## **Setup the Jetson Nano**

1. Create a Jetpack 4.4 Linux Image on an Sd Card of at least 64 GB

Follow the step here: <https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit>

1. To connect the board wirelessly to the internet use a Wifi Adapter such as [https://www.amazon.com/Edimax-EW-7811Un-N150-Wireless-Adapter/dp/B003MTTJOY/ref=as\_li\_ss\_tl](https://www.amazon.com/Edimax-EW-7811Un-N150-Wireless-Adapter/dp/B003MTTJOY/ref=as_li_ss_tl?keywords=Edimax+EW-7811Un&qid=1555076618&s=gateway&sr=8-1&linkCode=sl1&tag=thedailyack-20&linkId=1ed5ace0fd55f11b1d6bfe2b6780d699)

Install the necessary drivers to get the Wifi Adapter working.

1. Connect the Device to and HDMI display and get started.
2. Follow the <https://github.com/dusty-nv/jetson-inference#system-setup> tutorial to learn more about Object detection, classification, and image segmentation applications for the Jetson Nano.

## **Benchmarking Different Object Detection Models**

### Installing the required packages and dependencies:

|  |
| --- |
| $ sudo apt-get install libhdf5-serial-dev hdf5-tools $ sudo apt-get install python3-pip $ sudo apt-get install zlib1g-dev zip libjpeg8-dev libhdf5-dev $ pip3 install -U numpy grpcio absl-py py-cpuinfo psutil portpicker grpcio six mock requests gast h5py astor termcolor |

Install TensorFlow version 1.14 on the Jetson Nano to run TensorFlow supported machine learning models.

|  |
| --- |
| sudo apt-get install libhdf5-serial-dev hdf5-tools libhdf5-dev zlib1g-dev zip libjpeg8-dev liblapack-dev libblas-dev gfortran  sudo pip3 install -U pip  sudo pip3 install -U pip testresources setuptools numpy==1.16.1 future==0.17.1 mock==3.0.5 h5py==2.9.0 keras\_preprocessing==1.0.5 keras\_applications==1.0.8 gast==0.2.2 futures protobuf pybind11  # TF-1.15 $ sudo pip3 install --pre --extra-index-url https://developer.download.nvidia.com/compute/redist/jp/v44 'tensorflow<2' |

To run the benchmarking scripts we need to make sure the [Pillow](https://pillow.readthedocs.io/en/stable/) fork of the Python Imaging Library (PIL) is also installed along with some missing dependencies.

|  |
| --- |
| $ sudo apt-get install libfreetype6 libfreetype6-dev $ pip3 install Pillow |

We also need to install the object\_detection library which we’ll need for model optimization, starting with its dependencies.

|  |
| --- |
| $ sudo apt-get install protobuf-compiler python-pil python-lxml python-tk $ pip3 install --user Cython $ pip3 install --user contextlib2 $ pip3 install --user jupyter $ pip3 install --user matplotlib |

Now get the TensorFlow Models repository, which contains the TensorFlow Object Detection API and associated files, along with the COCO API repository.

|  |
| --- |
| $ git clone https://github.com/tensorflow/models.git $ git clone https://github.com/cocodataset/cocoapi.git |

Then build the COCO API and copy the pycocotools subdirectory into place inside the TensorFlow Models distribution.

|  |
| --- |
| $ sudo apt-get install python3-setuptools $ cd cocoapi/PythonAPI |

Since we’re using Python 3.x rather than Python 2.x we now need to make a quick edit to the Makefile as follows:

|  |
| --- |
| all: # install pycocotools locally python3 setup.py build\_ext --inplace rm -rf buildinstall: # install pycocotools to the Python site-packages python3 setup.py build\_ext install rm -rf build |

Then make pycocotools locally and copy them into the model’s directory

|  |
| --- |
| $ make $ cp -r pycocotools ~/models/research/ |

Build the Protobuf libraries.

|  |
| --- |
| $ cd ~/models/research $ protoc object\_detection/protos/\*.proto --python\_out=. |

Add the object\_detection directories to your PYTHONPATH,

|  |
| --- |
| $ export PYTHONPATH=$PYTHONPATH:/home/jnano/models/research:/home/jnano/models/research/slim |

Test the Installed Object Detection API

|  |
| --- |
| $ cd ~/models/research $ python3 object\_detection/builders/model\_builder\_test.py  .  .  . Ran 16 tests in 0.309s OK (skipped=1) |

Create a swap file on the Jetson Nano to help with larger models.

|  |
| --- |
| $ sudo fallocate -l 4G /var/swapfile $ sudo chmod 600 /var/swapfile $ sudo mkswap /var/swapfile $ sudo swapon /var/swapfile $ sudo bash -c 'echo "/var/swapfile swap swap defaults 0 0" >> /etc/fstab' |

## **Preparing a Model for Inference:**

**TensorRT**

The Jetpack SDK supports NVIDIA TensorRT which enables high-performance deep learning inference. Application using TensorRT accelerates its performance by 40 times faster

than CPU when real-time inference takes place. This allows optimizing Convolution Neural Nets (CNN) using many different deep learning frameworks, which can then be deployed to resource constraint embedded devices. TensorRT makes use of Nvidia’s CUDA platform to improve the inference for all major deep learning platform.

TensorRT scales down and quantizes the network parameter from FLOAT32 to INT8 and INT16 which reduces application latency required for many real-time embedded projects. The

batch size parameter can be specified in the model which determines the number of images on which object detection is performed. Multiple layers like ReLU, Convolution, Bias of the model is optimized by fusing them to create a layer fusion. Also, layers that share the same input are fused which is called layer aggregation. Tensor RT also helps to run certain operations such as FlattenConcat which are not supported by Nvidia’s Runtime framework.

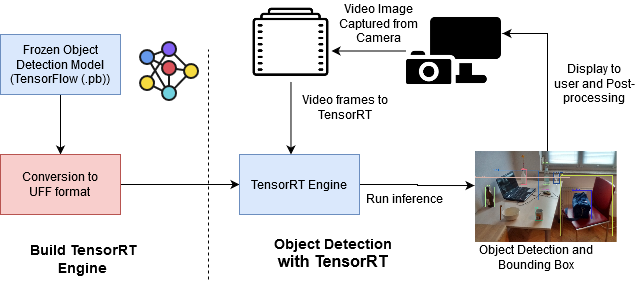


Fig. TensorRT Workflow.

In Fig, the entire workflow of the TensorRT and object detection is displayed. Any deep learning model in a framework such as TensorFlow, ONNX, PyTorch can be obtained. The frozen graph of the model (here TensorFlow model is depicted) is passed on the TensorRT build engine. The TensorRT build engine converts the model into Universal Frame Format (UFF). The engine then applies optimizations to the model such that it can take advantage of GPU based hardware and accelerate the performance and inference. The model can now detect the GPU and run code automatically using the GPU’s tensor core. Appropriate memory size is provided to the model by

the engine to perform and store the intermediate computations. A camera connected to the Jetson Nano captures live video images from the surrounding environment. The TensorRT

engine receives image frames from the captured video. It then performs pre-processing to the image such as shifting the order of the axis, normalizing, and flattening the image for faster

inferencing. A timer is used to measure the time taken for performing inference on the captured image. After performing the inference for object detection, the TensorRT engine will return the arrays consisting of bounding boxes for the located object, the confidence level for each object in the image, and the class to which the detected object belongs to. Finally, these arrays are used to overlay the image capture with the information about coordinates, confidence level, and class to the user. The user can take appropriate action.

### **Procedure for Converting a Model TensorRT**

* Download a pre-trained model from a repository such as Tensorflow Object detection Zoo. Following is the script to download pre-trained model weights and config files sourced from the TensorFlow models repository.

|  |
| --- |
| from tf\_trt\_models.detection import download\_detection\_model  config\_path, checkpoint\_path = download\_detection\_model('ssd\_inception\_v2\_coco') |

* Build the frozen graph from the downloaded model

|  |
| --- |
| from tf\_trt\_models.detection import build\_detection\_graph  frozen\_graph, input\_names, output\_names = build\_detection\_graph(  config=config\_path,  checkpoint=checkpoint\_path ) |

* We use the utility provided by Nvidia to Optimize the model with TensorRT. This utility converts a pre-trained object detection model such as a sdd\_mobilenet\_v2 TensorFlow object detection model to a UFF format and finally applies TensorRT optimizations.

|  |
| --- |
| import tensorflow.contrib.tensorrt as trt  trt\_graph = trt.create\_inference\_graph(  input\_graph\_def=frozen\_graph,  outputs=output\_names,  max\_batch\_size=1,  max\_workspace\_size\_bytes=1 << 25,  precision\_mode='FP16',  minimum\_segment\_size=50 ) |

* Setup the pre-trained model without any TensorRT optimizations by creating tensorflow sessions as follows:

|  |
| --- |
| input\_names = ['image\_tensor'] output\_names = ['detection\_boxes', 'detection\_classes', 'detection\_scores', 'num\_detections'] # Create session and load graph tf\_config = tf.ConfigProto() tf\_config.gpu\_options.allow\_growth = True tf\_sess = tf.Session(config=tf\_config) tf.import\_graph\_def(frozen\_graph, name='')  tf\_input = tf\_sess.graph.get\_tensor\_by\_name(input\_names[0] + ':0') tf\_scores = tf\_sess.graph.get\_tensor\_by\_name('detection\_scores:0') tf\_boxes = tf\_sess.graph.get\_tensor\_by\_name('detection\_boxes:0') tf\_classes = tf\_sess.graph.get\_tensor\_by\_name('detection\_classes:0') tf\_num\_detections = tf\_sess.graph.get\_tensor\_by\_name('num\_detections:0') |

* Run model repeated to benchmark the performance of object detection inference inference

|  |
| --- |
| import time times = [] for i in range(20):  start\_time = time.time()  scores, boxes, classes, num\_detections = tf\_sess.run([tf\_scores, tf\_boxes, tf\_classes, tf\_num\_detections], feed\_dict={  tf\_input: image[None, ...]  })   delta = (time.time() - start\_time)  times.append(delta) mean\_delta = np.array(times).mean() fps = 1/mean\_delta print('average(sec):{},fps:{}'.format(mean\_delta,fps)) |

### **Benchmarking TensorRt optimized model**

* Load the images and TensorRT optimized model

|  |
| --- |
| input\_names = ['image\_tensor'] IMAGE\_PATH = "./data/dogs.jpg"  # The TensorRT inference graph file downloaded from Colab or your local machine. pb\_fname = "./model/trt\_graph.pb"  def get\_frozen\_graph(graph\_file):  """Read Frozen Graph file from disk."""  with tf.gfile.FastGFile(graph\_file, "rb") as f:  graph\_def = tf.GraphDef()  graph\_def.ParseFromString(f.read())  return graph\_def  trt\_graph = get\_frozen\_graph(pb\_fname) |

* Create session and load graph

|  |
| --- |
| # Create session and load graph tf\_config = tf.ConfigProto() tf\_config.gpu\_options.allow\_growth = True tf\_sess = tf.Session(config=tf\_config) tf.import\_graph\_def(trt\_graph, name='') |

|  |
| --- |
| tf\_input = tf\_sess.graph.get\_tensor\_by\_name(input\_names[0] + ':0') tf\_scores = tf\_sess.graph.get\_tensor\_by\_name('detection\_scores:0') tf\_boxes = tf\_sess.graph.get\_tensor\_by\_name('detection\_boxes:0') tf\_classes = tf\_sess.graph.get\_tensor\_by\_name('detection\_classes:0') tf\_num\_detections = tf\_sess.graph.get\_tensor\_by\_name('num\_detections:0') |

* Run network on Image

|  |
| --- |
| scores, boxes, classes, num\_detections = tf\_sess.run([tf\_scores, tf\_boxes, tf\_classes, tf\_num\_detections], feed\_dict={  tf\_input: image[None, ...] }) boxes = boxes[0] # index by 0 to remove batch dimension scores = scores[0] classes = classes[0] num\_detections = int(num\_detections[0]) |

* Run model repeated to benchmark the performance of object detection inference inference

|  |
| --- |
| import time times = []  for i in range(20):  start\_time = time.time()  scores, boxes, classes, num\_detections = tf\_sess.run([tf\_scores, tf\_boxes, tf\_classes, tf\_num\_detections], feed\_dict={  tf\_input: image[None, ...]  })   delta = (time.time() - start\_time)  times.append(delta) mean\_delta = np.array(times).mean() fps = 1/mean\_delta print('average(sec):{},fps:{}'.format(mean\_delta, fps)) |

Refer to the Benchmarking code for more information.

## **Object Detection Using ZED camera**

Algorithm for developing Object detection model is as

Follows:

A. Python packages ‘jetson.inference’ and ‘jetsons.utils’ are imported which contains classes for object detection and camera capture.

B. The appropriate object detection network is loaded i.e. the 91-class SSD-Mobilenet-v2. The classification threshold of 0.5 is set.

C. The camera stream object is created for the attached ZED Camera via USB.

D. An OpenGL display loop is created to view the captured video frames in real-time.

E. The images are captured from the camera and passed on the Object detection network.

F. The detections obtained can be printed out and also displayed to the user with bounding boxes and labels in real-time.

G. Step E and F is repeated continuously to perform real-time object detection until the user quits.

Refer to the code for more information