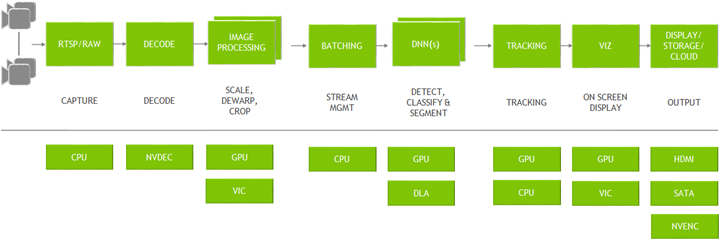
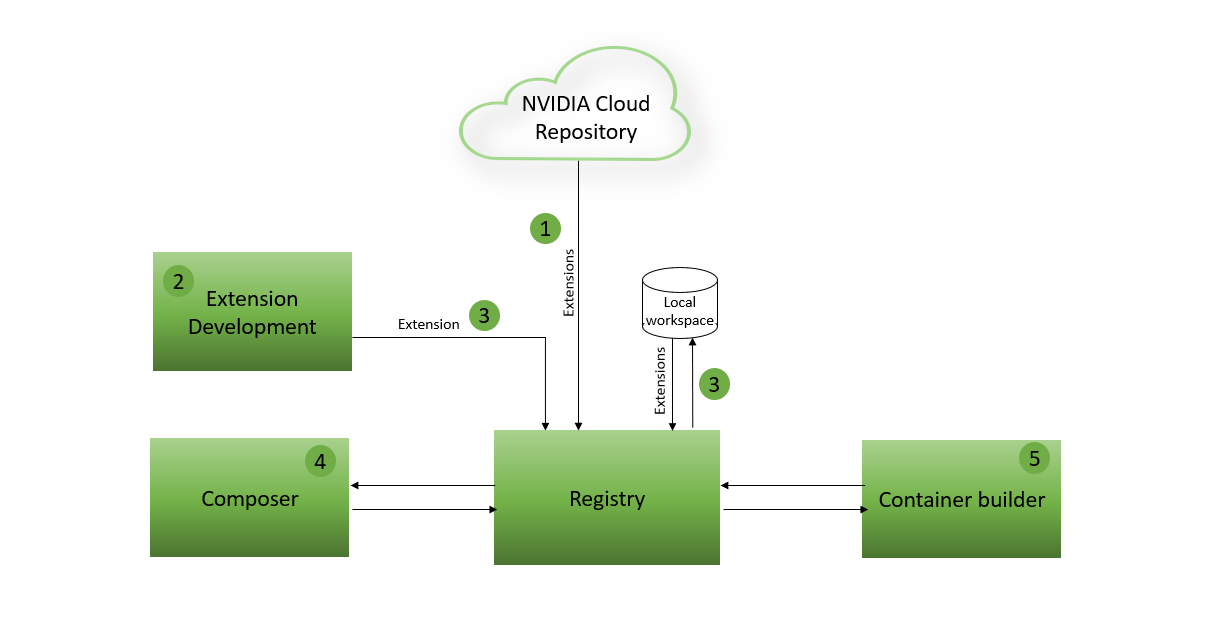
# DeepStream Notes (tutorial)

*Notes from* [*NVIDIA DeepStream development with Microsoft Azure - Learn | Microsoft Docs*](https://docs.microsoft.com/en-us/learn/paths/nvidia-deepstream-development-with-microsoft-azure/)

The DeepStream SDK allows for an end-to-end deployment of IVA (intelligent video analytics) solutions, all the way from raw video footage to inferenced output – this process includes decoding, image processing, inference, and more. The graphs which the video data flows through can be containerized and deployed remotely using services such as Azure IoT Hub and Azure Container Registry. Custom models, not just pre-trained models, can also be used. NVIDIA Graph composer is the graphical tool used to create these graphs using drag and drop.

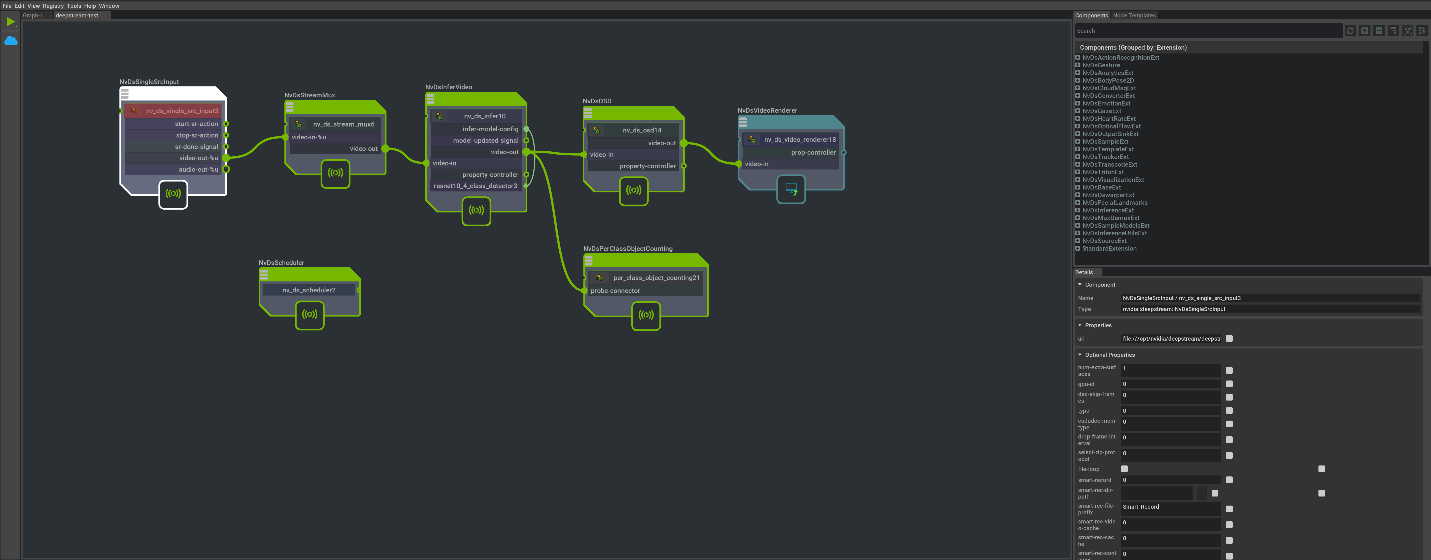


Additional extensions can add more components, which serve as “nodes” in the graph, to the NVIDIA Graph Composer, which can be dragged and dropped onto the graph. These extensions are obtained by connecting to NVIDIA’s cloud repository. They are then added to a registry which allows for them to be used in the graph composer. Those graphs then go through the registry and become containers at the container builder.



While in the graph composer, you can add components by dragging their names from the “components” sidebar at the right (you have to expand their groups first). The names of these components are displayed as headers, and at the left and right edges of these components are handles that can be connected between components to build pipelines: for example, linking a video input component output handle to a stream mux component (which performs batching of frames extracted from the video) input handle.

1. To edit the individual components of the graph, you can click on one of the horizontal slices (typically the first) of the component you want to edit and then manually edit their properties. For example, you might click on the first slice of the NvDsSignleSrcInput component and then edit the uri property to change the URI of the input stream.

This is what the above components do:

1. This application takes a video source as input by using NvDsSingleSrcInput
2. This video is passed into NvDsStreamMux (which could technically process multiple video inputs). The output of NvDsStreamMux provides a frame from each video input (batching).
3. These frames are then sent for processing in NvDsInferVideo. Custom model information can be passed in via a config text file parameter in this component.
4. The inference results of NvDsInferVideo are passed to both NvDsPerClassObjectCounting, to display a count for each detected class, and NvsOSD, which generates the on-screen detections with bounding boxes.
5. The boxes are displayed by the final connection to NvDsVideoRenderer.

Once this graph is created, you will want to make it into a containerized workload so that it can be run on the edge. Before doing this, you have to provide credentials for the NVIDIA GPU Containers, or NGC, service to Docker – NGC is NVIDIA’s official repository for distributing GPU accelerated containers (essentially containerized workloads which utilize GPU for ML computations), and Docker is a container distribution platform for getting containers to edge devices.

By using the container\_builder command and passing in some config files, you can create the container – specifically, you pass in the config file of the container builder (this includes a reference to the graph file itself) as well as the config file for the target graph, which basically just signifies the package versions to be used in the container when run (such as CUDA version and TensorRT version). When you run the containerized workload, an instance of a container image (whose name is input) is started and is given access to all GPUs as well as the X11 socket (for displaying the inference results in a GUI window). A key thing to understand is that an *image* is like the blueprint of the container that gets built, and the *container* itself is just an instance of this image. To view available images in docker, do sudo docker images.

These images can be tagged and pushed to container registries such as Azure Container Registry using Docker. From there, they can be pushed as modules to the Azure IoT Edge and to edge devices from there. These modules are then run and do inference.

\*\*\*

NOTE: This documentation requires Ubuntu 18.04 running on an x86-64 machine with NVIDIA graphics cards.

## Installing Package Dependencies & DeepStream SDK

[Link to tutorial](https://docs.microsoft.com/en-us/learn/modules/setup-configure-nvidia-deepstream-development/3-install-nvidia-deepstream-dependencies-sdk)

1. Run the following command to install these dependency packages needed for DeepStream:
   1. sudo apt install \

libssl1.0.0 \

libgstreamer1.0-0 \

gstreamer1.0-tools \

gstreamer1.0-plugins-good \

gstreamer1.0-plugins-bad \

gstreamer1.0-plugins-ugly \

gstreamer1.0-libav \

libgstrtspserver-1.0-0 \

libjansson4 \

gcc \

make \

git \

python3

1. Download NVIDIA drivers by doing sudo apt install nvidia-driver-470.
2. Run the following commands to install CUDA Toolkit 11.4:
   1. wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86\_64/cuda-ubuntu1804.pin
   2. sudo mv cuda-ubuntu1804.pin /etc/apt/preferences.d/cuda-repository-pin-600
   3. wget https://developer.download.nvidia.com/compute/cuda/11.4.1/local\_installers/cuda-repo-ubuntu1804-11-4-local\_11.4.1-470.57.02-1\_amd64.deb
   4. sudo dpkg -i cuda-repo-ubuntu1804-11-4-local\_11.4.1-470.57.02-1\_amd64.deb
   5. sudo apt-key add /var/cuda-repo-ubuntu1804-11-4-local/7fa2af80.pub
   6. sudo apt-get update
   7. sudo apt install nvidia-cuda-dev
   8. sudo apt upgrade nvidia-cuda-toolkit
   9. sudo apt-get update
   10. sudo apt-get -y install cuda
3. Run the following steps to install TensorRT:
   1. Run these commands:
      1. echo "deb https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86\_64 /" | sudo tee /etc/apt/sources.list.d/cuda-repo.list
      2. wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86\_64/7fa2af80.pub
      3. sudo apt-key add 7fa2af80.pub
      4. sudo apt-get update
   2. Open a web browser and download the [TensorRT 8.0.1 GA for Ubuntu 18.04 and CUDA 11.3 DEB local repo package](https://developer.nvidia.com/compute/machine-learning/tensorrt/secure/8.0.1/local_repos/nv-tensorrt-repo-ubuntu1804-cuda11.3-trt8.0.1.6-ga-20210626_1-1_amd64.deb). This will put the installation package into the Downloads folder.
   3. Install the package with the following commands:
      1. cd ~/Downloads
      2. sudo dpkg -i nv-tensorrt-repo-ubuntu1804-cuda11.3-trt8.0.1.6-ga-20210626\_1-1\_amd64
      3. sudo apt-key add /var/nv-tensorrt-repo-ubuntu1804-cuda11.3-trt8.0.1.6-ga-20210626/7fa2af80.pub
      4. sudo apt-get update
      5. sudo apt-get install \

libnvinfer8=8.0.1-1+cuda11.3 \

libnvinfer-plugin8=8.0.1-1+cuda11.3 \

libnvparsers8=8.0.1-1+cuda11.3 \

libnvonnxparsers8=8.0.1-1+cuda11.3 \

libnvinfer-bin=8.0.1-1+cuda11.3 \

libnvinfer-dev=8.0.1-1+cuda11.3 \

libnvinfer-plugin-dev=8.0.1-1+cuda11.3 \

libnvparsers-dev=8.0.1-1+cuda11.3 \

libnvonnxparsers-dev=8.0.1-1+cuda11.3 \

libnvinfer-samples=8.0.1-1+cuda11.3 \

libnvinfer-doc=8.0.1-1+cuda11.3

1. Run the following commands:
   1. sudo ln -s /usr/bin/python3 /usr/bin/python
   2. cd ~
   3. git clone https://github.com/edenhill/librdkafka.git
   4. cd librdkafka
   5. git reset --hard 7101c2310341ab3f4675fc565f64f0967e135a6a
   6. ./configure
   7. make
   8. sudo make install
   9. sudo mkdir -p /opt/nvidia/deepstream/deepstream-6.0/lib
   10. sudo cp /usr/local/lib/librdkafka\* /opt/nvidia/deepstream/deepstream-6.0/lib
2. Run 6a, then run sudo apt install libgstrtspserver-1.0-0 and sudo apt install libgstreamer-plugins-base1.0-dev before doing the installation with sudo dpkg -i deepstream-6.0\_6.0.0-1\_amd64
3. Install DeepStream SDK by going to [NVIDIA DeepStream - Version 6.0.0-1 Download](https://developer.nvidia.com/deepstream-6.0_6.0.0-1_amd64deb). Then, tun the following commands:
   1. sudo apt install libgstrtspserver-1.0-0
   2. sudo apt install libgstreamer-plugins-base1.0-dev
   3. cd ~/Downloads
   4. sudo apt-get install ./deepstream-6.0\_6.0.0-1\_amd64

## Install Graph Composer

[Link to tutorial](https://docs.microsoft.com/en-us/learn/modules/introduction-nvidia-deepstream-graph-composer-azure/3-install-nvidia-deepstream-graph-composer)

1. Run the following commands to install the Docker runtime:
   1. curl -fsSL https://get.docker.com -o get-docker.sh
   2. sudo sh get-docker.sh
2. Run the following commands to install some Graph Composer dependencies:
   1. sudo apt install \

mesa-utils \

vulkan-utils \

mesa-vulkan-drivers

1. Go to <https://developer.nvidia.com/graph_composer-1.0.0_x86_64deb> to install Graph Composer. Then, run the following commands, substituting in the name of the downloaded package:
   1. cd ~/Downloads
   2. sudo dpkg -i <name of downloaded package>
2. To install demo reference graphs, go to [Graph Composer reference examples](https://developer.nvidia.com/deepstream-reference-graphs-6.0deb) and install downloaded packages by doing the following, substituting in the name of the downloaded packages:
   1. cd ~/Downloads
   2. sudo dpkg -i <name of downloaded package>

## Get Graph Composer Extensions + Use Composer and Demo Graph

[Link to tutorial](https://docs.microsoft.com/en-us/learn/modules/introduction-nvidia-deepstream-graph-composer-azure/4-run-nvidia-deepstream-graph-composer-reference-application)

1. Run this command to sync with NVIDIA’s public Cloud Repository to get extensions for the base Graph Composer tool (it might take a few minutes to run, it took me 5): registry repo sync -n ngc-public
2. Verify that these extensions have installed correctly by running registry extn list. Note that these extensions have prefixes mapping to NVIDIA-optimized GStreamer plugins – for example the NvDsInferenceExt extensions provides a way to interact with Gst-nvinfer.
3. Launch the graph composer application with sudo composer. To add new components, drag them from the right-hand side of the screen onto the editor space. These components can be linked together with handles, and their properties can be altered by clicking on their slices and changing parameters in the right-hand side.
4. Launch the first demo graph by selecting **File** > **Open Graph** at the top of the screen and navigating to the */opt/nvidia/deepstream/deepstream/reference\_graphs/deepstream-test1* path and selecting the *deepstream-test1.yaml* file and then clicking **Okay**. Now that we’ve launched the graph, we can edit some of its properties. Click on the top slice of the first component, called *NvDsSingleSrcInput* – this will cause for a window to open at the right-hand side of the screen, detailing the various properties of this component. We will be changing the **URI**, or the input stream source, of this pipeline to *rtsp://wowzaec2demo.streamlock.net/vod/mp4:BigBuckBunny\_115k.mp4* – in order to set the URI to an RSTP source, we will need to set the **type** parameter equal to 4. Now, do ctrl + s to save the graph.
5. Step 6, when run, generates output based off the default input stream of a road – this is because the graph composer application doesn’t have the permission necessary to save the new graph with the Big Buck Bunny video source. To give it the permission, you have to do sudo composer when launching the graph composer, and you have to save the file in the graph composer by doing ctrl + s. You can either run the command, or you can press the green “run” button in the graph composer itself. This will run the graph just as it runs when you run the command. Pressing the run button automatically saves (or attempts to save) the graph – it won’t be able to save the graph in certain locations if the graph composer was not launched with sudo.
6. To run this graph, you can do one of two things: you can either just press the green “run” button at the top left corner of the screen, or you can run the following command, which runs the execute\_graph.sh function and takes in the graph config file, an optional parameters file, and a required parameter for the target device info:
7. cd /opt/nvidia/deepstream/deepstream/reference\_graphs/deepstream-test1
8. /opt/nvidia/graph-composer/execute\_graph.sh deepstream-test1.yaml parameters.yaml -d /opt/nvidia/graph-composer/config/target\_x86\_64\_cuda\_11\_4.yaml

## Containerize Graph from Composer

[Link to tutorial](https://docs.microsoft.com/en-us/learn/modules/introduction-nvidia-deepstream-graph-composer-azure/5-package-deepstream-graph-composer-app-containerized-workload)

1. Do the following to set up the NVIDIA Container Service:
   1. Set up an account with the NGC service by creating a new account / signing in at the [NVIDIA NGC sign-in page](https://ngc.nvidia.com/signin). Then, select your username at the top right corner and click Setup, click Get API Key, and then click Generate API Key.
   2. API key is to be stored in [Azure Key Vault](https://docs.microsoft.com/en-us/azure/key-vault/secrets/quick-create-portal) as a secret under the name Nvidia-NGC-API-Key (NGC stands for Nvidia GPU Containers service, which is NVIDIA’s official repository for the distribution of GPU-accelerated containers) in the DeepStream-Project-Vault Key Vault. You can store the API key in an alternate place if you have one.
   3. Now, give these credentials to the Docker runtime by doing sudo docker login nvcr.io and provide username as $oauthtoken and password as your generated API key. You should get a Login Succeeded message.
2. To use the container builder tool to build your graph into a container, run the following commands:
   1. cd /opt/nvidia/deepstream/deepstream-6.0/reference\_graphs/deepstream-test1
   2. sudo container\_builder -c ds\_test1\_container\_builder\_dgpu.yaml -d /opt/nvidia/graph-composer/config/target\_x86\_64\_cuda\_11\_4.yaml.

The -c parameter takes in the container builder config file, and the -d parameter takes in the target destination config. Step 1/9 might take a long time to run. This same thing could also be done by clicking the cloud icon in the NVIDIA Graph Composer GUI and providing these same config files.

1. Before running the container, we need to grant access to the X11 display environment by doing sudo xhost +
2. Before running the container, you need to get NVIDIA Container Toolkit. To do this, run the following commands:
   1. distribution=$(. /etc/os-release;echo $ID$VERSION\_ID) \

&& curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | sudo gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-keyring.gpg \

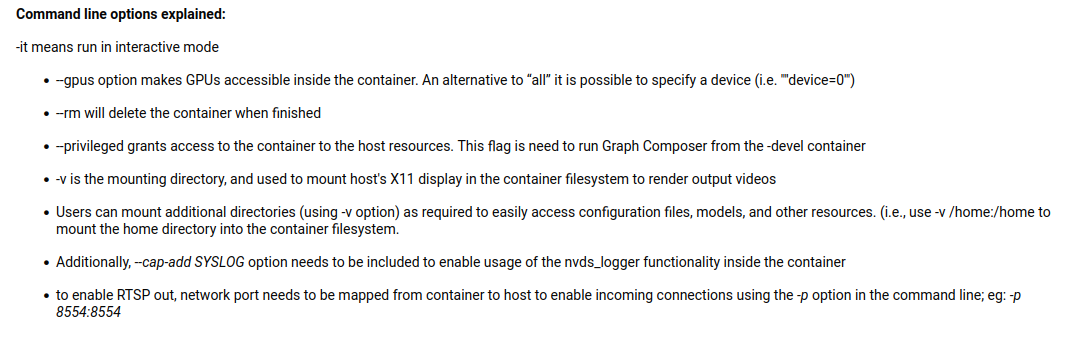
&& curl -s -L https://nvidia.github.io/libnvidia-container/experimental/$distribution/libnvidia-container.list | \

sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg] https://#g' | \

sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list

* 1. sudo apt update
  2. sudo apt-get install –y nvidia-docker2
  3. sudo systemctl restart docker

1. Once you do this, you can run the command sudo docker run -it --rm --gpus all -v /tmp/.X11-unix/:/tmp/.X11-unix/ -e DISPLAY=:0 deepstream\_test1\_dgpu to run the graph container. The last element of the command is just the image that is being run by docker.



NOTE: -e is used to set environment variables. Essentially, -e DISPLAY:=0 is just assigning / updating the DISPLAY environment variable to have the value 0 to change the display that object detection is shown on.

## Send Graph Container to Azure Container Registry

[Link to tutorial](https://docs.microsoft.com/en-us/learn/modules/introduction-nvidia-deepstream-graph-composer-azure/6-publish-deepstream-graph-composer-container-to-azure)

1. Create an azure container registry by following <https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-portal>
2. Go to your container registry and save the “Login Server” property’s value at the Overview page somewhere.
3. Then, go to the Access Keys tab at the left side of the screen and enable the option for Admin User. Then, make note of the username and password.
4. Now, log in to the container registry by running sudo docker login <login server>, providing the address obtained in step 2. Provide the username and password from step 3.
5. Before pushing the Docker image to the container registry, we have to tag it with the container registry name and the tag. Do this by running the following, swapping in the name of your login server: sudo docker tag deepstream\_test1\_dgpu <Login Server>/deepstream\_test1\_dgpu:v1
6. Push the Graph Composer container image to the container registry with sudo docker push <Login Server>/deepstream\_test1\_dgpu:v1. This will take a **very long time**, especially with low upload speed. For this example, it took 3 hours with 1 MBps upload speed.
7. Once the image is pushed, you can view it in the Azure Container Registry by clicking on the Repositories option at the left side of the screen in the Azure Container Registry viewing area website.
8. Before running this container, run sudo xhost + to allow for display access. Then, run the following command:

sudo docker run -it --rm --gpus all -v /tmp/.X11-unix/:/tmp/.X11-unix/ -e DISPLAY=:0 <Login server>/deepstream\_test1\_dgpu:v1

**NOTE: if you ever get an error saying, “failed to set GStreamer pipleine to PLAYING”, just run** sudo xhost + **again.**

# DeepStream Custom Model

## Run YOLO model (custom)

[Link to tutorial](https://github.com/marcoslucianops/DeepStream-Yolo)

NOTE: This part is mandatory for the next part (putting model into composer)

1. Download the DeepStream-Yolo repository necessary for doing this with the following commands:
   1. git clone https://github.com/marcoslucianops/DeepStream-Yolo.git
   2. cd DeepStream-Yolo
2. Get demo YOLOv3 model config and weight files by copying the yolov3.cfg and yolov3.weights files from /opt/nvidia/deepstream/deepstream-6.0/sources/objectDetector\_Yolo into the DeepStream-Yolo folder.
3. Compile the library we are using by running the following: CUDA\_VER=11.4 make -C nvdsinfer\_custom\_impl\_Yolo
4. To plug in the custom yolo model, run sudo gedit config\_infer\_primary.txt and enter in the model weight and cfg files at the custom-network-config and model-file parameters.
5. Run the custom model with deepstream-app -c deepstream\_app\_config.txt

## Convert YOLOv5 model into .wts format

[Link to tutorial](https://github.com/marcoslucianops/DeepStream-Yolo/blob/master/docs/YOLOv5.md)

NOTE: this takes in the .weights/.wts and .cfg files from training a yolo model.

First, you have to train a model in YOLOv5 format, as with [this tutorial](https://blog.roboflow.com/how-to-train-yolov5-on-a-custom-dataset/) which creates a model with the .pt format. Then you have to download the model file.

1. Now that we have the .pt file, we can begin to convert it. To start, we will need to get the YoloV5 directory. Get the terminal into the home directory and then run the following:

git clone <https://github.com/ultralytics/yolov5.git>

cd yolov5

pip3 install -r requirements.txt

Now, we will have to move the .pt model file as well as the file located at ~/DeepStream-Yolo/utils/gen\_wts\_yoloV5.py into the yolov5 folder.

1. Now, we will run the command python3 gen\_wts\_yoloV5.py -w weights\_file\_name.pt while inside the yolov5 folder (be sure to replace the weights file name). If you want to change the size of inference input images for the model (default is 640x640), you can simply add -s HEIGHT WIDTH as part of the command. This will generate a wts and cfg file in the current folder.

## Plug model in to Graph Composer

1. First, you want to move a few files into a single folder which will represent the current DeepStream project. In this folder, you will need the following:
   1. The .wts file and .cfg file from the model, generated above
   2. A labels.txt where each line represents one class’s name (order matters)
   3. The config\_nms.txt file, copied from ~/DeepStream-Yolo/config\_nms.txt
   4. The graph composer .yaml file, which will be created later on.

Create a config txt file for nvinfer that contains sample code copy pasted from here: <https://pastebin.com/SjD9ZJVQ>. This will be used to provide input. I have saved this config file to ~/Documents/deepstream\_custom\_nvinfer\_config.txt

1. Swap out the following parameters: (parameters explained at <https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_plugin_gst-nvinfer.html#gst-nvinfer-file-configuration-specifications>)
   1. Custom-network-config is to be set to the .cfg file for the yolo model
   2. Model-file is to be set to the .weights/.wts file for the yolo model
   3. Model-engine-file is optional, but if you have it then set it to the model engine
   4. Labelfile-path is to be set to the labels text file path
   5. Custom-lib-path should just be changed to reflect the place where DeepStream-Yolo is saved for you (just the prefix of the path)
   6. Num-detected-classes should be changed to the number of classes the model has
   7. Create a graph composer graph at the folder created in step 1 and create an NvDsInferVideo node. Click on the first slice and set the config-file-path parameter to the path of the document created above.
2. The graph can then be run by pressing the run button.

## Container Builder Notes

NOTE: These notes refer to the container builder config file required to build the container of a graph, specifically from the container builder config file provided for the DeepStream test 1 demo.

1. From what I have gathered, parameters.yaml is a file that is created manually to override specific parts of the graph composer config file. It is optional.
2. The manifest file mentioned is generated by DeepStream when you run a graph in the Composer GUI and check the box to generate a fresh manifest file. It is stored in the /tmp folder so you will have to search for it there and put it into the same folder as the rest of your graph-related documents and then reference its file name in the container builder yaml file.
3. I tried doing container builder on a prebuilt model graph with the parameters.yaml file as well. It still failed for unknown reasons.

## DeepStream Issues list

1. How does one create a container builder config file? Is there a proper template for doing this? Are there any required parts of the file that it won’t work without? When creating a container while using a custom model, do any custom model files have to be moved inside the container image?
2. What is the proper way to put a custom model into DeepStream other than just editing the config file? Is there a template to easily implement, say, a custom ONNX model? How can I access custom model implementations for other file types than YOLO?

## Moving on from here

To progress past the point I have reached, you would have to figure out how to containerize a custom graph composer graph, as it isn’t working for me. I tried doing it by editing the container builder config file that is needed to run it, putting in my own graph names, but when I run a graph built by this container builder, it throws an error related to GStreamer / gxe. Info about the config file and various options regarding it can be found here: [Container Builder — DeepStream 6.1 Release documentation (nvidia.com)](https://docs.nvidia.com/metropolis/deepstream/dev-guide/graphtools-docs/docs/text/GraphComposer_Container_Builder.html). Custom graphs with prebuilt models also fail to properly containerize for unknown reasons – perhaps this is a localized issue to my PC, so it might work on others.

To push images from the container registry onto IoT Edge devices in the Azure IoT Hub, you can follow steps here: [NVIDIA DeepStream embedded device deployment with Azure - Learn | Microsoft Docs](https://docs.microsoft.com/en-us/learn/modules/nvidia-deepstream-embedded-device-deployment-azure/)

# Unused (not needed)

ONNX to TensorRT engine is not needed because DeepStream automatically generates an engine file when provided with an ONNX model file in the nvinfer config.

## Convert ONNX to TensorRT (demo)

[Link to tutorial](https://forums.developer.nvidia.com/t/creating-a-tensorrt-engine-from-an-onnx-model-file/106979)

1. First, clone the TensorRT repository by doing git clone <https://github.com/NVIDIA/TensorRT.git> and enter it by doing cd TensorRT. Enter the specific repository for ONNX to TensorRT by doing cd ~/TensorRT/samples/python/yolov3\_onnx
2. Once here, run the following commands:
   1. python3 –m pip install numpy==1.19.4
   2. python3 –m pip install protobuf==3.15.7
   3. python3 –m pip install onnx==1.9.0
   4. python3 –m pip install Pillow
   5. python3 –m pip install pycuda==2020.1
   6. python3 –m pip install nvidia-tensorrt
   7. python3 –m pip install nvidia-pyindex
   8. python3 -m pip install --upgrade PyYAML
   9. python3 -m pip install tqdm
   10. sudo apt-get install python3-libnvinfer
3. Now, we need to download some sample data. We will store it in a new folder ~/TensorRT/samples/python/yolov3\_onnx/downloaded\_data by doing mkdir downloaded\_data to create the directory. To download the data, run the following:

python3 ~/TensorRT/samples/python/downloader.py -d ~/TensorRT/samples/python/yolov3\_onnx/downloaded\_data -f ~/TensorRT/samples/python/yolov3\_onnx/download.yml

1. Now, we can finally run the ONNX to TensorRT command with Python. To do this, we have to turn our downloaded YoloV3 model into a ONNX model by doing python3 yolov3\_to\_onnx.py -d downloaded\_data and then turn this ONNX model into TensorRT Engine by doing python3 onnx\_to\_tensorrt.py -d downloaded\_data
2. Now, in the yolov3\_onnx directory, we can see that an onnx file and a trt file have been generated.

## Convert ONNX to TensorRT (custom model) (OPTIONAL)

[Link to tutorial](https://docs.nvidia.com/deeplearning/tensorrt/quick-start-guide/index.html#convert-onnx-engine)

1. First, clone the TensorRT repository by doing git clone <https://github.com/NVIDIA/TensorRT.git> and enter it by doing cd TensorRT. Enter the specific repository for ONNX to TensorRT by doing cd ~/TensorRT/samples/python/yolov3\_onnx
2. Once here, run the following commands:
   1. python3 –m pip install numpy==1.19.4
   2. python3 –m pip install protobuf==3.15.7
   3. python3 –m pip install onnx==1.9.0
   4. python3 –m pip install Pillow
   5. python3 –m pip install pycuda==2020.1
   6. python3 –m pip install nvidia-tensorrt
   7. python3 –m pip install nvidia-pyindex
   8. python3 -m pip install --upgrade PyYAML
   9. python3 -m pip install tqdm
   10. sudo apt-get install python3-libnvinfer
3. To convert from ONNX to TensorRT .engine, we will take some code from the ONNX to TensorRT sample from NVIDIA and modify it slightly. I have put this code into a pastebin here: <https://pastebin.com/Jvx36n0L>. Copy this code into a notepad / text editor file titled something along the lines of Onnx-to-RT-build-engine.py (that is what I named it) - ***this file must be saved in the***

~/TensorRT/samples/python/yolov3\_onnx ***directory, since it calls locally defined packages***. Essentially, you edit the file with the ONNX file path and the Engine saving path each time you use it – I don’t know how to use argparse to make this all inputable from the command line, but you can add that yourself. This code will open the ONNX file and save it as a TensorRT engine file at the defined location – just be sure to replace the last two elements of list [1, 3, 608, 608] to whatever matches the dimensions of the model you are using. If you don’t have the batch size at 1, then you will run into errors when trying to turn the onnx file into the .engine format. **I have saved my file at** ~/TensorRT/samples/python/yolov3\_onnx/Onnx-to-RT-build-engine.py

# Unused (requires Jetson device)

These below steps refer to the Microsoft Learn tutorial used at the beginning of this documentation for setup of DeepStream.

## Configure DeepStream graphs to publish to AZ IoT Hub

[Link to tutorial](https://docs.microsoft.com/en-us/learn/modules/nvidia-deepstream-embedded-device-deployment-azure/3-configure-deepstream-graph-publish-data-azure-iot-hub)

1. Follow the first step to create an IoT Hub.
2. This step is just instructions, so we can move onwards.
3. For this step, follow normally, but note the components NvDsSampleProbeMessageMetaCreation and NvDsMsgConvBroker which work to send messages to the cloud. NvDsSampleProbeMessageMetaCreation transforms the metadata generated by the pipeline into a different form of metadata which is serialized by NvDsMsgConvBroker and sent to the cloud by a message broker protocol.
4. Follow the fourth step normally. Take note of how we change the msg-conv-config file to follow the azure edge protocol – this allows for usage with Azure IoT Edge

## Build and publish cross-platform container images

[Link to tutorial](https://docs.microsoft.com/en-us/learn/modules/nvidia-deepstream-embedded-device-deployment-azure/4-build-publish-cross-platform-deepstream-container-images)

1. Run the first step normally.
2. Run the second step normally.
3. For the third step, replace the command vi with gedit. This will make editing significantly easier. I am slightly unsure of what is done in this step, but it is necessary.
4. For the fourth step, do the same as before (replace vi with gedit) and follow the steps normally. This essentially sets the protocol for messaging to be compatible with azure edge and changes the format of the output object detection.
5. Run the fifth step normally. Note that the target graph format config file is now of arch64, which is the architecture of Jetson devices as well as Raspberry Pi 3