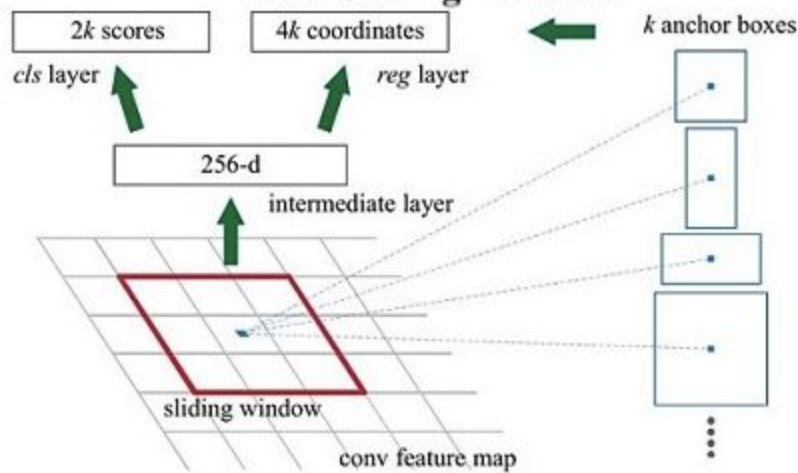
## 3. RPN (Region Proposal Network):

In the RPN, there are actually two processes happened simultaneously in the correct process. One of them is to calculate the loss function of softmax between foreground and background(a.k.a ground truth) IoU. The other one is to calculate the smooth L1 loss function of bounding box regression between the foreground and background bounding box "location". The sum of the two loss function can balance the feature proposal region close to the real object. For example, as shown in the Fig. 5, the green box is the background, the red box is the foreground. Sometimes in this situation, the IoU will give the foreground a high probability and the softmax loss will be low, however, if only consider the loss function, the object detection result in the later process will give a wrong label. But when we come to the smooth L1 loss, the distance between the two boxes will increase the smooth L1 loss and therefore increase the the final loss. Thus, RPN will force the foreground and background box to optimize their location to give a lower loss and a reasonable proposal region. At the same time, it will pick off those proposals with extremely small size and over bounding size.



For more detailed demonstration, as shown in Fig. 7 in ZF model, if apply  $k$  anchors for each  $3 \times 3$  sliding window, in the classification layer each anchor will generate foreground and background score separately, which return  $2k$  scores, in regression layer each anchor will generate four score for each position of the proposal box, which return  $4k$  coordinates.

**Fig. 7 The Output Dimension from Each Sliding Window**



#### 4. Roi Pooling:

The input in Roi pooling layer are raw feature maps and internal proposal region with different size. Same reason with fast RCNN in this layer, the Roi pooling is to avoid the local variance caused by cropping and wrapping for different input size to next convolutional layer.

#### 5. Classification:

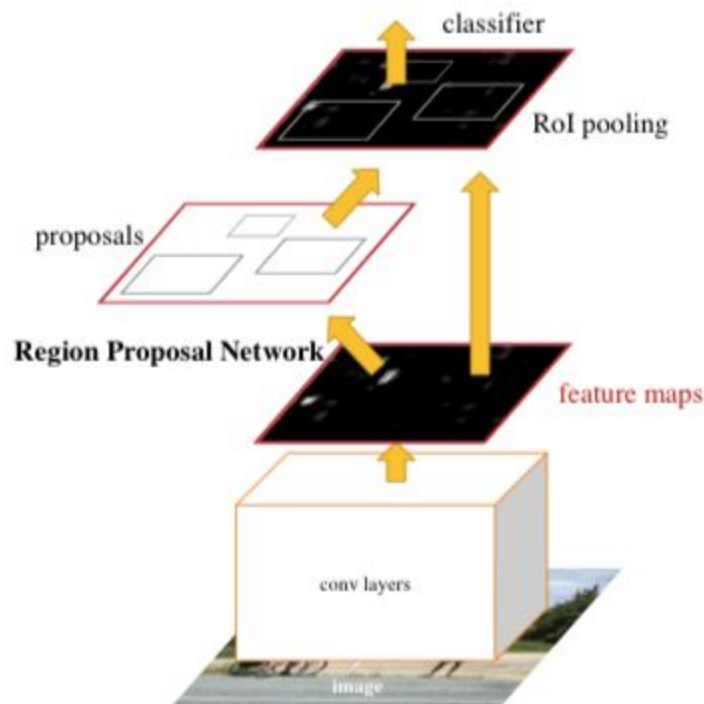
After fully connected layer, we obtain the proposal feature map, and classification layer is to calculate the softmax classification loss and bounding box regression loss again to update the object proposal region and to return the class of this object and a probability of this classification.

## Training Procedure

Faster RCNN is trained on the pre-trained model (VGG 16, Resnet 50, Inception V4). There are three different training methods on paper, alternating training, approximate joint training and non-approximate training. We utilize the approximate joint training in our network.

Different with alternating training, approximate joint training doesn't train the RPN and fast RCNN interactively, instead, it trains the RPN and fast RCNN as a whole network, as shown in Fig. 8. The thumb reason for the approximate joint training is the shorter training time than the alternating training. Since the approximate joint training only compute the Stochastic Gradient Descent(SGD) on the softmax loss function part and ignore the Smooth L1 loss function part, the training process update convolutional layer backwards with less parameters. And that's why it called "approximate". To be more concrete, for each SGD process, it will calculate  $6N$  parameters ( $2N$  for two softmax loss function and  $4N$  for four smooth L1 loss function), but in approximate joint training, it only calculate  $2N$  parameters each time, and this will vanish  $2/3$  percent computation complexity, which can be used to estimate the training time with around 60% less.

**Fig. 8 Approximate Joint Training**



**GPU demonstration:**

For GPU, we deployed two kinds of GPU on Google Cloud including Tesla K80 & P100. The screenshot of K80 and links to two instances are listed below.

Instance1 link :<https://35.227.148.57:5000/> password: (Plz send email for access)

Instance2 link:<https://35.196.117.225:5000/> password: (Plz send email for access)

Screenshot:

```
Mon Dec 18 17:40:47 2017
```

NVIDIA-SMI 387.26				Driver Version: 387.26			
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr. ECC	
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	
0	Tesla K80	Off	00000000:00:04.0	Off		0	
N/A	48C	P8	30W / 149W	16MiB / 11439MiB	0%	Default	

Processes:				GPU Memory	
GPU	PID	Type	Process name	Usage	
0	2041	G	/usr/lib/xorg/Xorg	15MiB	

## Performance Analysis

### 👑 Model Description:

#### 1.Capability and Principle of model:

- (1). In the training process, we use a predefined class, Config, to store all information in the Faster RCNN model, and then save it as a pickle file. In addition, we store weights from training process of the model in a h5py file. For more details,
- (2). In the test process, we load corresponding pickle file to build a same model as information stored in the Config object. Then loads weights stored in the test process to detect object in the unlabeled(without target label and bbox data) dataset and make the visualization.
- (3) The model result are stored in the pickle files and h5py files.
- (4) Our Faster RCNN model is interactive with detailed description, which makes it more available and accessible. [model details are in sharedCNN\_model folder]

#### 2. Work accomplished:

- (1). Now, we have successfully built Faster RCNN model based on VGG16, Resnet50, and Inception V4. We also build another Faster RCNN model with Inception V3 net, which is still training now. Training Pascal Voc 2007 and Pascal Voc 2012 in a Faster rcnn model is time consuming(one epoch needs about 30 minutes with NVIDIA K80 GPU), thus we trained VGG16 with 800~1000 epoches, Resnet 50 with 20~30 epoches, Inception V4 with 20 epoches

on Tensorflow with GPU due to the limitation of time. Based on experience and information we collect, we believe that a well-trained model at least need 1000~2000 epoches during the training process.

(2). We provide one sample of training process (Frcnn\_train.html) and two samples of testing process(*Test\_case\_inception.html* & *Test\_case\_vgg.html*).

## 👑 Speed Analysis:

### 1. Training speed estimation:

We analyze mean training time during 50 to 100 batches in the 1 epoch to indicate the average performance on training speed, Because first 50 batches are not stable.

#### (1) VGG16: 1 min07s (for 50 batches)

```
vgg
Epoch 1/5
 1/1000 [.....] - ETA: 2:20:29 - rpn_cls: 6.5692 - rpn_regr: 0.0691 - detector_cls: 3.0445
- detector_regr: 0.3163

vgg
Epoch 1/5
 50/1000 [>.....] - ETA: 44:38 - rpn_cls: 4.7711 - rpn_regr: 0.1525 - detector_cls: 2.1983 -
detector_regr: 0.4464

vgg
Epoch 1/5
100/1000 [==>.....] - ETA: 43:31 - rpn_cls: 5.0864 - rpn_regr: 0.1872 - detector_cls: 2.0293 -
detector_regr: 0.3448
```

#### (2) Resnet50: 4min47s (for 50 batches)

```
resnet50
Epoch 1/5
 1/1000 [.....] - ETA: 4:15:16 - rpn_cls: 10.2358 - rpn_regr: 0.0214 - detector_cls: 3.044
5 - detector_regr: 0.4818

resnet50
Epoch 1/5
 51/1000 [>.....] - ETA: 28:50 - rpn_cls: 5.3656 - rpn_regr: 0.2366 - detector_cls: 2.1729 -
detector_regr: 0.5535

resnet50
Epoch 1/5
100/1000 [==>.....] - ETA: 24:03 - rpn_cls: 4.5575 - rpn_regr: 0.2441 - detector_cls: 1.9130 -
detector_regr: 0.5806
```

#### (3) Inception V4: 5min40s (for 50 batches)

```
Epoch 1/5
 1/1000 [.....] - ETA: 19:19:19 - rpn_cls: 9.3775 - rpn_regr: 0.0860 - detector_cls: 3.044
5 - detector_regr: 0.0000e+00

Epoch 1/5
 50/1000 [>.....] - ETA: 1:04:05 - rpn_cls: 6.7471 - rpn_regr: 0.1326 - detector_cls: 2.1892
- detector_regr: 0.1479

Epoch 1/5
100/1000 [==>.....] - ETA: 48:25 - rpn_cls: 6.3929 - rpn_regr: 0.1332 - detector_cls: 1.7981 -
detector_regr: 0.2071
```

## 2. Conclusion:

(1) Compared with VGG16, Resnet 50 and Inception V4 with a larger raw feature extraction net, which leads to a relatively long training process

(2) Based on previous analysis, Resnet 50 has a better performance in detection than VGG 16, thus we are interested to test the performance of Inception V4 in detection and to find out whether a deeper structure could lead to a better performance in detection.

## 👑 Detection Analysis:

### 1. Reality:

These pictures we used to do the detection task shows that these faster rcnn model can not detect target without enough training epochs. (please visit github for more details)

Example: the first generated image in test process:

(1) Vgg 16: with 800~1000 epochs



(2) Inception V4: with 20 epochs:



(3) Resnet 50: with 20~30 epochs:



**2. Conclusion:**

(1) At least ( $\approx$ )1000 epochs are needed to for Faster RCNN to be relatively well-trained and to accomplish detection.

(2) As such, in order to do model comparison, we need more training epochs in each model, which will leads to a more time-consuming task or a more powerful GPU.

**👑 Future Work:**

1. Without limitation of time, we can train each model with much more epoches, then we can get well-trained weights, which means a better model.

2. With well-trained model, we can compare the accuracy mathematically and explore detection performance of each model.

3. Based on results from 2, combined with speed analysis, we can compare the performance in detection of these four models.



## Appendix:

### Zipped Package Detail:

1. sharedCNN\_model:(html)[CORE ]presentation of construction for each raw feature extraction model, including VGG16, Resnet50, Inception V4, Inception V3
2. Train\_Test: (html)[CORE]presentation of training and test process
3. Keras\_frcnn: (py) functions used in training and testing. Both from reference and our own .py file(each own file has a presentation in sharedCNN\_model)
4. Pretrained\_weights: weights to initialize raw feature extraction net
5. Model\_Store: Store pickle(model information) and h5py(weights information)
6. Spark: spark process

### Figure Source:

1. Fig. 7, 8: Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks : <https://arxiv.org/pdf/1506.01497.pdf>
2. Fig. 1: <http://blog.csdn.net/shenxiaolu1984/article/details/51036677>
3. Fig. 2, 3, 4: <http://blog.csdn.net/shenxiaolu1984/article/details/51152614>
4. Fig. 5, 6: <http://blog.csdn.net/AliceHzj/article/details/78066947>