with Deep Learning

Bharath Raj Follow

Aug 3, 2018 · 14 min read

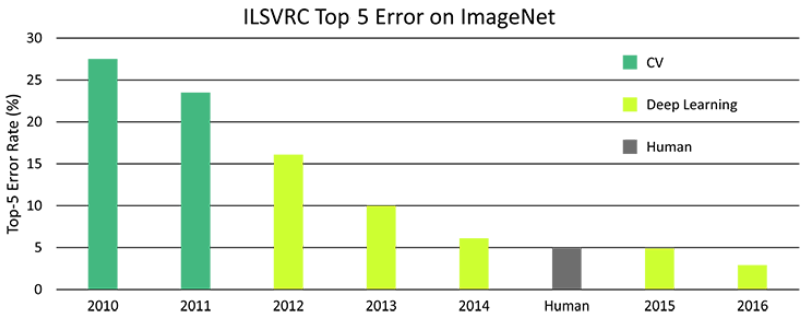
> *This article is a quick tutorial for implementing a surveillance system using Object Detection based on Deep Learning. It also compares the performance of different Object Detection models using GPU multiprocessing for inference, on Pedestrian Detection.*

Surveillance is an integral part of security and patrol. For the most part, the job entails extended periods of looking out for something undesirable to happen. It is crucial that we do this, but also it is a very mundane task.

Wouldn't life be much simpler if there was something that could do the "watching and waiting" for us? Well, you're in luck. With the advancements in technology over the past few years, we could write some scripts to automate the above tasks—and that too, rather easily. But, before we dive deeper, let us ask ourselves:

Are machines are good as humans?

Anyone familiar with Deep Learning would know that image classifiers have surpassed human level accuracy.



Error rate on the ImageNet dataset over time, for Humans, Traditional Computer Vision (CV) and Deep Learning. (Image source: Link)

So yes, a machine can keep a lookout for objects at the same standard (or better) when compared to a human. With that being said, using technology to perform surveillance is much more efficient.

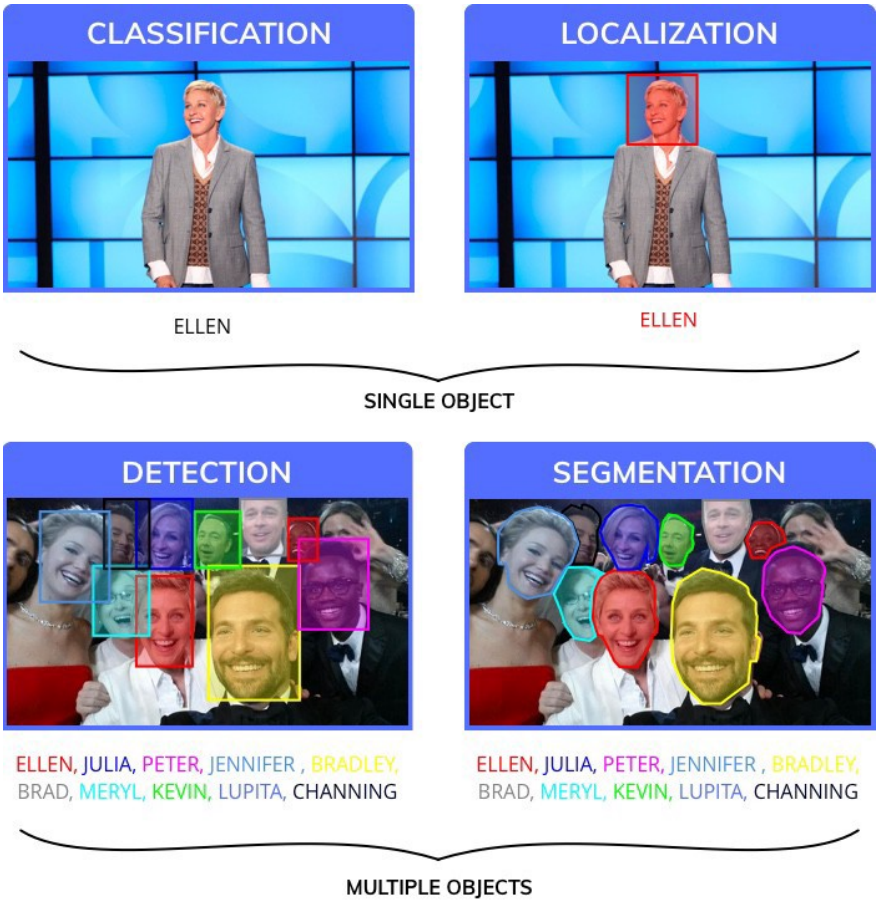
- Surveillance is a repetitive and mundane task. This may cause performance dips for us human beings. By letting technology do the surveillance, we could focus on taking action if something goes amiss.
- To survey a large strip of land, you need lots of personnel. Stationary cameras also have a limited range of view. With mobile surveillance bots (such as micro drones) these problems can be mitigated.

Moreover, the same technology can be used for a variety of applications which are not limited to security, such as baby monitors or automated product delivery.

Fair enough. But how do we automate it?

Before we contrive complicated theories, let us think about how surveillance works normally. We look at a video feed, and if we spot something abnormal, we take action. So in essence, our technology should peruse every frame of the video, hoping to spot something abnormal. Does this process ring a bell?

As you may have guessed, this is the very essence of **Object Detection with Localization.** It is slightly different from classification that, we need to know the exact location of the object. Moreover, we may have multiple objects in a single image.

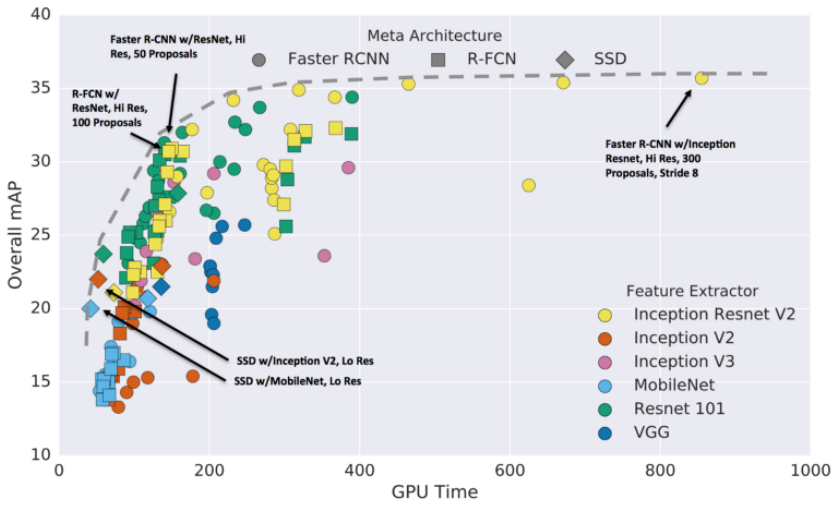


To find the exact location, our algorithm should inspect every portion of the image to find the existence of a class. It is harder than it sounds. But since 2014, continuous iterative research in Deep Learning has introduced heavily engineered neural networks that can detect objects in real time.

	R-CNN	Fast R-CNN	Faster R-CNN
Test time per image (with proposals)	50 seconds	2 seconds	**0.2 seconds**
(Speedup)	1x	25x	**250x**
mAP (VOC 2007)	66.0	**66.9**	**66.9**

Look at how performance increased over just a span of 2 years!

There are several Deep Learning architectures, that use different methods internally, to perform the same task. The most popular variants are the Faster RCNN, YOLO and the SSD networks.



Speed vs accuracy trade-off. A higher mAP and a lower GPU Time is optimal.

Each model depends on a base classifier, which greatly affects the final accuracy and model size. Moreover, the choice of the object detector can heavily influence computational complexity and final accuracy.

> *There is always a Speed vs Accuracy vs Size trade-off when choosing an Object Detection algorithm.*

In this blog post, we will learn how to build a a simple but effective surveillance system, using Object Detection. Let us first discuss the constraints we are bound to because of the nature of the surveillance task.

Constraints for Deep Learning in Surveillance

Often we would like to keep a look-out over a large stretch of land. This brings forth a couple of factors that we may need to consider before automating surveillance.

1. Video Feed

Naturally, to keep a look-out over a large area, we may require multiple cameras. Moreover, these cameras need to store this data somewhere; either locally, or to a remote location.

Typical surveillance cameras. (Photo by Scott Webb on Unsplash)

A higher quality video will take a lot more memory than a lower quality one. Moreover, an RGB input stream is 3x larger than a BW input stream. Since we can only store a finite amount of the input stream, the quality is often lowered to maximize storage.

Therefore, a scalable surveillance system should be able to interpret low quality images. Hence, our Deep Learning algorithm must be

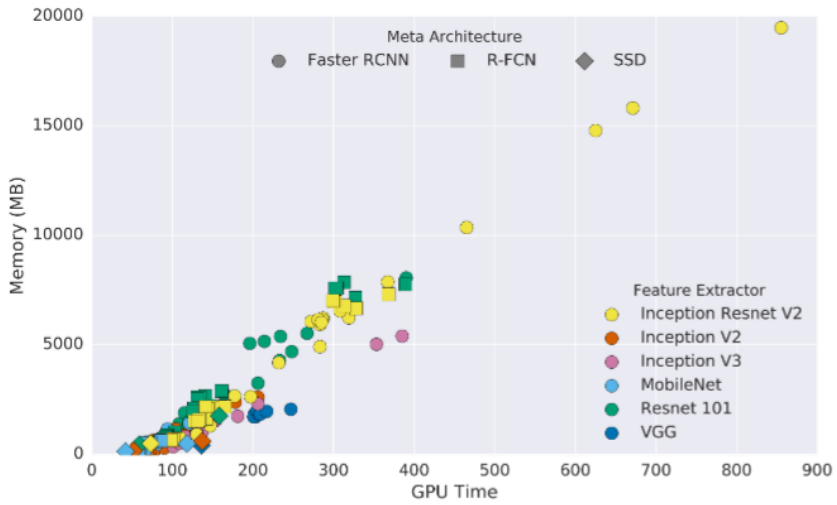
trained on such low quality images as well.

2. Processing Power

Now that we have resolved the input constraint, we can answer a bigger question. Where do we process the data obtained from camera sources? There are two methods of doing this.

- **Processing on a centralized server:**

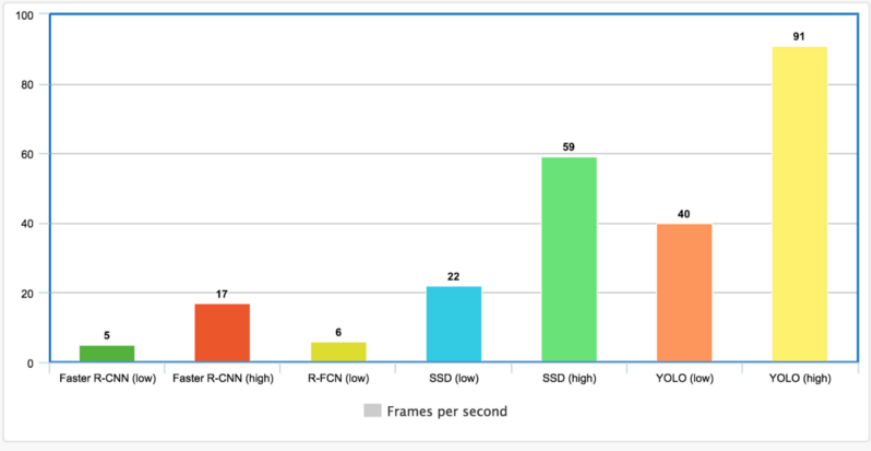
The video streams from the cameras are processed frame by frame on a remote server or a cluster. This method is robust, and enables us to reap the benefits of complex models with high accuracies. The obvious problem is latency; you need a fast Internet connection for limited delay. Moreover, if you are not using a commerical API, the server setup and maintenance costs can be high.



Memory consumption vs Inference GPU Time (milliseconds). Most high performance models consume a lot of memory. (Source)

- **Processing on the edge:**

By attaching a small microcontroller, we can perform realtime inference on the camera itself. There is no transmission delay, and abnormalities can be reported faster than the previous method. Moreover, this is an excellent add on for bots that are mobile, so that they need not be constrained by range of WiFi/Bluetooth available. (such as microdrones).



FPS capability of various object detectors. (Source)

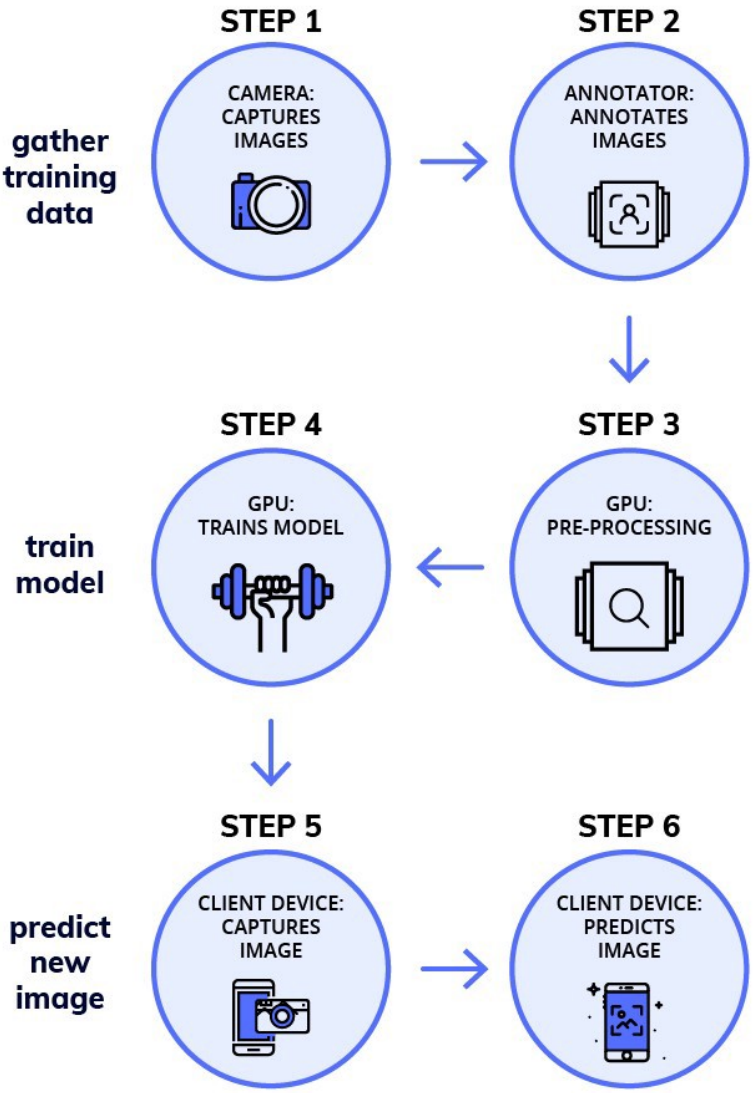
The disadvantage is that, microcontrollers aren't as powerful as GPUs, and hence you may be forced to use models with lower accuracy. This issue can be circumvented by using onboard GPUs, but that is an expensive solution. An interesting solution would be to use software such as TensorRT, which can optimize your program for inference.

Training a Surveillance System

In this section, we will checkout how to identify pedestrians using Object Detection. We'll use the TensorFlow Object Detection API to create our Object Detection module. We will explore in brief on how to set up the API and train it for our surveillance task. For a more detailed explanation, you can checkout this blog post.

The entire process can be summarized in three phases:

1. Data preparation
2. Training the model
3. Inference



The workflow involved in training an Object Detection model.

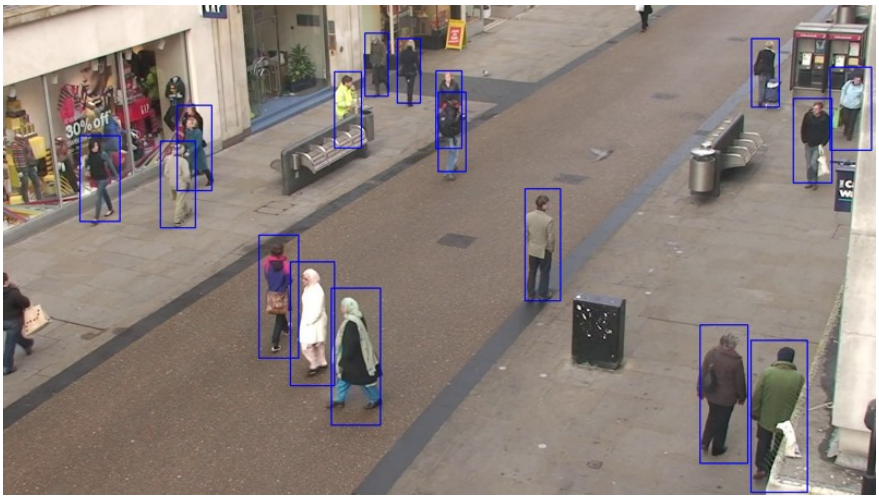
If you feel like seeing the results would motivate you more to try it out, feel free to scroll down to Phase 3!

Phase 1: Data Preparation

Step 1: Obtain the dataset

Surveillance footage taken in the past is probably the most accurate dataset you can get. But, it's often hard to obtain such surveillance

footage for most cases. In that case, we can train our object detector to generally recognize our targets from normal images.



Sample annotated image from our dataset.

As discussed before, the images in your camera feed maybe of lower quality. So you must train your model to work in such conditions. A very elegant way of doing that is by performing data augmentation, which is explained in detail here. Essentially, we have to add some noise to degrade the image quality of the dataset. We could also experiment with blur and erosion effects.

We'll use the TownCentre Dataset for our object detection task. We'll use the first 3600 frames of the video for training and validation, and the remaining 900 for testing. You can use the scripts in my GitHub repo to extract the dataset.

Step 2: Annotate the dataset

You could use a tool such as LabelImg to perform the annotations. This is a tedious task, but important all the same. The annotations are saved as XML files.

Luckily, the owners of the TownCentre Dataset have provided annotations in csv format. I wrote a quick script to convert the annotations to the required XML format, which you can find in my GitHub repo.

Step 3: Clone the repository

Clone the repository. Run the following commands to install requirements, compile some Protobuf libraries and set path variables

```
pip install -r requirements.txt
sudo apt-get install protobuf-compiler
protoc object_detection/protos/*.proto --python_out=.
export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim
```

Step 4: Prepare the supporting inputs

We need to assign an ID to our target. We define the ID in file called label_map.pbtxt as follows

```
item {
 id: 1
 name: 'target'
}
```

Next, you must create a text file with the names of the XML and image files. For instance, if you have img1.jpg, img2.jpg and img1.xml, img2.xml in your dataset, you trainval.txt file should look like this:

```
img1
img2
```

Separate your dataset into two folders, namely **images** and **annotations**. Place the **label_map.pbtxt** and **trainval.txt** inside your annotations folder. Create a folder named **xmls** inside the annotations folder and place all your XMLs inside that. Your directory hierarchy should look something like this:

```
-base_directory
|-images
|-annotations
||-xmls
||-label_map.pbtxt
||-trainval.txt
```

Step 5: Create TF Records

The API accepts inputs in the **TFRecords** file format. Use the **create_tf_record.py** file provided in my repo to convert your dataset into TFRecords. You should execute the following command in your base directory:

```
python create_tf_record.py \
    --data_dir=`pwd` \
    --output_dir=`pwd`
```

You will find two files, **train.record** and **val.record**, after the program finishes its execution.

Phase 2: Training the model

Step 1: Model Selection

As mentioned before, there is a trade off between speed and accuracy. Also, building and training an object detector from scratch would be extremely time consuming. So, the TensorFlow Object Detection API provides a bunch of pre-trained models, which you can fine tune to your use case. This process is known as Transfer Learning, and it speeds up your training process by an enormous amount.

Model name	Speed (ms)	COCO mAP[^1]	Outputs
ssd_mobilenet_v1_coco	30	21	Boxes
ssd_inception_v2_coco	42	24	Boxes
faster_rcnn_inception_v2_coco	58	28	Boxes
faster_rcnn_resnet50_coco	89	30	Boxes
faster_rcnn_resnet50_lowproposals_coco	64		Boxes
rfcn_resnet101_coco	92	30	Boxes
faster_rcnn_resnet101_coco	106	32	Boxes
faster_rcnn_resnet101_lowproposals_coco	82		Boxes
faster_rcnn_inception_resnet_v2_atrous_coco	620	37	Boxes
faster_rcnn_inception_resnet_v2_atrous_lowproposals_coco	241		Boxes
faster_rcnn_nas	1833	43	Boxes
faster_rcnn_nas_lowproposals_coco	540		Boxes

A bunch of models pre-trained on the MS COCO Dataset

Download one of these models, and extract the contents into your base directory. You will receive the model checkpoints, a frozen inference graph, and a pipeline.config file.

Step 2: Defining the training job

You have to define the "training job" in the **pipeline.config** file. Place the file in the base directory. What really matters is the last few lines of the file—you only need to set the highlighted values to your respective file locations.

```
gradient_clipping_by_norm: 10.0
  fine_tune_checkpoint: "model.ckpt"
  from_detection_checkpoint: true
  num_steps: 200000
}
train_input_reader {
  label_map_path: "annotations/label_map.pbtxt"
  tf_record_input_reader {
    input_path: "train.record"
  }
}
eval_config {
  num_examples: 8000
  max_evals: 10
  use_moving_averages: false
}
eval_input_reader {
  label_map_path: "annotations/label_map.pbtxt"
  shuffle: false
  num_epochs: 1
  num_readers: 1
  tf_record_input_reader {
```

```
    input_path: "val.record"
  }
}
```

Step 3: Commence training

Execute the below command to start the training job. It's recommended to use a machine with a large enough GPU (provided you installed the gpu version of tensorflow) to accelerate the training process.

```
python object_detection/train.py \
--logtostderr \
--pipeline_config_path=pipeline.config \
--train_dir=train
```

Phase 3: Inference

Step 1: Export the trained model

Before you can use the model, you need to export the trained checkpoint files to a frozen inference graph. It's actually easier done than said—just execute the code below (Replace 'xxxxx' with the checkpoint number):

```
python object_detection/export_inference_graph.py \
--input_type=image_tensor \
--pipeline_config_path=pipeline.config \
--trained_checkpoint_prefix=train/model.ckpt-xxxxx \
--output_directory=output
```

You will obtain a file named **frozen_inference_graph.pb**, along with a bunch of checkpoint files.

Step 2: Use it on a video stream

We need to extract individual frames from our video source. It can done by using OpenCV's VideoCapture method, as follows:

```
cap = cv2.VideoCapture()
flag = True


while(flag):
    flag, frame = cap.read()
    ## -- Object Detection Code --
```

The data extraction code used in Phase 1 automatically creates a folder ‘test_images’ with our test set images. We can run our model on the test set by executing the following:

```
python object_detection/inference.py \
--input_dir={PATH} \
--output_dir={PATH} \
--label_map={PATH} \
--frozen_graph={PATH} \
--num_output_classes=1 \
--n_jobs=1 \
--delay=0
```

Experiments

As mentioned earlier, there is trade off between speed and accuracy while choosing an object detection model. I ran some experiments which measured the FPS and count accuracy of the people detected using three different models. Moreover, the experiments were run on different resource constraints (GPU parallelism constraints) . The outcome of these experiments can give you some valuable insights while selecting an object detection model.

Setup

The following models were selected for our experiment. These are available in the TensorFlow Object Detection API’s Model Zoo.

- Faster RCNN with ResNet 50
- SSD with MobileNet v1
- SSD with InceptionNet v2

All models were trained on Google Colab for 10k steps (or until their loss saturated). For inference, an AWS p2.8xlarge instance was used. The count accuracy was measured by comparing the number of people detected by the model and the ground truth. The inference speed in Frames per Second (FPS) was tested under the following constraints:

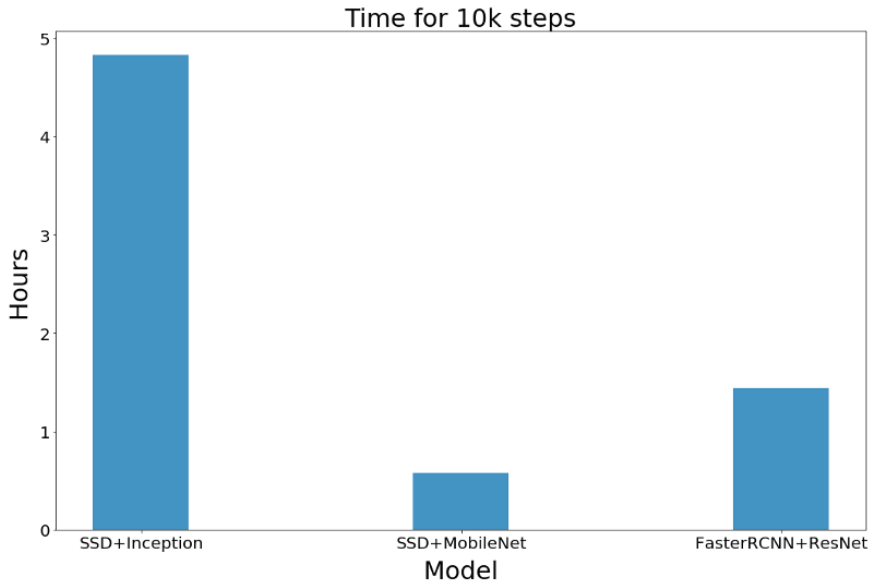
- Single GPU
- Two GPUs in parallel
- Four GPUs in parallel
- Eight GPUs in parallel

Results

Here's an excerpt from the output produced by using FasterRCNN on our test set. I've also attached a video comparing the output produced by each model near the end of this blog. Feel free to scroll down and check it out!

Training Time

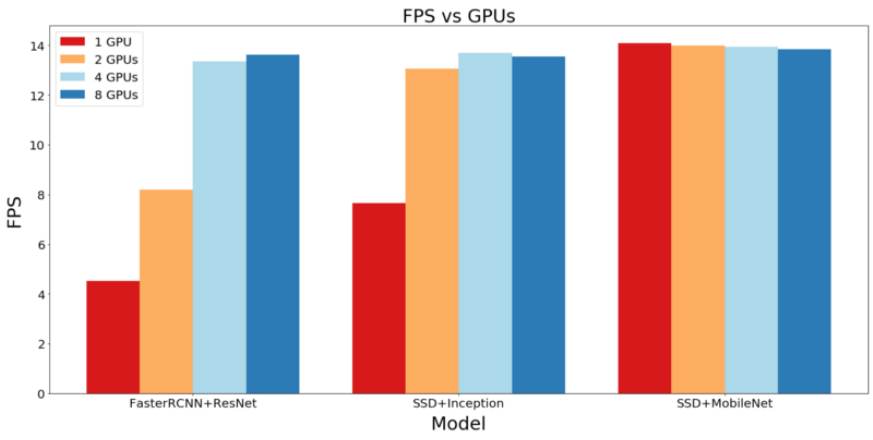
The plot below shows the time needed to train each model for 10k steps(in hours). This is excluding the time required for a hyperparameter search.



When your application is very different from the pretrained model you use for transfer learning, you may need to heavily adjust the hyperparameters. However, when your application is similar, you wouldn't need to do an extensive search. Nonetheless, you may still require to experiment with training parameters such as the learning rate and choice of optimizer.

Speed (Frames per Second)

This was the most interesting part of our experiment. As stated earlier, we measured the FPS performance of our three models on five different resource constraints. The results are shown below:



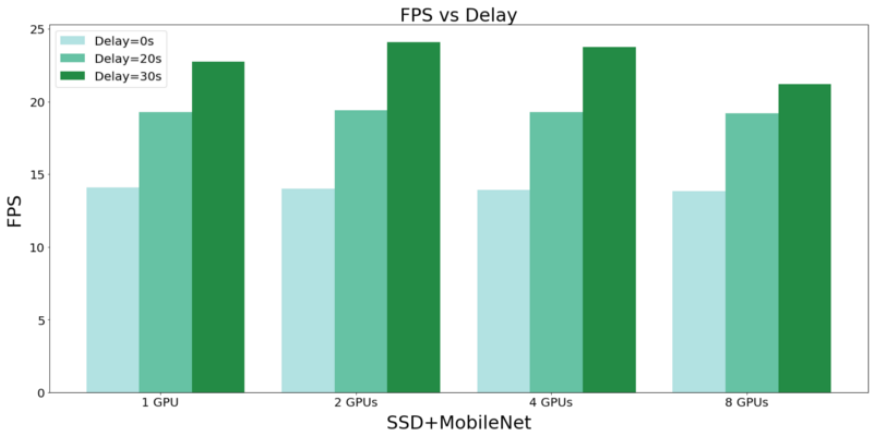
SSDs are extremely fast, easily beating Faster RCNN's speed when we use a single GPU. However, Faster RCNN quickly catches up with SSD

when we increase the number of GPUs (working in parallel). Needless to say, SSD with MobileNet is much faster than SSD with InceptionNet at a low GPU environment.

One notable feature from the above graph is that, FPS slightly decreases when we increase the number of GPUs for SSD with MobileNet. There's actually a simple answer to this apparent paradox. It turns out that our setup processed the images faster than they were being supplied by the image read function!

> *Speed of your video processing system can not be greater than the speed at which images are fed to the system.*

To prove my hypothesis, I gave the image read function a head-start. The plot below shows the improvement in FPS for SSD with MobileNet when a delay was added. The slight reduction in FPS in the earlier graph is because of the overhead involved due to multiple GPUs requesting for input.



Needless to say, we observe a sharp increase in FPS if we introduce delays. The bottom line is, we need to have an optimized image transfer pipeline to prevent a bottleneck for speed. But since our intended use case is surveillance, we have an additional bottleneck. The FPS of the surveillance camera sets the upper limit for the FPS of our system.

Count Accuracy

We define count accuracy as the percentage of people correctly recognized by our object detection system. I felt like it's more apt with respect to surveillance. Here's how each of our models performed:

Needless to say, Faster RCNN is the most accurate model. Also surprisingly MobileNet performs better than InceptionNet.

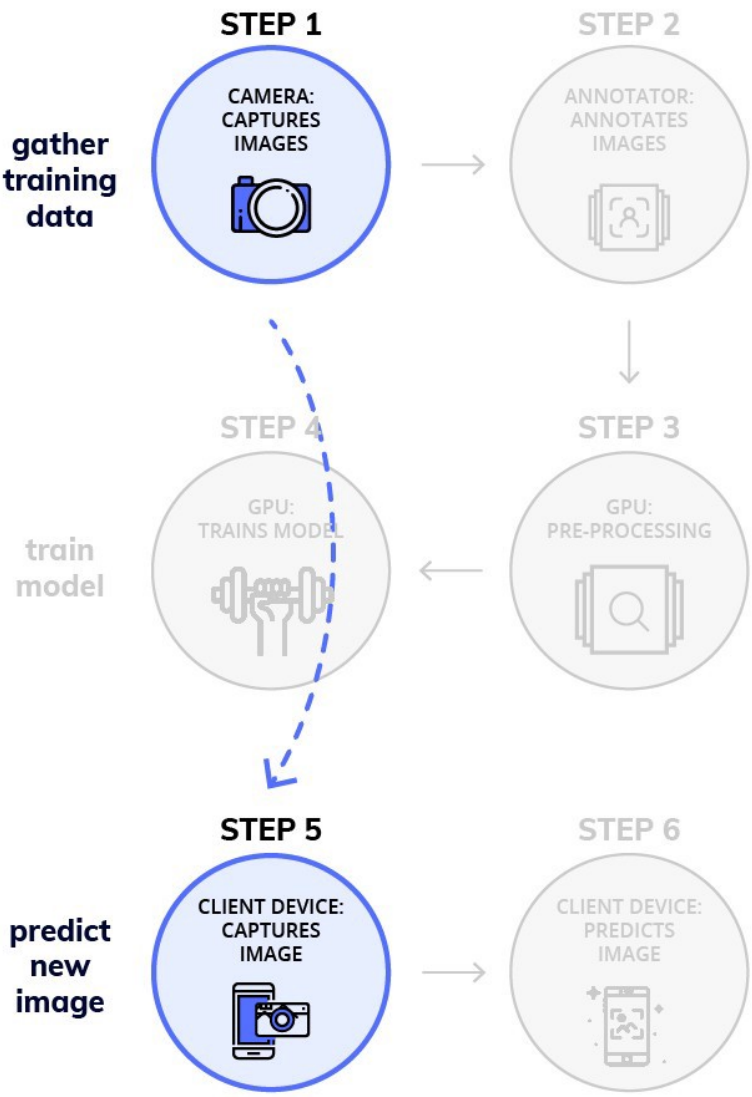
Based on the experiments, it is evident that there is indeed a speed vs accuracy trade-off. However, we can use a model with high accuracy at a good FPS rate if we have enough resources. We observe that Faster RCNN with ResNet-50 offers the best accuracy, and a very good FPS rating when deployed on 4+ GPUs in parallel.

That was a lot of steps!

Well.. I wouldn't argue. It is indeed a lot of steps. Moreover, setting a up cloud instance for this model to work in real time would be burdensome and expensive.

A better solution would be to use an API service that is already deployed on servers so that you can just worry about developing your product. That's where **Nanonets** kicks in. They have their API deployed on quality hardware with GPUs such that you get insane performance with none of the hassle!

I converted my existing XML annotations to JSON format and fed it to the Nanonets API. As a matter of fact, if you dont want to manually annotate your dataset, you can request them to annotate it for you. Here's the reduced workflow when Nanonets takes care of the heavy lifting.



Reduced workflow with Nanonets

Earlier, I mentioned how mobile surveillance units such as micro drones can greatly enhance efficiency. We can create such drones quite easily using micro controllers such the Raspberry Pi, and we can use API calls to perform inference.

It's pretty simple to get started with the Nanonets API for Object Detection, but for a well explained guide, you can checkout this blog post.

Results with Nanonets

It took about 2 hours for Nanonets to finish the training process. This is including the time required for hyperparameter search. In terms of time taken for training, Nanonets is the clear winner. Nanonets also defeated FasterRCNN in terms of count accuracy.

```
FasterRCNN Count Accuracy = 88.77%
Nanonets Count Accuracy = 89.66%
```

Here is the performance of all four models on our test dataset. It is evident that both SSD models are a bit unstable and have lower accuracy. Moreover, even though FasterRCNN and Nanonets have comparable accuracies, the latter has bounding boxes that are more stable.

• • •

Is automated surveillance accountable?

Deep learning is an amazing tool that provides exemplary results with ease. But, to what extent can we trust our surveillance system to act on its own? There are a few instances where automation is questionable.

> ***Update:*** *In light of GDPR and the reasons stated below, it is* ***imperative*** *that we ponder about the legality and ethical issues concerning automation of surveillance. This blog is for educational purposes only, and it used a publicly available dataset. It is your responsibility to make sure that your automated system complies with the law in your region.*

1. Dubious Conclusions

We do not know how a deep learning algorithm arrives at a conclusion. Even if the data feeding process is impeccable, there may be a lot of spurious hits. For instance, this AI profanity filter used by British cops kept removing pictures of sand dunes thinking they were obscene images. Techniques such as guided backpropagation can explain decisions to some extent, but we still have a long way to go.

2. Adversarial Attacks

Deep Learning systems are fragile. Adversarial attacks are akin to optical illusions for image classifiers. But the scary part is, a calculated unnoticeable perturbation can force a deep learning model to mis-classify. Using the same principle, researchers have been able to circumvent surveillance systems based on deep learning by using "adversarial glasses".

3. False positives

Another problem is, what do we do in the event of false positives. The severity of the issue depends on the application itself. For instance, a false positive on a border patrol system may be more significant than a garden monitoring system. There should be some amount of human intervention to avoid mishaps.

4. Similar faces

Sadly, your look is not as unique as your fingerprint. It is possible for two people (or more) to look very similar. Identical twins are one of the prime examples. It was reported that, Apple Face ID failed to distinguish two unrelated Chinese coworkers. This could make surveillance and identifying people harder.

5. Lack of diversity in datasets

Deep Learning algorithms are only as good as the data your provide it. Most popular datasets of human faces, only have samples of white people. While it may seem obvious to a child that humans can exist in various colors, Deep Learning algorithms are sort of dumb. In fact, Google got into trouble because it classified a black person incorrectly as a gorilla.

. . .

> ***About Nanonets:*** *Nanonets is building APIs to simplify deep learning for developers. Visit us at https://www.nanonets.com for more)*