Natural Language Processing Lab

Object detection using CNN

Samin Payrosangari

Supervisor: Diego Esteves

Winter semester 2017/18

Abstract

In recent years, various approaches for object detection have been designed using CNNs. The main idea of these approaches is to move throughout the image with small paces and discover if the desired object exists in that specific part of the image. These approaches perform better and faster than previous approaches using computer graphics. In this report, I am going to shortly explain these methods in section 1, explain how we could implement and train them with my desired datasets using google cloud platform in section 2, and analyze some graphs displaying their loss and distribution of their weights to decide which model operates better and with higher accuracy in section 3. Finally, I make a conclusion about them in section 4. You could also see the list of references in section 5.

1 Object Detection using CNN

The first model which uses CNNs for object detection is “R-CNN”. Thereafter, other approaches called fast-R-CNN and faster-R-CNN proposed to increase the speed and efficiency of R-CNN. Moreover, two other methods called R-FCN and SSD are discussed which try to decrease the amount of required computations.

1.1 R-CNN

R-CNN or “**R**egion-based **C**onvolutional **N**eural **N**etwork” gives us an overall view of the functionality of object detection methods using CNNs.
The Idea is to:

1. Scan the input image for possible objects using an algorithm called Selective Search, generating ~2000 **region proposals**
2. Run a convolutional neural net (**CNN**) on top of each of these region proposals
3. Take the output of each **CNN** and feed it into a) an SVM to classify the region and b) a linear regressor to tighten the bounding box of the object, if such an object exists [1].

These steps are easily observable in the following figure:

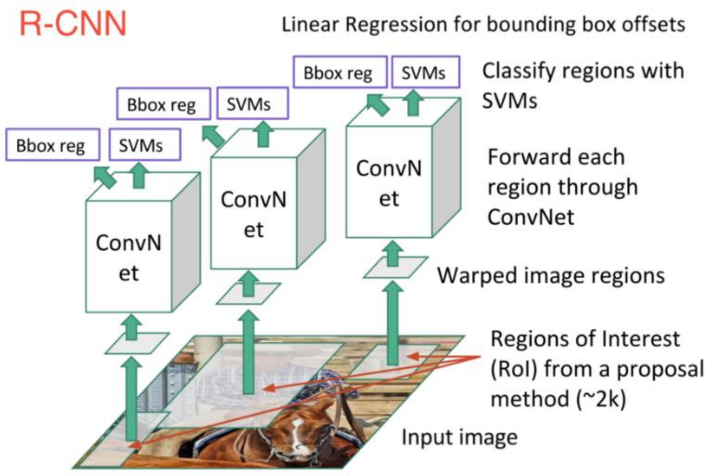
R-CNN
Linear Regression for bounding box offsets
Bbox reg
SVMs
Classify regions with SVMs
ConvNet
Forward each region through ConvNet
Warped image regions
Regions of Interest (RoI) from a proposal method (~2k)
Input image

Figure 1. R-CNN

R-CNN was very intuitive, but very slow according to the selective search.

1.2 Fast R-CNN

Fast R-CNN, improves the detection speed of R-CNN by running only one CNN through the whole image and extract the features, and then generate the region proposals on the feature map. So, it has only one CNN instead of several. Moreover, in this method a softmax layer is used for classification rather than the SVM. These changes, enhance the speed in comparison with R-CNN.

1.3 Faster R-CNN

Faster -CNN proposes an even faster approach by replacing the slow selective search method by **region proposal network** (RPN). This RPN is operated on the last feature map of the CNN. As I mentioned before, the image is passed to CNN at first and then the object detection task is done on the last feature map. A 3*3 sliding window moves across the feature map and while the sliding window is moving, multiple possible regions based on *k* fixed-ratio **anchor boxes** are generated on that position. In other words, we look at each location in our last feature map and consider *k* different boxes centered around it: a tall box, a wide box, a large box, etc. For each of those boxes, we output an objectness score, and the coordinates for that box are [1]. If the score is higher than a threshold, this box is proposed as a region for object detection. Then We add a pooling layer, some fully-connected layers, and finally a softmax classification layer and bounding box regressor at top of the region proposals to classify objects in these regions.

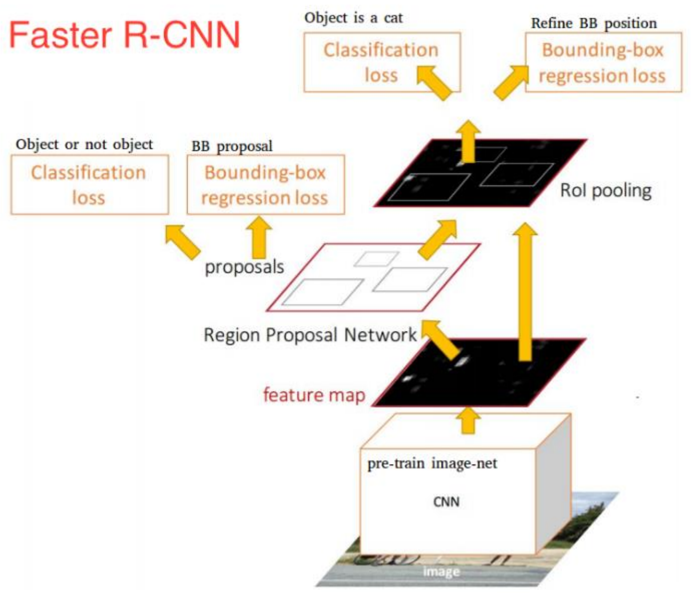

Figure 2. Faster R-CNN

1.4 R-FCN

R-FCN or "**R**egion-based **F**ully **C**onvolutional **N**et "**,** tries to share the whole computations among all the steps and all the regions. As I mentioned before, fast R-CNN also used a shared CNN to increase the speed and decrease the amount of computations, and RFCN benefits from the same idea and is fully convolutional. For this purpose, R-FCN uses position-sensitive score

maps**.** Essentially, these score maps are **convolutional feature maps that have been trained to recognize certain parts of each object**. For example, one score map might activate wherever it detects the *top-right* of a *cat*. Another score map might activate where it sees the *bottom-left* of a *cat [1]*.

1.5 SSD

The final model is SSD, which stands for Single-Shot Detector. Our first two models performed region proposals and region classifications in two separate steps. First, they used a region proposal network to generate regions of interest; next, they used either fully-connected layers or position-sensitive convolutional layers to classify those regions. SSD does the two in a "single shot," simultaneously predicting the bounding box and the class as it processes the image [1]:

Goes through convolutional layers → yielding several sets of feature maps at different scales (e.g. 10x10, then 6x6, then 3x3, etc.)

1. For each location in *each* of these feature maps → considers small set of default bounding boxes.
2. For each box→ **simultaneously** predict the bounding box offset and the class probabilities

2 Setup the environment

Tensorflow library for python is one of the most famous libraries for implementing machine learning algorithms and methods. Recently, google has come up with Object detection API designed on top of Tensorflow and contains various object detection models like SSD, R-FCN and faster R-CNN which I mentioned in previous section. Furthermore, these models are pretrained with bug datasets such as COCO dataset and pets dataset. I used these models and trained them with my own dataset. Using an initial point where the model has already learned something looks better than a totally random initial point which we might need more efforts. Moreover, I used google cloud platform to run my training and evaluation tasks as it provides users with advanced CPU and GPU resources and reduces the training time significantly in comparison to running the task on private laptops. In this section I briefly describe the workflow step be step [2], [3], [4].

2.1 preparing the datasets

For this task I wanted to train a model to detect persons, mountains, forests, highways, buildings and logos. Therefore I used 3 prepared datasets: Caltech 101 for images of faces of people, scene 13 for images of landscapes and belgalogos dataset for logos.

The inputs of tensorflow models are TFrecord formats, so I had to extract required info from images into csv files and the encode the csv file to TFrecord format. So, at first I extracted the filename, width, heigh and coordination of the bounding box around the objects in the images into a xml files using “script.py”, and then turned these xml files into two csv files using “xml2csvscript.py”[1], one corresponding the train images (train.csv) and the another corresponding to evaluation images (test.csv). Subsequently, running “tfRecordConversion.py” encodes the images and their csv file into TFrecod files called train.record and test.record. These files should be saved on google cloud as I discuss in further steps.

2.2 Installing requirements

for using object detection api, tensorflow, and google cloud platform, firstly I installed some prerequisite:

- tensorflow library for python 3.6
- Setup object detection api, generate a google cloud project and generate a Bucket in my google cloud account to store required data on it[2].
- Install google cloud SDK to be able to connect to google cloud platform from your lovcal system (this service uses python 2.7 so I installed both python 3.6 and 2.7)[3]

Resource allocations for all the experiments are as mentioned bellow:

2.3 Manipulation of some files to avoid errors

Some parts of the object detection api need to be changes and some lines of codes should be added so that you could run your application properly on google cloud [5].

2.4 manipulating configuration files

[1] The script for converting xml to csv and csv to TFrecord are copied from this repository: https://github.com/datitran/raccoon_dataset

[2] https://cloud.google.com/solutions/creating-object-detection-application-tensorflow

[3] https://cloud.google.com/sdk/downloads

For object detection I needed three other important files config file of the models I wanted to use, cloud.yml file which contain the required configurations for running my project on cloud and Label Map file which contains a mapping of name of object classes to numbers. These files are already located in object detection api and I had to change the paths and other configuration according to my own project.

The models I used were:

- SSD pretrained on COCO dataset
- Fast R-CNN pretrained on COCO dataset
- R-FCN pretrained on COCO dataset

Fall the models were trained for 100000 iterations, in all experiments.

2.5 Uploading required data on google cloud platform

To implement the project on google cloud I had to upload the training and evaluation TFrecord files, model config files, and label map to my BUCKET (storage) on google cloud. The paths also had to point to the location of these files on google cloud. After uploading the files and packaging the configuration files on my laptop, I was able to use google cloud sdk to run the training task on google cloud.

2.6 Monitoring training task

The training and evaluation progress are visible at “tensorboard”. I called tensorbard from google cloud shell, and it displayed charts and graphs regarding the training and evaluation task to me. These charts are shown on Experiments section.

2.7 Test the model

Various checkpoints are saved to my storage after specific number of iterations while training. So, I was able to retrieve the model after any number of iterations and test it.

3 Experiments

To be able to compare the performance of three models on datasets, same configurations for all of the are used. Some important aspects of the models are:

- Number of iterations: 100000
- momentum_optimizer_value: 0.9
- Transition function:
  - SSD: Sigmoid
  - R-FCN and Faster R-CNN: Softmax
- batch size:
  - SSD: 24
  - R-FCN and Faster R-CNN: 1

3.1 Resource allocations for running this project

For running big machine learning projects, we need several powerful resources as training takes lots of time. So, cloud environment provides us with our required resources like large amount of memory and enough CPU and GPU resources. I configured these resources for my project as:

- **Machine type**: **8 vCPUs**.
- **Memory**: **8Gb**.
- **GPU**: **none**
- **Firewall**: **Allow HTTP traffic**.
- **Assigned a static IP address**

3.2 First Experiment: Training SSD on Adidas logos

The losses are too high. It seems that as the images are so big and the logos are only a small part of images the model need even more than 100000 steps to learn the object detection task. After 100000 iteration, the model only was able to classify an image into having an Adidas logo, but still was unable to localize the logo. Image 1 is a sample of dataset training images and image 2 shows a test image.

Image 1. Image containing Adidas logo

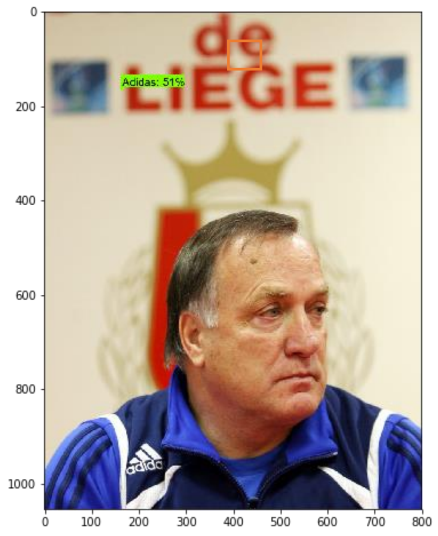

Image 2. Testing the model on an image: the model could only propose that there is an Adidas logo in the image, but can't localize it

TensorBoard's Scalar Dashboard visualizes scalar statistics that vary over time; for example, you might want to track the model's loss or learning rate.

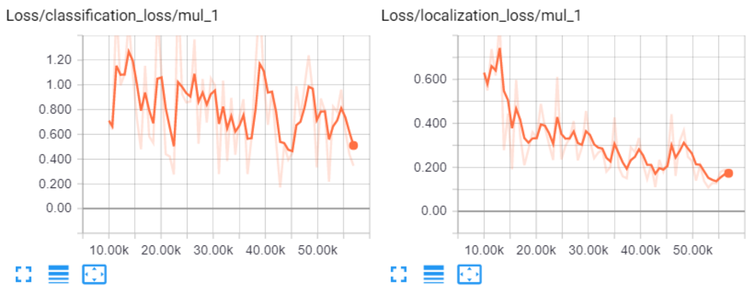

Figure 3. classification and localization loss of experiment 1

As shown in figure 3, training the SSD using ends up in high localization and classification loss. The loss plot over iterations is very nasty and abnormal.

The TensorBoard Histogram Dashboard displays how the distribution of some Tensor in your TensorFlow graph has changed over time. In Tensorboard many histograms visualizations of your tensor at different points in time are displayed. In figure 4 you observe the histograms of experiment 1.

3.2 Experiment 2: Training SSD with 5 classes of person, forest, mountain, highway and buildings

The SSD model on this dataset works better than the previous one but still it is not that much powerful. Maybe, the result of performance improvement is that the training images of datasets only contain one object of one of the specified class. Actually, the whole image is the object and no irrelevant object is included in a photo. some example of images in this dataset are shown in image 3.

Image 3. Samples of training images of the dataset of 5 classes.

The SSD model was really successful to classify the objects of teszt images, however, for localization it was not that much precise although the localization loss looks near zero as shown in figure 5.

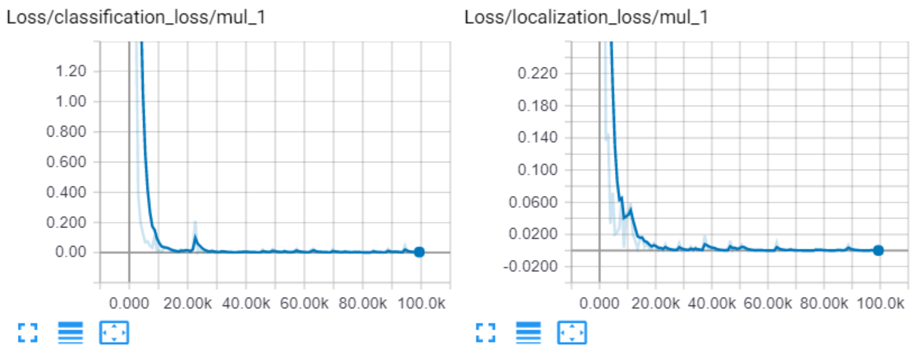

Figure 4. classification and localization loss of experiment 2.

The set of test images shown in image 4, are used for all the models so that we could compare the performance of these models. As shown in this image, most of the SSD has localized the whole photo as the desired object. It seems that, as the coordination of bounding boxes of train images were the total width and height of the image, SSD has learned this behavior and it has maked the whole image as bounding box in test images. It seems that SSD is adopted to training images too much and can't localize the test images well. It does not seem that increasing the number of iterations could help SSD to better localize. However, it has classified many images.

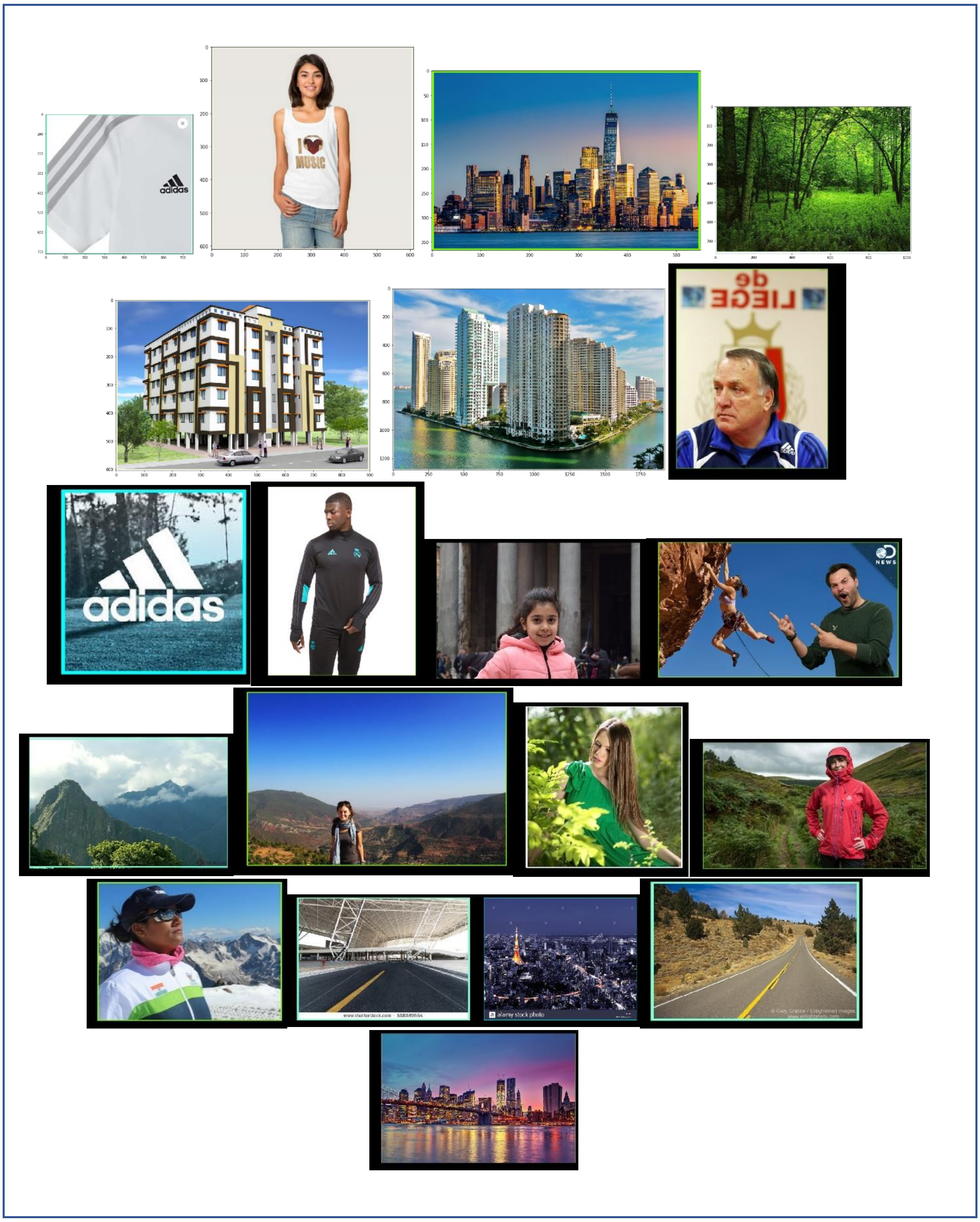

Image 4. Test images on SSD model trained with 5 classes dataset

Figure 5 shows the histograms of experiment 2. So, in this figure you could observe the change in distribution of weights and biases and other parameters per iteration.

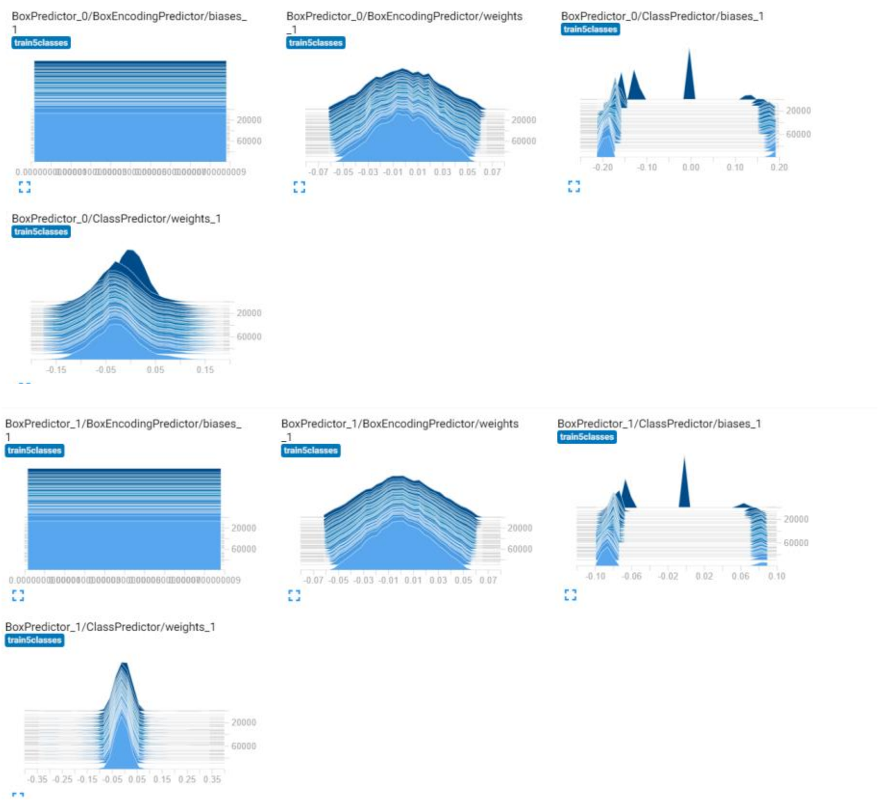

Figure 5. Histograms of experiment 2

3.2 Experiment 3: Training RFCN on dataset of 5 classes

RFCN looks really more powerful than SSD for learning our dataset, total loss look low enough as shown in figure 6. RFCN acted good on localizing the objects, however, as shown in image 5, it was unable to classify some of the photos of test set. RFCN has learned one class (person class) very well, but it is still unable to detect the objects of some other classes like forest class. Maybe, increasing the number of iterations could help RFCN to learn other classes.

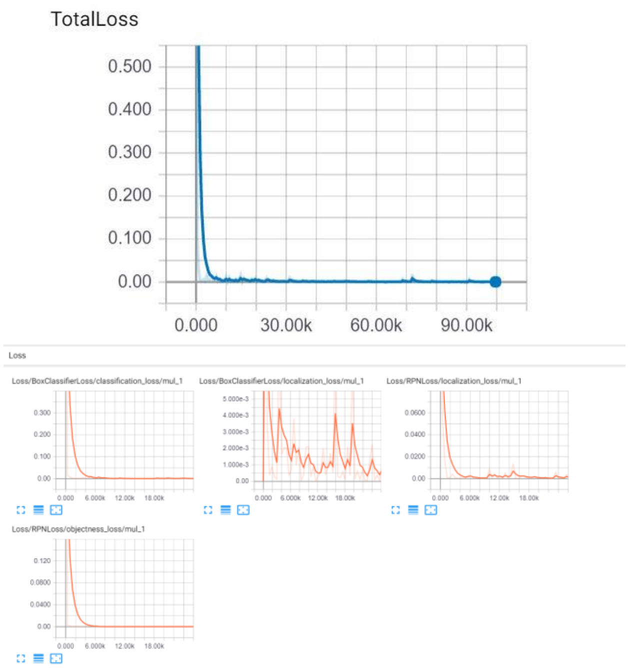

Figure 6. Total loss (shown in blue), classification loss, localization loss, localization loss of RPN network and objectness loss of RFCN when training on images of 5 classes.

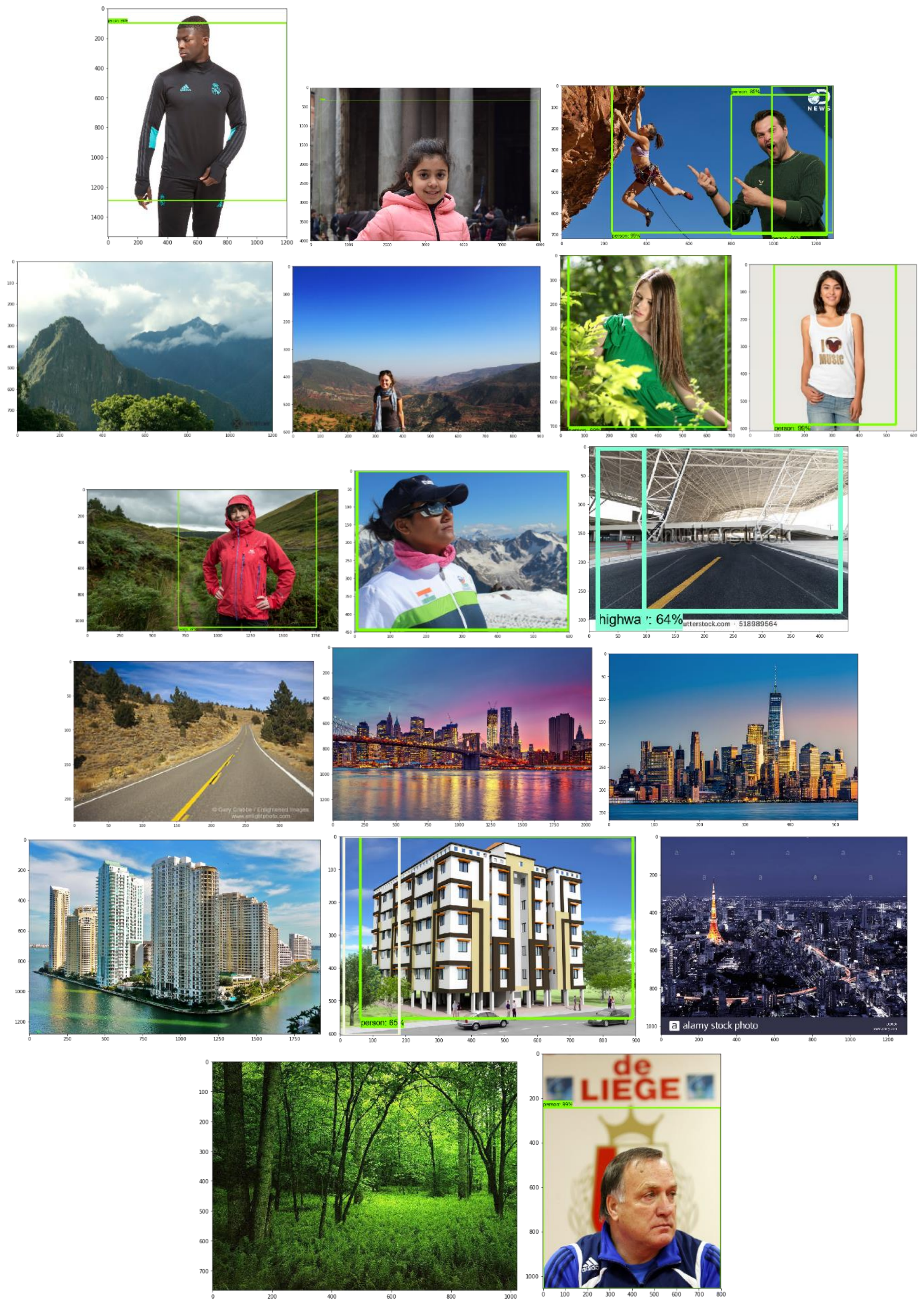

Image 5. RFCN performance on test images

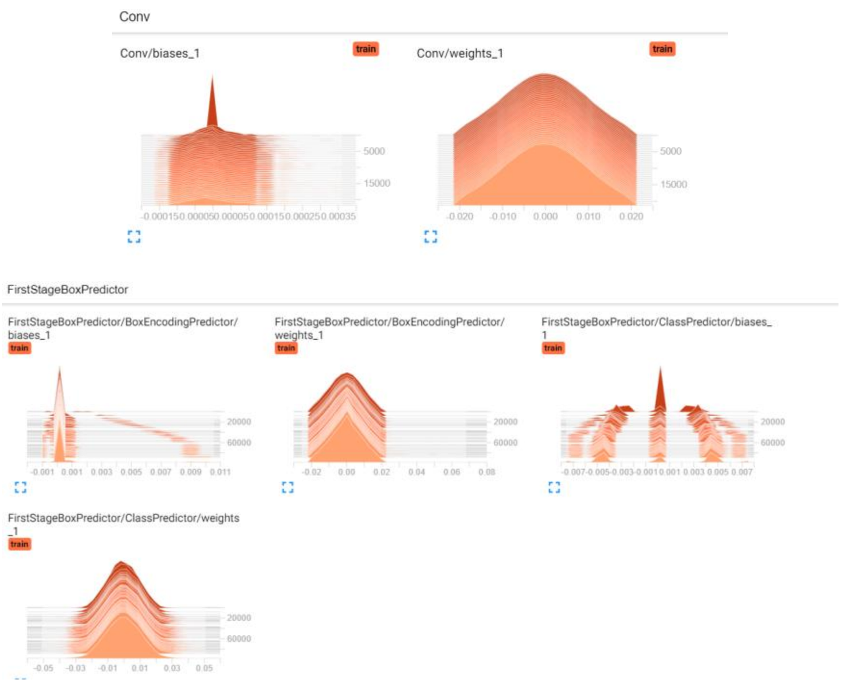

Figure 7. Histograms of RFCN, these histograms show that the distribution of some of the weights has not changed over iterations however, the distribution of biases has changed a lot.

3.4 Experiment 4: Faster R-CNN on five classes

Faster R-CNN looks the best and most accurate model for learning our dataset, although it's a bit slower than the other two models and needed more time to accomplish training. The total loss is very low, it has learned the training data well but still able to work well on test data. Image 6 shows its performance on test dataset. It has classifies all the objects with good localization of the object location. It still has some issues and has classified a highway as a mountain. But, I think increasing the number of iterations could help faster R-CNN to learn better. Moreover, looking at figure 8 and 9 you could observe that the training losses are low, and the histograms show changed in the weights and bias distribution over time, which means that Faster R-CNN has learned datasets.

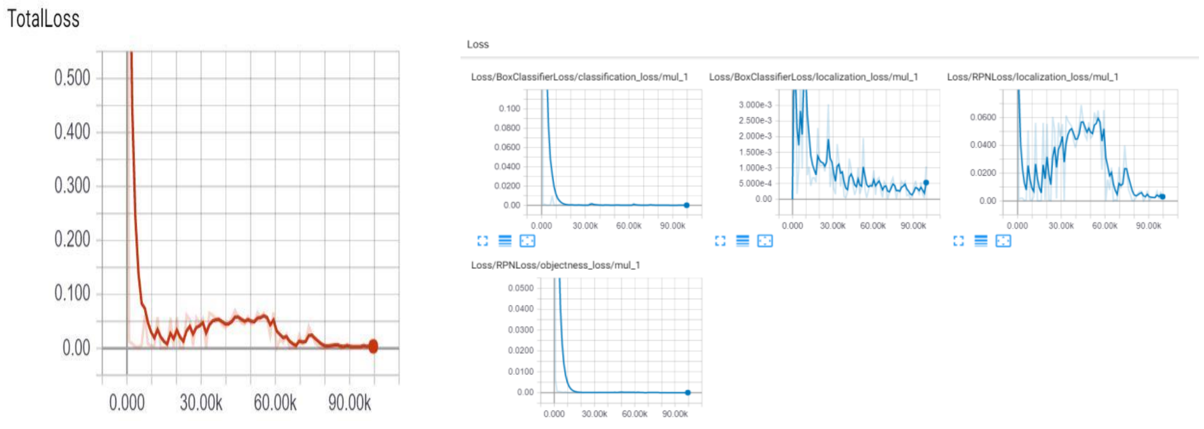

Figure 8. losses of Faster R-CNN

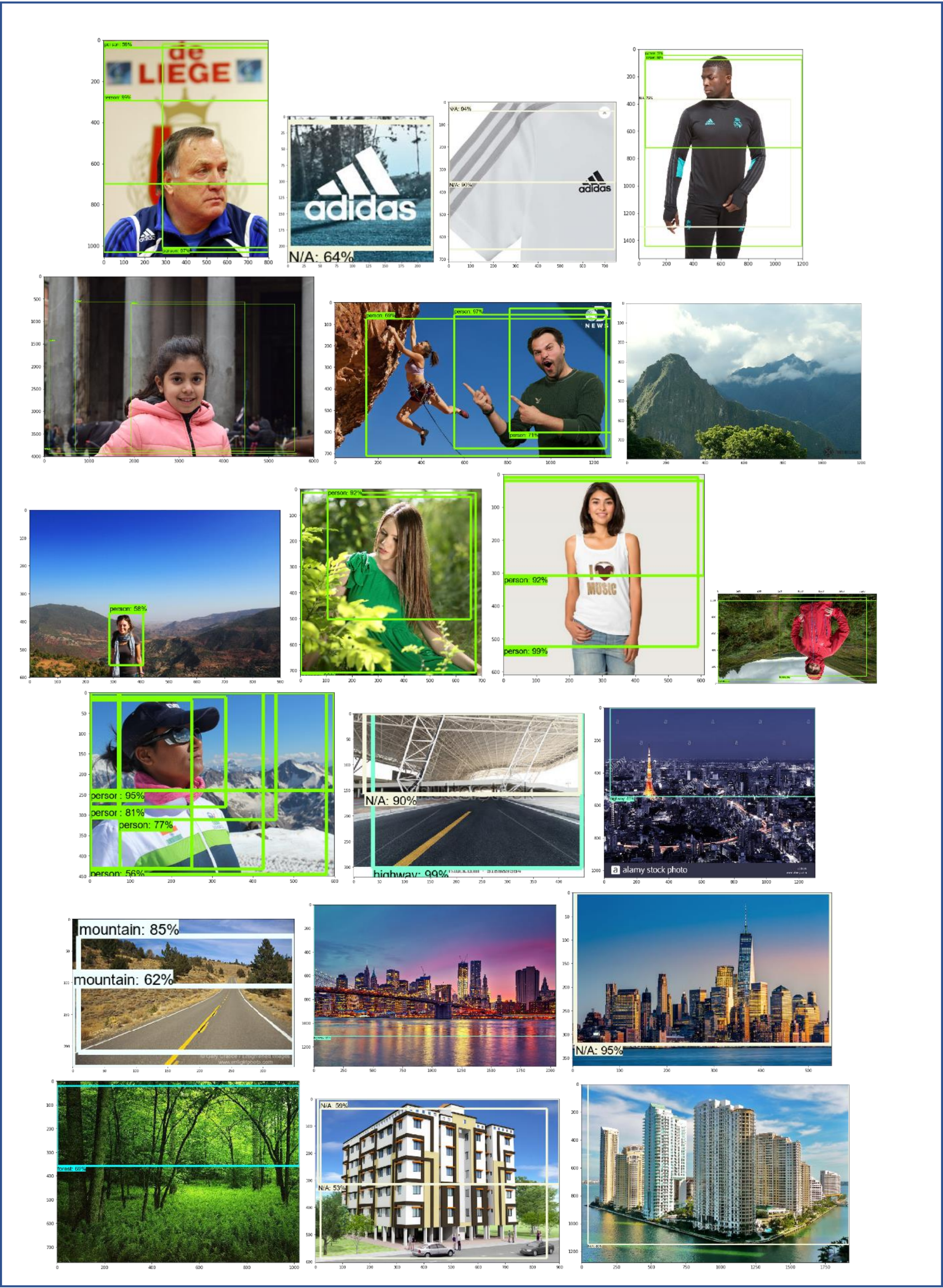

Image 6. Faster R-CNN on test images

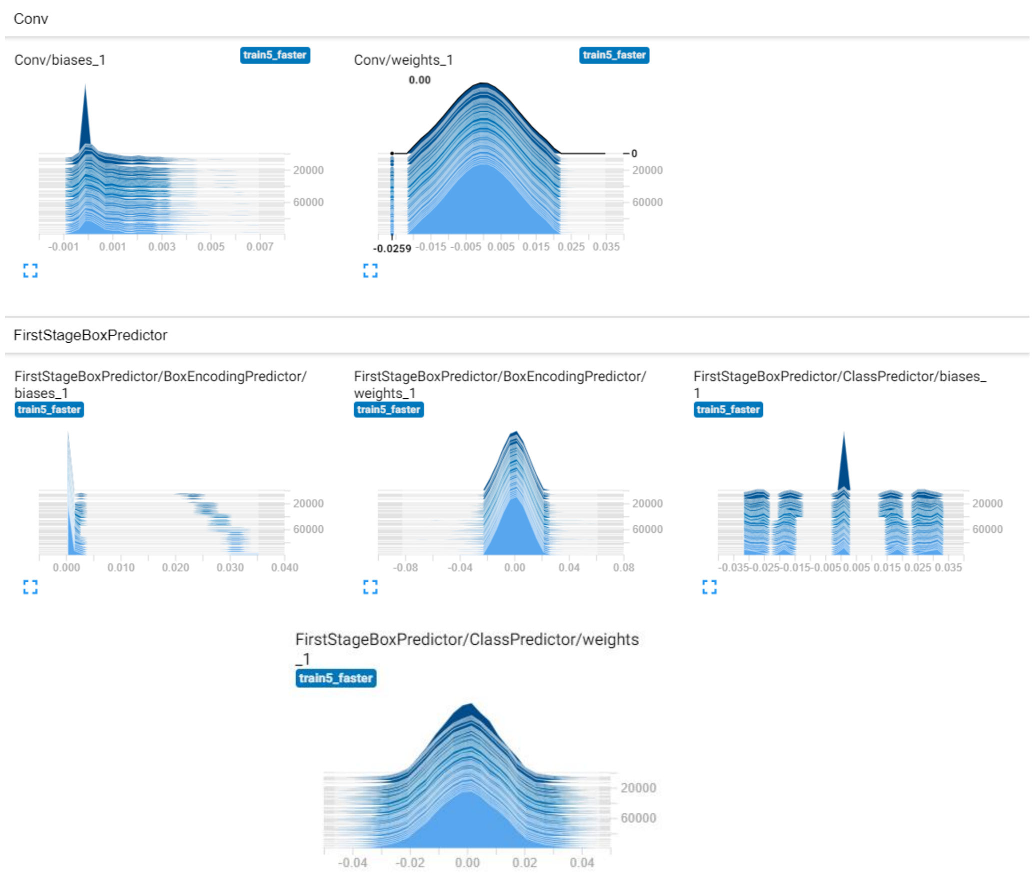

Figure 9. Histograms of faster R-CNN, the distribution of biases and weights has changed over time and shows that the model has learned the datasets

4 Conclusion

In this Lab course, we trained three objects detection models SSD, Faster R-CNN and RFCN on our dataset containing 5 classes of objects. Although all the models show low total loss after 100000 iterations, SSD and RFCN don't show good performance on test images. SSD model looks completely over trained on training images and it does not perform good on localizing objects. RFCN operates better but still weak to detect objects in some images, and has only learned two classes of objects well.

Faster R-CNN has the best performance on test images. It detects most of the objects and localize them accurately. However, it still has classified one image wrong, which could be improved by increasing the number of iterations.

5 References

[1] https://towardsdatascience.com/deep-learning-for-object-detection-a-comprehensive-review-73930816d8d9

[2] https://medium.com/google-cloud/object-detection-tensorflow-and-google-cloud-platform-72e0a3f3bdd6

[3] https://cloud.google.com/solutions/creating-object-detection-application-tensorflow

[4] https://towardsdatascience.com/how-to-train-your-own-object-detector-with-tensorflows-object-detector-api-bec72ecfe1d9

[5] https://github.com/tensorflow/models/issues/2739