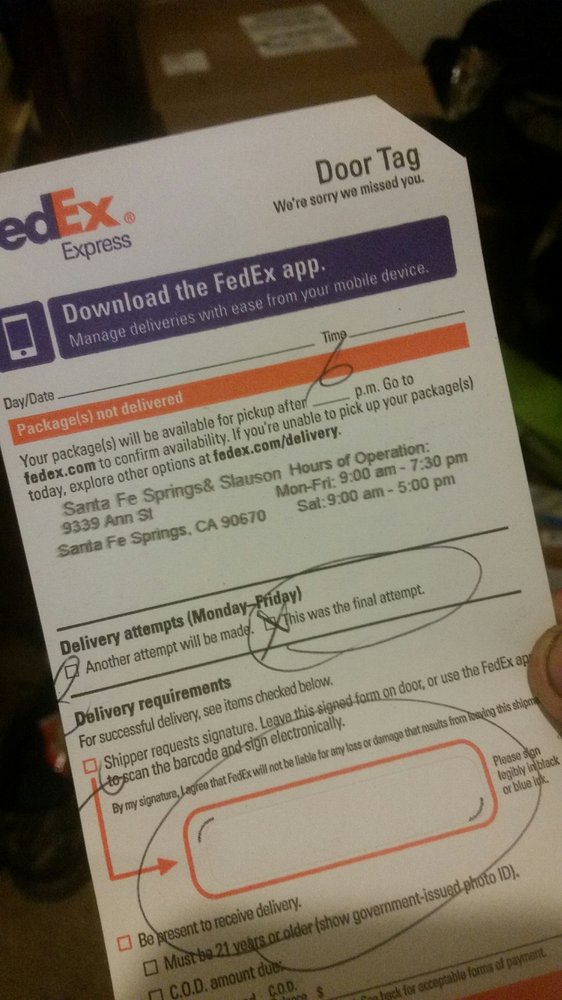
**Building a Real-Time FedEx/UPS/USPS detection system with YOLO**

In this article, I will share with you the steps to build a real-time object detection system to detect FedEx/UPS/USPS delivery trucks with Yolo. This is the 1st part of a 2-parts project:

* 1. 1st part (this article): Motivation, a quick introduction of Yolo, and how to train and test the model.
  2. 2nd part (coming soon): Connect with Raspberry Pi and camera for real life detection via Darkflow implementation of Yolo.

**You can find the code on my GitHub repo here...**



**Title:** “Another attempt will be made.”

**Motivation**

*“Another attempt will be made.”* It is not uncommon for you to receive a missing delivery note like this for signature required packages. There is only one more attempt left, so you decide to change your schedule to stay home the next day. However, your apartment is in the back of the building, and most of the time the delivery person would just attempt to deliver at front door or middle door, but not your rear door, no matter how detailed of an instruction you provide when ordering the products. You get tired of this, so next time you take initiative by calling delivery service to request for your package to be held at their main office, just to realize that the seller does not allow them to do so. You feel desperate, so you start putting all kind of instructions at front door, middle door, rear door, with your phone number on it, which they’re not allowed to call!!

Yes, these ironic situations have been happening to my roommates and I, as well as too many friends and family we know. The last time when I was waiting for my new pc rig, I had to spend two hours sitting in front of the building, waiting and killing time reading a book. Thus, my roommate and I have decided to step up and create our own object detection system, that would notify us when a delivery vehicle stops, or drives in front of our building.

**Why Yolo?**

Among state-of-the-art methods for deep learning object detection ( Faster R-CNN, SSD, YOLO , …), Yolo stands out due to its great balance between speed and accuracy. In Yolo, each input image is divided into a S x S grid. For each cell in the grid, some bounding boxes predictions are generated simultaneously with class probabilities for predicting objects associated with that grid cell. Each grid cell predicts B (B = 5, 7, 9 …) bounding boxes and confidence scores for those boxes. That score reflects how confident the model is that the box contains an object, and also how accurate it thinks the box is that it predicts.

You can find some useful links for Yolo explanation here:

<https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b>

<https://medium.com/diaryofawannapreneur/yolo-you-only-look-once-for-object-detection-explained-6f80ea7aaa1e>

<https://hackernoon.com/understanding-yolo-f5a74bbc7967>

In case you have inquiry regarding Yolo, I highly recommend this google group: <https://groups.google.com/forum/#!forum/darknet> .

In summary:

* Yolo’s detection speed makes it an ideal candidate for real-time detection (GTX 1070 GPU can process video at 32 FPS for Yolov3, and 64 FPS for Yolov2, at default resolution of 416 x 416).
* Yolo reasons globally about the image when making decisions …
* Yolo learns generalizable representations of objects => highly generalizable => less likely to break down when applied to new domains or unexpected inputs. **(Attach some good links about YOLO here)**

**Training and Testing the model**

For training the model, we use the optimized Yolo implementation by AlexeyAB ( <https://github.com/AlexeyAB/darknet> ). (**Should we need to elaborate how good this implementation is?)**

1. **Gathering training data**

First, we started by scraping data for training. Our target data set was Fedex/ UPS/ USPS delivery trucks/logos. Other service like DHL could easily be expanded later. To avoid manually downloading the images, we used the google images scraping tool from <https://github.com/hardikvasa/google-images-download>. We downloaded around 400 images for each category, but ended up labeling only around 200 images for each class, since that should be enough for the first pilot experiment. For better training, we also included negative examples, i.e. images that do not include trucks/logos from the above companies. Ideally, these images should include different kinds of vehicles, houses, trees... that resemble similar environment around our houses. Keep in mind that, in his GitHub repo ( <https://github.com/AlexeyAB/darknet#how-to-improve-object-detection> ), Alexey recommends using as many images of negative examples as there are positive images for better detection. Based on my personal experience on some other Yolo projects, perhaps 40% - 50% of negative images is a good range, and more than 50% might make the model become less “sensitive”.

1. **Labeling images**

Next step was to label data for training. In our case, objects that we were labeling was the company’s logo. We decided to exclude whole vehicle and draw bounding box only around the logo. However, in the future, we would experiment with labeling the whole vehicles for detection as well, since trucks like UPS and DHL are very distinctive/ recognizable. We used BBox-Label-Tool for labeling images ( <https://github.com/puzzledqs/BBox-Label-Tool> )  
Since BBox-Label-Tool does not automatically resize the image to fit the computer screen, oversized images should be resized before labelling to prevent missing part of logo(s) in the images.

1. **Training the model**

You can train the model on the cloud if your GPU is not quite strong. I have just bought a new desktop with GTX 1070, so it’s more than enough for training. I have experimented with both darknet version from the original author of Yolo ( <https://pjreddie.com/darknet/yolo/> ), and the improved version from AlexeyAB ( <https://github.com/AlexeyAB/darknet> ). Based on my experience for a few projects, including this one, I notice AlexeyAB’s version is indeed way faster, in both training and detecting phase ( <https://github.com/AlexeyAB/darknet/issues/529#issuecomment-377204382> ).

To train the model, we need to have the model configuration file, which contains several hyperparameters for training. You can look for the .cfg on my GitHub repo (link).

In this project, we experimented with both Yolov2 and Yolov3.

*Disclaimer:* Training Yolov2 might take way more iterations than Yolov3 before the model start detecting objects ( <https://github.com/thtrieu/darkflow/issues/80> ).

In our case, it took more than 12,000 iterations for Yolov2, and 3000 iterations for Yolov3 to start detecting the delivery trucks. In general, Yolov3 would be way better in terms of accuracy and it’s significantly better at detecting small objects. However, we ended up using Yolov2, since we wanted to use DarkFlow for the Raspberry Pi web server set up (next article), and Yolov3 has not been made available with DarkFlow yet.

Training with Yolov2 is way faster than Yolov3, but as I mentioned above, you might need way more training iterations for detecting your desired objects. The model is trained on 3 classes: FedEx, UPS, and USPS, on top of a pre-trained weights: <http://pjreddie.com/media/files/darknet19_448.conv.23>   
Detailed instruction on how to train your custom objects with Yolov2 can be found here: <https://github.com/AlexeyAB/darknet/tree/47c7af1cea5bbdedf1184963355e6418cb8b1b4f#how-to-train-to-detect-your-custom-objects>   
Or if you want an actual one class detection instruction: <https://medium.com/@manivannan_data/how-to-train-yolov2-to-detect-custom-objects-9010df784f36>

1. **Testing the model (attach some images here and there)**

Knowing when to stop training, and which set of weights to choose for detection is quite important. You’d gain more experience and intuition once you’ve played around enough with training and testing the model, but when you first started, having a guideline like this might be extremely helpful: <https://github.com/AlexeyAB/darknet#when-should-i-stop-training> .   
Keep in mind that having a small training data set might lead to overfitting quickly. In this case, since the data set is small, and I’ve got enough experience from the other project, I decided to use all of the data for training, and chose the best weight by testing on a few clips, instead of splitting my data into train and test set. In general, we should split data into training and testing, and check the accuracy of the model by looking at precision & recall, mAP (mean Average Precision), IoU (Intersection over Union). One good explanation about these measures can be found here: <https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173>

Here are the results when tested on a nice encounter of the three competitors in the Loop – downtown Chicago:



You can also find a demo clip of FedEx/UPS/USPS detection of our project in this link ( <https://youtu.be/qqFTzfZhMMY> ). This is a quite good result, given the size of training dataset, and the fact that it was tested at resolution 416 x 416. You can improve detection by changing resolution from 416 x 416 to 608 x 608 or more, as in AlexeyAB’s instruction. To create a more generalized and robust detection system, we just need to create more labeled data for training.

**Conclusion**

Give me a clap (or 50 :p) if you like this post. Feel free to download the weights and configuration files and test it out yourself. Second part is coming soon. We are basically done with the implementation, and managed to have a real-life detection with FedEx, UPS, and USPS delivery trucks already. Feel free to leave any comment and feedback below!