NVIDIA TRANSFER LEARNING TOOLKIT FOR INTELLIGENT VIDEO ANALYTICS

DU-09243-003 _v2.0 | August 2020

Getting Started Guide

TABLE OF CONTENTS

Chapter 1. Overview 1
Chapter 2. Transfer Learning Toolkit Requirements and Installation 8
2.1. Installation 9
2.1.1. Running the Transfer Learning Toolkit 9
2.1.2. Downloading the models 10
Chapter 3. Supported model architecture 12
Chapter 4. Purpose-built models 17
Chapter 5. Augmenting a dataset 20
5.1. Configuring the augmentor 21
5.1.1. Spatial augmentation config 23
5.1.1.1. Rotation config 24
5.1.1.2. Shear config 24
5.1.1.3. Flip config 24
5.1.1.4. Translation config 25
5.1.2. Color augmentation config 25
5.1.2.1. Hue saturation config 26
5.1.2.2. Brightness config 27
5.1.2.3. Contrast config 27
5.1.3. Dataset config 27
5.1.4. Blur config 28
5.2. Running the augmentor tool 29
Chapter 6. Preparing input data structure 31
6.1. Data input for classification 31
6.2. Data input for object detection 31
6.2.1. KITTI file format 32
6.2.2. Label files 32
6.2.3. Sequence mapping file 34
6.3. Conversion to TFRecords 34
6.3.1. Configuration file for dataset converter 35
6.3.2. Sample usage of the dataset converter tool 38
Chapter 7. Creating an experiment spec file 40
7.1. Specification file for classification 40
7.2. Specification file for DetectNet_v2 41
7.2.1. Model config 42
7.2.2. BBox ground truth generator 47
7.2.3. Post processor 49
7.2.4. Cost function 52
7.2.5. Trainer 53
7.2.6. Augmentation module 58
7.2.7. Configuring the evaluator 65

7.2.8. Dataloader....69
7.2.9. Specification file for inference....70
7.2.9.1. Inferencer....70
7.2.9.2. TLT_Config....74
7.2.9.3. Bbox handler....74
7.3. Specification file for FasterRCNN....78
7.3.1. Network config....82
7.3.2. Training Configuration....89
7.4. Specification file for SSD....104
7.4.1. Training config....104
7.4.2. Evaluation config....105
7.4.3. NMS config....105
7.4.4. Augmentation config....106
7.4.5. Dataset config....106
7.4.6. SSD config....106
7.5. Specification file for DSSD....110
7.5.1. Training config....112
7.5.2. Evaluation config....113
7.5.3. NMS config....113
7.5.4. Augmentation config....114
7.5.5. Dataset config....114
7.5.6. DSSD config....114
7.6. Specification file for RetinaNet....121
7.6.1. Training config....123
7.6.2. Evaluation config....124
7.6.3. NMS config....124
7.6.4. Augmentation config....125
7.6.5. Dataset config....125
7.6.6. RetinaNet config....125
7.7. Specification file for YOLOv3....130
7.7.1. Training config....132
7.7.2. Evaluation config....133
7.7.3. NMS config....133
7.7.4. Augmentation config....134
7.7.5. Dataset config....134
7.7.6. YOLOv3 config....134
7.8. Specification file for RetinaNet....137
7.8.1. Training config....138
7.8.2. Evaluation config....140
7.8.3. NMS config....140
7.8.4. Augmentation config....141
7.8.5. Dataset config....141
7.9. Specification file for MaskRCNN....141

7.9.1. MaskRCNN config.. 149
7.9.2. Data config... 151
Chapter 8. Training the model.. 153
8.1. Quantization Aware Training... 153
8.2. Automatic Mixed Precision... 154
8.3. Training a classification model.. 154
8.4. Training a DetectNet_v2 model.. 155
8.5. Training a FasterRCNN model...156
8.6. Training an SSD model.. 157
8.7. Training a DSSD model..157
8.8. Training a YOLOv3 model..158
8.9. Training a RetinaNet model.. 158
8.10. Training a MaskRCNN model... 159
Chapter 9. Evaluating the model... 160
9.1. Evaluating a classification model.. 161
9.2. Evaluating a DetectNet_v2 model..161
9.3. Evaluating a FasterRCNN model...163
9.4. Evaluating an SSD model.. 164
9.5. Evaluating a DSSD model.. 164
9.6. Evaluating a YOLOv3 model..165
9.7. Evaluating a RetinaNet model.. 165
9.8. Evaluating a MaskRCNN model...165
Chapter 10. Using inference on a model... 166
10.1. Running inference on a classification model.. 166
10.2. Running inference on a DetectNet_v2 model..167
10.3. Running inference on a FasterRCNN model... 167
10.4. Running inference on an SSD model... 168
10.5. Running inference on a DSSD model... 169
10.6. Running inference on a YOLOv3 model...169
10.7. Running inference on a RetinaNet model... 170
10.8. Running inference on a MaskRCNN model..171
Chapter 11. Pruning the model..173
Chapter 12. Exporting the model... 175
Chapter 13. Deploying to DeepStream.. 181
13.1. TensorRT Open Source Software (OSS)..182
13.2. Generating an engine using tlt-converter... 185
13.3. Integrating the model to DeepStream... 188
13.3.1. Integrating a Classification model...189
13.3.2. Integrating a DetectNet_v2 model... 191
13.3.3. Integrating an SSD model..194
13.3.4. Integrating a FasterRCNN model... 197
13.3.5. Integrating a YOLOv3 model.. 200
13.3.6. Integrating a DSSD model..203

13.3.7. Integrating a RetinaNet model..206
13.3.8. Integrating Purpose-built models..208
13.3.9. Integrating a MaskRCNN model.. 210

Chapter 1. OVERVIEW

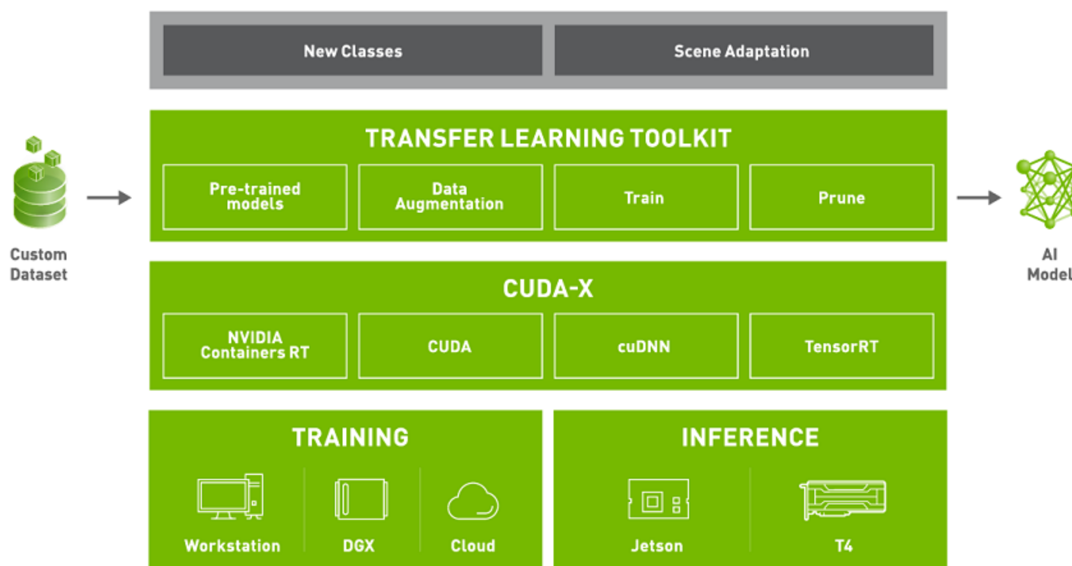
"**Transfer learning**" is the process of transferring learned features from one application to another. It is a commonly used training technique where you use a model trained on one task and re-train to use it on a different task. It works surprisingly well since a lot of the early layers in a neural network are primarily used to identify outlines, curves, and other features in an image. This can easily be transferred to other domains. An example would be if you want to identify different breeds of dogs, but you only have few images per breed. So, what you can do is take a model that was trained on recognizing animals and apply transfer learning to train the model to recognize breeds of dogs with your own images of dogs. Features to recognize animals can be transferred over for your use case.

Transfer learning is very useful when data collection and annotation is difficult or expensive. With transfer learning, less data is required to train accurately as compared to if you were to train from scratch. This reduces the overall training time and cost. Because you are running over a smaller dataset, you can train quicker and minimize the cost of collecting and annotating data. To learn more about transfer learning, read this blog.

NVIDIA Transfer Learning Toolkit

NVIDIA Transfer Learning Toolkit (TLT) is a simple, easy-to-use training toolkit that requires minimal to zero coding to create vision AI models using the user's own data. Using TLT users can transfer learning from NVIDIA pre-trained models to your own model. Users can add new classes to an existing pre-trained model, or they can re-train the model to adapt to their use case. Users can use model pruning capability to reduce the overall size of the model.

Getting started with TLT is very easy. Training AI models using TLT does not require expertise in AI or deep learning. Users with basic knowledge of deep learning, can get started building their own custom models using a simple spec file and pre-trained model.



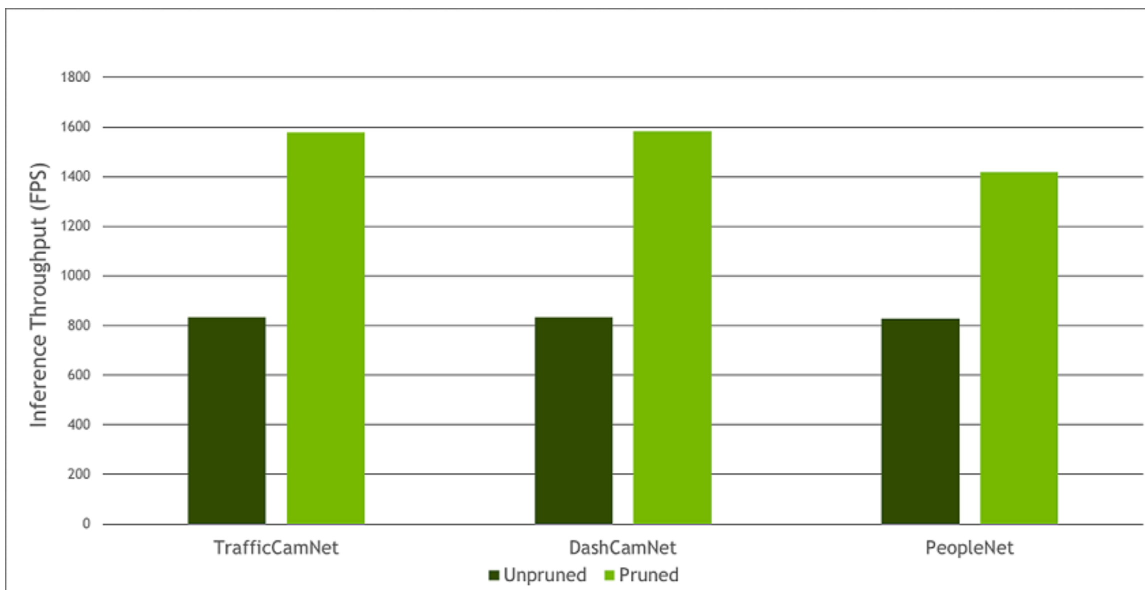
Transfer Learning Toolkit is a simplified toolkit where users start with our pre-trained models and their own custom dataset. Transfer learning toolkit is available in a docker container that can be downloaded from NGC, NVIDIA GPU cloud registry. The container comes with all the dependencies required to train. For more information about TLT requirements and installation, see TLT requirements and installation. The pre-trained models can also be downloaded from NGC. The toolkit consists of a command line interface (CLI) that can be run from the Jupyter notebooks, which are packaged inside the docker container. TLT consists of a few simple commands such as data augmentation, training, pruning and model export. The output of TLT is a trained model that can be deployed for inference on NVIDIA edge devices usingDeepStream andTensorRT.

TLT builds on top of CUDA-X stack which contains all the lower level NVIDIA libraries. These are NVIDIA container RT for GPU acceleration from within the containers, CUDA and cuDNN for a lot of DL operations and TensorRT for generating TensorRT compatible models for deployment. TensorRT is NVIDIA's inference runtime which optimizes the runtime model based on the targeted hardware. The models that are generated with TLT are completely accelerated with TensorRT, so users can expect maximum inference performance without any extra effort.

TLT is designed to run on x86 systems with a NVIDIA GPU such as a GPU-powered workstation or a DGX system or can be run in any cloud with a NVIDIA GPU. For inference, models can be deployed on any edge device such as the embedded Jetson platform or in data center GPUs like T4.

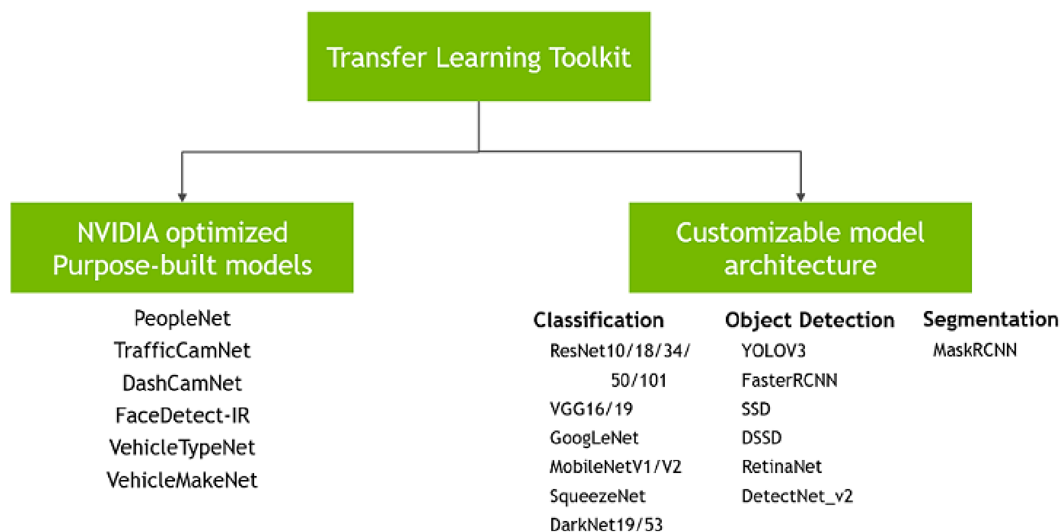
Model pruning is one of the key differentiators for TLT. Pruning means removing nodes in the neural network which contribute less to the overall accuracy of the model. With pruning, users can reduce the overall size of the model significantly which results in much lower memory footprint and higher inference throughput, which are very important for edge deployment. The graph below shows the performance gain from going from an unpruned model to a pruned model on NVIDIA T4. TrafficCamNet,

DashCamNet and PeopleNet are 3 of the custom pre-trained models that are available with NGC. More on these models below.



Pre-trained models

There are 2 types of pre-trained models that users can start with. One is the purpose-built pre-trained models. These are highly accurate models that are trained on millions of objects for a specific task. The other type of models are meta-architecture vision models. The pre-trained weights for these models merely act as a starting point to build more complex models. These pre-trained weights are trained on Open image dataset and they provide a much better starting point for training versus starting from scratch or starting from random weights. With the latter choice, users can choose from 80+ permutations of model architecture and backbone. See the illustration below.



The purpose-built models are built for high accuracy and performance. These models can be deployed out of the box for applications in smart city or smart places or can also be used to re-train with user's own data. All 6 models are trained on millions of objects

and can achieve more than 80% accuracy on our test data. More information about each of these models is available in Purpose built models or in the individual model cards. Typical use cases and some model KPIs are provided in the table below. PeopleNet can be used for detecting and counting people in smart buildings, retail, hospitals, etc. For smart traffic applications, TrafficCamNet and DashCamNet can be used to detect and track vehicles on the road.

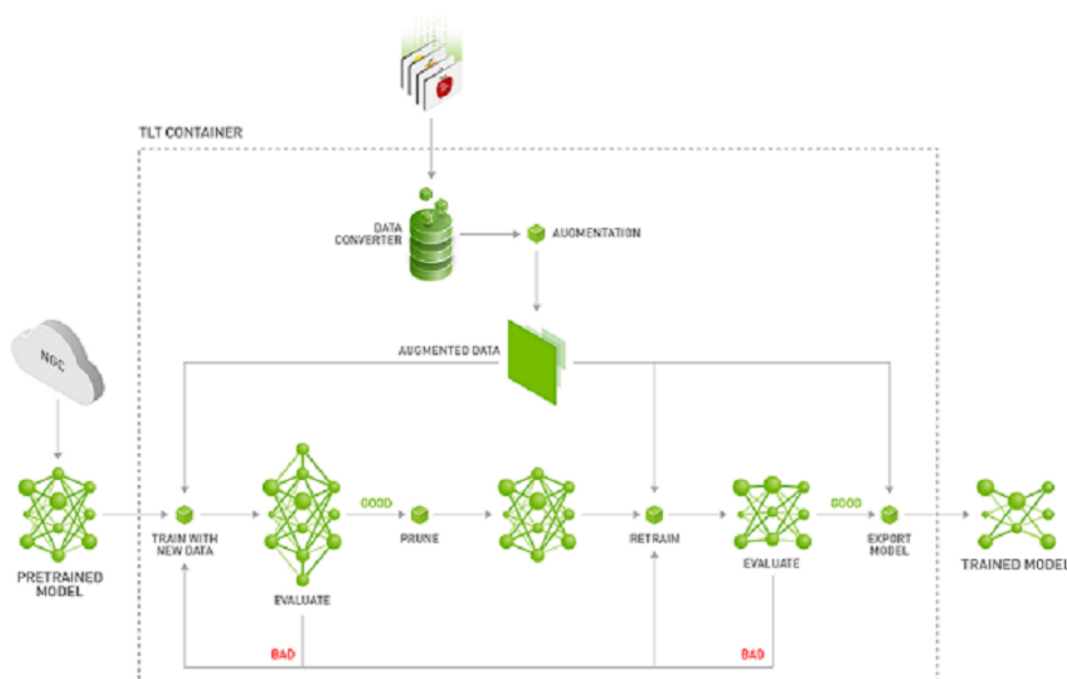
Model Name	Network Architecture	Number of classes	Accuracy	Use Case
TrafficCamNet	DetectNet_v2-ResNet18	4	83.5%	Detect and track cars.
PeopleNet	DetectNet_v2-ResNet18/34	3	84%	People counting, heatmap generation, social distancing.
DashCamNet	DetectNet_v2-ResNet18	4	80%	Identify objects from a moving object.
FaceDetectIR	DetectNet_v2-ResNet18	1	96%	Detect face in a dark environment with IR camera.
VehicleMakeNet	ResNet18	20	91%	Classifying car models.
VehicleTypeNet	ResNet18	6	96%	Classifying type of cars as coupe, sedan, truck, etc.

In the architecture specific models bucket, users can train an image classification model, an object detection model or an instance segmentation model. For classification, users can train using one of 13 available architectures such as ResNet, VGG, MobileNet, GoogLeNet, SqueezeNet or DarkNet architecture. For object detection tasks, users can choose from wildly popular YOLOV3, FasterRCNN, SSD as well as RetinaNet, DSSD and NVIDIA's own DetectNet_v2 architecture. Finally, for instance segmentation, users can use the MaskRCNN architecture. This gives users the flexibility and control to build AI models for any number of applications, from smaller light weight models for edge GPUs to larger models for more complex tasks. For all the permutations and combinations, see the table below and see Supported model architectures.

	Image Classification	Object Detection						Instance Segmentation
		DetectNet_V2	FasterRCNN	SSD	YOLOV3	RetinaNet	DSSD	MaskRCNN
ResNet 10/18/34/50/101	✓	✓	✓	✓	✓	✓	✓	✓
VGG16/19	✓	✓	✓	✓	✓	✓	✓	
GoogLeNet	✓	✓	✓	✓	✓	✓	✓	
MobileNet V1/V2	✓	✓	✓	✓	✓	✓	✓	
SqueezeNet	✓	✓		✓	✓	✓	✓	
DarkNet 19/53	✓	✓	✓	✓	✓	✓	✓	

TLT workflow

The goal of TLT is to train and fine-tune a model using the user's own dataset. In the workflow diagram shown below, a user typically starts with a pre-trained model from NGC; either the highly accurate purpose-built model or just the pre-trained weights of the architecture of their choice. The other input is the user's own dataset. The dataset is fed into the data converter, which can augment the data while training to introduce variations in the dataset. This is very important in training as the data variation improves the overall quality of the model and prevents overfitting. Users can also do offline augmentation with TLT, where the dataset is augmented before training. More information about offline augmentation is provided in Augmenting a dataset.



Once the dataset is prepared and augmented, the next step in the training process is to start training. The training hyperparameters are chosen through the spec file. To learn about all the knobs that users can tune, see Creating an experiment spec file. After the first training phase, users evaluate the model against a test set to see how the model works on the data it has never seen before. Once the model is deemed accurate, the next step is model pruning. If accuracy is not as expected, then the user might have to tune some hyperparameters and re-train. Training is a very iterative process, so you might have to try a few times before converging on the right model.

In model pruning, TLT will algorithmically remove neurons from the neural network which does not contribute significantly to the overall accuracy. The model pruning step will inadvertently reduce the accuracy of the model. So after pruning, the next step is to re-train the model on the same dataset to recover the lost accuracy. After re-train, the user will evaluate the model on the same test set. If the accuracy is back to what was before pruning, then the user can move to the model export step. At this point, the user feels confident in accuracy of the model as well as inference performance. The exported

model will be in '.etlt' format which can be deployed directly on any NVIDIA GPU using DeepStream and TensorRT. In the export step, users can optionally generate an INT8 calibration cache that quantizes the floating-point weights to integer. Running inference at INT8 precision can provide more than 2x performance over FP16 or FP32 precision without sacrificing the accuracy of the model. To learn more about model export and deployment, see Exporting the model and Deploying to DeepStream.

To learn more about how to use TLT, read the technical blogs which provide step-by-step guide to training with TLT:

- Learn toTrain with PeopleNet and other pre-trained model using TLT
- Learn how to trainInstance segmentation model using MaskRCNN with TLT
- Learn how to improve INT8 accuracy usingQuantization aware training(QAT) with TLT

Use the Transfer Learning Toolkit to perform these tasks:

- Download the model - Download pre-trained models.
- Prepare the dataset - Evaluate models for target predictions.
- Train the model - Train or re-train data to create and refine models.
- Evaluate the model - Evaluate models for target predictions.
- Prune the model - Prune models to reduce size.
- Export the model - Export models for TensorRT inference.

Chapter 2. TRANSFER LEARNING TOOLKIT REQUIREMENTS AND INSTALLATION

Using the Transfer Learning Toolkit requires the following:

Hardware Requirements

Minimum

- 4 GB system RAM
- 4 GB of GPU RAM
- Single core CPU
- 1 NVIDIA GPU
- 50 GB of HDD space

Recommended

- 32 GB system RAM
- 32 GB of GPU RAM
- 8 core CPU
- 1 NVIDIA V100 GPU
- 100 GB of SSD space

Currently TLT is not supported on GA-100 GPU's.

Software Requirements

- Ubuntu 18.04 LTS
- NVIDIA GPU Cloud account and API key - https://ngc.nvidia.com/
- docker-ce installed, https://docs.docker.com/install/linux/docker-ce/ubuntu/
- Nvidia docker installed, instructions at https://github.com/NVIDIA/nvidia-docker

- NVIDIA GPU driver v410.xx or above

DeepStream 5.0 - NVIDIA SDK for IVA inference https://developer.nvidia.com/deepstream-sdk is recommended.

Installation Prerequisites

- Install Docker. See: https://www.docker.com/.
- NVIDIA GPU driver v410.xx or above. Download from https://www.nvidia.com/Download/index.aspx?lang=en-us.
- Install NVIDIA Docker from: https://github.com/NVIDIA/nvidia-docker.

Get an NGC API key

- NVIDIA GPU Cloud account and API key - https://ngc.nvidia.com/
    1. Go to NGC and click the **Transfer Learning Toolkit** container in the **Catalog** tab. This message is displayed, **Sign in to access the PULL feature of this repository**.
    2. Enter your email address and click **Next** or click **Create an Account**.
    3. Choose your **organization** when prompted for Organization/Team.
    4. Click **Sign In**.
    5. Select the **Containers** tab on the left navigation pane and click the **Transfer Learning Toolkit** tile.

Download the docker container

- Execute **`docker login nvcr.io`** from the command line and enter these login credentials:
    - Username: $oauthtoken
    - Password: YOUR_NGC_API_KEY
- Execute **`docker pull nvcr.io/nvidia/tlt-streamanalytics:<version>`**

2.1. Installation

The Transfer Learning Toolkit (TLT) is available to download from the NGC. You must have an NGC account and an API key associated with your account. See the Installation Prerequisites section in Chapter 2 for details on creating an NGC account and obtaining an API key.

2.1.1. Running the Transfer Learning Toolkit

Use this procedure to run the Transfer Learning Toolkit.

- **Run the toolkit**: Run the toolkit using this command. The docker starts in the **/workplace** folder by default.

```
docker run --runtime=nvidia -it nvcr.io/nvidia/tlt-streamanalytics:<version> /bin/bash
```

- **Access local directories**: To access local directories from inside the docker you need to mount them in the docker. Use this option, -v <source_dir>:<mount_dir>, to mount local directories in the docker. For example the command to run the toolkit mounting the **/home/<username>/tlt-experiments directory** in your disk to the **/workspace/tlt-experiments** in docker would be:

```
docker run --runtime=nvidia -it -v /home/<username>/tlt-experiments:/workspace/tlt-experiments nvcr.io/nvidia/tlt-streamanalytics:<version> /bin/bash
```

  It is useful to mount separate volumes for the dataset and the experiment results so that they persist outside of the docker. In this way the data is preserved after the docker is closed. Any data that is generated to, or referred from a directory inside the docker, will be lost if it is not either copied out of the docker, or written to or read from volumes outside of the docker.

- **Use the examples**: Examples of DetectNet_v2, SSD, DSSD, RetinaNet, YOLOv3 and FasterRCNN with ResNet18 backbone for detecting objects that are available as Jupyter Notebooks. To run the examples that are available, enable the jupyter notebook included in the docker to run in your browser:

```
docker run --runtime=nvidia -it -v /home/<username>/tlt-experiments:/workspace/tlt-experiments -p 8888:8888 nvcr.io/nvidia/tlt-streamanalytics:<version>
```

  Go to the examples folder: **cd examples/**

  Execute this command from inside the docker to start the jupyter notebook:

```
jupyter notebook --ip 0.0.0.0 --allow-root
```

  Copy and paste the link produced from this command into your browser to access the notebook. The **/workspace/examples** folder will contain a demo notebook.

For all the detector notebooks, the **tlt-train** tool does not support training on images of multiple resolutions, or resizing images during training. All of the images must be resized offline to the final training size and the corresponding bounding boxes must be scaled accordingly.

2.1.2. Downloading the models

The Transfer Learning Toolkit docker gives you access to a repository of pretrained models that can serve as a starting point when training deep neural networks. These models are hosted on the NGC. The TLT docker interfaces with NGC via the NGC Catalog CLI. More information about the NGC Catalog CLI is available here: https://docs.nvidia.com/ngc/ngc-catalog-cli-user-guide/index.html". Please follow the instructions given here to configure the NGC CLI and download the models.

Configure the NGC API key

Using the NGC API Key obtained in Transfer Learning Toolkit Requirements and Installation, configure the enclosed ngc cli by executing this command and following the prompts:

`ngc config set`

Getting a list of models

Use this command to get a list of models that are hosted in the NGC model registry:

```
ngc registry model list <model_glob_string>
```

Here is an example of using this command:

```
ngc registry model list nvidia/tlt_pretrained_*
```

All our classification models have names based on this template **`nvidia/tlt_pretrained_classification:<template>`**.

Downloading a model

Use this command to download the model you have chosen from the NGC model registry:

```
ngc registry model download-version <ORG/model_name:version> -dest
 <path_to_download_dir>
```

For example, use this command to download the resnet 18 classification model to the **`$USER_EXPERIMENT_DIR`** directory.

```
ngc registry model download-version
nvidia/tlt_pretrained_classification:resnet18 --dest
$USER_EXPERIMENT_DIR/pretrained_resnet18
```

```


Downloaded 82.41 MB in 9s, Download speed: 9.14 MB/s
----------------------------------------------------
Transfer id: tlt_iva_classification_resnet18_v1 Download status: Completed.
Downloaded local path: /workspace/tlt-experiments/pretrained_resnet18/
tlt_resnet18_classification_v1
Total files downloaded: 2
Total downloaded size: 82.41 MB
Started at: 2019-07-16 01:29:53.028400
Completed at: 2019-07-16 01:30:02.053016
Duration taken: 9s seconds
```

Chapter 3. SUPPORTED MODEL ARCHITECTURE

Transfer Learning Toolkit supports image classification, 6 object detection architectures, including: **YOLOV3**, **FasterRCNN**, **SSD**, **DSSD**, **RetinaNet**, and **DetectNet_v2** and 1 instance segmentation architecture, namely **MaskRCNN**. In addition, there are 13 classification backbones supported by TLT. For a complete list of all the permutations that are supported by TLT, please see the matrix below:

	Image Classification	**Object Detection**						Instance Segmentation
Backbone		**DetectNet_V2**	**FasterRCNN**	**SSD**	**YOLOV3**	**RetinaNet**	**DSSD**	**MaskRCNN**
ResNet10/18/34/50/101	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
VGG 16/19	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
GoogLeNet	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
MobileNet V1/V2	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
SqueezeNet	Yes	Yes	No	Yes	Yes	Yes	Yes	
DarkNet 19/53	Yes	Yes	Yes	Yes	Yes	Yes	Yes	

Model Requirements

Classification

- Input size: 3 * H * W (W, H >= 16)
- Input format: JPG, JPEG, PNG

Classification input images do not need to be manually resized. The input dataloader resizes images as needed.

Object Detection

DetectNet_v2

- Input size: C * W * H (where C = 1 or 3, W > =480, H >=272 and W, H are multiples of 16)
- Image format: JPG, JPEG, PNG
- Label format: KITTI detection

The `tlt-train` tool does not support training on images of multiple resolutions, or resizing images during training. All of the images must be resized offline to the final training size and the corresponding bounding boxes must be scaled accordingly.

FasterRCNN

- Input size: C * W * H (where C = 1 or 3; W > =160; H >=160)
- Image format: JPG, JPEG, PNG
- Label format: KITTI detection

The `tlt-train` tool does not support training on images of multiple resolutions, or resizing images during training. All of the images must be resized offline to the final training size and the corresponding bounding boxes must be scaled accordingly.

SSD

- Input size: C * W * H (where C = 1 or 3, W >= 128, H >= 128, W, H are multiples of 32)
- Image format: JPG, JPEG, PNG
- Label format: KITTI detection

The `tlt-train` tool does not support training on images of multiple resolutions, or resizing images during training. All of the images must be resized offline to the final training size and the corresponding bounding boxes must be scaled accordingly.

DSSD

- Input size: C * W * H (where C = 1 or 3, W >= 128, H >= 128, W, H are multiples of 32)
- Image format: JPG, JPEG, PNG
- Label format: KITTI detection

The `tlt-train` tool does not support training on images of multiple resolutions, or resizing images during training. All of the images must be resized offline to the final training size and the corresponding bounding boxes must be scaled accordingly.

YOLOv3

- Input size: C * W * H (where C = 1 or 3, W >= 128, H >= 128, W, H are multiples of 32)
- Image format: JPG, JPEG, PNG
- Label format: KITTI detection

The `tlt-train` tool does not support training on images of multiple resolutions, or resizing images during training. All of the images must be resized offline to the final training size and the corresponding bounding boxes must be scaled accordingly.

RetinaNet

- Input size: C * W * H (where C = 1 or 3, W >= 128, H >= 128, W, H are multiples of 32)
- Image format: JPG, JPEG, PNG
- Label format: KITTI detection

The **tlt-train** tool does not support training on images of multiple resolutions, or resizing images during training. All of the images must be resized offline to the final training size and the corresponding bounding boxes must be scaled accordingly.

Instance Segmentation

MaskRCNN

- Input size: C * W * H (where C = 3, W > =128, H >=128 and W, H are multiples of 32)
- Image format: JPG
- Label format: COCO detection

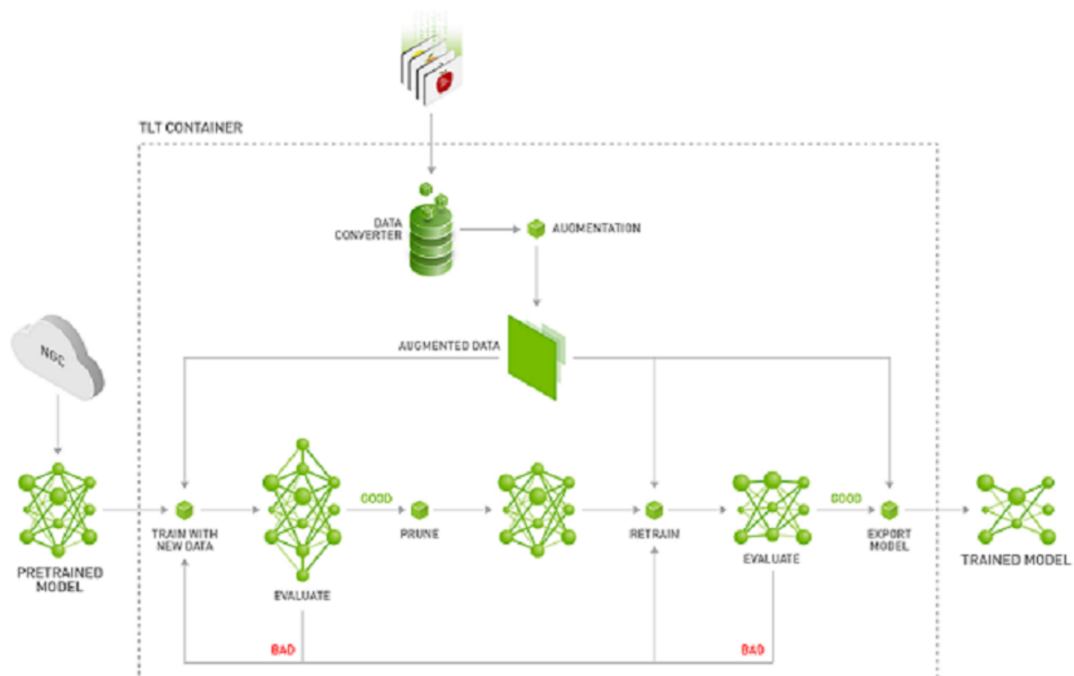
Training

The TLT container contains Jupyter notebooks and the necessary spec files to train any network combination. The pre-trained weight for each backbone is provided on NGC. The pre-trained model is trained on Open image dataset. The pre-trained weights provide a great starting point for applying transfer learning on your own dataset.

To get started, first choose the type of model that you want to train, then go to the appropriate model card on NGC and then choose one of the supported backbones.

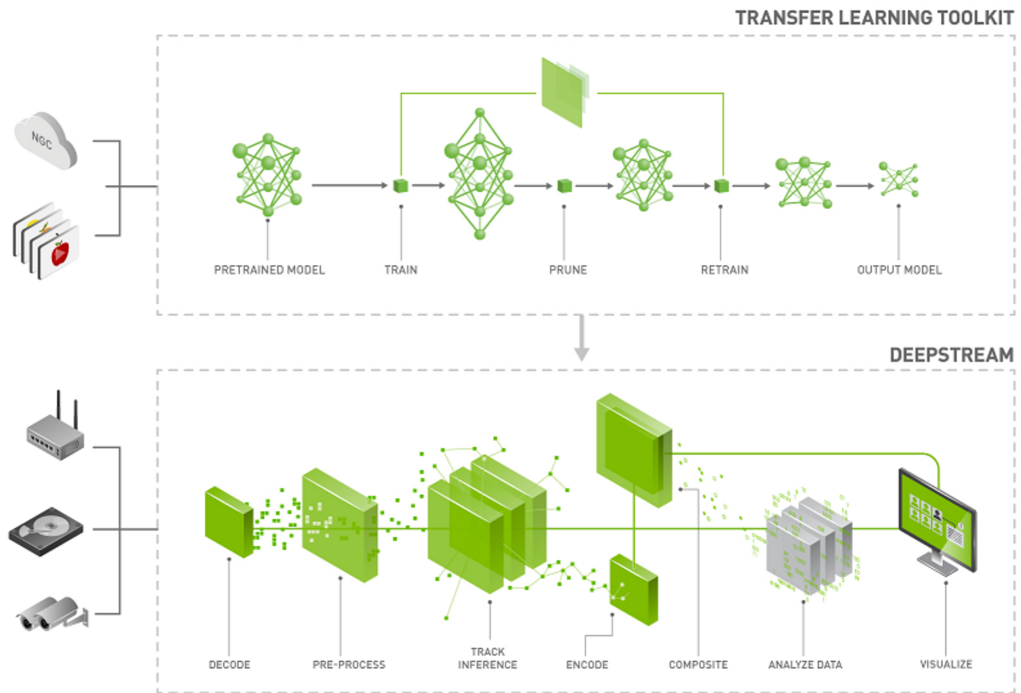
Model to train	NGC model card	Supported Backbone
YOLOv3 SSD FasterRCNN RetinaNet DSSD	TLT object detection	resnet10, resnet18, resnet34, resnet50, resnet101, vgg16, vgg19, googlenet, mobilenet_v1, mobilenet_v2, squeezenet, darknet19, darknet53
DetectNet_v2	TLT DetectNet_v2 detection	resnet10, resnet18, resnet34, resnet50, resnet101, vgg16, vgg19, googlenet, mobilenet_v1, mobilenet_v2, squeezenet, darknet19, darknet53
MaskRCNN	TLT instance segmentation	resnet10, resnet18, resnet34, resnet50, resnet101
Image Classification	TLT image classification	resnet10, resnet18, resnet34, resnet50, resnet101, vgg16, vgg19, googlenet, mobilenet_v1, mobilenet_v2, squeezenet, darknet19, darknet53

Once you pick the appropriate pre-trained model, follow the TLT workflow to take your dataset and pre-trained model and export a tuned model that is adapted to your use case. Chapters 4 to 11 walks through all the steps in training.



Deployment

You can deploy your trained model on any edge device using DeepStream and TensorRT. See Deploying to DeepStream for deployment instructions.


TRANSFER LEARNING TOOLKIT
NGC
PRETRAINED MODEL
TRAIN
PRUNE
RETRAIN
OUTPUT MODEL
DEEPSTREAM
DECODE
PRE-PROCESS
TRACK
INFERENCE
ENCODE
COMPOSITE
ANALYZE DATA
VISUALIZE

Chapter 4. PURPOSE-BUILT MODELS

The purpose-built AI models are primarily built for applications in smart cities, parking management, smart buildings and are trained on millions of images. Both unpruned and pruned versions of these models are available on NGC. The unpruned models are used with TLT to re-train with your dataset. On the other hand, pruned models are deployment ready that allows you to directly deploy on your edge device. In addition, the pruned model also contains a calibration table for INT8 precision. The pruned INT8 model will provide the highest inference throughput.

The table below shows the network architecture and accuracy measured on our dataset.

Model Name	Network Architecture	Number of classes	Accuracy
TrafficCamNet	DetectNet_v2-ResNet18	4	83.5%
PeopleNet	DetectNet_v2-ResNet34	3	84%
	DetectNet_v2-ResNet18	3	80%
DashCamNet	DetectNet_v2-ResNet18	4	80%
FaceDetect-IR	DetectNet_v2-ResNet18	1	96%
VehicleMakeNet	ResNet18	20	91%
VehicleTypeNet	ResNet18	6	96%

Training

The PeopleNet, TrafficCamNet, DashCamNet and FaceDetect-IR are detection models based on DetectNet_v2 and either ResNet18 or ResNet34 backbone. To re-train these models with your data, use the unpruned model from NGC and follow the DetectNet_v2 object detection training guidelines from chapters Preparing input data structure to Exporting the model. The entire training workflow is given in the prior section.

The VehicleMakeNet and VehicleTypeNet are classification models based on the ResNet18 backbone. To re-train these models, use the unpruned model from NGC and follow the Image classification training guideline from chapters Preparing input data structure to Exporting the model.

Deployment

You can deploy your own trained or the provided pruned model on any edge device using DeepStream. The deployment instructions are provided in Deploying to DeepStream.

TrafficCamNet

TrafficCamNet is a 4-class object detection network built on NVIDIA's detectnet_v2 architecture with ResNet18 as the backbone feature extractor. It's trained on 544x960 RGB images to detect cars, persons, road signs and two wheelers. The dataset contains images from real traffic intersections from cities in the US (at about 20ft vantage point). This model is trained to overcome the problem of separating a line of cars as they come to stop at a red traffic light or a stop sign. This model is ideal for smart city applications, where you want to count the number of cars on the road and understand flow of traffic.

PeopleNet

PeopleNet is a 3-class object detection network built on NVIDIA's detectnet_v2 architecture with ResNet34 as the backbone feature extractor. It's trained on 544x960 RGB images to detect person, bag, and face. Several million images of both indoor and outdoor scenes were labeled in-house to adapt to a variety of use cases, such as airports, shopping malls and retail stores. This dataset contains images from various vantage points. PeopleNet can be used for smart places or building applications where you need to accurately count people in a crowded environment for security or higher level business insights.

DashCamNet

DashCamNet is a 4-class object detection network built on NVIDIA's detectnet_v2 architecture with ResNet18 as the backbone feature extractor. It's trained on 544x960 RGB images to detect cars, pedestrians, traffic signs and two wheelers. The training data for this network contains real images collected, annotated and curated in-house from different dashboard cameras in cars at about 4-5ft height in vantage point. Unlike the other models the camera in this case is moving. The use case for this model is to identify objects from a moving object, which can be a car or a robot.

FaceDetect-IR

FaceDetect_IR is a single class face detection network built on NVIDIA's detectnet_v2 architecture with ResNet18 as the backbone feature extractor. The model is trained on 384x240x3 IR (infrared) images augmented with synthetic noises and is trained for use cases where the person's face is close to the camera, such as a laptop camera during video conferencing or a camera placed inside a vehicle to observe a distracted driver. When infrared illuminators are used this model can continue to work even when visible light conditions are considered too dark for normal color cameras.

VehicleMakeNet

VehicleMakeNet is a classification network based on ResNet18, which aims to classify car images of size 224 x 224. This model can identify 20 popular car makes. VehicleMakeNet is generally cascaded with DashCamNet or TrafficCamNet for smart city applications. For example, DashCamNet or TrafficCamNet acts as a primary detector, detecting the objects of interest and for each detected car the VehicleMakeNet acts as a secondary classifier determining the make of the car. Businesses such as smart

parking or gas stations can use the insights of the make of vehicles to understand their customers.

VehicleTypeNet

VehicleTypeNet is a classification network based on ResNet18, which aims to classify cropped vehicle images of size 224 x 224 into 6 classes: Coupe, Large Vehicle, Sedan, SUV, Truck, and Vans. The typical use case for this model is in smart city applications such as smart garage or toll booth, where you can charge based on size of the vehicle.

Chapter 5. AUGMENTING A DATASET

Training a deep neural network can be a daunting task, and the most important component of training a model is the data. Acquiring curated and annotated dataset can be a very tiring and manual process, involving thousands of man hours of painstaking labelling. In spite of planning and collecting data, it is very difficult to estimate all the corner cases that a network may go through, and repeating the process of collecting the missing data and annotating is very expensive and has long turnover times.

Online augmentation in the training data loader is a good way to increase the variation in the dataset. However, the augmented data is generated randomly based on the distribution the data loader follows when sampling the data and in order to achieve good accuracy, the model may need to be trained for a long time. Inorder to circumvent this and generate a dataset with the required augmentations and give control to the user, TLT provides an offline augmentation tool called **`tlt-augment`**. Offline augmentation can dramatically increase the size of the dataset when collecting and labeling data is expensive or not possible. The **`tlt-augment`** tools provides several custom GPU accelerated augmentation routines categorized into:

1. Spatial augmentation
2. Color space augmentation
3. Image Blur

Spatial augmentation comprises routines where data is augmented in space. The following spatial augmentation operations are supported in TLT.

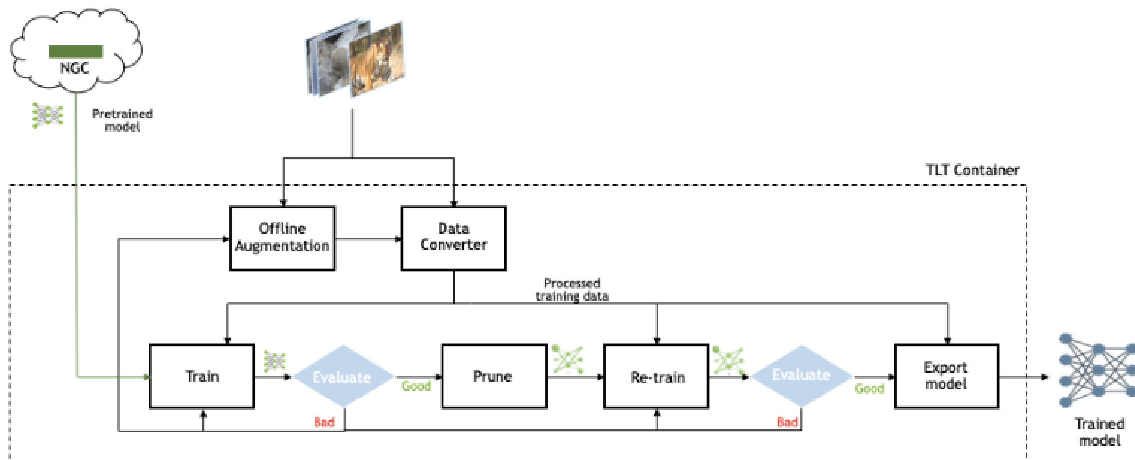
1. Rotate
2. Resize
3. Translate
4. Shear
5. Flip

Color space augmentation comprises routines where the image data is augmented in the color space. The following color augmentations operators are supported.

1. Hue Rotation
2. Brightness offset
3. Contrast shift

Along with the above mentioned augmentation operations **`tlt-augment`** also enables use to blur images, using a Gaussian blur operator. More information about the operation is described in Blur config.

All augmentation routines currently provided with **`tlt-augment`** are supported only for an object detection dataset. The spatial augmentation routines are applied to the images as well as the labelled data coordinates, while the color augmentation routines and channel-wise blue operator is applied only to images as the object labels are not affected. The sample workflow of using tlt-augment is as follows:



> The data is expected in KITTI format, as described in Data input for objection detection. The following sections describe how to use the augmentation tool.

5.1. Configuring the augmentor

The augmentor has several components which the user can configure by using a simple protobuf based configuration file. The configuration file is divided into 4 major components.

1. Spatial augmentation config
2. Color augmentation config
3. Blur config
4. Data dimensions - output image width, output image height, output image channel, image extension.

This configuration file contains several nested protobuf elements, and global parameters which are defined below.

Parameter	Datatype	Description	Supported Values
spatial_config	Protobuf message	This protobuf message configures the spatial augmentation.	Protobuf definition provided in Spatial augmentation config.

color_config	Protobuf message	This protobuf message configures the color space augmentation operator.	Protobuf definition provided in Color augmentation config.
blur_config	Protobuf message	This protobuf message configures the gaussian blue operator to be applied on the image. The blur is computed channel wise and then concatenated based on the number of image channels.	Protobuf definition provided in Blur config.
dataset_config	Protobuf message	This protobuf message configures the relative paths of the images and labels path from the input dataset root defined over the **tlt-augment** command line.	Protobuf definition provided in Dataset config.
output_image_width	int32	This parameter defines the width of the output image.	
output_image_height	int32	This parameter defines the height of the output image.	
output_image_channel	int32	This parameter defines the number of channels in the output image.	1, 3
image_extension	string	The extension of the input image. Note that all the images in the input dataset are expected to be of the same extension.	.png, .jpeg, .jpg

5.1.1. Spatial augmentation config

Spatial augmentation config contains parameters to configure the spatial augmentation routines. This is a nested protobuf element called **`spatial_config`** containing protobuf elements for all the spatial augmentation operations.

Parameter	Datatype	Description	Supported Values
rotation_config	Protobuf message	This protobuf message configures the rotate augmentation operator. Defining this activates rotation.	{ angle: 0.5 units: degrees } See Rotation config.
flip_config	Protobuf message	This protobuf message configures the flip augmentation operator. Defining this activates flip along the horizontal and/or vertical axes.	{ flip_vertical: true flip_horizontal: true } See Flip config.
translation_config	Protobuf message	This protobuf message configures the translation augmentation operator. Defining this activates translating the images across the x and/or y axes.	{ translate_x: 8 translate_y: 8 } See Translation config.
shear_config	Protobuf message	This protobuf message configures the shear augmentation operator. Defining this activates adds a shear to the images across the x and/or y axes.	{ shear_ratio_x: 0.2 shear_ratio_y: 0.2 } See Shear config.

The augmentation operators may be enabled by simply defining the corresponding proto associated with it. When defining multiple proto elements, it implies that all the augmentation operations are cascaded.

If you don't wish to introduce any of the supported augmentation operations, simply omit the field you wish to drop. The configurable parameters for the individual spatial augmentation operators are mentioned in the table below.

5.1.1.1. Rotation config

The rotation operation rotates the image at an angle. The transformation matrix for shear operation is computed as:

```
[x_new, y_new, 1] = [x, y, 1] * [[cos(angle) sin(angle)  zero]
                                 [-sin(angle)        cos(angle)  zero]
                                 [x_t          y_t          one]]
Where x_t, y_t are defined as
x_t = height * sin(angle) / 2.0 - width * cos(angle) / 2.0 + width / 2.0
y_t = -1 * height * cos(angle) / 2.0 + height / 2.0 - width * sin_(angle) / 2.0
Here height = height of the output image, width = width of the output image.
```

Parameter	Datatype	Description	Supported Values
angle	float	The angle of the rotation to be applied to the image and the coordinates.	+/- 0 - 360 (degrees) +/- 0 - 2# (radians)
units	string	The unit in which the angle parameter mentioned below is mentioned.	"degrees", "radians"

5.1.1.2. Shear config

The shear operation introduces a slant to the object along the x or the y dimension. The transformation matrix for shear operation is computed as:

```
[x_new, y_new, 1] = [x, y, 1] * [[1.0             shear_ratio_y,    0],
                                   [shear_ratio_x,       1.0,         0],
                                   [x_t,                 y_t,       1.0]]
X_t = -height * shear_ratio_x / 2.
Y_t = -width * shear_ratio_y / 2.
Here height = height of the output image, width = width of the output image.
```

Parameter	Datatype	Description	Supported Values
shear_ratio_x	float32	The amount of horizontal shift per y row.	
shear_ratio_y	float32	The amount of vertical shift per x column.	

5.1.1.3. Flip config

This element configures the flip operator of tlt-augment. The operator flips an image and the bounding box coordinates along the horizontal and vertical axis.

Parameter	Datatype	Description	Supported Values
flip_horizontal	bool	The flag to enable flipping an image horizontally.	true, false
flip_vertical	bool	The flag to enable flipping an image vertically.	true, false

Please note that at least one of the two flags must be set when defining this parameter.

5.1.1.4. Translation config

This protobuf message configures the translation operator for tlt-augment. The operator translates the image and polygon coordinates along the x and/or y axis.

Parameter	**Datatype**	**Description**	**Supported Values**
translate_x	int	The number of pixels to translate the image along the x axis.	0 - image_width
translate_y	int	The number of pixels to translate the image along the y axis.	0 - image_height

5.1.2. Color augmentation config

Color augmentation config contains parameters to configure the color space augmentation routines. This is a nested protobuf element called **`color_config`** containing protobuf elements for all the color augmentation operations.

Parameter	**Datatype**	**Description**	**Supported Values**

hue_saturation_config	Protobuf message	This augmentation operator applies hue rotation and color saturation augmentation.	{
			 hue_rotation_angle: 30
			 saturation_shift: 1.0
			 }
			 See Hue saturation config.

contrast_config	Protobuf message	This augmentation operator applies contrast scaling.	{ contrast: 0.0

			center: 127.5 } See Contrast config.

brightness_config	Protobuf message	This protobuf message configures the translation augmentation operator. Defining this activates translating the images across the x and/or y axes.	{
			translate_x: 8
			translate_y: 8
			}
			SeeBrightness config

The augmentation operators may be enabled by simply defining the corresponding proto associated with it. When defining multiple proto elements, it implies that all the augmentation operations are cascaded.

If you don't want to introduce any of the supported augmentation operations, simply omit the field you wish to drop. The configurable parameters for the individual color augmentation operators are

mentioned in the table below.

5.1.2.1. Hue saturation config

This augmentation operator applies a color space manipulation by converting the RGB image to HSV applying hue rotation and saturation shift and then returning with the corresponding RGB image.

Parameter	Datatype	Description	Supported Values
hue_rotation_angle	float32	hue rotation in degrees (scalar or vector). A value of 0.0 (modulo 360) leaves the hue unchanged.	0 - 360 (the angles are computed as angle % 360)
saturation_shift	float32	Saturation shift multiplier. A value of 1.0 leaves the saturation unchanged. A value of 0 removes all saturation from the image and makes all channels equal in value.	0.0 - 1.0

5.1.2.2. Brightness config

This augmentation operator applies a channel-wise brightness shift.

Parameter	Datatype	Description	Supported Values
offset	float32	Offset value per color channel	0 - 255

5.1.2.3. Contrast config

This augmentation operator applies contrast scaling across a center point to an image.

Parameter	Datatype	Description	Supported Values
contrast	float32	Contrast scale value. A value 0 leaves the contrast unchanged.	0 - 1.0
center	float32	Center value for the image. In our case, the images are scaled between 0-255 (8 bit images), therefore setting a value of 127.5 is the common value.	0.0 - 1.0

5.1.3. Dataset config

```
dataset_config {
  data_sources: {
    tfrecords_path: "/path/to/tfrecords/root/*"
    image_directory_path: "/path/to/dataset/root"
  }
  image_extension: "png"
  target_class_mapping {
      key: "car"
      value: "car"
  }
  target_class_mapping {
      key: "pedestrian"
      value: "pedestrian"
  }
  target_class_mapping {
      key: "cyclist"
      value: "cyclist"
  }
  target_class_mapping {
      key: "van"
      value: "car"
  }
  target_class_mapping {
      key: "person_sitting"
      value: "pedestrian"
  }
```

```
  validation_fold: 0
}
```

See Dataloader for more information.

5.1.4. Blur config

This protobuf element configures the gaussian blur operator to an image. A gaussian kernel is formulated based on the parameters mentioned below and then a 2D convolution is performed between this image and kernel per channel.

Parameter	Datatype	Description	Supported Values
size	int	Size of the kernel to be convolved.	>0
std	float	Standard deviation of the gaussian filter to blurring.	>0.0

As an example, a configuration file to augment the image by

1. rotate an image by 5 deg
2. shear along x axis by a ratio of 0.3
3. Translate along x axis by 8 pixels

```
# Spec file for tlt-augment.
spatial_config{
  rotation_config{
    angle: 5.0
    units: "degrees"
  }
  shear_config{
    shear_ratio_x: 0.3
  }
  translation_config{
    translate_x: 8
  }
}
color_config{
  hue_saturation_config{
    hue_rotation_angle: 25.0
    saturation_shift: 1.0
  }
}
# Setting up dataset config.
dataset_config{
  image_path: "image_2"
  label_path: "label_2"
}
output_image_width: 1248
output_image_height: 384
output_image_channel: 3
image_extension: ".png"
```

5.2. Running the augmentor tool

The **tlt-augment** tool has a simple command line interface and its usage may be defined as follows.

```
tlt-augment [-h] -d /path/to/the/dataset/root
                 -a /path/to/augmentation/spec/file
                 -o /path/to/the/augmented/output
                 [-v]
```

Here are the command line parameters:

- **-h, --help**: show this help message and exit
- **-d, --dataset-folder**: Path to the detection dataset
- **-a, --augmentation-proto**: Path to augmentation spec file.
- **-o, --output-dataset**: Path to the augmented output dataset.
- **-v, --verbose**: Flag to get detailed logs during the augmentation process.

The augmented images and labels are generated in the path mentioned in the output-dataset parameter under the following directories.

- Augmented images: **/path/to/augmented/output/images**
- Augmented labels: **/path/to/augmented/output/labels**

When running tlt-augment with the verbose flag set, tlt-augment generates augmented images with the bbox outputs rendered under **/path/to/augmented/output/images/annotated**.

The log from a successful run of **tlt-augment** is mentioned below:

```
Using TensorFlow backend.
2020-07-10 16:19:18,980 [INFO] iva.augment.spec_handler.spec_loader: Merging
 specification from /path/to/augmentor/spec/file.txt
2020-07-10 16:19:18,992 [INFO] iva.augment.build_augmentor: Input dataset: /
path/to/input/dataset/root
2020-07-10 16:19:18,992 [INFO] iva.augment.build_augmentor: Output dataset: /
path/to/augmented/output
2%|███#                                    | 167/7481 [00:13<10:04, 12.09it/s]
```

The dataset thus generated may then be used with **tlt-dataset-convert** tool to be converted to TFRecords so that it may be ingested by **tlt-train**. The details about converting the data to TFRecords are described in Data input for object detection and training a model with this dataset is described in chapter 6.

The **tlt-augment** only applies the spatial augmentation operators to the bounding box coordinates fields in the label files of the input dataset, as only the bbox coordinates are relevant to us. All the other fields are just propagated as from the input labels to the output labels.

Sample rendered augmented images are shown below.

Input image rotated by 5 degrees.

Image rotated by 5 degrees, hue rotation by 25 degrees and saturation shift of 0.0.

Chapter 6. PREPARING INPUT DATA STRUCTURE

The chapter provides instructions on preparing your data for use by the Transfer Learning Toolkit (TLT).

6.1. Data input for classification

Classification expects a directory of images with the following structure, where each class has its own directory with the class name. The naming convention for **train/val/test** can be different, because the path of each set is individually specified in the spec file. See Specification file for classification for more information.

```
|--dataset_root:
    |--train
        |--audi:
            |--1.jpg
            |--2.jpg
        |--bmw:
            |--01.jpg
            |--02.jpg
    |--val
        |--audi:
            |--3.jpg
            |--4.jpg
        |--bmw:
            |--03.jpg
            |--04.jpg
    |--test
        |--audi:
            |--5.jpg
            |--6.jpg
        |--bmw:
            |--05.jpg
            |--06.jpg
```

6.2. Data input for object detection

The object detection apps in TLT expect data in KITTI file format for training and evaluation. For DetectNet_v2, SSD, DSSD, YOLOv3, and FasterRCNN, this data is

converted to TFRecords for training. TFRecords help iterate faster through the data. The steps to convert the data for TFRecords are covered in Conversion to TFRecords.

6.2.1. KITTI file format

Using the KITTI format requires data to be organized in this structure:

```
.
|--dataset root
  |-- images
      |-- 000000.jpg
      |-- 000001.jpg
            .
            .
      |-- xxxxxx.jpg
  |-- labels
      |-- 000000.txt
      |-- 000001.txt
            .
            .
      |-- xxxxxx.txt
  |-- kitti_seq_to_map.json
```

Here's a description of the structure:

- The **images** directory contains the images to train on.
- The **labels** directory contains the labels to the corresponding images. Details of this file are included in the Label files section.

The images and labels have the same file id's before the extension. The image to label correspondence is maintained using this file name.

- **`kitti_seq_to_map.json`**: This file contains a sequence to frame id mapping for the frames in the images directory. This is an optional file, and is useful if the data needs to be split into N folds sequence wise. In case the data is to be split into a random 80:20 train:val split, then this file may be ignored.

All the images and labels in the training dataset should be of the same resolution. For DetectNet_v2, SSD, DSSD, YOLOv3 and FasterRCNN notebooks, the **`tlt-train`** tool does not support training on images of multiple resolutions, or resizing images during training. All of the images must be resized offline to the final training size and the corresponding bounding boxes must be scaled accordingly.

6.2.2. Label files

A KITTI format label file is a simple text file containing one line per object. Each line has multiple fields. Here is a description of these fields:

Num elements	Parameter name	Description	Type	Range	Example
1	Class names	The class to which	String	N/A	Person, car, Road_Sign

Num elements	Parameter name	Description	Type	Range	Example
		the object belongs.			
1	Truncation	How much of the object has left image boundaries.	Float	0.0, 0.1	0.0
1	Occlusion	Occlusion state [0 = fully visible, 1 = partly visible, 2 = largely occluded, 3 = unknown].	Integer	[0,3]	2
1	Alpha	Observation Angle of object	Float	[-pi, pi]	0.146
4	Bounding box coordinates: [xmin, ymin, xmax, ymax]	Location of the object in the image	Float(0 based index)	[0 to image width],[0 to image_height], [top_left, image_width], [bottom_right, image_height]	100 120 180 160
3	3-D dimension	Height, width, length of the object (in meters)	Float	N/A	1.65, 1.67, 3.64
3	Location	3-D object location x, y, z in camera coordinates (in meters)	Float	N/A	-0.65,1.71, 46.7
1	Rotation_y	Rotation ry around the Y-axis in camera coordinates	Float	[-pi, pi]	-1.59

The sum of the total number of elements per object is 15. Here is a sample text file:

```
car 0.00 0 -1.58 587.01 173.33 614.12 200.12 1.65 1.67 3.64 -0.65 1.71 46.70
 -1.59
cyclist 0.00 0 -2.46 665.45 160.00 717.93 217.99 1.72 0.47 1.65 2.45 1.35 22.10
 -2.35
pedestrian 0.00 2 0.21 423.17 173.67 433.17 224.03 1.60 0.38 0.30 -5.87 1.63
 23.11 -0.03
```

This indicates that in the image there are 3 objects with parameters mentioned as above. Currently, for detection the toolkit only requires the class name and bbox coordinates fields to be populated. This is because the TLT training pipe supports training only for class and bbox coordinates. The remaining fields may be set to 0. Here is a sample file for a custom annotated dataset:

```
car 0.00 0 0.00 587.01 173.33 614.12 200.12 0.00 0.00 0.00 0.00 0.00 0.00 0.00
cyclist 0.00 0 0.00 665.45 160.00 717.93 217.99 0.00 0.00 0.00 0.00 0.00 0.00
 0.00
pedestrian 0.00 0 0.00 423.17 173.67 433.17 224.03 0.00 0.00 0.00 0.00 0.00 0.00
 0.00
```

6.2.3. Sequence mapping file

This is an optional **json** file that captures the mapping between the frames in **images** directory and the names of video sequences from which these frames were extracted. This information is needed while doing an N-fold split of the dataset. This way frames from one sequence don't repeat in other folds and one of the folds for could be used for validation. Here's an example of the json dictionary file.

```
{
  "video_sequence_name": [list of strings(frame idx)]
}
```

Here's an example of a **`kitti_seq_to_frames.json`** file with a sample dataset with six sequences.

```
{
  "2011_09_28_drive_0165_sync": ["003193", "003185", "002857", "001864",
 "003838",
  "007320", "003476", "007308", "000337", "004165", "006573"],
  "2011_09_28_drive_0191_sync": ["005724", "002529", "004136", "005746"],
  "2011_09_28_drive_0179_sync": ["005107", "002485", "006089", "000695"],
  "2011_09_26_drive_0079_sync": ["005421", "000673", "002064", "000783",
 "003068"],
  "2011_09_28_drive_0035_sync": ["005540", "002424", "004949", "004996",
 "003969"],
  "2011_09_28_drive_0117_sync": ["007150", "003797", "002554", "001509"]
}
```

6.3. Conversion to TFRecords

The SSD, DSSD, YOLOv3, FasterRCNN, and DetectNet_v2 apps, as mentioned in Data input for object detection, require KITTI format data to be converted to TFRecords. To do so, the Transfer Learning Toolkit includes the **`tlt-dataset-convert`** tool. This

tool requires a configuration file as input. Configuration file details and sample usage examples are included in the following sections.

6.3.1. Configuration file for dataset converter

The dataio conversion tool takes a spec file as input to define the parameters required to convert a KITTI format data to the TFRecords that the detection models ingest. This is a prototxt format file with two global parameters:

- **`kitti_config`** field: This is a nested prototxt configuration with multiple input parameters.
- **`image_directory_path`**: Path to the dataset root. This image_dir_name is appended to this path to get the input images, and must be the same path as mentioned in the experiment spec file

Here are descriptions of the configurable parameters for the **`kitti_config`** field:

Parameter	Datatype	Default	Description	Supported Values
root_directory _path	string	-	Path to the dataset root directory	-
image_dir_name	string	-	Relative path to the directory containing images from the path in root_ directory_path	-
label_dir_name	string	-	Relative path to the directory containing labels from the path in root_ directory_path	-
partition_mode	string	-	The method employed when partitioning the data to multiple folds. Two methods are supported: ▸ Random partitioning: Where the data is divided in	▸ random ▸ sequence

Parameter	Datatype	Default	Description	Supported Values
			to 2 folds namely, train and val. This mode requires that the val_split parameter be set. - Sequence-wise partitioning: Where the data is divided into n partitions (defined by num _partitions parameter) based on the number of sequences available.	
num_partitions	int	2 (if partition_mode is random)	Number of partitions to split the data (N folds). This field is ignored when the partition model is set to random, as by default only 2 partitions are generated. Val and train. In sequence mode the data is split into n-folds. The number of partitions is ideally lesser	- n=2 for random partition - n< number of sequences in the **kitti_ sequence _to_frames _file**

Parameter	Datatype	Default	Description	Supported Values
			than the total number of sequences in the kitti_sequence _to_frames_file .	
image_extension	str	".png"	The extension of the images in the image_dir_name parameter.	▸ .png ▸ .jpg ▸ .jpeg
val_split	float	20	Percentage of data to be separated for validation. This only works under "random" partition mode. This partition is available in fold 0 of the TFrecords generated. Please set the validation fold to 0 in the **`dataset_config`**.	0-100
kitti_sequence _to _frames_file	str		Name of the kitti sequence to frame mapping file. This file must be present within the dataset root as mentioned in the root_directory _path .	

Parameter	Datatype	Default	Description	Supported Values
num_shards	int	10	Number of shards per fold.	1-20

A sample configuration file to convert the pascal voc dataset with 80% training data and 20 % validation data is mentioned below. This assumes that the data has been converted to KITTI format and is available for ingestion in the root directory path.

```
kitti_config {
  root_directory_path: "/workspace/tlt-experiments/data/VOCtrainval_11-May-2012/
VOCdevkit/VOC2012"
  image_dir_name: "JPEGImages_kitti/test"
  label_dir_name: "Annotations_kitti/test"
  image_extension: ".jpg"
  partition_mode: "random"
  num_partitions: 2
  val_split: 20
  num_shards: 10
}
image_directory_path: "/workspace/tlt-experiments/data/VOCtrainval_11-May-2012/
VOCdevkit/VOC2012"
```

6.3.2. Sample usage of the dataset converter tool

KITTI is the accepted dataset format for image detection. The KITTI dataset must be converted to the TFRecord file format before passing to detection training. Use this command to do the conversion:

```
tlt-dataset-convert [-h] -d DATASET_EXPORT_SPEC -o OUTPUT_FILENAME
                         [-f VALIDATION_FOLD]
```

You can use these optional arguments:

- **`-h, --help`**: Show this help message and exit
- **`-d, --dataset-export-spec`**: Path to the detection dataset spec containing config for exporting .tfrecords.
- **`-o output_filename`**: Output file name.
- **`-f, -validation-fold`**: Indicate the validation fold in 0-based indexing. This is required when modifying the training set but otherwise optional.

Here's an example of using the command with the dataset:

```
tlt-dataset-convert -d <path_to_tfrecords_conversion_spec> -o
 <path_to_output_tfrecords>
```

Output log from executing **`tlt-dataset-convert`**:

```
Using TensorFlow backend.
2019-07-16 01:30:59,073 - iva.detectnet_v2.dataio.build_converter - INFO -
 Instantiating a kitti converter
2019-07-16 01:30:59,243 - iva.detectnet_v2.dataio.kitti_converter_lib - INFO -
 Num images in
Train: 10786    Val: 2696
2019-07-16 01:30:59,243 - iva.detectnet_v2.dataio.kitti_converter_lib - INFO -
 Validation data in partition 0. Hence, while choosing the validation set during
 training choose validation_fold 0.
```

```
2019-07-16 01:30:59,251 - iva.detectnet_v2.dataio.dataset_converter_lib - INFO -
 Writing partition 0, shard 0
/usr/local/lib/python2.7/dist-packages/iva/detectnet_v2/dataio/
kitti_converter_lib.py:265: VisibleDeprecationWarning: Reading unicode strings
 without specifying the encoding argument is deprecated. Set the encoding, use
 None for the system default.
2019-07-16 01:31:01,226 - iva.detectnet_v2.dataio.dataset_converter_lib - INFO -
 Writing partition 0, shard 1
. .
sheep: 242
bottle: 205
..
boat: 171
car: 418
2019-07-16 01:31:20,772 - iva.detectnet_v2.dataio.dataset_converter_lib - INFO -
 Writing partition 1, shard 0
..
2019-07-16 01:32:40,338 - iva.detectnet_v2.dataio.dataset_converter_lib - INFO -
 Writing partition 1, shard 9
2019-07-16 01:32:49,063 - iva.detectnet_v2.dataio.dataset_converter_lib - INFO -
Wrote the following numbers of objects:
sheep: 695
..
car: 1770

2019-07-16 01:32:49,064 - iva.detectnet_v2.dataio.dataset_converter_lib - INFO -
 Cumulative object statistics
2019-07-16 01:32:49,064 - iva.detectnet_v2.dataio.dataset_converter_lib - INFO -
Wrote the following numbers of objects:
sheep: 937
..
car: 2188
2019-07-16 01:32:49,064 - iva.detectnet_v2.dataio.dataset_converter_lib - INFO -
 Class map.
Label in GT: Label in tfrecords file
sheep: sheep
..

boat: boat
For the dataset_config in the experiment_spec, please use labels in the
 tfrecords file, while writing the classmap.



2019-07-16 01:32:49,064 - iva.detectnet_v2.dataio.dataset_converter_lib - INFO -
 Tfrecords generation complete.
```

The tlt-dataset-convert tool updates the class names in the KITTI formatted data files to lowercase alphabets. Therefore, please do make sure to use the updated lowercase class names in the dataset_config section under target class mapping, when configuring a training experiment. Using incorrect class names here, can lead invalid training experiments with 0 mAP.

When using the tool to create separate tfrecords for evaluation, which may be defined under the dataset_config using the parameter validation_data_source, we advise you to set partition_mode to random with 2 partitions, and an arbitrary val_split (1-100). The dataloader takes care of traversing through all the folds and generating the mAP accordingly.

Chapter 7. CREATING AN EXPERIMENT SPEC FILE

This chapter describes how to create a specification file for model training, inference and evaluation.

7.1. Specification file for classification

Here is an example of a specification file for model classification.

```
model_config {

  # Model architecture can be chosen from:
  # ['resnet', 'vgg', 'googlenet', 'alexnet', 'mobilenet_v1', 'mobilenet_v2',
 'squeezenet', 'darknet', 'googlenet']

  arch: "resnet"

  # for resnet --> n_layers can be [10, 18, 34, 50, 101]
  # for vgg --> n_layers can be [16, 19]
  # for darknet --> n_layers can be [19, 53]


  n_layers: 18
  use_bias: True
  use_batch_norm: True
  all_projections: True
  use_pooling: False
  freeze_bn: False
  freeze_blocks: 0
  freeze_blocks: 1

  # image size should be "3, X, Y", where X,Y >= 16
  input_image_size: "3,224,224"
}

eval_config {
  eval_dataset_path: "/path/to/your/eval/data"
  model_path: "/path/to/your/model"
  top_k: 3
  conf_threshold: 0.5
  batch_size: 256
  n_workers: 8

}
```

```
train_config {
  train_dataset_path: "/path/to/your/train/data"
  val_dataset_path: "/path/to/your/val/data"
  pretrained_model_path: "/path/to/your/pretrained/model"
  # optimizer can be chosen from ['adam', 'sgd']

  optimizer: "sgd"
  batch_size_per_gpu: 256
  n_epochs: 80
  n_workers: 16

  # regularizer
  reg_config {
    type: "L2"
    scope: "Conv2D,Dense"
    weight_decay: 0.00005

  }

  # learning_rate

  lr_config {

    # "step" and "soft_anneal" are supported.

    scheduler: "soft_anneal"

    # "soft_anneal" stands for soft annealing learning rate scheduler.
    # the following 4 parameters should be specified if "soft_anneal" is used.
    learning_rate: 0.005
    soft_start: 0.056
    annealing_points: "0.3, 0.6, 0.8"
    annealing_divider: 10
    # "step" stands for step learning rate scheduler.
    # the following 3 parameters should be specified if "step" is used.
    # learning_rate: 0.006
    # step_size: 10
    # gamma: 0.1

    # "cosine" stands for soft start cosine learning rate scheduler.
    # the following 2 parameters should be specified if "cosine" is used.
    # learning_rate: 0.05
    # soft_start: 0.01

  }
}
```

7.2. Specification file for DetectNet_v2

To do training, evaluation and inference for DetectNet_v2, several components need to be configured, each with their own parameters. The **`tlt-train`** and **`tlt-evaluate`** commands for a DetectNet_v2 experiment share the same configuration file. The **`tlt-infer`** command uses a separate configuration file.

The training and inference tools use a specification file for object detection. The specification file for detection training configures these components of the training pipe:

- Model
- BBox ground truth generation
- Post processing module

- Cost function configuration
- Trainer
- Augmentation model
- Evaluator
- Dataloader

7.2.1. Model config

Core object detection can be configured using the **`model_config`** option in the spec file. Here are the parameters:

Parameter	Datatype	Default	Description	Supported Values
all_projections	bool	False	For templates with shortcut connections, this parameter defines whether or not all shortcuts should be instantiated with 1x1 projection layers irrespective of whether there is a change in stride across the input and output.	True/False (only to be used in resnet templates)
arch	string	resnet	This defines the architecture of the back bone feature extractor to be used to train.	▸ resnet ▸ vgg ▸ mobilenet_v1 ▸ mobilenet_v2 ▸ googlenet
num_layers	int	18	Depth of the feature extractor for scalable templates.	▸ resnets: 10, 18, 34, 50, 101 ▸ vgg: 16, 19

Parameter	Datatype	Default	Description	Supported Values
pretrained model file	string	-	This parameter defines the path to a pretrained tlt model file. If the **load_graph** flag is set to **False,** it is assumed that only the weights of the pretrained model file is to be used. In this case, TLT train constructs the feature extractor graph in the experiment and loads the weights from the pretrained model file whose layer names match. Thus, transfer learning across different resolutions and domains are supported. For layers that may be absent in the pretrained model, the tool initializes them with random weights and skips import for that layer.	Unix path
use_pooling	Boolean	False	Choose between using strided convolutions	False/True

Parameter	Datatype	Default	Description	Supported Values
			or MaxPooling while downsampling. When true, MaxPooling is used to down sample, however for the object detection network, NVIDIA recommends setting this to False and using strided convolutions.	
use_batch_norm	Boolean	False	Boolean variable to use batch normalization layers or not.	True/False
objective_set	Proto Dictionary	-	This defines what objectives is this network being trained for. For object detection networks, set it to learn cov and bbox. These parameters should not be altered for the current training pipeline.	cov {} bbox { scale: 35.0 offset: 0.5 }
dropout_rate	Float	0.0	Probability for drop out	0.0-0.1
training precision	Proto Dictionary	-	Contains a nested parameter that sets the precision of the back-	backend_floatx: FLOAT32

Parameter	Datatype	Default	Description	Supported Values
			end training framework.	
load_graph	Boolean	False	Flag to define whether or not to load the graph from the pretrained model file, or just the weights. For a **pruned,** please remember to set this parameter as **True.** Pruning modifies the original graph, hence the pruned model graph and the weights need to be imported.	True/False
freeze_blocks	float (repeated)	-	This parameter defines which blocks may be frozen from the instantiated feature extractor template, and is different for different feature extractor templates.	- **ResNet series.** For the ResNet series, the block ID's valid for freezing is any subset of [0, 1, 2, 3] (inclusive) - **VGG series.** For the VGG series, the block ID's valid for freezing is any subset of [1, 2, 3, 4, 5] (inclusive) - **MobileNet V1.** For the

Parameter	Datatype	Default	Description	Supported Values
				MobileNet V1, the block ID's valid for freezing is any subset of [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] (inclusive) ▸ **MobileNet V2.** For the MobileNet V2, the block ID's valid for freezing is any subset of [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] (inclusive) ▸ **GoogLeNet.** For the GoogLeNet, the block ID's valid for freezing is any subset of [0, 1, 2, 3, 4, 5, 6, 7] (inclusive)
freeze_bn	Boolean	False	You can choose to freeze the Batch Normalization layers in the model during training.	True/False

Here's a sample model config to instantiate a resnet18 model with pretrained weights and freeze blocks 0 and 1, with all shortcuts being set to projection layers.

```
# Sample model config for to instantiate a resnet18 model with pretrained
 weights and freeze blocks 0, 1
# with all shortcuts having projection layers.
model_config {
  arch: "resnet"
  pretrained_model_file: <path_to_model_file>
  freeze_blocks: 0
  freeze_blocks: 1
  all_projections: True
  num_layers: 18
  use_pooling: False
  use_batch_norm: True
  dropout_rate: 0.0
  training_precision: {
    backend_floatx: FLOAT32
  }
  objective_set: {
    cov {}
    bbox {
      scale: 35.0
      offset: 0.5
    }
  }
}
```

7.2.2. BBox ground truth generator

DetectNet_v2 generates 2 tensors, **cov** and **bbox**. The image is divided into 16x16 grid cells. The cov tensor(short for coverage tensor) defines the number of gridcells that are covered by an object. The bbox tensor defines the normalized image coordinates of the object (x1, y1) top_left and (x2, y2) bottom right with respect to the grid cell. For best results, you can assume the coverage area to be an ellipse within the bbox label, with the maximum confidence being assigned to the cells in the center and reducing coverage outwards. Each class has its own coverage and bbox tensor, thus the shape of the tensors are:

- cov: Batch_size, Num_classes, image_height/16, image_width/16
- bbox: Batch_size, Num_classes * 4, image_height/16, image_width/16 (where 4 is the number of coordinates per cell)

The bbox_rasterizer has the following parameters that are configurable.

Parameter	Datatype	Default	Description	Supported Values
deadzone radius	float	0.67	The area to be considered as dormant (or area of no bboxes) around the ellipse of an object. This is particularly useful in cases	0-1.0

Parameter	Datatype	Default	Description	Supported Values
			of overlapping objects, so that foreground objects and background objects are not confused.	
target_class _config	proto dictionary		This is a nested configuration field that defines the coverage region for an object of a given class. For each class, this field is repeated. The configurable parameters of the target_class _config include: ▸ cov_center_x (float): x-coordinate of the center of the object. ▸ cov_center_y (float): y-coordinate of the center of the object. ▸ cov_radius_x (float): x-radius of the coverage ellipse ▸ cov_radius_y (float): y-radius of the	▸ cov_center _x: 0.0 - 1.0 ▸ cov_center _y: 0.0 - 1.0 ▸ cov_radius _x: 0.0 - 1.0 ▸ cov_radius _y: 0.0 - 1.0 ▸ bbox_min _radius: 0.0 - 1.0

Parameter	Datatype	Default	Description	Supported Values
			coverage ellipse ▸ bbox_min_radius (float): minimum radius of the coverage region to be drawn for boxes.	

Here is a sample rasterizer config for a 3 class detector:

```
# Sample rasterizer configs to instantiate a 3 class bbox rasterizer
bbox_rasterizer_config {
  target_class_config {
    key: "car"
    value: {
      cov_center_x: 0.5
      cov_center_y: 0.5
      cov_radius_x: 0.4
      cov_radius_y: 0.4
      bbox_min_radius: 1.0
    }
  }
  target_class_config {
    key: "cyclist"
    value: {
      cov_center_x: 0.5
      cov_center_y: 0.5
      cov_radius_x: 0.4
      cov_radius_y: 0.4
      bbox_min_radius: 1.0
    }
  }
  target_class_config {
    key: "pedestrian"
    value: {
      cov_center_x: 0.5
      cov_center_y: 0.5
      cov_radius_x: 0.4
      cov_radius_y: 0.4
      bbox_min_radius: 1.0
    }
  }
  deadzone_radius: 0.67
}
```

7.2.3. Post processor

The post processor module generates renderable bounding boxes from the raw detection output. The process includes:

- Filtering out valid detections by thresholding objects using the confidence value in the coverage tensor
- Clustering the raw filtered predictions using DBSCAN to produce the final rendered bounding boxes
- Filtering out weaker clusters based on the final confidence threshold derived from the candidate boxes that get grouped into a cluster

This section defines parameters that configure the post processor. For each class you can train for, the **`postprocessing_config`** has a **`target_class_config`** element, which defines the clustering parameters for this class. The parameters for each target class include:

Parameter	Datatype	Default	Description	Supported Values
key	string	-	The names of the class for which the post processor module is being configured.	The network object class name, which are mentioned in the cost_function_config.
value	clustering _config proto	-	The nested clustering config proto parameter that configures the postprocessor module. The parameters for this module are defined in the next table.	Encapsulated object with parameters defined below.

The **`clustering_config`** element configures the clustering block for this class. Here are the parameters for this element.

Parameter	Datatype	Default	Description	Supported Values
coverate _threshold	float	-	The minimum threshold of the coverage tensor output to be considered as a valid candidate box for clustering. The 4 coordinates from the bbox tensor at the	0.0 - 1.0

Parameter	Datatype	Default	Description	Supported Values
			corresponding indices are passed for clustering.	
dbscan_eps	float	-	The maximum distance between two samples for one to be considered as in the neighborhood of the other. This is not a maximum bound on the distances of points within a cluster. The greater the eps, more boxes are grouped together.	0.0 - 1.0
dbscan _min_samples	float	-	The total weight in a neighborhood for a point to be considered as a core point. This includes the point itself.	0.0 - 1.0
minimum _bounding _box_height	int	-	Minimum height in pixels to consider as a valid detection post clustering.	0 - input image height.

Here is an example of the definition of the postprocessor for a 3 class network learning for **car**, **cyclist**, and **pedestrian**:

```
postprocessing_config {
  target_class_config {
    key: "car"
    value: {
      clustering_config {
        coverage_threshold: 0.005
        dbscan_eps: 0.15
```

```
        dbscan_min_samples: 0.05
        minimum_bounding_box_height: 20
      }
    }
  }
  target_class_config {
    key: "cyclist"
    value: {
      clustering_config {
        coverage_threshold: 0.005
        dbscan_eps: 0.15
        dbscan_min_samples: 0.05
        minimum_bounding_box_height: 20
      }
    }
  }
  target_class_config {
    key: "pedestrian"
    value: {
      clustering_config {
        coverage_threshold: 0.005
        dbscan_eps: 0.15
        dbscan_min_samples: 0.05
        minimum_bounding_box_height: 20
      }
    }
  }
}
```

7.2.4. Cost function

This section helps you configure the cost function to include the classes that you are training for. For each class you want to train, add a new entry of the **target classes** to the spec file. NVIDIA recommends not changing the parameters within the spec file for best performance with these classes. The other parameters remain unchanged here.

```
cost_function_config {
  target_classes {
    name: "car"
    class_weight: 1.0
    coverage_foreground_weight: 0.05
    objectives {
      name: "cov"
      initial_weight: 1.0
      weight_target: 1.0
    }
    objectives {
      name: "bbox"
      initial_weight: 10.0
      weight_target: 10.0
    }
  }
  target_classes {
    name: "cyclist"
    class_weight: 1.0
    coverage_foreground_weight: 0.05
    objectives {
      name: "cov"
      initial_weight: 1.0
      weight_target: 1.0
    }
    objectives {
      name: "bbox"
      initial_weight: 10.0
```

```
      weight_target: 1.0
    }
  }
  target_classes {
    name: "pedestrian"
    class_weight: 1.0
    coverage_foreground_weight: 0.05
    objectives {
      name: "cov"
      initial_weight: 1.0
      weight_target: 1.0
    }
    objectives {
      name: "bbox"
      initial_weight: 10.0
      weight_target: 10.0
    }
  }
  enable_autoweighting: True
  max_objective_weight: 0.9999
  min_objective_weight: 0.0001
}
```

7.2.5. Trainer

Here are the parameters used to configure the trainer:

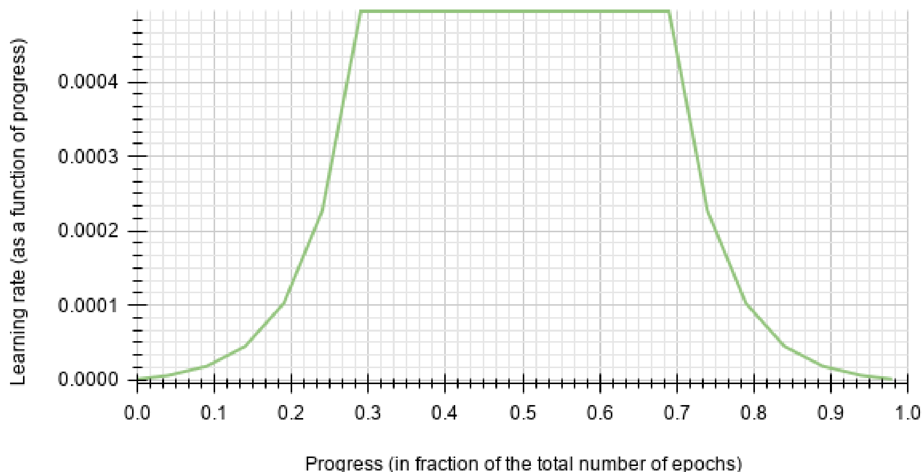
Parameter	Datatype	Default/ Suggested value	Description	Supported values
batch_size_per _gpu	int	32	This parameter defines the number of images per batch per gpu.	>1
num_epochs	int	120	This parameter defines the total number of epochs to run the experiment.	
enable_qat	bool	False	This parameter enables training a model using Quantization Aware Training (QAT). For more information about QAT see Quantization Aware Training.	True, False
learning rate	learning rate scheduler proto	soft_start _annealing	This parameter configures the learning rate	annealing: 0.0-1.0 and

Parameter	Datatype	Default/ Suggested value	Description	Supported values
		_schedule	schedule for the trainer. Currently detectnet_v2 only supports softstart annealing learning rate schedule, and maybe configured using the following parameters: ▸ soft_start (float): Defines the time to ramp up the learning rate from minimum learning rate to maximum learning rate ▸ annealing (float): Defines the time to cool down the learning rate from maximum learning rate to minimum learning rate ▸ minimum _learning _rate	greater than soft_start Soft_start: 0.0 - 1.0 A sample lr plot for a soft start of 0.3 and annealing of 0.1 is shown in the figure below.

Parameter	Datatype	Default/ Suggested value	Description	Supported values
			(float): Minimum learning rate in the learning rate schedule. ▸ maximum_learning_rate (float): Maximum learning rate in the learning rate schedule.	
regularizer	regularizer proto config		This parameter configures the type and the weight of the regularizer to be used during training. The two parameters include: ▸ type: The type of the regularizer being used. ▸ weight: The floating point weight of the regularizer.	The supported values for type are: ▸ NO_REG ▸ L1 ▸ L2
optimizer	optimizer proto config		This parameter defines which optimizer to use for training, and the parameters	

Parameter	Datatype	Default/ Suggested value	Description	Supported values
			to configure it, namely: ▸ epsilon (float): Is a very small number to prevent any division by zero in the implemen -tation ▸ beta1 (float) ▸ beta2 (float)	
cost_scaling	costscaling _config		This parameter enables cost scaling during training. Please leave this parameter untouched currently for the detectnet_v2 training pipe.	cost_scaling { enabled: False initial_exponent: 20.0 increment: 0.005 decrement: 1.0 }
checkpoint interval	float	0/10	The interval (in epochs) at which tlt-train saves intermediate models.	0 to num_epochs

Detectnet_v2 currently supports the soft-start annealing learning rate schedule. The learning rate when plotted as a function of the training progress (0.0, 1.0) results in the following curve.



In this experiment, the soft start was set as 0.3 and annealing as 0.7, with minimum learning rate as 5e-6 and a maximum learning rate or base_lr as 5e-4.

NVIDIA suggests using L1 regularizer when training a network before pruning as L1 regularization helps making the network weights more easily pruned. After pruning, when retraining the networks, NVIDIA recommends turning regularization off by setting the regularization type to **NO_REG**.

Here's a sample training_config block to configure a detectnet_v2 trainer:

```
training_config {
  batch_size_per_gpu: 16
  num_epochs: 80
  learning_rate {
    soft_start_annealing_schedule {
      min_learning_rate: 5e-6
      max_learning_rate: 5e-4
      soft_start: 0.1
      annealing: 0.7
    }
  }
  regularizer {
    type: L1
    weight: 3e-9
  }
  optimizer {
    adam {
      epsilon: 1e-08
      beta1: 0.9
      beta2: 0.999
    }
  }
  cost_scaling {
    enabled: False
    initial_exponent: 20.0
    increment: 0.005
    decrement: 1.0
  }
}
```

7.2.6. Augmentation module

The augmentation module provides some basic pre-processing and augmentation when training. The augmentation_config contains three elements :

- **`preprocessing`**: This nested field configures the input image and ground truth label pre-processing module. It sets the shape of the input tensor to the network. The ground truth labels are pre-processed to meet the dimensions of the input image tensors.

Parameter	Datatype	Default/ Suggested value	Description	Supported Values
output _image _width	int	--	The width of the augmen- tation output. This is the same as the width of the network input and must be a multiple of 16.	>480
output _image _height	int	--	The height of the augmen- tation output. This is the same as the height of the network input and must be a multiple of 16.	>272
output _image	int	1, 3	The channel	1,3

Parameter	Datatype	Default/ Suggested value	Description	Supported Values
_channel			depth of the augmentation output. This is the same as the channel depth of the network input. Currently 1-channel input is not recommended for datasets with jpg images. For png images, both 3 channel RGB and 1 channel monochrome images are supported.	
Min_bbox _height	float		The minimum height of the object labels to be considered for training.	0 - output_image_height
Min_bbox _width	float		The minimum width of the object	0 - output_image_width

Parameter	Datatype	Default/ Suggested value	Description	Supported Values
			labels to be considered for training.	
crop_right	int		The right boundary of the crop to be extracted from the original image.	0 - input image width
crop_left	int		The left boundary of the crop to be extracted from the original image.	0 - input image width
crop_top	int		The top boundary of the crop to be extracted from the original image.	0 - input image height
crop_bottom	int		The bottom boundary of the crop to be extracted from the original image.	0 - input image height
scale_height	float		The floating point factor to scale the height	> 0.0

Parameter	Datatype	Default/ Suggested value	Description	Supported Values
			of the cropped images.	
scale_width	float		The floating point factor to scale the width of the cropped images.	> 0.0

- **`spatial_augmentation`**: This module supports basic spatial augmentation such as flip, zoom and translate which may be configured.

Parameter	Datatype	Default/ Suggested value	Description	Supported Values
hflip _probability	float	0.5	The probability to flip an input image horizontally.	0.0-1.0
vflip _probability	float	0.0	The probability to flip an input image vertically.	0.0-1.0
zoom_min	float	1.0	The minimum zoom scale of the input image.	>0.0
zoom_max	float	1.0	The maximum zoom scale of the input image.	>0.0

Parameter	Datatype	Default/ Suggested value	Description	Supported Values
translate_max_x	int	8.0	The maximum translation to be added across the x axis.	0.0 - output_image_width
translate_max_y	int	8.0	The maximum translation to be added across the y axis	0.0 - output_image_height
rotate_rad_max	float	0.69	The angle of rotation to be applied to the images and the training labels. The range is defined between [-rotate_rad_max, rotate_rad_max]	> 0.0 (modulo 2*pi

- **`color_augmentation`**: This module configures the color space transformations, such as color shift, hue_rotation, saturation shift, and contrast adjustment.

Parameter	Datatype	Default/ Suggested value	Description	Supported Values
color_shift_stddev	float	0.0	The standard devidation value for the color shift.	0.0-1.0
hue_rotation	float	25.0	The maximum rotation angle for	0.0-360.0

Parameter	Datatype	Default/ Suggested value	Description	Supported Values
_max			the hue rotation matrix.	
saturation _shift_max	float	0.2	The maximum shift that changes the saturation. A value of 1.0 means no change in saturation shift.	0.0 - 1.0
contrast _scale_max	float	0.1	The slope of the contrast, as rotated around the provided center. A value of 0.0 leaves the contrast unchanged.	0.0 - 1.0
contrast _center	float	0.5	The center around which the contrast is rotated. Ideally this is set to half of the maximum pixel value. (Since our input images are scaled between 0 and 1.0,	0.5

Parameter	Datatype	Default/ Suggested value	Description	Supported Values
			you can set this value to 0.5).	

The dataloader online augmentation pipeline applies spatial and color-space augmentation transformations in the below mentioned order.

1. The dataloader first performs the pre-processing operations on the input data (image and labels) read from the tfrecords files. Here the images and labels cropped and scaled based on the parameters mentioned in the **`preprocessing`** config. The boundaries of generating the cropped image and labels from the original image is defined by the **`crop_left`**, **`crop_right`**, **`crop_top`** and **`crop_bottom`** parameters. This cropped data is then scaled by the scale factors defined by **`scale_height`** and **`scale_width`**. These transformation matrices for these operations are computed globally and do not change per image.
2. The net tensors generated from the pre-processing blocks are then passed through a pipeline of random augmentations in spatial and color domain. The spatial augmentations are applied to both images and the label coordinates, while the color augmentations are applied only to the images. Inorder to apply color augmentations the output_image_channel parameter must be set to 3. For monochrome tensors color augmentations are not applied. The spatial and color transformation matrices are computed per image based on a uniform distribution along the max and min ranges defined by the **`spatial_augmentation`** and **`color_augmentation`** config parameters.
3. Once the spatial and color augmented net input tensors are generated, the output is then padded with zeros or clipped along the right and bottom edge of the image to fit the output dimensions defined in the **`preprocessing`** config.

Here is a sample augmentation config element:

```
# Sample augementation config for
augmentation_config {
  preprocessing {
    output_image_width: 960
    output_image_height: 544
    output_image_channel: 3
    min_bbox_width: 1.0
    min_bbox_height: 1.0
  }
  spatial_augmentation {

    hflip_probability: 0.5
    vflip_probability: 0.0
    zoom_min: 1.0
    zoom_max: 1.0
    translate_max_x: 8.0
    translate_max_y: 8.0
  }
  color_augmentation {
    color_shift_stddev: 0.0
    hue_rotation_max: 25.0
    saturation_shift_max: 0.2
    contrast_scale_max: 0.1
```

```
    contrast_center: 0.5
  }
}
```

If the output image height and the output image width of the preprocessing block doesn't match with the dimensions of the input image, the dataloader either pads with zeros, or crops to fit to the output resolution. It does not resize the input images and labels to fit.

7.2.7. Configuring the evaluator

The evaluator in the detection training pipe can be configured using the **`evaluation_config`** parameters.

Parameter	Datatype	Default/ Suggested value	Description	Supported Values
average_precision_mode		Sample	The mode in which the average precision for each class is calculated.	‣ SAMPLE: This is the ap calculation mode using 11 evenly spaced recall points as used in the Pascal VOC challenge 2007. ‣ INTEGRATE: This is the ap calculation mode as used in the 2011 challenge
validation_period_during_training	int	10	The interval at which evaluation is run during training. The evaluation is run at this interval starting from the value of the first	1 - total number of epochs

Parameter	Datatype	Default/ Suggested value	Description	Supported Values
			validation epoch parameter as specified below.	
first _validation _epoch	int	30	The first epoch to start running validation. Ideally it is preferred to wait for atleast 20-30% of the total number of epochs before starting evaluation, since the predictions in the initial epochs would be fairly inaccurate. Too many candidate boxes may be sent to clustering and this can cause the evaluation to slow down.	1 - total number of epochs
minimum _detection _ground_truth _overlap	proto dictionary		Minimum IOU between ground truth and predicted box after clustering to call a valid detection. This parameter is a repeatable dictionary, and a separate one must be defined for every class. The members include:	

Parameter	Datatype	Default/ Suggested value	Description	Supported Values
			▶ key (string): class name ▶ value (float): intersection over union value	
evaluation _box_config	proto dictionary		This nested configuration field configures the min and max box dimensions to be considered as a valid ground truth and prediction for AP calculation.	

The evaluation_box_config field has these configurable inputs.

Parameter	Datatype	Default/ Suggested value	Description	Supported Value
minimum _height	float	10	Minimum height in pixels for a valid ground truth and prediction bbox.	0. - model image height
minimum _width	float	10	Minimum width in pixels for a valid ground truth and prediction bbox.	0. - model image width
maximum _height	float	9999	Maximum height in pixels for a valid ground truth and prediction bbox.	minimum_height - model image height

Parameter	Datatype	Default/ Suggested value	Description	Supported Value
maximum _width	float	9999	Maximum width in pixels for a valid ground truth and prediction bbox.	minimum _width - model image width

```
# Sample evaluation config to run evaluation in integrate mode for the given 3
 class model,
# at every 10th epoch starting from the epoch 1.
evaluation_config {
  average_precision_mode: INTEGRATE
  validation_period_during_training: 10
  first_validation_epoch: 1
  minimum_detection_ground_truth_overlap {
    key: "car"
    value: 0.7
  }
  minimum_detection_ground_truth_overlap {
    key: "person"
    value: 0.5
  }
  minimum_detection_ground_truth_overlap {
    key: "bicycle"
    value: 0.5
  }
  evaluation_box_config {
    key: "car"
    value {
      minimum_height: 4
      maximum_height: 9999
      minimum_width: 4
      maximum_width: 9999
    }
  }
  evaluation_box_config {
    key: "person"
    value {
      minimum_height: 4
      maximum_height: 9999
      minimum_width: 4
      maximum_width: 9999
    }
  }
  evaluation_box_config {
    key: "bicycle"
    value {
      minimum_height: 4
      maximum_height: 9999
      minimum_width: 4
      maximum_width: 9999
    }
  }
}
```

7.2.8. Dataloader

This section defines the parameters to configure the dataloader. Here, you define the path to the data you want to train on and the class mapping for classes in the dataset that the network is to be trained for. The parameters in the dataset config are:

- **data_sources**: Captures the path to TFrecords to train on. This field contains 2 parameters:
  - **tfrecords_path**: Path to the individual TFrecords files. This path follows the UNIX style pathname pattern extension, so a common pathname pattern that captures all the tfrecords files in that directory can be used.
  - **image_directory_path**: Path to the training data root from which the tfrecords was generated.
- **image_extension**: Extension of the images to be used.
- **target_class_mapping**: This parameter maps the class names in the tfrecords to the target class to be trained in the network. An element is defined for every source class to target class mapping. This field was included with the intention of grouping similar class objects under one umbrella. For eg: car, van, heavy_truck etc may be grouped under automobile. The “key” field is the value of the class name in the tfrecords file, and “value” field corresponds to the value that the network is expected to learn.
- **validation_fold**: In case of an **n** fold tfrecords, you define the index of the fold to use for validation. For **sequence wise** validation please choose the validation fold in the range [0, N-1]. For a **random split** partitioning, please force the validation fold index to 0 as the tfrecord is just 2-fold.

The class names key in the target_class_mapping must be identical to the one shown in the dataset converter log, so that the correct classes are picked up for training.

```
dataset_config {
  data_sources: {
    tfrecords_path: "<path to the training tfrecords root/tfrecords train
 pattern>"
    image_directory_path: "<path to the training data source>"
  }
  image_extension: "jpg"
  target_class_mapping {
      key: "car"
      value: "car"
  }
  target_class_mapping {
      key: "automobile"
      value: "car"
  }
  target_class_mapping {
      key: "heavy_truck"
      value: "car"
  }
  target_class_mapping {
```

```
      key: "person"
      value: "pedestrian"
  }
  target_class_mapping {
      key: "rider"
      value: "cyclist"
  }
  validation_fold: 0
}
```

In this example the tfrecords is assumed to be multi-fold, and the fold number to validate on is defined. However, evaluation doesn't necessarily have to be run on a split of the training set. Many ML engineers choose to evaluate the model on a well chosen evaluation dataset that is exclusive of the training dataset. If you prefer to run evaluation on a different validation dataset as opposed to a split from the training dataset, then please convert this dataset into tfrecords as well using the **`tlt-dataset-convert`** tool as mentioned in the here, and use the **`validation_data_source`** field in the **`dataset_config`** to define this. In this case, please do not forget to remove the **`validation_fold`** field from the spec. When generating the TFRecords for evaluation by using the **`validation_data_source`** field, please review the notes here.

```
validation_data_source: {
    tfrecords_path: " <path to tfrecords to validate on>/tfrecords validation
 pattern>"
    image_directory_path: " <path to validation data source>"
}
```

7.2.9. Specification file for inference

This spec file configures the tlt-infer tool of detectnet to generate valid bbox predictions. The inference tool consists of 2 blocks, namely the inferencer and the bbox handler. The inferencer instantiates the model object and preprocessing pipe, which the bbox handler handles the post processing, rendering of bounding boxes and the serialization to KITTI format output labels.

7.2.9.1. Inferencer

The inferencer instantiates a model object that generates the raw predictions from the trained model. The model may be defined to run inference in the TLT backend or the TensorRT backend. The inferencer_config parameters are explained in the table below.

Parameter	Datatype	Default	Description	Supported Value
target_classes	String (repeated)	None	The names of the target classes the model should output. For a multi-class model this parameter is repeated N times. The	For example, for the 3 class kitti model it will be: car cyclist pedestrian

Parameter	Datatype	Default	Description	Supported Value
			number of classes must be equal to the number of classes and the order must be the same as the classes in costfunction_config of the training config file.	
batch_size	int	1	The number of images per batch of inference	Max number of images that can be fit in 1 GPU
image_height	int	384	The height of the image in pixels at which the model will be inferred.	>16
image_width	int	1248	The width of the image in pixels at which the model will be inferred.	>16
image_channels	int	3	The number of channels per image.	1,3
gpu_index	int	0	The index of the GPU to run inference on. This is useful only in TLT inference. For tensorRT inference, by default, the GPU of choice in ‘0’.	
tensorrt_config	TensorRTConfig	None	Proto config to instantiate a TensorRT object	

Parameter	Datatype	Default	Description	Supported Value
tlt_config	TLTConfig	None	Proto config to instantiate a TLT model object.	

As mentioned earlier, the tlt-infer tool is capable of running inference using the native TLT backend and the TensorRT backend. They can be configured by using the tensorrt_config proto element, or the tlt_config proto element respectively. You may use only one of the two in a single spec file. The definitions of the two model objects are:

Parameter	Datatype	Default	Description	Supported Value
parser	enum	ETLT	The tensorrt parser to be invoked. Only ETLT parser is supported.	ETLT
etlt_model	string	None	Path to the exported etlt model file.	Any existing etlt file path.
backend_data_type	enum	FP32	The data type of the backend TensorRT inference engine. For int8 mode, please be sure to mention the calibration_cache.	FP32 FP16 INT8
save_engine	bool	False	Flag to save a TensorRT engine from the input etlt file. This will save initialization time if inference needs to be run on the same etlt file and there are no changes needed to be made to the inferencer object.	True, False

Parameter	Datatype	Default	Description	Supported Value
trt_engine	string	None	Path to the TensorRT engine file. This acts asa n I/O parameter. If the path defined here is not an engine file, then the tlt-infer tool creates a new TensorRT engine from the etlt file. If there exists an engine already, the tool, re-instantiates the inferencer from the engine defined here.	UNIX path string
calibration _config	CalibratorConfig Proto	None	This is a required parameter when running in the int8 inference mode. This proto object contains parameters used to define a calibrator object. Namely: calibration_cache: path to the calibration cache file generated using tlt-export	

7.2.9.2. TLT_Config

Parameter	Datatype	Default	Description	Supported Values
model	string	None	The path to the .tlt model file.	UNIX Path string

Since detectnet is a full convolutional neural net, the model can be inferred at a different inference resolution than the resolution at which it was trained. The input dims of the network will be overridden to run inference at this resolution, if they are different from the training resolution. There may be some regression in accuracy when running inference at a different resolution since the convolutional kernels don't see the object features at this shape.

A sample inferencer_config element for the inferencer spec is defined here.

```
inferencer_config{
  # defining target class names for the experiment.
  # Note: This must be mentioned in order of the networks classes.
  target_classes: "car"
  target_classes: "cyclist"
  target_classes: "pedestrian"
  # Inference dimensions.
  image_width: 1248
  image_height: 384
  # Must match what the model was trained for.
  image_channels: 3
  batch_size: 16
  gpu_index: 0
  # model handler config
  tensorrt_config{
    parser:  ETLT
    etlt_model: "/path/to/model.etlt"
    backend_data_type: INT8
    save_engine: true
    trt_engine: "/path/to/trt/engine/file"
    calibrator_config{
        calibration_cache: "/path/to/calibration/cache"
        n_batches: 10
        batch_size: 16
    }
  }
}
```

7.2.9.3. Bbox handler

The bbox handler takes care of the post processing the raw outputs from the inferencer. It performs the following steps:

1. Thresholding the raw outputs to defines grid-cells where the detections may be present per class.
2. Reconstructing the image space coordinates from the raw coordinates of the inferencer.
3. Clustering the raw thresholded predictions.

4. Filtering the clustered predictions per class.
5. Rendering the final bounding boxes on the image in its input dimensions and serializing them to KITTI format metadata.

The parameters to configure the bbox handler are defined below.

Parameter	Datatype	Default	Description	Supported Value
kitti_dump	bool	false	Flag to enable saving the final output predictions per image in KITTI format.	true, false
disable_overlay	bool	true	Flag to disable bbox rendering per image.	true, false
overlay _linewidth	int	1	Thickness in pixels of the bbox boundaries.	>1
classwise_bbox _handler_config	ClasswiseCluster Config (repeated)	None	This is a repeated class-wise dictionary of post-processing parameters. DetectNet_v2 uses dbscan clustering to group raw bboxes to final predictions. For models with several output classes, it may be cumbersome to define a separate dictionary for each class. In such a situation, a default class may be used for all classes in the network.	

The **classwise_bbox_handler_config** is a Proto object containing several parameters to configure the clustering algorithm as well as the bbox renderer.

Parameter	Datatype	Default	Description	Supported Value
confidence_model	string	aggregate_cov	Algorithm to compute the final confidence of the clustered bboxes. In the aggregate_cov mode, the final confidence of a detection is the sum of the confidences of all the candidate bboxes in a cluster. In mean_cov mode, the final confidence is the mean confidence of all the bboxes in the cluster.	aggregate_cov, mean_cov
confidence_threshold	float	0.9 in aggregate_cov mode. 0.1 in mean_cov_mode.	The threshold applied to the final aggregate confidence values to render the bboxes.	In aggregate_cov: Maybe tuned to any float value > 0.0 In mean_cov: 0.0 - 1.0
bbox_color	BBoxColor Proto Object	None	RGB channel wise color intensity per box.	R: 0 - 255 G: 0 - 255 B: 0 - 255
clustering_config	ClusteringConfig	None	Proto object to configure the DBSCAN clustering algorithm. Contains the	coverage_threshold: 0.005 dbscan_eps: 0.3

Parameter	Datatype	Default	Description	Supported Value
			following sub parameters. coverage_threshold: The threshold applied to the raw network confidence predictions as a first stage filtering technique. dbscan_eps: (float) The search distance to group together boxes into a single cluster. The lesser the number, the more boxes are detected. Eps of 1.0 groups all boxes into a single cluster. dbscan_min_samples: (float) The weight of the boxes in a cluster. min_bbox_height: (int) The minimum height of the bbox to be clustered.	dbscan_min_samples: 0.05 minimum_bounding_box_height: 4

A sample bbox_handler_config element is defined below.

```
bbox_handler_config{
  kitti_dump: true
  disable_overlay: false
  overlay_linewidth: 2
  classwise_bbox_handler_config{
    key:"car"
    value: {
      confidence_model: "aggregate_cov"
      output_map: "car"
      confidence_threshold: 0.9
```

```
      bbox_color{
        R: 0
        G: 255
        B: 0
      }
      clustering_config{
        coverage_threshold: 0.00
        dbscan_eps: 0.3
        dbscan_min_samples: 0.05
        minimum_bounding_box_height: 4
      }
    }
  }
  classwise_bbox_handler_config{
    key:"default"
    value: {
      confidence_model: "aggregate_cov"
      confidence_threshold: 0.9
      bbox_color{
        R: 255
        G: 0
        B: 0
      }
      clustering_config{
        coverage_threshold: 0.00
        dbscan_eps: 0.3
        dbscan_min_samples: 0.05
        minimum_bounding_box_height: 4
      }
    }
  }
}
```

7.3. Specification file for FasterRCNN

Below is a sample of the FasterRCNN spec file. It has two major components: **`network_config`** and **`training_config`**, explained below in detail. The format of the spec file is a protobuf text(prototxt) message and each of its fields can be either a basic data type or a nested message. The top level structure of the spec file is summarized in the table below.

Here's a sample of the FasterRCNN spec file:

```
random_seed: 42
enc_key: 'tlt'
verbose: True
network_config {
  input_image_config {
    image_type: RGB
    image_channel_order: 'bgr'
    size_height_width {
      height: 384
      width: 1248
    }
    image_channel_mean {
      key: 'b'
      value: 103.939
    }
    image_channel_mean {
      key: 'g'
      value: 116.779
    }
    image_channel_mean {
```

```
      key: 'r'
      value: 123.68
    }
    image_scaling_factor: 1.0
    max_objects_num_per_image: 100
  }
  feature_extractor: "resnet:18"
  anchor_box_config {
    scale: 64.0
    scale: 128.0
    scale: 256.0
    ratio: 1.0
    ratio: 0.5
    ratio: 2.0
  }
  freeze_bn: True
  freeze_blocks: 0
  freeze_blocks: 1
  roi_mini_batch: 256
  rpn_stride: 16
  conv_bn_share_bias: False
  roi_pooling_config {
    pool_size: 7
    pool_size_2x: False
  }
  all_projections: True
  use_pooling:False
}
training_config {
  kitti_data_config {
    data_sources: {
      tfrecords_path: "/workspace/tlt-experiments/tfrecords/kitti_trainval/
kitti_trainval*"
      image_directory_path: "/workspace/tlt-experiments/data/training"
    }
    image_extension: 'png'
    target_class_mapping {
      key: 'car'
      value: 'car'
    }
    target_class_mapping {
      key: 'van'
      value: 'car'
    }
    target_class_mapping {
      key: 'pedestrian'
      value: 'person'
    }
    target_class_mapping {
      key: 'person_sitting'
      value: 'person'
    }
    target_class_mapping {
      key: 'cyclist'
      value: 'cyclist'
    }
    validation_fold: 0
  }
  data_augmentation {
    preprocessing {
      output_image_width: 1248
      output_image_height: 384
      output_image_channel: 3
      min_bbox_width: 1.0
      min_bbox_height: 1.0
    }
    spatial_augmentation {
```

```
      hflip_probability: 0.5
      vflip_probability: 0.0
      zoom_min: 1.0
      zoom_max: 1.0
      translate_max_x: 0
      translate_max_y: 0
    }
    color_augmentation {
      hue_rotation_max: 0.0
      saturation_shift_max: 0.0
      contrast_scale_max: 0.0
      contrast_center: 0.5
    }
  }
  enable_augmentation: True
  batch_size_per_gpu: 16
  num_epochs: 12
  pretrained_weights: "/workspace/tlt-experiments/data/faster_rcnn/resnet18.h5"
  #resume_from_model: "/workspace/tlt-experiments/data/faster_rcnn/
resnet18.epoch2.tlt"
  #retrain_pruned_model: "/workspace/tlt-experiments/data/faster_rcnn/
model_1_pruned.tlt"
  output_model: "/workspace/tlt-experiments/data/faster_rcnn/
frcnn_kitti_resnet18.tlt"
  rpn_min_overlap: 0.3
  rpn_max_overlap: 0.7
  classifier_min_overlap: 0.0
  classifier_max_overlap: 0.5
  gt_as_roi: False
  std_scaling: 1.0
  classifier_regr_std {
    key: 'x'
    value: 10.0
  }
  classifier_regr_std {
    key: 'y'
    value: 10.0
  }
  classifier_regr_std {
    key: 'w'
    value: 5.0
  }
  classifier_regr_std {
    key: 'h'
    value: 5.0
  }
  rpn_mini_batch: 256
  rpn_pre_nms_top_N: 12000
  rpn_nms_max_boxes: 2000
  rpn_nms_overlap_threshold: 0.7
  reg_config {
    reg_type: 'L2'
    weight_decay: 1e-4
  }
  optimizer {
    adam {
      lr: 0.00001
      beta_1: 0.9
      beta_2: 0.999
      decay: 0.0
    }
  }
  lr_scheduler {
    step {
      base_lr: 0.00016
      gamma: 1.0
      step_size: 30
```

```
    }
  }
  lambda_rpn_regr: 1.0
  lambda_rpn_class: 1.0
  lambda_cls_regr: 1.0
  lambda_cls_class: 1.0
  inference_config {
    images_dir: '/workspace/tlt-experiments/data/testing/image_2'
    model: '/workspace/tlt-experiments/data/faster_rcnn/
frcnn_kitti_resnet18.epoch12.tlt'
    detection_image_output_dir: '/workspace/tlt-experiments/data/faster_rcnn/
inference_results_imgs'
    labels_dump_dir: '/workspace/tlt-experiments/data/faster_rcnn/
inference_dump_labels'
    rpn_pre_nms_top_N: 6000
    rpn_nms_max_boxes: 300
    rpn_nms_overlap_threshold: 0.7
    bbox_visualize_threshold: 0.6
    classifier_nms_max_boxes: 300
    classifier_nms_overlap_threshold: 0.3
  }
  evaluation_config {
    model: '/workspace/tlt-experiments/data/faster_rcnn/
frcnn_kitti_resnet18.epoch12.tlt'
    labels_dump_dir: '/workspace/tlt-experiments/data/faster_rcnn/
test_dump_labels'
    rpn_pre_nms_top_N: 6000
    rpn_nms_max_boxes: 300
    rpn_nms_overlap_threshold: 0.7
    classifier_nms_max_boxes: 300
    classifier_nms_overlap_threshold: 0.3
    object_confidence_thres: 0.0001
    use_voc07_11point_metric:False
  }
}
```

Field	Description	Data Type and Constraints	Recommended/ Typical Value
random_seed	The random seed for the experiment.	Unsigned int	42
enc_key	The encoding and decoding key for the TLT models, can be override by the command line arguments of **tlt-train**, **tlt-evaluate** and **tlt-infer** for FasterRCNN.	Str, should not be empty	-
verbose	Controls the logging level during the experiments. Will print more logs if **True**.	Boolean(**True** or **False**)	False

Field	Description	Data Type and Constraints	Recommended/ Typical Value
network_config	The architecture of the model and its input format.	message	-
training_config	The configurations for the training, evaluation and inference for this experiment.	message	-

7.3.1. Network config

The network config(network_config) defines the model structure and the its input format. This model is used for training, evaluation and inference. Detailed description is summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/ Typical Value
input_image _config	Defines the input image format, including the image channel number, channel order, width and height, and the preprocessings (subtract per-channel mean and divided by a scaling factor) for it before feeding input the model. See below for details.	message	-
input_image _config.image_type	The image type, can be either RGB or gray-scale image.	enum type. Either **RGB** or **GRAYSCALE**	RGB
input_image _config.image _channel_order	The image channel order.	str type. If **image_type** is RGB, **'rgb'** or **'bgr'** is valid. If the **image_type** is **GRAYSCALE**, only **'l'** is valid.	**'bgr'**

Field	Description	Data Type and Constraints	Recommended/ Typical Value
input_image _config.size _height_width	The height and width as the input dimension of the model.	message	-
input_image _config.image _channel_mean	Per-channel mean value to subtract by for the image preprocessing.	map(dict) type from channel names to the corresponding mean values. Each of the mean values should be non-negative	`image_channel_mean { key: 'b' value: 103.939 } image_channel_mean { key: 'g' value: 116.779 } image_channel_mean { key: 'r' value: 123.68 }`
input_image _config.image _scaling_factor	Scaling factor to divide by for the image preprocessing.	float type, should be a positive scalar.	1.0
input_image _config.max_objects _num_per_image	The maximum number of objects in an image for the dataset. Usually, the number of objects in different images is different, but there is a maximum number. Setting this field to be no less than this maximum number. This field is used to pad the objects number to the same value so you can make multi-batch and multi-gpu training of FasterRCNN possible.	unsigned int, should be positive.	100

Field	Description	Data Type and Constraints	Recommended/ Typical Value
feature_extractor	The feature extractor(backbone) for the FasterRCNN model. FasterRCNN supports 12 backbones. Note: FasterRCNN actually supports another backbone: **vgg**. This backbone is a VGG16 backbone exactly the same as in Keras applications. The layer names matter when loading a pretrained weights. If you want to load a pretrained weights that has the same names as VGG16 in the Keras applications, you should use this backbone. Since this is indeed duplicated with the **vgg:16** backbone, you might consider using **vgg:16** for production. The only use case for the vgg backbone is to reproduce the original Caffe implementation of VGG16 FasterRCNN that uses ImageNet weights as pretrained weights.	str type. The architecture can be ResNet, VGG , GoogLeNet, MobileNet or DarkNet. For each specific architecture, it can have different layers or versions. Details listed below. ResNet series: **resnet:10**, **resnet:18**, **resnet:34**, **resnet:50**, **resnet:101** VGG series: **vgg:16**, **vgg:19** GoogLeNet: **googlenet** MobileNet series: **mobilenet_v1**, **mobilenet_v2** DarkNet: **darknet:19**, **darknet:53** Here a notational convention can be used, i.e., for models that can have different numbers of layers, use a colon followed by the layer number as the suffix of the model name. E.g., **resnet:<layer_number>**	-
anchor_box _config	The anchor box configuration defines the set of anchor box sizes	Message type that contains two sub-fields: **scale** and **ratio**. Each of them	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
	and aspect ratios in a FasterRCNN model.	is a list of floating point numbers. The **`scale`** field defines the absolute anchor sizes in pixels(at input image resolution). The **`ratio`** field defines the aspect ratios of each anchor.	
`freeze_bn`	whether or not to freeze all the BatchNormalization layers in the model. You can choose to freeze the BatchNormalization layers in the model during training. This is a common trick when training a FasterRCNN model. Note: Freezing the BatchNormalization layer will only freeze the moving mean and moving variance in it, while the gamma and beta parameters are still trainable.	Boolean (**`True`** or **`False`**)	If you train with a small batch size, usually you need to set the field to be **`True`** and use good pretrained weights to make the training converge well. But if you train with a large batch size(e.g., >=16), you can set it to be **`False`** and let the BatchNormalization layer to calculate the moving mean and moving variance by itself.
`freeze_blocks`	The list of block IDs to be frozen in the model during training. You can choose to freeze some of the CNN blocks in the model to make the training more stable and/or easier to converge. The	list(repeated integers) **ResNet series** - For the ResNet series, the block IDs valid for freezing is any subset of `[0, 1, 2, 3]`(inclusive) **VGG series** - For the VGG series,	Leave it empty(`[]`)

Field	Description	Data Type and Constraints	Recommended/ Typical Value
	definition of a block is heuristic for a specific architecture. For example, by stride or by logical blocks in the model, etc. However, the block ID numbers identify the blocks in the model in a sequential order so you don't have to know the exact locations of the blocks when you do training. A general principle to keep in mind is: the smaller the block ID, the closer it is to the model input; the larger the block ID, the closer it is to the model output. You can divide the whole model into several blocks and optionally freeze a subset of it. Note that for FasterRCNN you can only freeze the blocks that are before the ROI pooling layer. Any layer after the ROI pooling layer will not be frozen any way. For different backbones, the number of blocks and the block ID for each block are different. It deserves some detailed explanations on	the block IDs valid for freezing is any subset of `[1, 2, 3, 4, 5]` (inclusive) **GoogLeNet**- For the GoogLeNet, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7]` (inclusive) **MobileNet V1**- For the MobileNet V1, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]` (inclusive) **MobileNet V2**- For the MobileNet V2, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]` (inclusive) **DarkNet** - For the DarkNet 19 and DarkNet 53, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4,5]` (inclusive)	

Field	Description	Data Type and Constraints	Recommended/ Typical Value
	how to specify the block ID's for each backbone.		
roi_mini_batch	The batch size used to train the RCNN after ROI pooling.	A positive integer, usually uses **128**, **256**, etc.	256
RPN_stride	The cumulative stride from the model input to the RPN. This value is fixed(**16**) for current implementation.	positive integer	16
conv_bn_share_bias	A Boolean value to indicate whether or not to share the bias of the convolution layer and the BatchNormalization (BN) layer immediately after it. Usually you share the bias between them to reduce the model size and avoid redundancy of parameters. When using the pretrained weights, make sure the value of this parameter aligns with the actual configuration in the pretrained weights otherwise error will be raised when loading the pretrained weights.	Boolean (**True** or **False**)	True
roi_pooling_config	The configuration for the ROI pooling layer.	Message type that contains two sub-fields: **pool_size** and **pool_size_2x**. See below for details.	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
roi_pooling _config.pool _size	The output spatial size(height and width) of ROIs. Only square spatial size is supported currently, i.e. height=width.	unsigned int, should be positive.	7
roi_pooling _config.pool _size_2x	A Boolean value to indicate whether to do the ROI pooling at **2*pool_size** followed by a 2 x 2 pooling operation or do ROI pooling directly at **pool_size** without pooling operation. E.g. if **pool_size = 7**, and **pool_size_2x=True**, it means you do ROI pooling to get an output that has a spatial size of 14 x 14 followed by a 2 x 2 pooling operation to get the final output tensor.	Boolean (**True** or **False**)	-
all_projections	This field is only useful for models that have shortcuts in it. These models include ResNet series and the MobileNet V2. If **all_projections =True**, all the pass-through shortcuts will be replaced by a projection layer that has the same number of output channels as it.	Boolean (**True** or False)	True

Field	Description	Data Type and Constraints	Recommended/ Typical Value
use_pooling	This parameter is only useful for VGG series and ResNet series. When **use_pooling=True**, you can use pooling in the model as the original implementation, otherwise use strided convolution to replace the pooling operations in the model. If you want to improve the inference FPS(Frame Per Second) performance, you can try to set **use_pooling=False**.	Boolean (**True** or False)	False

7.3.2. Training Configuration

The training configuration(**training_config**) defines the parameters needed for the training, evaluation and inference. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/ Typical Value
kitti_data_config	The dataset used for training, evaluation and inference.	Message type. It has the same structure as the **dataset_config** message in DetectNet_v2 spec file. Refer to the DetectNet_v2 **dataset_config** documentation for the details.	-
data_augmentation	Defines the data augmentation pipeline during training.	Message type. It has the same structure as the **data_augmentation** message in the	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
		DetectNet_v2 spec file. Refer to the DetectNet_v2 **data_augmentation** documentation for the details.	
enable _augmentation	Whether or not to enable the data augmentation during training. If this parameter is **False**, the training will not have any data augmentation operation even if you have already defined the data augmentation pipeline in the **data_augmentation** field in spec file. This feature is mostly used for debugging of the data augmentation pipeline.	Boolean(**True** or **False**)	**True**
batch_size _per_gpu	The training batch size on each GPU device. The actual total batch size will be **batch_size _per_gpu** multiplied by the number of GPUs in a multi-gpu training scenario.	unsigned int, positive.	Change the **batch_size_per _gpu** to adapt the capability of your GPU device.
num_epochs	The number of epochs for the training.	unsigned int, positive.	**20**
pretrained _weights	Absolute path to the pretrained weights file used to initialize the training model. The pretrained	Str type. Can be left empty. In that case, the FasterRCNN model will use random	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
	weights file can be either a Keras weights file(with **`.h5`** suffix), a Keras model file(with **`.hdf5`** suffix) or a TLT model(with **`.tlt`** suffix, trained by TLT). If the file is a model file(**`.tlt`** or **`.hdf5`**), TLT will extract the weights from it and then load the weights for initialization. Files with any other formats are not supported as pretrained weights. Note that the pretrained weights file is agnostic to the input dimensions of the FasterRCNN model so the model you are training can have different input dimensions from the input dimensions specified in the pretrained weights. Normally, the pretrained weights file is only useful during the initial training phase in a TLT workflow.	initialization for its weights. Usually, FasterRCNN model needs a pretrained weights for good convergence of training.	
`resume_from_model`	Absolute path to the checkpoint **`.tlt`** model that you want to resume the training from. This is useful in some cases when the training process is interrupted for	Str type. Leave it empty when you are not resuming the training, i.e., train from epoch 0.	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
	some reason and you don't want to redo the training from epoch 0(or 1 in 1-based indexing). In that case, you can use the last checkpoint as the model you will resume from, to save the training time.		
retrain_pruned_model	Path to the pruned model that you can load and do the retraining. This is used in the retraining phase in a TLT workflow. The model is the output model of the pruning phase.	Str type. Leave it empty when you are not in the retraining phase.	-
output_model	Absolute path to the output **.tlt** model that the training/retraining will save. Note that this path is not the actual path of the **.tlt** models. For example, if the **output_model** is **'/workspace/tlt_training/resnet18.tlt'**, then the actual output model path will be **'/workspace/tlt_training/resnet18 .epoch<k>.tlt'** where **<k>** denotes the epoch number of during training. In this way, you	Str type. Cannot be empty.	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
	can distinguish the output models for different epochs. Here, the epoch number **<k>** is a 1-based index.		
checkpoint _interval	The epoch interval that controls how frequent TLT will save the checkpoint during training. TLT will save the checkpoint at every **checkpoint _interval** epoch(1 based index). For example, if the **num_epochs** is 12 and **checkpoint _interval** is 3, then TLT will save checkpoint at the end of epoch 3, 6, 9, and 12. If this parameter is not specified, then it defaults to **checkpoint _interval=1**.	unsigned int, can be omitted(defaults to 1).	**-**
rpn_min_overlap	The lower IoU threshold used to map the anchor boxes to ground truth boxes. If the IoU of an anchor box and any ground truth box is below this threshold, you can treat this anchor box as a negative anchor box.	Float type, scalar. Should be in the interval **(0, 1)**.	**0.3**
rpn_max_overlap	The upper IoU threshold used to map the anchor boxes to ground	Float type, scalar. Should be in the interval **(0, 1)**	**0.7**

Field	Description	Data Type and Constraints	Recommended/ Typical Value
	truth boxes. If the IoU of an anchor box and at least one ground truth box is above this threshold, you can treat this anchor box as a positive anchor box.	and greater than **`rpn_min_overlap`**.	
`classifier_min_overlap`	The lower IoU threshold to generate the proposal target. If the IoU of an ROI and a ground truth box is above the threshold and below the **`classifier_max_overlap`**, then this ROI is regarded as a negative ROI(background) when training the RCNN.	floating-point number, scalar. Should be in the interval `[0, 1)`.	`0.0`
`classifier_max_overlap`	Similar to the **`classifier_min_overlap`**. If the IoU of a ROI and a ground truth box is above this threshold, then this ROI is regarded as a positive ROI and this ground truth box is treated as the target(ground truth) of this ROI when training the RCNN.	Float type, scalar. Should be in the interval **`(0, 1)`** and greater than **`classifier_min_overlap`**.	`0.5`
`gt_as_roi`	A Boolean value to specify whether or not to include the ground truth boxes	Boolean(**`True`** or **`False`**)	**`False`**

Field	Description	Data Type and Constraints	Recommended/ Typical Value
	into the positive ROI to train the RCNN.		
`std_scaling`	The scaling factor to multiply by for the RPN regression loss when training the RPN.	Float type, should be positive.	**`1.0`**
`classifier_regr_std`	The scaling factor to divide by for the RCNN regression loss when training the RCNN.	map(dict) type. Map from **`'x'`**, **`'y'`**, **`'w'`**, **`'h'`** to its corresponding scaling factor. Each of the scaling factors should be a positive float number.	**`classifier_regr_std {`** **`key: 'x'`** **`value: 10.0`** **`}`** **`classifier_regr_std {`** **`key: 'y'`** **`value: 10.0`** **`}`** **`classifier_regr_std {`** **`key: 'w'`** **`value: 5.0`** **`}`** **`classifier_regr_std {`** **`key: 'h'`** **`value: 5.0`** **`}`**
`rpn_mini_batch`	The anchor batch size used to train the RPN.	unsigned int, positive.	**`256`**
`rpn_pre_nms_top_N`	The number of boxes to be retained before the NMS in Proposal layer.	unsigned int, positive.	**`-`**
`rpn_nms_max_boxes`	The number of boxes to be retained	unsigned int, positive and	**`-`**

Field	Description	Data Type and Constraints	Recommended/ Typical Value
	after the NMS in Proposal layer.	should be no greater than the **rpn_pre_nms_top_N**	
rpn_nms_overlap _threshold	The IoU threshold for the NMS in Proposal layer.	Float type, should be in the interval **(0, 1)**.	**0.7**
reg_config	Regularizer configuration of the model weights, including the regularizer type and weight decay.	message that contains two sub-fields: **reg_type** and **weight_decay**. See below for details.	-
reg_config.reg _type	The regularizer type. Can be either **'L1'**(L1 regularizer), **'L2'**(L2 regularizer), or **'none'**(No regularizer).	Str type. Should be one of the below: **'L1'**, **'L2'**, or **'none'**.	-
reg_config.weight _decay	The weight decay for the regularizer.	Float type, should be a positive scalar. Usually this number should be smaller than 1.0	-
optimizer	The Optimizer used for the training. Can be either SGD, RMSProp or Adam.	**oneof** message type that can be one of **sgd** message, **rmsprop** message or **adam** message. See below for the details of each message type.	-
adam	Adam optimizer.	message type that contains the 4 sub-fields: **lr**, **beta_1**, **beta_2**, and **epsilon**. See the Keras 2.2.4 documentation for the meaning of each field.	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
		Note# When the learning rate scheduler is enabled, the learning rate in the optimizer is overridden by the learning rate scheduler and the one specified in the optimizer(**lr**) is irrelevant.	
sgd	SGD optimizer	message type that contains the following fields: **lr**, **momentum**, **decay**and **nesterov**. See the Keras 2.2.4 documentation for the meaning of each field. Note# When the learning rate scheduler is enabled, the learning rate in the optimizer is overridden by the learning rate scheduler and the one specified in the optimizer(**lr**) is irrelevant.	-
rmsprop	RMSProp optimizer	message type that contains only one field: **lr**(learning rate). Note# When learning rate scheduler is enabled, the learning rate in the optimizer is	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
		overridden by the learning rate scheduler and the one specified in the optimizer(**lr**) is irrelevant.	
lr_scheduler	The learning rate scheduler.	message type that can be **step**or **soft_start**. **step**scheduler is the same as **step**scheduler in classification, while **soft_start**is the same as **soft_anneal**in classification. Refer to the classification spec file documentation for details.	-
lambda_rpn_regr	The loss scaling factor for RPN deltas regression loss.	Float typer. Should be a positive scalar.	**1.0**
lambda_rpn_class	The loss scaling factor for RPN classification loss.	Float type. Should be a positive scalar.	**1.0**
lambda_cls_regr	The loss scaling factor for RCNN deltas regression loss.	Float type. Should be a positive scalar.	**1.0**
lambda_cls_class	The loss scaling factor for RCNN classification loss.	Float type. Should be a positive scalar.	**1.0**
inference_config	The inference configuration for **tlt-infer**.	message type. See below for details.	-
inference_config	**images_dir** The absolute path to the image directory that **tlt-infer**will do inference on.	Str type. Should be a valid Unix path.	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
inference _config.model	The absolute path to the the **.tlt** model that **tlt-infer**will do inference for.	Str type. Should be a valid Unix path.	-
inference _config.detection _image_output_dir	The absolute path to the output image directory for the detection result. If the path doesn't exist **tlt-infer**will create it. If the directory already contains images **tlt-infer**will overwrite them.	Str type. Should be a valid Unix path.	-
inference _config.labels _dump_dir	The absolute path to the directory to save the detected labels in KITTI format. **tlt-infer**will create it if it doesn't exist beforehand. If it already contains label files, **tlt-infer**will overwrite them.	Str type. Should be a valid Unix path.	-
inference _config.rpn _pre_nms_top_N	The number of top ROI's to be retained before the NMS in Proposal layer.	unsigned int, positive.	-
inference _config.rpn _nms_max_boxes	The number of top ROI's to be retained after the NMS in Proposal layer.	unsigned int, positive.	-
inference_config _overlap_threshold	**rpn_nms** The IoU threshold for the NMS in Proposal layer.	Float type, should be in the interval **(0, 1)**.	**0.7**
inference_config _visualize _threshold	**bbox** The confidence threshold for the bounding boxes to be regarded as valid detected objects in the images.	Float type, should be in the interval **(0, 1)**.	**0.6**

Field	Description	Data Type and Constraints	Recommended/ Typical Value
`inference_config.classifier_nms_max_boxes`	The number of bounding boxes to be retained after the NMS in RCNN.	unsigned int, positive.	**`300`**
`inference_config.classifier_nms_overlap_threshold`	The IoU threshold for the NMS in RCNN.	Float type. Should be in the interval **`(0, 1)`**.	**`0.3`**
`inference_config.bbox_caption_on`	Whether or not to show captions for each bounding box in the detected images. The captions include the class name and confidence probability value for each detected object.	Boolean(**`True`** or **`False`**)	**`False`**
`inference_config.trt_inference`	The TensorRT inference configuration for **`tlt-infer`**in TensorRT backend mode.	Message type. This can be not present, and in this case, **`tlt-infer`**will use TLT as a backend for inference. See below for details.	-
`inference_config.trt_inference.trt_infer_model`	The model configuration for the **`tlt-infer`**in TensorRT backend mode. It is a **`oneof`**wrapper of the two possible model configurations: **`trt_engine`**and **`etlt_model`**. Only one of them can be specified if run **`tlt-infer`**in TensorRT backend. If **`trt_engine`**is provided, **`tlt-infer`**will run TensorRT inference on the TensorRT	message type, oneof wrapper of **`trt_engine`**and **`etlt_model`**. See below for details.	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
	engine file. If **.etlt** model is provided, **tlt-infer**will run TensorRT inference on the **.etlt** model. If in INT8 mode a calibration cache file should also be provided along with the **.etlt** model.		
inference_config.trt_inference.trt_engine	The absolute path to the TensorRT engine file for **tlt-infer**in TensorRT backend mode. The engine should be generated via the **tlt-export**or **tlt-converter**command line tools.	Str type.	-
inference_config.trt_inference .etlt_model	The configuration for the **.etlt** model and the calibration cache(only needed in INT8 mode) for **tlt-infer**in TensorRT backend mode. The **.etlt** model(and calibration cache, if needed) should be generated via the **tlt-export** command line tool.	message type that contains two string type sub-fields: **model** and **calibration_cache**. See below for details.	-
inference_config.trt_inference.etlt_model.model	The absolute path to the **.etlt** model that **tlt-infer** will use to run TensorRT based inference.	Str type.	-
inference_config.trt_inference.etlt_model.calibration_cache	The path to the TensorRT INT8 calibration cache file in the case of **tlt-infer** run	Str type.	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
	with **.etlt** model in INT8 mode.		
inference _config.trt _inference.trt _data_type	The TensorRT inference data type if **tlt-infer** runs with TensorRT backend. The data type is only useful when running on a **.etlt** model. In that case, if the data type is **'int8'**, a calibration cache file should also be provided as mentioned above. If running on a TensorRT engine file directly, this field will be ignored since the engine file already contains the data type information.	String type. Valid values are **'fp32'**, **'fp16'** and **'int8'**.	**'fp32'**
evaluation _config	The configuration for the **tlt-evaluate** in FasterRCNN.	message type that contains the below fields. See below for details.	-
evaluation _config.model	The absolute path to the **.tlt** model that **tlt-evaluate** will do evaluation for.	Str type. Should be a valid Unix path.	-
evaluation _config .labels_dump_dir	The absolute path to the directory of detected labels that **tlt-evaluate** will save. If it doesn't exist, **tlt-evaluate** will create it. If it already contains label files, **tlt-evaluate** will overwrite them.	Str type. Should be a valid Unix path.	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
evaluation_config .rpn_pre_nms_top_N	The number of top ROIs to be retained before the NMS in Proposal layer in **tlt-evaluate**.	unsigned int, positive.	-
evaluation_config .rpn_nms_max_boxes	The number of top ROIs to be retained after the NMS in Proposal layer in **tlt-evaluate**.	unsigned int, positive. Should be no greater than the **evaluation_config.rpn_pre_nms_top_N**.	-
evaluation_config .rpn_nms_iou_threshold	The IoU threshold for the NMS in Proposal layer in **tlt-evaluate**.	Float type in the interval **(0, 1)**.	**0.7**
evaluation_config .classifier_nms_max_boxes	The number of top bounding boxes to be retained after the NMS in RCNN in **tlt-evaluate**.	Unsigned int, positive.	-
evaluation_config .classifier_nms_overlap_threshold	The IoU threshold for the NMS in RCNN in **tlt-evaluate**.	Float typer in the interval **(0, 1)**.	**0.3**
evaluation_config .object_confidence_thres	The confidence threshold above which a bounding box can be regarded as a valid object detected by FasterRCNN. Usually you can use a small threshold to improve the recall and mAP as in many object detection challenges.	Float type in the interval **(0, 1)**.	**0.0001**
evaluation_config. use_voc07_11point_metric	Whether to use the VOC2007 mAP calculation method when computing the mAP of the FasterRCNN model	Boolean (**True** or **False**)	**False**

Field	Description	Data Type and Constraints	Recommended/ Typical Value
	on a specific dataset. If this is **False**, you can use VOC2012 metric instead.		

7.4. Specification file for SSD

Here is a sample of the SSD spec file. It has 6 major components: **ssd_config**, **training_config**, **eval_config**, **nms_config**, **augmentation_config**, and **dataset_config**. The format of the spec file is a protobuf text(prototxt) message and each of its fields can be either a basic data type or a nested message. The top level structure of the spec file is summarized in the table below.

7.4.1. Training config

The training configuration(**training_config**) defines the parameters needed for the training, evaluation and inference. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
batch_size_per_gpu	The batch size for each GPU, so the effective batch size is batch_size_per_gpu * num_gpus	Unsigned int, positive	–
num_epochs	The anchor batch size used to train the RPN.	Unsigned int, positive.	–
enable_qat	Whether to use quantization aware training	Boolean	–
learning_rate	Only soft_start_annealing_schedule with these nested parameters is supported. 1. **min_learning_rate**: minimum learning late to be seen during the entire experiment. 2. **max_learning_rate**: maximum learning rate to be seen during the entire experiment	Message type.	–

	3. **soft_start**: Time to be lapsed before warm up (expressed in percentage of progress between 0 and 1) 4. **annealing**: Time to start annealing the learning rate		
`regularizer`	This parameter configures the regularizer to be used while training and contains the following nested parameters. 1. **type**: The type or regularizer to use. NVIDIA supports NO_REG, L1 or L2 2. **weight**: The floating point value for regularizer weight	Message type.	`L1` (**Note**: NVIDIA suggests using L1 regularizer when training a network before pruning as L1 regularization helps making the network weights more prunable.)

7.4.2. Evaluation config

The evaluation configuration (**`eval_config`**) defines the parameters needed for the evaluation either during training or standalone. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`validation_period_during_training`	The number of training epochs per which one validation should run.	Unsigned int, positive	`10`
`average_precision_mode`	Average Precision (AP) calculation mode can be either **SAMPLE** or **INTEGRATE**. SAMPLE is used as VOC metrics for VOC 2009 or before. INTEGRATE is used for VOC 2010 or after that.	ENUM type (**SAMPLE** or **INTEGRATE**)	`SAMPLE`
`matching_iou_threshold`	The lowest iou of predicted box and ground truth box that can be considered a match.	Boolean	`0.5`

7.4.3. NMS config

The NMS configuration (**`nms_config`**) defines the parameters needed for the NMS postprocessing. NMS config applies to the NMS layer of the model in training, validation, evaluation, inference and export. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
confidence_threshold	Boxes with a confidence score less than confidence_threshold are discarded before applying NMS	float	0.01
cluster_iou_threshold	IOU threshold below which boxes will go through NMS process	float	0.6
top_k	top_k boxes will be outputted after the NMS keras layer. If the number of valid boxes is less than k, the returned array will be padded with boxes whose confidence score is 0.	Unsigned int	200

7.4.4. Augmentation config

The augmentation configuration (**augmentation_config**) defines the parameters needed for data augmentation. The configuration is shared with DetectNet_v2. See Augmentation module for more information.

7.4.5. Dataset config

The dataset configuration (**dataset_config**) defines the parameters needed for the data loader. The configuration is shared with DetectNet_v2. See Dataloader for more information.

7.4.6. SSD config

The SSD configuration (**ssd_config**) defines the parameters needed for building the SSD model. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
aspect_ratios_global	Anchor boxes of aspect ratios defined in aspect_ratios_global will be generated for each feature layer used for prediction. **Note:** Only one of **aspect_ratios_global** or **aspect_ratios** is required.	string	“[1.0, 2.0, 0.5, 3.0, 0.33]”
aspect_ratios	The length of the outer list must be equivalent to the number of	string	“[[1.0,2.0,0.5], [1.0,2.0,0.5], [1.0,2.0,0.5],

	feature layers used for anchor box generation. And the i-th layer will have anchor boxes with aspect ratios defined in aspect_ratios[i]. **Note:** Only one of **`aspect_ratios_global`** or **`aspect_ratios`** is required.		[1.0,2.0,0.5], [1.0,2.0,0.5], [1.0, 2.0, 0.5, 3.0, 0.33]]"
`two_boxes_for_ar1`	This setting is only relevant for layers that have 1.0 as the aspect ratio. If two_boxes_for_ar1 is true, two boxes will be generated with an aspect ratio of 1. One whose scale is the scale for this layer and the other one whose scale is the geometric mean of the scale for this layer and the scale for the next layer.	Boolean	**`True`**
`clip_boxes`	If true, all corner anchor boxes will be truncated so they are fully inside the feature images.	Boolean	**`False`**
`scales`	**scales** is a list of positive floats containing scaling factors per convolutional predictor layer. This list must be one element longer than the number of predictor layers, so if two_boxes_for_ar1 is true, the second aspect ratio 1.0 box for the last layer can have a proper scale. Except for the last element in this list, each positive float is the scaling factor for boxes in that layer. For example, if for one layer the scale is 0.1, then the generated anchor box with aspect ratio 1 for that layer (the first aspect ratio 1 box if two_boxes_for_ar1 is true) will have its height and width as 0.1*min(img_h, img_w).	string	`"[0.05, 0.1, 0.25, 0.4, 0.55, 0.7, 0.85]"`

	min_scale and max_scale are two positive floats. If both of them appear in the config, the program can automatically generate the scales by evenly splitting the space between min_scale and max_scale.		
min_scale/max_scale	If both appear in the config, scales will be generated evenly by splitting the space between min_scale and max_scale.	float	-
loss_loc_weight	This is a positive float controlling how much location regression loss should contribute to the final loss. The final loss is calculated as classification_loss + loss_loc_weight * loc_loss	float	1.0
focal_loss_alpha	Alpha is the focal loss equation.	float	0.25
focal_loss_gamma	Gamma is the focal loss equation.	float	2.0
variances	Variances should be a list of 4 positive floats. The four floats, in order, represent variances for box center x, box center y, log box height, log box width. The box offset for box center (cx, cy) and log box size (height/width) w.r.t. anchor will be divided by their respective variance value. Therefore, larger variances result in less significant differences between two different boxes on encoded offsets.		
steps	An optional list inside quotation marks whose length is the number of feature layers for prediction. The elements should be floats or tuples/lists of two floats. Steps define	string	-

	how many pixels apart the anchor box center points should be. If the element is a float, both vertical and horizontal margin is the same. Otherwise, the first value is step_vertical and the second value is step_horizontal. **If steps are not provided, anchor boxes will be distributed uniformly inside the image.**		
`offsets`	An optional list of floats inside quotation marks whose length is the number of feature layers for prediction. The first anchor box will have offsets[i]*steps[i] pixels margin from the left and top borders. **If offsets are not provided, 0.5 will be used as default value.**	string	–
`arch`	Backbone for feature extraction. Currently, “resnet”, “vgg”, “darknet”, “googlenet”, “mobilenet_v1”, “mobilenet_v2” and “squeezenet” are supported.	string	**`resnet`**
`nlayers`	Number of conv layers in specific **arch.** For “resnet”, 10, 18, 34, 50 and 101 are supported. For “vgg”, 16 and 19 are supported. For “darknet”, 19 and 53 are supported. All other networks don’t have this configuration and users should just delete this config from the config file.	Unsigned int	–
`freeze_bn`	Whether to freeze all batch normalization layers during training.	boolean	**`False`**
`freeze_blocks`	The list of block IDs to be frozen in the model during training. You can choose to freeze some of the CNN blocks in the model to make	list(repeated integers) • **ResNet series.** For the ResNet series, the block IDs valid for freezing is any subset of `[0, 1, 2, 3]` (inclusive)	–

	the training more stable and/or easier to converge. The definition of a block is heuristic for a specific architecture. For example, by stride or by logical blocks in the model, etc. However, the block ID numbers identify the blocks in the model in a sequential order so you don't have to know the exact locations of the blocks when you do training. A general principle to keep in mind is: the smaller the block ID, the closer it is to the model input; the larger the block ID, the closer it is to the model output. You can divide the whole model into several blocks and optionally freeze a subset of it. Note that for FasterRCNN you can only freeze the blocks that are before the ROI pooling layer. Any layer after the ROI pooling layer will not be frozen any way. For different backbones, the number of blocks and the block ID for each block are different. It deserves some detailed explanations on how to specify the block ID's for each backbone.	• **VGG series.** For the VGG series, the block IDs valid for freezing is any subset of `[1, 2, 3, 4, 5]` (inclusive) • **GoogLeNet.** For the GoogLeNet, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7]` (inclusive) • **MobileNet V1.** For the MobileNet V1, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]`(inclusive) • **MobileNet V2.** For the MobileNet V2, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]`(inclusive) • **DarkNet**. For the DarkNet 19 and DarkNet 53, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5]`(inclusive)

7.5. Specification file for DSSD

Below is a sample for the DSSD spec file. It has 6 major components: **dssd_config, training_config, eval_config, nms_config, augmentation_config and dataset_config**. The format of the spec file is a protobuf text(prototxt) message and each of its fields can be either a basic data type or a nested message. The top level structure of the spec file is summarized in the table below.

```
random_seed: 42
dssd_config {
  aspect_ratios_global: "[1.0, 2.0, 0.5, 3.0, 1.0/3.0]"
  scales: "[0.05, 0.1, 0.25, 0.4, 0.55, 0.7, 0.85]"
```

```
  two_boxes_for_ar1: true
  clip_boxes: false
  loss_loc_weight: 0.8
  focal_loss_alpha: 0.25
  focal_loss_gamma: 2.0
  variances: "[0.1, 0.1, 0.2, 0.2]"
  arch: "resnet"
  nlayers: 18
  pred_num_channels: 512
  freeze_bn: false
  freeze_blocks: 0
}
training_config {
  batch_size_per_gpu: 16
  num_epochs: 80
  enable_qat: false
  learning_rate {
  soft_start_annealing_schedule {
    min_learning_rate: 5e-5
    max_learning_rate: 2e-2
    soft_start: 0.15
    annealing: 0.8
    }
  }
  regularizer {
    type: L1
    weight: 3e-5
  }
}
eval_config {
  validation_period_during_training: 10
  average_precision_mode: SAMPLE
  batch_size: 16
  matching_iou_threshold: 0.5
}
nms_config {
  confidence_threshold: 0.01
  clustering_iou_threshold: 0.6
  top_k: 200
}
augmentation_config {
  preprocessing {
    output_image_width: 1248
    output_image_height: 384
    output_image_channel: 3
    crop_right: 1248
    crop_bottom: 384
    min_bbox_width: 1.0
    min_bbox_height: 1.0
  }
  spatial_augmentation {
    hflip_probability: 0.5
    vflip_probability: 0.0
    zoom_min: 0.7
    zoom_max: 1.8
    translate_max_x: 8.0
    translate_max_y: 8.0
  }
  color_augmentation {
    hue_rotation_max: 25.0
    saturation_shift_max: 0.20000000298
    contrast_scale_max: 0.10000000149
    contrast_center: 0.5
  }
}
dataset_config {
  data_sources: {
```

```
    tfrecords_path: "/workspace/tlt-experiments/data/tfrecords/kitti_trainval/
kitti_trainval*"
    image_directory_path: "/workspace/tlt-experiments/data/training"
  }
  image_extension: "png"
  target_class_mapping {
      key: "car"
      value: "car"
  }
  target_class_mapping {
      key: "pedestrian"
      value: "pedestrian"
  }
  target_class_mapping {
      key: "cyclist"
      value: "cyclist"
  }
  target_class_mapping {
      key: "van"
      value: "car"
  }
  target_class_mapping {
      key: "person_sitting"
      value: "pedestrian"
  }
validation_fold: 0
}
```

7.5.1. Training config

The training configuration(**training_config**) defines the parameters needed for the training, evaluation and inference. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
batch_size_per_gpu	The batch size for each GPU, so the effective batch size is batch_size_per_gpu * num_gpus	Unsigned int, positive	-
num_epochs	The anchor batch size used to train the RPN.	Unsigned int, positive.	-
enable_qat	Whether to use quantization aware training	Boolean	-
learning_rate	Only soft_start_annealing_schedule with these nested parameters is supported. 1. **min_learning_rate**: minimum learning late to be seen during the entire experiment. 2. **max_learning_rate**: maximum learning rate to be seen during the entire experiment	Message type.	-

	3. **soft_start**: Time to be lapsed before warm up (expressed in percentage of progress between 0 and 1) 4. **annealing**: Time to start annealing the learning rate		
`regularizer`	This parameter configures the regularizer to be used while training and contains the following nested parameters. 1. **type**: The type or regularizer to use. NVIDIA supports NO_REG, L1 or L2 2. **weight**: The floating point value for regularizer weight	Message type.	`L1` (**Note**: NVIDIA suggests using L1 regularizer when training a network before pruning as L1 regularization helps making the network weights more prunable.)

7.5.2. Evaluation config

The evaluation configuration (**`eval_config`**) defines the parameters needed for the evaluation either during training or standalone. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`validation_period_during_training`	The number of training epochs per which one validation should run.	Unsigned int, positive	`10`
`average_precision_mode`	Average Precision (AP) calculation mode can be either **SAMPLE** or **INTEGRATE**. SAMPLE is used as VOC metrics for VOC 2009 or before. INTEGRATE is used for VOC 2010 or after that.	ENUM type (**SAMPLE** or **INTEGRATE**)	`SAMPLE`
`matching_iou_threshold`	The lowest iou of predicted box and ground truth box that can be considered a match.	Boolean	`0.5`

7.5.3. NMS config

The NMS configuration (**`nms_config`**) defines the parameters needed for the NMS postprocessing. NMS config applies to the NMS layer of the model in training, validation, evaluation, inference and export. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`confidence_threshold`	Boxes with a confidence score less than confidence_threshold are discarded before applying NMS	float	`0.01`
`cluster_iou_threshold`	IOU threshold below which boxes will go through NMS process	float	`0.6`
`top_k`	top_k boxes will be outputted after the NMS keras layer. If the number of valid boxes is less than k, the returned array will be padded with boxes whose confidence score is 0.	Unsigned int	`200`

7.5.4. Augmentation config

The augmentation configuration (**`augmentation_config`**) defines the parameters needed for data augmentation. The configuration is shared with DetectNet_v2. See Augmentation module for more information.

7.5.5. Dataset config

The dataset configuration (**`dataset_config`**) defines the parameters needed for the data loader. The configuration is shared with DetectNet_v2. See Dataloader for more information.

7.5.6. DSSD config

The DSSD configuration (**`dssd_config`**) defines the parameters needed for building the DSSD model. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`aspect_ratios_global`	Anchor boxes of aspect ratios defined in aspect_ratios_global will be generated for each feature layer used for prediction. **Note:** Only one of **`aspect_ratios_global`** or **`aspect_ratios`** is required.	string	“[1.0, 2.0, 0.5, 3.0, 0.33]”
`aspect_ratios`	The length of the outer list must be equivalent to the number of	string	“[[1.0,2.0,0.5], [1.0,2.0,0.5], [1.0,2.0,0.5],

	feature layers used for anchor box generation. And the i-th layer will have anchor boxes with aspect ratios defined in aspect_ratios[i]. **Note:** Only one of **`aspect_ratios_global`** or **`aspect_ratios`** is required.		[1.0,2.0,0.5], [1.0,2.0,0.5], [1.0, 2.0, 0.5, 3.0, 0.33]]”
`two_boxes_for_ar1`	This setting is only relevant for layers that have 1.0 as the aspect ratio. If two_boxes_for_ar1 is true, two boxes will be generated with an aspect ratio of 1. One whose scale is the scale for this layer and the other one whose scale is the geometric mean of the scale for this layer and the scale for the next layer.	Boolean	**`True`**
`clip_boxes`	If true, all corner anchor boxes will be truncated so they are fully inside the feature images.	Boolean	**`False`**
`scales`	**scales** is a list of positive floats containing scaling factors per convolutional predictor layer. This list must be one element longer than the number of predictor layers, so if two_boxes_for_ar1 is true, the second aspect ratio 1.0 box for the last layer can have a proper scale. Except for the last element in this list, each positive float is the scaling factor for boxes in that layer. For example, if for one layer the scale is 0.1, then the generated anchor box with aspect ratio 1 for that layer (the first aspect ratio 1 box if two_boxes_for_ar1 is true) will have its height and width as 0.1*min(img_h, img_w).	string	**`“[0.05, 0.1, 0.25, 0.4, 0.55, 0.7, 0.85]”`**

	min_scale and max_scale are two positive floats. If both of them appear in the config, the program can automatically generate the scales by evenly splitting the space between min_scale and max_scale.		
min_scale/max_scale	If both appear in the config, scales will be generated evenly by splitting the space between min_scale and max_scale.	float	-
loss_loc_weight	This is a positive float controlling how much location regression loss should contribute to the final loss. The final loss is calculated as classification_loss + loss_loc_weight * loc_loss	float	`1.0`
focal_loss_alpha	Alpha is the focal loss equation.	float	`0.25`
focal_loss_gamma	Gamma is the focal loss equation.	float	`2.0`
variances	Variances should be a list of 4 positive floats. The four floats, in order, represent variances for box center x, box center y, log box height, log box width. The box offset for box center (cx, cy) and log box size (height/width) w.r.t. anchor will be divided by their respective variance value. Therefore, larger variances result in less significant differences between two different boxes on encoded offsets.		
steps	An optional list inside quotation marks whose length is the number of feature layers for prediction. The elements should be floats or tuples/lists of two floats. Steps define	string	-

	how many pixels apart the anchor box center points should be. If the element is a float, both vertical and horizontal margin is the same. Otherwise, the first value is step_vertical and the second value is step_horizontal. **If steps are not provided, anchor boxes will be distributed uniformly inside the image.**		
offsets	An optional list of floats inside quotation marks whose length is the number of feature layers for prediction. The first anchor box will have offsets[i]*steps[i] pixels margin from the left and top borders. **If offsets are not provided, 0.5 will be used as default value.**	string	–
arch	Backbone for feature extraction. Currently, “resnet”, “vgg”, “darknet”, “googlenet”, “mobilenet_v1”, “mobilenet_v2” and “squeezenet” are supported.	string	**resnet**
nlayers	Number of conv layers in specific **arch.** For “resnet”, 10, 18, 34, 50 and 101 are supported. For “vgg”, 16 and 19 are supported. For “darknet”, 19 and 53 are supported. All other networks don’t have this configuration and users should just delete this config from the config file.	Unsigned int	–
pred_num_channels	This setting controls the number of channels of the convolutional layers in the DSSD prediction module. Setting this value to 0 will disable the DSSD prediction module. Supported values for this setting are 0, 256, 512 and	Unsigned int	**512**

	1024. A larger value gives a larger network and usually means the network is harder to train.		
`freeze_bn`	Whether to freeze all batch normalization layers during training.	boolean	**`False`**
`freeze_blocks`	The list of block IDs to be frozen in the model during training. You can choose to freeze some of the CNN blocks in the model to make the training more stable and/or easier to converge. The definition of a block is heuristic for a specific architecture. For example, by stride or by logical blocks in the model, etc. However, the block ID numbers identify the blocks in the model in a sequential order so you don't have to know the exact locations of the blocks when you do training. A general principle to keep in mind is: the smaller the block ID, the closer it is to the model input; the larger the block ID, the closer it is to the model output. You can divide the whole model into several blocks and optionally freeze a subset of it. Note that for FasterRCNN you can only freeze the blocks that are before the ROI pooling layer. Any layer after the ROI pooling layer will not be frozen any way. For different backbones, the number of blocks and the block ID for each block are different. It deserves some detailed explanations on how to	list(repeated integers) • **ResNet series.** For the ResNet series, the block IDs valid for freezing is any subset of `[0, 1, 2, 3]` (inclusive) • **VGG series.** For the VGG series, the block IDs valid for freezing is any subset of `[1, 2, 3, 4, 5]` (inclusive) • **GoogLeNet.** For the GoogLeNet, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7]` (inclusive) • **MobileNet V1.** For the MobileNet V1, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]` (inclusive) • **MobileNet V2.** For the MobileNet V2, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]` (inclusive) • **DarkNet**. For the DarkNet 19 and DarkNet 53, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5]` (inclusive)	-

	specify the block ID's for each backbone.

```
dssd_config {
  aspect_ratios_global: "[1.0, 2.0, 0.5, 3.0, 0.33]"
  scales: "[0.1, 0.24166667, 0.38333333, 0.525, 0.66666667, 0.80833333, 0.95]"
  two_boxes_for_ar1: true
  clip_boxes: false
  loss_loc_weight: 1.0
  focal_loss_alpha: 0.25
  focal_loss_gamma: 2.0
  variances: "[0.1, 0.1, 0.2, 0.2]"
  pred_num_channels: 0
  arch: "resnet"
  nlayers: 18
  freeze_bn: True
  freeze_blocks: 0
  freeze_blocks: 1}
```

Using aspect_ratios_global or aspect_ratios

Only one of **aspect_ratios_global** or **aspect_ratios** is required.

aspect_ratios_global should be a 1-d array inside quotation marks. Anchor boxes of aspect ratios defined in **aspect_ratios_global** will be generated for each feature layer used for prediction. Example: "[1.0, 2.0, 0.5, 3.0, 0.33]"

aspect_ratios should be a list of lists inside quotation marks. The length of the outer list must be equivalent to the number of feature layers used for anchor box generation. And the i-th layer will have anchor boxes with aspect ratios defined in aspect_ratios[i]. Here's an example:

```
"[[1.0,2.0,0.5], [1.0,2.0,0.5], [1.0,2.0,0.5], [1.0,2.0,0.5], [1.0,2.0,0.5],
 [1.0, 2.0, 0.5, 3.0, 0.33]]"
```

two_boxes_for_ar1

This setting is only relevant for layers that have 1.0 as the aspect ratio. If **two_boxes_for_ar1** is true, two boxes will be generated with an aspect ratio of 1. One whose scale is the scale for this layer and the other one whose scale is the geometric mean of the scale for this layer and the scale for the next layer.

Scales or combination of min_scale and max_scale

Only one of **scales** and the combination of **min_scale** and **max_scale** is required.

Scales should be a 1-d array inside quotation marks. It is a list of positive floats containing scaling factors per convolutional predictor layer. This list must be one element longer than the number of predictor layers, so if **two_boxes_for_ar1** is true, the second aspect ratio 1.0 box for the last layer can have a proper scale. Except for the last element in this list, each positive float is the scaling factor for boxes in that layer. For example, if for one layer the scale is 0.1, then the generated anchor box with aspect ratio 1 for that layer (the first aspect ratio 1 box if two_boxes_for_ar1 is true) will have its height and width as *0.1*min(img_h, img_w)*.

min_scale and **max_scale** are two positive floats. If both of them appear in the config, the program can automatically generate the scales by evenly splitting the space between **min_scale** and **max_scale**.

clip_boxes

If true, all corner anchor boxes will be truncated so they are fully inside the feature images.

loss_loc_weight

This is a positive float controlling how much location regression loss should contribute to the final loss. The final loss is calculated as *classification_loss + loss_loc_weight * loc_loss*

focal_loss_alpha and focal_loss_gamma

Focal loss is calculated as:

$$loss = -\alpha(1 - p_t)^{\gamma} log(p_t), \text{ where } p_t = \begin{cases} p, \text{ if } y_{true} = 1 \\ 1 - p, \text{ if } y_{true} = 0 \end{cases}$$

focal_loss_alpha defines α and focal_loss_gamma defines γ in the formula. NVIDIA recommends α=0.25 and γ=2.0 if you don't know what values to use.

variances

Variances should be a list of 4 positive floats. The four floats, in order, represent variances for box center x, box center y, log box height, log box width. The box offset for box center (cx, cy) and log box size (height/width) w.r.t. anchor will be divided by their respective variance value. Therefore, larger variances result in less significant differences between two different boxes on encoded offsets. The formula for offset calculation is:

$$e_{cx} = \frac{cx_{gt} - cx_{anchor}}{w_{anchor} * variance_{cx}}$$

$$e_{cy} = \frac{cy_{gt} - cy_{anchor}}{w_{anchor} * variance_{cy}}$$

$$e_{logw} = \frac{log\left(\frac{w_{gt}}{w_{anchor}}\right)}{variance_{w}}$$

$$e_{logh} = \frac{log\left(\frac{h_{gt}}{h_{anchor}}\right)}{variance_{h}}$$

steps

An optional list inside quotation marks whose length is the number of feature layers for prediction. The elements should be floats or tuples/lists of two floats. Steps define how many pixels apart the anchor box center points should be. If the element is a float, both vertical and horizontal margin is the same. Otherwise, the first value is step_vertical and the second value is step_horizontal. If steps are not provided, anchorboxes will be distributed uniformly inside the image.

offsets

An optional list of floats inside quotation marks whose length is the number of feature layers for prediction. The first anchor box will have offsets[i]*steps[i] pixels margin from the left and top borders. If offsets are not provided, 0.5 will be used as default value.

arch

A string indicating which feature extraction architecture you want to use. Currently, “resnet”, “vgg”, “darknet”, “googlenet”, “mobilenet_v1”, “mobilenet_v2” and “squeezenet” are supported.

nlayers

An integer specifying the number of layers of the selected arch. For “resnet”, 10, 18, 34, 50 and 101 are supported. For “vgg”, 16 and 19 are supported. For “darknet”, 19 and 53 are supported. All other networks don’t have this configuration and users should just delete this config from the config file.

freeze_bn

Whether to freeze all batch normalization layers during training.

freeze_blocks

Optionally, you can have more than 1 **`freeze_blocks`** field. Weights of layers in those blocks will be freezed during training. See Model config for more information.

7.6. Specification file for RetinaNet

Below is a sample for the RetinaNet spec file. It has 6 major components: **retinanet_config, training_config, eval_config, nms_config, augmentation_config and dataset_config**. The format of the spec file is a protobuf text(prototxt) message and each of its fields can be either a basic data type or a nested message. The top level structure of the spec file is summarized in the table below.

```
random_seed: 42
retinanet_config {
  aspect_ratios_global: "[1.0, 2.0, 0.5]"
  scales: "[0.045, 0.09, 0.2, 0.4, 0.55, 0.7]"
  two_boxes_for_ar1: false
  clip_boxes: false
  loss_loc_weight: 0.8
  focal_loss_alpha: 0.25
  focal_loss_gamma: 2.0
  variances: "[0.1, 0.1, 0.2, 0.2]"
  arch: "resnet"
  nlayers: 18
  n_kernels: 1
  feature_size: 256
  freeze_bn: false
  freeze_blocks: 0
}
training_config {
  enable_qat: False
  batch_size_per_gpu: 24
  num_epochs: 100
  learning_rate {
  soft_start_annealing_schedule {
    min_learning_rate: 4e-5
    max_learning_rate: 1.5e-2
```

```
    soft_start: 0.15
    annealing: 0.5
    }
  }
  regularizer {
    type: L1
    weight: 2e-5
  }
}
eval_config {
  validation_period_during_training: 10
  average_precision_mode: SAMPLE
  batch_size: 32
  matching_iou_threshold: 0.5
}
nms_config {
  confidence_threshold: 0.01
  clustering_iou_threshold: 0.6
  top_k: 200
}
augmentation_config {
  preprocessing {
    output_image_width: 1248
    output_image_height: 384
    output_image_channel: 3
    crop_right: 1248
    crop_bottom: 384
    min_bbox_width: 1.0
    min_bbox_height: 1.0
  }
  spatial_augmentation {
    hflip_probability: 0.5
    vflip_probability: 0.0
    zoom_min: 0.7
    zoom_max: 1.8
    translate_max_x: 8.0
    translate_max_y: 8.0
  }
  color_augmentation {
    hue_rotation_max: 25.0
    saturation_shift_max: 0.2
    contrast_scale_max: 0.1
    contrast_center: 0.5
  }
}
dataset_config {
  data_sources: {
    tfrecords_path: "/workspace/tlt-experiments/data/tfrecords/kitti_trainval/
kitti_trainval*"
    image_directory_path: "/workspace/tlt-experiments/data/training"
  }
  image_extension: "png"
  target_class_mapping {
      key: "car"
      value: "car"
  }
  target_class_mapping {
      key: "pedestrian"
      value: "pedestrian"
  }
  target_class_mapping {
      key: "cyclist"
      value: "cyclist"
  }
  target_class_mapping {
      key: "van"
      value: "car"
```

```
  }
  target_class_mapping {
      key: "person_sitting"
      value: "pedestrian"
  }
validation_fold: 0
}
```

7.6.1. Training config

The training configuration(**`training_config`**) defines the parameters needed for the training, evaluation and inference. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`batch_size_per_gpu`	The batch size for each GPU, so the effective batch size is batch_size_per_gpu * num_gpus	Unsigned int, positive	–
`num_epochs`	The anchor batch size used to train the RPN.	Unsigned int, positive.	–
`enable_qat`	Whether to use quantization aware training	Boolean	–
`learning_rate`	Only soft_start_annealing_schedule with these nested parameters is supported. 1. **min_learning_rate**: minimum learning late to be seen during the entire experiment. 2. **max_learning_rate**: maximum learning rate to be seen during the entire experiment 3. **soft_start**: Time to be lapsed before warm up (expressed in percentage of progress between 0 and 1) 4. **annealing**: Time to start annealing the learning rate	Message type.	–
`regularizer`	This parameter configures the regularizer to be used while training and contains the following nested parameters.	Message type.	**`L1`** (**Note**: NVIDIA suggests using L1 regularizer when training a network before pruning as L1 regularization helps making the network weights more prunable.)

	1. **type**: The type or regularizer to use. NVIDIA supports NO_REG, L1 or L2 2. **weight**: The floating point value for regularizer weight		

7.6.2. Evaluation config

The evaluation configuration (**`eval_config`**) defines the parameters needed for the evaluation either during training or standalone. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`validation_period_during_training`	The number of training epochs per which one validation should run.	Unsigned int, positive	`10`
`average_precision_mode`	Average Precision (AP) calculation mode can be either **SAMPLE** or **INTEGRATE**. SAMPLE is used as VOC metrics for VOC 2009 or before. INTEGRATE is used for VOC 2010 or after that.	ENUM type (**SAMPLE** or **INTEGRATE**)	`SAMPLE`
`matching_iou_threshold`	The lowest iou of predicted box and ground truth box that can be considered a match.	Boolean	`0.5`

7.6.3. NMS config

The NMS configuration (**`nms_config`**) defines the parameters needed for the NMS postprocessing. NMS config applies to the NMS layer of the model in training, validation, evaluation, inference and export. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`confidence_threshold`	Boxes with a confidence score less than confidence_threshold are discarded before applying NMS	float	`0.01`
`cluster_iou_threshold`	IOU threshold below which boxes will go through NMS process	float	`0.6`
`top_k`	top_k boxes will be outputted after the NMS keras layer. If	Unsigned int	`200`

	the number of valid boxes is less than k, the returned array will be padded with boxes whose confidence score is 0.		

7.6.4. Augmentation config

The augmentation configuration (**`augmentation_config`**) defines the parameters needed for data augmentation. The configuration is shared with DetectNet_v2. See Augmentation module for more information.

7.6.5. Dataset config

The dataset configuration (**`dataset_config`**) defines the parameters needed for the data loader. The configuration is shared with DetectNet_v2. See Dataloader for more information.

7.6.6. RetinaNet config

The RetinaNet configuration (**`retinanet_config`**) defines the parameters needed for building the RetinaNet model. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`aspect_ratios_global`	Anchor boxes of aspect ratios defined in aspect_ratios_global will be generated for each feature layer used for prediction. **Note:** Only one of **`aspect_ratios_global`** or **`aspect_ratios`** is required.	string	"[1.0, 2.0, 0.5]"
`aspect_ratios`	The length of the outer list must be equivalent to the number of feature layers used for anchor box generation. And the i-th layer will have anchor boxes with aspect ratios defined in aspect_ratios[i]. **Note:** Only one of **`aspect_ratios_global`** or **`aspect_ratios`** is required.	string	"[[1.0,2.0,0.5], [1.0,2.0,0.5], [1.0,2.0,0.5], [1.0,2.0,0.5], [1.0,2.0,0.5], [1.0, 2.0, 0.5, 3.0, 0.33]]"
`two_boxes_for_ar1`	This setting is only relevant for layers that have 1.0 as the aspect ratio. If two_boxes_for_ar1 is	Boolean	**`True`**

	true, two boxes will be generated with an aspect ratio of 1. One whose scale is the scale for this layer and the other one whose scale is the geometric mean of the scale for this layer and the scale for the next layer.		
`clip_boxes`	If true, all corner anchor boxes will be truncated so they are fully inside the feature images.	Boolean	**`False`**
`scales`	**scales** is a list of positive floats containing scaling factors per convolutional predictor layer. This list must be one element longer than the number of predictor layers, so if two_boxes_for_ar1 is true, the second aspect ratio 1.0 box for the last layer can have a proper scale. Except for the last element in this list, each positive float is the scaling factor for boxes in that layer. For example, if for one layer the scale is 0.1, then the generated anchor box with aspect ratio 1 for that layer (the first aspect ratio 1 box if two_boxes_for_ar1 is true) will have its height and width as 0.1*min(img_h, img_w). min_scale and max_scale are two positive floats. If both of them appear in the config, the program can automatically generate the scales by evenly splitting the space between min_scale and max_scale.	string	**`“[0.05, 0.1, 0.25, 0.4, 0.55, 0.7, 0.85]”`**
`min_scale/max_scale`	If both appear in the config, scales will be generated evenly by splitting the space	float	-

	between min_scale and max_scale.		
loss_loc_weight	This is a positive float controlling how much location regression loss should contribute to the final loss. The final loss is calculated as classification_loss + loss_loc_weight * loc_loss	float	1.0
focal_loss_alpha	Alpha is the focal loss equation.	float	0.25
focal_loss_gamma	Gamma is the focal loss equation.	float	2.0
variances	Variances should be a list of 4 positive floats. The four floats, in order, represent variances for box center x, box center y, log box height, log box width. The box offset for box center (cx, cy) and log box size (height/width) w.r.t. anchor will be divided by their respective variance value. Therefore, larger variances result in less significant differences between two different boxes on encoded offsets.		
steps	An optional list inside quotation marks whose length is the number of feature layers for prediction. The elements should be floats or tuples/lists of two floats. Steps define how many pixels apart the anchor box center points should be. If the element is a float, both vertical and horizontal margin is the same. Otherwise, the first value is step_vertical and the second value is step_horizontal. **If steps are not provided, anchor boxes will be distributed uniformly inside the image.**	string	-

offsets	An optional list of floats inside quotation marks whose length is the number of feature layers for prediction. The first anchor box will have offsets[i]*steps[i] pixels margin from the left and top borders. **If offsets are not provided, 0.5 will be used as default value.**	string	-
arch	Backbone for feature extraction. Currently, “resnet”, “vgg”, “darknet”, “googlenet”, “mobilenet_v1”, “mobilenet_v2” and “squeezenet” are supported.	string	**resnet**
nlayers	Number of conv layers in specific **arch.** For “resnet”, 10, 18, 34, 50 and 101 are supported. For “vgg”, 16 and 19 are supported. For “darknet”, 19 and 53 are supported. All other networks don’t have this configuration and users should just delete this config from the config file.	Unsigned int	-
freeze_bn	Whether to freeze all batch normalization layers during training.	boolean	**False**
freeze_blocks	The list of block IDs to be frozen in the model during training. You can choose to freeze some of the CNN blocks in the model to make the training more stable and/or easier to converge. The definition of a block is heuristic for a specific architecture. For example, by stride or by logical blocks in the model, etc. However, the block ID numbers identify the blocks in the model in a sequential order so you don't have to know the exact locations of the blocks when you	list(repeated integers) • **ResNet series.** For the ResNet series, the block IDs valid for freezing is any subset of `[0, 1, 2, 3]` (inclusive) • **VGG series.** For the VGG series, the block IDs valid for freezing is any subset of `[1, 2, 3, 4, 5]` (inclusive) • **GoogLeNet.** For the GoogLeNet, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7]` (inclusive) • **MobileNet V1.** For the MobileNet V1,	-

	do training. A general principle to keep in mind is: the smaller the block ID, the closer it is to the model input; the larger the block ID, the closer it is to the model output. You can divide the whole model into several blocks and optionally freeze a subset of it. Note that for FasterRCNN you can only freeze the blocks that are before the ROI pooling layer. Any layer after the ROI pooling layer will not be frozen any way. For different backbones, the number of blocks and the block ID for each block are different. It deserves some detailed explanations on how to specify the block ID's for each backbone.	the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]`(inclusive) • **MobileNet V2.** For the MobileNet V2, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]`(inclusive) • **DarkNet**. For the DarkNet 19 and DarkNet 53, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5]`(inclusive)	
`n_kernels`	This setting controls the number of convolutional layers in the RetinaNet subnets for classification and anchor box regression. A larger value generates a larger network and usually means the network is harder to train.	Unsigned int	`2`
`feature_size`	This setting controls the number of channels of the convolutional layers in the RetinaNet subnets for classification and anchor box regression. A larger value gives a larger network and usually means the network is harder to train. Note that RetinaNet FPN generates 5 feature maps, thus the **scales** field requires a list of 6 scaling factors. The last number is not used if two_boxes_for_ar1 is set to False. There are	Unsigned int	`256`

	also three underlying scaling factors at each feature map level (2^0, 2^#, 2^#).	

Focal loss is calculated as:

$$loss = -\alpha(1-p_t)^{\gamma} log(p_t), \text{ where } p_t = \begin{cases} p, \text{ if } y_{true} = 1 \\ 1-p, \text{ if } y_{true} = 0 \end{cases}$$

Variances

$$e_{cx} = \frac{cx_{gt} - cx_{anchor}}{w_{anchor} * variance_{cx}}$$
$$e_{cy} = \frac{cy_{gt} - cy_{anchor}}{w_{anchor} * variance_{cy}}$$
$$e_{logw} = \frac{log\left(\frac{w_{gt}}{w_{anchor}}\right)}{variance_w}$$
$$e_{logh} = \frac{log\left(\frac{h_{gt}}{h_{anchor}}\right)}{variance_h}$$

7.7. Specification file for YOLOv3

Below is a sample for the YOLOv3 spec file. It has 6 major components: **yolo_config, training_config, eval_config, nms_config, augmentation_config and dataset_config**. The format of the spec file is a protobuf text(prototxt) message and each of its fields can be either a basic data type or a nested message. The top level structure of the spec file is summarized in the table below.

```
random_seed: 42
yolo_config {
  big_anchor_shape: "[(116,90), (156,198), (373,326)]"
  mid_anchor_shape: "[(30,61), (62,45), (59,119)]"
  small_anchor_shape: "[(10,13), (16,30), (33,23)]"
  matching_neutral_box_iou: 0.5
  arch: "darknet"
  nlayers: 53
  arch_conv_blocks: 2
  loss_loc_weight: 5.0
  loss_neg_obj_weights: 50.0
  loss_class_weights: 1.0
  freeze_bn: True
  freeze_blocks: 0
  freeze_blocks: 1}
training_config {
  batch_size_per_gpu: 16
  num_epochs: 80
  enable_qat: false
  learning_rate {
  soft_start_annealing_schedule {
    min_learning_rate: 5e-5
    max_learning_rate: 2e-2
    soft_start: 0.15
    annealing: 0.8
```

```
    }
  }
  regularizer {
    type: L1
    weight: 3e-5
  }
}
eval_config {
  validation_period_during_training: 10
  average_precision_mode: SAMPLE
  batch_size: 16
  matching_iou_threshold: 0.5
}
nms_config {
  confidence_threshold: 0.01
  clustering_iou_threshold: 0.6
  top_k: 200
}
augmentation_config {
  preprocessing {
    output_image_width: 1248
    output_image_height: 384
    output_image_channel: 3
    crop_right: 1248
    crop_bottom: 384
    min_bbox_width: 1.0
    min_bbox_height: 1.0
  }
  spatial_augmentation {
    hflip_probability: 0.5
    vflip_probability: 0.0
    zoom_min: 0.7
    zoom_max: 1.8
    translate_max_x: 8.0
    translate_max_y: 8.0
  }
  color_augmentation {
    hue_rotation_max: 25.0
    saturation_shift_max: 0.20000000298
    contrast_scale_max: 0.10000000149
    contrast_center: 0.5
  }
}
dataset_config {
  data_sources: {
    tfrecords_path: "/workspace/tlt-experiments/data/tfrecords/kitti_trainval/
kitti_trainval*"
    image_directory_path: "/workspace/tlt-experiments/data/training"
  }
  image_extension: "png"
  target_class_mapping {
      key: "car"
      value: "car"
  }
  target_class_mapping {
      key: "pedestrian"
      value: "pedestrian"
  }
  target_class_mapping {
      key: "cyclist"
      value: "cyclist"
  }
  target_class_mapping {
      key: "van"
      value: "car"
  }
  target_class_mapping {
```

```
      key: "person_sitting"
      value: "pedestrian"
  }
validation_fold: 0
}
```

7.7.1. Training config

The training configuration(**`training_config`**) defines the parameters needed for the training, evaluation and inference. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`batch_size_per_gpu`	The batch size for each GPU, so the effective batch size is batch_size_per_gpu * num_gpus	Unsigned int, positive	-
`num_epochs`	The anchor batch size used to train the RPN.	Unsigned int, positive.	-
`enable_qat`	Whether to use quantization aware training	Boolean	-
`learning_rate`	Only soft_start_annealing_schedule with these nested parameters is supported. 1. **min_learning_rate**: minimum learning late to be seen during the entire experiment. 2. **max_learning_rate**: maximum learning rate to be seen during the entire experiment 3. **soft_start**: Time to be lapsed before warm up (expressed in percentage of progress between 0 and 1) 4. **annealing**: Time to start annealing the learning rate	Message type.	-
`regularizer`	This parameter configures the regularizer to be used while training and contains the following nested parameters. 1. **type**: The type or regularizer to	Message type.	**`L1`** (**Note**: NVIDIA suggests using L1 regularizer when training a network before pruning as L1 regularization helps making the network weights more prunable.)

	use. NVIDIA supports NO_REG, L1 or L2 2. **weight**: The floating point value for regularizer weight		

7.7.2. Evaluation config

The evaluation configuration (**`eval_config`**) defines the parameters needed for the evaluation either during training or standalone. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`validation_period_during_training`	The number of training epochs per which one validation should run.	Unsigned int, positive	`10`
`average_precision_mode`	Average Precision (AP) calculation mode can be either **SAMPLE** or **INTEGRATE**. SAMPLE is used as VOC metrics for VOC 2009 or before. INTEGRATE is used for VOC 2010 or after that.	ENUM type (**SAMPLE** or **INTEGRATE**)	**`SAMPLE`**
`matching_iou_threshold`	The lowest iou of predicted box and ground truth box that can be considered a match.	Boolean	`0.5`

7.7.3. NMS config

The NMS configuration (**`nms_config`**) defines the parameters needed for the NMS postprocessing. NMS config applies to the NMS layer of the model in training, validation, evaluation, inference and export. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`confidence_threshold`	Boxes with a confidence score less than confidence_threshold are discarded before applying NMS	float	`0.01`
`cluster_iou_threshold`	IOU threshold below which boxes will go through NMS process	float	`0.6`
`top_k`	top_k boxes will be outputted after the NMS keras layer. If the number of valid boxes is less than k,	Unsigned int	`200`

	the returned array will be padded with boxes whose confidence score is 0.		

7.7.4. Augmentation config

The augmentation configuration (**`augmentation_config`**) defines the parameters needed for data augmentation. The configuration is shared with DetectNet_v2. See Augmentation module for more information.

7.7.5. Dataset config

The dataset configuration (**`dataset_config`**) defines the parameters needed for the data loader. The configuration is shared with DetectNet_v2. See Dataloader for more information.

7.7.6. YOLOv3 config

The YOLOv3 configuration (**`yolo_config`**) defines the parameters needed for building the DSSD model. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`big_anchor_shape`, **`mid_anchor_shape`** **`and`** **`small_anchor_shape`**	Those settings should be 1-d arrays inside quotation marks. The elements of those arrays are tuples representing the pre-defined anchor shape in the order of width, height. The default YOLOv3 has 9 predefined anchor shapes. They are divided into 3 groups corresponding to big, medium and small objects. The detection output corresponding to different groups are from different depths in the network. Users should run the kmeans.py file attached with the example notebook to determine the best anchor shapes for their own dataset and put those anchor shapes in the spec file. It is worth noting that the number of anchor shapes for any field is	string	Use **`kmeans.py`** attached in **`examples/yolo`** inside docker to generate those shapes

	not limited to 3. Users only need to specify at least 1 anchor shape in each of those three fields.		
matching_neutral_box_iou	This field should be a float number between 0 and 1. Any anchor not matching to ground truth boxes, but with IOU higher than this float value with any ground truth box, will not have their objectiveness loss back-propagated during training. This is to reduce false negatives.	float	0.5
arch_conv_blocks	Supported values are 0, 1 and 2. This value controls how many convolutional blocks are present among detection output layers. Setting this value to 2 if you want to reproduce the meta architecture of the original YOLOv3 model coming with DarkNet 53. Please note this config setting only controls the size of the YOLO meta architecture and the size of the feature extractor has nothing to do with this config field.	0, 1 or 2	2
loss_loc_weight, loss_neg_obj_weights and loss_class_weights	Those loss weights can be configured as float numbers. The YOLOv3 loss is a summation of localization loss, negative objectiveness loss, positive objectiveness loss and classification loss. The weight of positive objectiveness loss is set to 1 while the weights of other losses are read from config file.	float	loss_loc_weight: 5.0 loss_neg_obj_weights: 50.0 loss_class_weights: 1.0
arch	Backbone for feature extraction. Currently, “resnet”, “vgg”, “darknet”, “googlenet”, “mobilenet_v1”,	string	**resnet**

	"mobilenet_v2" and "squeezenet" are supported.		
`nlayers`	Number of conv layers in specific **arch.** For "resnet", 10, 18, 34, 50 and 101 are supported. For "vgg", 16 and 19 are supported. For "darknet", 19 and 53 are supported. All other networks don't have this configuration and users should just delete this config from the config file.	Unsigned int	–
`freeze_bn`	Whether to freeze all batch normalization layers during training.	boolean	**`False`**
`freeze_blocks`	The list of block IDs to be frozen in the model during training. You can choose to freeze some of the CNN blocks in the model to make the training more stable and/or easier to converge. The definition of a block is heuristic for a specific architecture. For example, by stride or by logical blocks in the model, etc. However, the block ID numbers identify the blocks in the model in a sequential order so you don't have to know the exact locations of the blocks when you do training. A general principle to keep in mind is: the smaller the block ID, the closer it is to the model input; the larger the block ID, the closer it is to the model output. You can divide the whole model into several blocks and optionally freeze a subset of it. Note that for FasterRCNN you can only freeze the blocks that are before the ROI pooling layer.	list(repeated integers) • **ResNet series.** For the ResNet series, the block IDs valid for freezing is any subset of `[0, 1, 2, 3]` (inclusive) • **VGG series.** For the VGG series, the block IDs valid for freezing is any subset of `[1, 2, 3, 4, 5]` (inclusive) • **GoogLeNet.** For the GoogLeNet, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7]` (inclusive) • **MobileNet V1.** For the MobileNet V1, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]` (inclusive) • **MobileNet V2.** For the MobileNet V2, the block IDs valid for freezing is any subset of `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]` (inclusive) • **DarkNet**. For the DarkNet 19 and DarkNet 53, the block IDs valid for freezing is any	–

	Any layer after the ROI pooling layer will not be frozen any way. For different backbones, the number of blocks and the block ID for each block are different. It deserves some detailed explanations on how to specify the block ID's for each backbone.	subset of `[0, 1, 2, 3, 4, 5]`(inclusive)

7.8. Specification file for RetinaNet

Below is a sample for the RetinaNet spec file. It has 6 major components: **retinanet_config, training_config, eval_config, nms_config, augmentation_config and dataset_config**. The format of the spec file is a protobuf text(prototxt) message and each of its fields can be either a basic data type or a nested message. The top level structure of the spec file is summarized in the table below.

```
random_seed: 42
retinanet_config {
  aspect_ratios_global: "[1.0, 2.0, 0.5]"
  scales: "[0.045, 0.09, 0.2, 0.4, 0.55, 0.7]"
  two_boxes_for_ar1: false
  clip_boxes: false
  loss_loc_weight: 0.8
  focal_loss_alpha: 0.25
  focal_loss_gamma: 2.0
  variances: "[0.1, 0.1, 0.2, 0.2]"
  arch: "resnet"
  nlayers: 18
  n_kernels: 1
  feature_size: 256
  freeze_bn: false
  freeze_blocks: 0
}
training_config {
  enable_qat: False
  batch_size_per_gpu: 24
  num_epochs: 100
  learning_rate {
  soft_start_annealing_schedule {
    min_learning_rate: 4e-5
    max_learning_rate: 1.5e-2
    soft_start: 0.15
    annealing: 0.5
    }
  }
  regularizer {
    type: L1
    weight: 2e-5
  }
}
eval_config {
  validation_period_during_training: 10
  average_precision_mode: SAMPLE
  batch_size: 32
  matching_iou_threshold: 0.5
}
nms_config {
```

```
  confidence_threshold: 0.01
  clustering_iou_threshold: 0.6
  top_k: 200
}
augmentation_config {
  preprocessing {
    output_image_width: 1248
    output_image_height: 384
    output_image_channel: 3
    crop_right: 1248
    crop_bottom: 384
    min_bbox_width: 1.0
    min_bbox_height: 1.0
  }
  spatial_augmentation {
    hflip_probability: 0.5
    vflip_probability: 0.0
    zoom_min: 0.7
    zoom_max: 1.8
    translate_max_x: 8.0
    translate_max_y: 8.0
  }
  color_augmentation {
    hue_rotation_max: 25.0
    saturation_shift_max: 0.2
    contrast_scale_max: 0.1
    contrast_center: 0.5
  }
}
dataset_config {
  data_sources: {
    tfrecords_path: "/workspace/tlt-experiments/data/tfrecords/kitti_trainval/
kitti_trainval*"
    image_directory_path: "/workspace/tlt-experiments/data/training"
  }
  image_extension: "png"
  target_class_mapping {
      key: "car"
      value: "car"
  }
  target_class_mapping {
      key: "pedestrian"
      value: "pedestrian"
  }
  target_class_mapping {
      key: "cyclist"
      value: "cyclist"
  }
  target_class_mapping {
      key: "van"
      value: "car"
  }
  target_class_mapping {
      key: "person_sitting"
      value: "pedestrian"
  }
validation_fold: 0
}
```

7.8.1. Training config

The training configuration(**`training_config`**) defines the parameters needed for the training, evaluation and inference. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`batch_size_per_gpu`	The batch size for each GPU, so the effective batch size is batch_size_per_gpu * num_gpus	Unsigned int, positive	–
`num_epochs`	The anchor batch size used to train the RPN.	Unsigned int, positive.	–
`enable_qat`	Whether to use quantization aware training	Boolean	–
`learning_rate`	Only soft_start_annealing_schedule with these nested parameters is supported. 1. **min_learning_rate**: minimum learning late to be seen during the entire experiment. 2. **max_learning_rate**: maximum learning rate to be seen during the entire experiment 3. **soft_start**: Time to be lapsed before warm up (expressed in percentage of progress between 0 and 1) 4. **annealing**: Time to start annealing the learning rate	Message type.	–
`regularizer`	This parameter configures the regularizer to be used while training and contains the following nested parameters. 1. **type**: The type or regularizer to use. NVIDIA supports NO_REG, L1 or L2 2. **weight**: The floating point value for regularizer weight	Message type.	`L1` (**Note**: NVIDIA suggests using L1 regularizer when training a network before pruning as L1 regularization helps making the network weights more prunable.)

7.8.2. Evaluation config

The evaluation configuration (**eval_config**) defines the parameters needed for the evaluation either during training or standalone. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`validation_period_during_training`	The number of training epochs per which one validation should run.	Unsigned int, positive	`10`
`average_precision_mode`	Average Precision (AP) calculation mode can be either **SAMPLE** or **INTEGRATE**. SAMPLE is used as VOC metrics for VOC 2009 or before. INTEGRATE is used for VOC 2010 or after that.	ENUM type (**SAMPLE** or **INTEGRATE**)	`SAMPLE`
`matching_iou_threshold`	The lowest iou of predicted box and ground truth box that can be considered a match.	Boolean	`0.5`

7.8.3. NMS config

The NMS configuration (**nms_config**) defines the parameters needed for the NMS postprocessing. NMS config applies to the NMS layer of the model in training, validation, evaluation, inference and export. Details are summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`confidence_threshold`	Boxes with a confidence score less than confidence_threshold are discarded before applying NMS	float	`0.01`
`cluster_iou_threshold`	IOU threshold below which boxes will go through NMS process	float	`0.6`
`top_k`	top_k boxes will be outputted after the NMS keras layer. If the number of valid boxes is less than k, the returned array will be padded with boxes whose confidence score is 0.	Unsigned int	`200`

7.8.4. Augmentation config

The augmentation configuration (**`augmentation_config`**) defines the parameters needed for data augmentation. The configuration is shared with DetectNet_v2. See Augmentation module for more information.

7.8.5. Dataset config

The dataset configuration (**`dataset_config`**) defines the parameters needed for the data loader. The configuration is shared with DetectNet_v2. See Dataloader for more information.

7.9. Specification file for MaskRCNN

Below is a sample for the MaskRCNN spec file. It has 3 major components: top level experiment configs , **`data_config`** and **`maskrcnn_config`**, explained below in detail. The format of the spec file is a protobuf text(prototxt) message and each of its fields can be either a basic data type or a nested message. The top level structure of the spec file is summarized in the table below.

Here's a sample of the MaskRCNN spec file:

```
seed: 123
use_amp: False
warmup_steps: 0
checkpoint: "/workspace/tlt-experiments/maskrcnn/pretrained_resnet50/
tlt_instance_segmentation_vresnet50/resnet50.hdf5"
learning_rate_steps: "[60000, 80000, 100000]"
learning_rate_decay_levels: "[0.1, 0.02, 0.002]"
total_steps: 120000
train_batch_size: 2
eval_batch_size: 4
num_steps_per_eval: 10000
momentum: 0.9
l2_weight_decay: 0.0001
warmup_learning_rate: 0.0001
init_learning_rate: 0.02

data_config{
        image_size: "(832, 1344)"
        augment_input_data: True
        eval_samples: 500
        training_file_pattern: "/workspace/tlt-experiments/data/train*.tfrecord"
        validation_file_pattern: "/workspace/tlt-experiments/data/val*.tfrecord"
        val_json_file: "/workspace/tlt-experiments/data/annotations/
instances_val2017.json"

        # dataset specific parameters
        num_classes: 91
        skip_crowd_during_training: True
}

maskrcnn_config {
        nlayers: 50
        arch: "resnet"
        freeze_bn: True
        freeze_blocks: "[0,1]"
```

```
        gt_mask_size: 112

        # Region Proposal Network
        rpn_positive_overlap: 0.7
        rpn_negative_overlap: 0.3
        rpn_batch_size_per_im: 256
        rpn_fg_fraction: 0.5
        rpn_min_size: 0.

        # Proposal layer.
        batch_size_per_im: 512
        fg_fraction: 0.25
        fg_thresh: 0.5
        bg_thresh_hi: 0.5
        bg_thresh_lo: 0.

        # Faster-RCNN heads.
        fast_rcnn_mlp_head_dim: 1024
        bbox_reg_weights: "(10., 10., 5., 5.)"

        # Mask-RCNN heads.
        include_mask: True
        mrcnn_resolution: 28

        # training
        train_rpn_pre_nms_topn: 2000
        train_rpn_post_nms_topn: 1000
        train_rpn_nms_threshold: 0.7

        # evaluation
        test_detections_per_image: 100
        test_nms: 0.5
        test_rpn_pre_nms_topn: 1000
        test_rpn_post_nms_topn: 1000
        test_rpn_nms_thresh: 0.7

        # model architecture
        min_level: 2
        max_level: 6
        num_scales: 1
        aspect_ratios: "[(1.0, 1.0), (1.4, 0.7), (0.7, 1.4)]"
        anchor_scale: 8

        # localization loss
        rpn_box_loss_weight: 1.0
        fast_rcnn_box_loss_weight: 1.0
        mrcnn_weight_loss_mask: 1.0
}
```

Field	Description	Data Type and Constraints	Recommended/ Typical Value
seed	The random seed for the experiment.	Unsigned int	123
warmup_steps	The steps taken for learning rate to ramp up to the **init_learning_rate**.	Unsigned int	-
warmup_learning_rate	The initial learning rate during in the warmup phase.	float	-

Field	Description	Data Type and Constraints	Recommended/ Typical Value
`learning_rate_steps`	List of steps, at which the learning rate decays by the factor specified in **`learning_rate_decay_levels`**.	string	-
`learning_rate_decay_levels`	List of decay factors. The length should match the length of **`learning_rate_steps`**.	string	-
`total_steps`	Total number of training iterations.	Unsigned int	-
`train_batch_size`	Batch size during training.	Unsigned int	4
`eval_batch_size`	Batch size during validation or evaluation.	Unsigned int	8
`num_steps_per_eval`	Save a checkpoint and run evaluation every N steps.	Unsigned int	-
`momentum`	Momentum of SGD optimizer.	float	0.9
`l2_weight_decay`	L2 weight decay	float	0.0001
`use_amp`	Whether to use Automatic Mixed Precision training.	boolean	**`False`**
`checkpoint`	Path to a pretrained model.	string	-
`maskrcnn_config`	The architecture of the model.	message	-
`data_config`	Input data configuration.	message	-
`skip_checkpoint_variables`	If specified, the weights of the layers with matching regular expressions will not be loaded. This is especially helpful for transfer learning.	string	-

When using skip_checkpoint_variables, you can first find the model structure in the training log (Part of MaskRCNN+ResNet50 model structure is shown below). If, for example, you want to retrain all prediction heads, you can set skip_checkpoint_variables to “head”. TLT uses Python **re** library to check whether “head” matches any layer name or re.search($skip_checkpoint_variables, $layer_name).

```
[MaskRCNN] INFO    : ================ TRAINABLE VARIABLES ==================
[MaskRCNN] INFO    : [#0001] conv1/kernel:0
          => (7, 7, 3, 64)
[MaskRCNN] INFO    : [#0002] bn_conv1/gamma:0
          => (64,)
[MaskRCNN] INFO    : [#0003] bn_conv1/beta:0
          => (64,)
[MaskRCNN] INFO    : [#0004] block_1a_conv_1/kernel:0
          => (1, 1, 64, 64)
[MaskRCNN] INFO    : [#0005] block_1a_bn_1/gamma:0
          => (64,)
[MaskRCNN] INFO    : [#0006] block_1a_bn_1/beta:0
          => (64,)
[MaskRCNN] INFO    : [#0007] block_1a_conv_2/kernel:0
          => (3, 3, 64, 64)
[MaskRCNN] INFO    : [#0008] block_1a_bn_2/gamma:0
          => (64,)
[MaskRCNN] INFO    : [#0009] block_1a_bn_2/beta:0
          => (64,)
[MaskRCNN] INFO    : [#0010] block_1a_conv_3/kernel:0
          => (1, 1, 64, 256)
[MaskRCNN] INFO    : [#0011] block_1a_bn_3/gamma:0
          => (256,)
[MaskRCNN] INFO    : [#0012] block_1a_bn_3/beta:0
          => (256,)
[MaskRCNN] INFO    : [#0110] block_3d_bn_3/gamma:0
          => (1024,)
[MaskRCNN] INFO    : [#0111] block_3d_bn_3/beta:0
          => (1024,)
[MaskRCNN] INFO    : [#0112] block_3e_conv_1/kernel:0
          => (1, 1, 1024, [MaskRCNN] INFO    : [#0144] block_4b_bn_1/beta:0
                                    => (512,)
                                …     …      …    …
                                             ...
[MaskRCNN] INFO    : [#0174] fpn/post_hoc_d5/kernel:0
          => (3, 3, 256, 256)
[MaskRCNN] INFO    : [#0175] fpn/post_hoc_d5/bias:0
          => (256,)
[MaskRCNN] INFO    : [#0176] rpn_head/rpn/kernel:0
          => (3, 3, 256, 256)
[MaskRCNN] INFO    : [#0177] rpn_head/rpn/bias:0
          => (256,)
[MaskRCNN] INFO    : [#0178] rpn_head/rpn-class/kernel:0
          => (1, 1, 256, 3)
[MaskRCNN] INFO    : [#0179] rpn_head/rpn-class/bias:0
          => (3,)
[MaskRCNN] INFO    : [#0180] rpn_head/rpn-box/kernel:0
          => (1, 1, 256, 12)
[MaskRCNN] INFO    : [#0181] rpn_head/rpn-box/bias:0
          => (12,)
[MaskRCNN] INFO    : [#0182] box_head/fc6/kernel:0
          => (12544, 1024)
[MaskRCNN] INFO    : [#0183] box_head/fc6/bias:0
          => (1024,)
[MaskRCNN] INFO    : [#0184] box_head/fc7/kernel:0
          => (1024, 1024)
```

```
[MaskRCNN] INFO    : [#0185] box_head/fc7/bias:0
         => (1024,)
[MaskRCNN] INFO    : [#0186] box_head/class-predict/kernel:0
         => (1024, 91)
[MaskRCNN] INFO    : [#0187] box_head/class-predict/bias:0
         => (91,)
[MaskRCNN] INFO    : [#0188] box_head/box-predict/kernel:0
         => (1024, 364)
[MaskRCNN] INFO    : [#0189] box_head/box-predict/bias:0
         => (364,)
[MaskRCNN] INFO    : [#0190] mask_head/mask-conv-l0/kernel:0
         => (3, 3, 256, 256)
[MaskRCNN] INFO    : [#0191] mask_head/mask-conv-l0/bias:0
         => (256,)
[MaskRCNN] INFO    : [#0192] mask_head/mask-conv-l1/kernel:0
         => (3, 3, 256, 256)
[MaskRCNN] INFO    : [#0193] mask_head/mask-conv-l1/bias:0
         => (256,)
[MaskRCNN] INFO    : [#0194] mask_head/mask-conv-l2/kernel:0
         => (3, 3, 256, 256)
[MaskRCNN] INFO    : [#0195] mask_head/mask-conv-l2/bias:0
         => (256,)
[MaskRCNN] INFO    : [#0196] mask_head/mask-conv-l3/kernel:0
         => (3, 3, 256, 256)
[MaskRCNN] INFO    : [#0197] mask_head/mask-conv-l3/bias:0
         => (256,)
[MaskRCNN] INFO    : [#0198] mask_head/conv5-mask/kernel:0
         => (2, 2, 256, 256)
[MaskRCNN] INFO    : [#0199] mask_head/conv5-mask/bias:0
         => (256,)
[MaskRCNN] INFO    : [#0200] mask_head/mask_fcn_logits/kernel:0
         => (1, 1, 256, 91)
[MaskRCNN] INFO    : [#0201] mask_head/mask_fcn_logits/bias:0
         => (91,)
```

maskrcnn config

The maskrcnn configuration (maskrcnn_config) defines the model structure. This model is used for training, evaluation and inference. Detailed description is summarized in the table below. Currently, MaskRCNN only supports ResNet10/18/34/50/101 as its backbone.

Field	Description	Data Type and Constraints	Recommended/ Typical Value
`nlayers`	Number of layers in ResNet arch	message	50
`arch`	The backbone feature extractor name .	string	**`resnet`**
`freeze_bn`	Whether to freeze all BatchNorm layers in the backbone.	boolean	**`False`**

`freeze_blocks`	List of conv blocks in the backbone to freeze.	string **ResNet** For the ResNet series, the block IDs valid for freezing is any subset of **`[0, 1, 2, 3]`** (inclusive)	-
`gt_mask_size`	Groundtruth mask size.	Unsigned int	112
`rpn_positive_overlap`	Lower bound threshold to assign positive labels for anchors.	float	0.7
`rpn_positive_overlap`	Upper bound threshold to assign negative labels for anchors.	float	0.3
`rpn_batch_size_per_im`	The number of sampled anchors per image in RPN.	Unsigned int	256
`rpn_fg_fraction`	Desired fraction of positive anchors in a batch.	Unsigned int	0.5
`rpn_min_size`	Minimum proposal height and width.		0
`batch_size_per_im`	RoI minibatch size per image.	Unsigned int	512
`fg_fraction`	The target fraction of RoI minibatch that is labeled as foreground.	float	0.25
`fast_rcnn_mlp_head_dim`	fastrcnn classification head dimension.	Unsigned int	1024
`bbox_reg_weights`	Bounding box regularization weights.	string	**`"(10, 10, 5, 5)"`**
`include_mask`	Whether to include mask head.	boolean	**`True`**

mrcnn_resolution	Mask head resolution.	Unsigned int	28
train_rpn_pre_nms_topn	Number of top scoring RPN proposals to keep before applying NMS (per FPN level).	Unsigned int	2000
train_rpn_post_nms_topn	Number of top scoring RPN proposals to keep after applying NMS (total number produced).	Unsigned int	1000
train_rpn_nms_threshold	NMS IOU threshold in RPN during training.	float	0.7
test_detections_per_image	Number of bounding box candidates after NMS.	Unsigned int	100
test_nms	NMS IOU threshold during test.	float	0.5
test_rpn_pre_nms_topn	Number. of top scoring RPN proposals to keep before applying NMS (per FPN level) during test.	Unsigned int	1000
test_rpn_post_nms_topn	Number of top scoring RPN proposals to keep after applying NMS (total number produced) during test.	Unsigned int	1000
test_rpn_nms_threshold	NMS IOU threshold in RPN during test.	float	0.7
min_level	Minimum level of the output feature pyramid.	Unsigned int	2

max_level	Maximum level of the output feature pyramid.	Unsigned int	6
num_scales	Number of anchor octave scales on each pyramid level (e.g. if it's set to 3, the anchor scales are [2^0, 2^(1/3), 2^(2/3)])	Unsigned int	1
aspect_ratios	List of tuples representing the aspect ratios of anchors on each pyramid level.	string	"[(1.0, 1.0), (1.4, 0.7), (0.7, 1.4)]"
anchor_scale	Scale of base anchor size to the feature pyramid stride.	Unsigned int	8
rpn_box_loss_weight	Weight for adjusting RPN box loss in the total loss.	float	1.0
fast_rcnn_box_loss_weight	Weight for adjusting FastRCNN box regression loss in the total loss.	float	1.0
mrcnn_weight_loss_mask	Weight for adjusting mask loss in the total loss.	float	1.0

The **min_level**, **max_level**, **num_scales**, **aspect_ratios** and **anchor_scale** are used to determine MaskRCNN's anchor generation. anchor_scale is the base anchor's scale. And min_level and max_level sets the range of the scales on different feature maps. For example, the actual anchor scale for the feature map at min_level will be **anchor_scale * 2^min_level** and the actual anchor scale for the feature map at max_level will be **anchor_scale * 2^max_level**. And it will generate anchors of different **aspect_ratios**based on the actual anchor scale.

data config

The data configuration (data_config) specifies the input data source and format. This model is used for training, evaluation and inference. Detailed description is summarized in the table below. Currently, MaskRCNN only supports ResNet10/18/34/50/101 as its backbone.

Field	Description	Data Type and Constraints	Recommended/ Typical Value
image_size	Image dimension	Unsigned int	123
augment_input_data	Whether to augment data	boolean	True
eval_samples	Number of samples for evaluation.	Unsigned int	-
training_file_pattern	TFRecord path for training.	string	-
validation_file_pattern	TFRecord path for validation.	string	-
val_json_file	The annotation file path for validation.	string	-
num_classes	Number of classes.	Unsigned int	-
skip_crowd_druing_training	Whether to skip crowd during training .	boolean	True

7.9.1. MaskRCNN config

The maskrcnn configuration (maskrcnn_config) defines the model structure. This model is used for training, evaluation and inference. Detailed description is summarized in the table below. Currently, MaskRCNN only supports ResNet10/18/34/50/101 as its backbone.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`nlayers`	Number of layers in ResNet arch	message	`50`
`arch`	The backbone feature extractor name	string	`resnet`
`freeze_bn`	Whether to freeze all BatchNorm layers in the backbone	boolean	`False`
`freeze_blocks`	List of conv blocks in the backbone to freeze	string • **ResNet.** For the ResNet series, the block IDs valid for freezing is any subset of `[0, 1, 2, 3]` (inclusive)	-
`gt_mask_size`	Groundtruth mask size	Unsigned int	`112`

`rpn_positive_overlap`	Lower bound threshold to assign positive labels for anchors	float	`0.7`
`rpn_positive_overlap`	Upper bound threshold to assign negative labels for anchors	float	`0.3`
`rpn_batch_size_per_im`	The number of sampled anchors per image in RPN	Unsigned int	`256`
`rpn_fg_fraction`	Desired fraction of positive anchors in a batch	Unsigned int	`0.5`
`rpn_min_size`	Minimum proposal height and width		`0`
`batch_size_per_im`	RoI minibatch size per image	Unsigned int	`512`
`fg_fraction`	The target fraction of RoI minibatch that is labeled as foreground	float	`0.25`
`fast_rcnn_mlp_head_dim`	fast rcnn classification head dimension	Unsigned int	`1024`
`bbox_reg_weights`	Bounding box regularization weights	string	`“(10, 10, 5, 5)”`
`include_mask`	Whether to include mask head	boolean	`True` (currently only True is supported)
`mrcnn_resolution`	Mask head resolution	Unsigned int	`28`
`train_rpn_pre_nms_topn`	Number of top scoring RPN proposals to keep before applying NMS (per FPN level)	Unsigned int	`2000`
`train_rpn_post_nms_topn`	Number of top scoring RPN proposals to keep after applying NMS (total number produced)	Unsigned int	`1000`
`train_rpn_nms_threshold`	NMS IOU threshold in RPN during training	float	`0.7`
`test_detections_per_image`	Number of bounding box candidates after NMS	Unsigned int	`100`
`test_nms`	NMS IOU threshold during test	float	`0.5`
`test_rpn_pre_nms_topn`	Number of top scoring RPN proposals to keep before applying NMS (per FPN level) during test	Unsigned int	1000
`test_rpn_post_nms_topn`	Number of top scoring RPN proposals to keep after applying NMS	Unsigned int	1000

	(total number produced) during test		
`test_rpn_nms_threshold`	NMS IOU threshold in RPN during test	float	0.7
`min_level`	Minimum level of the output feature pyramid	Unsigned int	2
`max_level`	Maximum level of the output feature pyramid	Unsigned int	6
`num_scales`	Number of anchor octave scales on each pyramid level (e.g. if it's set to 3, the anchor scales are [2^0, 2^(1/3), 2^(2/3)])	Unsigned int	1
`aspect_ratios`	List of tuples representing the aspect ratios of anchors on each pyramid level	string	"[(1.0, 1.0), (1.4, 0.7), (0.7, 1.4)]"
`anchor_scale`	Scale of base anchor size to the feature pyramid stride	Unsigned int	8
`rpn_box_loss_weight`	Weight for adjusting RPN box loss in the total loss	float	1.0
`fast_rcnn_box_loss_weight`	Weight for adjusting FastRCNN box regression loss in the total loss	float	1.0
`mrcnn_weight_loss_mask`	Weight for adjusting mask loss in the total loss	float	1.0

The min_level, max_level, num_scales, aspect_ratios and anchor_scale are used to determine MaskRCNN's anchor generation. anchor_scale is the base anchor's scale. And min_level and max_level sets the range of the scales on different feature maps. For example, the actual anchor scale for the feature map at min_level will be **anchor_scale * 2^min_level** and the actual anchor scale for the feature map at max_level will be **anchor_scale * 2^max_level**. And it will generate anchors of different **aspect_ratios** based on the actual anchor scale.

7.9.2. Data config

The data configuration (data_config) specifies the input data source and format. This is used for training, evaluation and inference. Detailed description is summarized in the table below.

Field	Description	Data Type and Constraints	Recommended/Typical Value
`image_size`	Image dimension as a tuple within quote marks. "(height, width)" indicates the dimension	string	`"(832, 1344)"`

	of the resized and padded input		
augment_input_data	Whether to augment data	boolean	**True**
eval_samples	Number of samples for evaluation	Unsigned int	–
training_file_pattern	TFRecord path for training	string	–
validation_file_pattern	TFRecord path for validation	string	–
val_json_file	The annotation file path for validation	string	–
num_classes	Number of classes	Unsigned int	–
skip_crowd_druing_training	Whether to skip crowd during training	boolean	**True**

Chapter 8. TRAINING THE MODEL

You can use the **tlt-train** command to train models with single and multiple GPUs. The NVIDIA Transfer Learning Toolkit provides a simple command line interface to train a deep learning model for classification, object detection, and instance segmentation. It includes the **tlt-train** command to do this. To speed up the training process, the **tlt-train** command supports multiGPU training. You can invoke a multi GPU training session by using the **--gpus N** option, where **N** is the number of GPUs you want to use. **N** must be less than the number of GPUs available in the given node for training.

Currently, only single-node multiGPU is supported.

The other optimizations included with tlt-train are:

- Quantization Aware Training (QAT)
- Automatic Mixed Precision (AMP)

8.1. Quantization Aware Training

TLT now supports Quantization-Aware-Training (QAT) for its object detection networks namely, DetectNet_v2, SSD, DSSD, YOLOv3, RetinaNet and FasterRCNN. Quantization Aware Training emulates the inference time quantization when training a model that may then be used by downstream inference platforms to generate actual quantized models. The error from quantizating weights and tensors to INT8 is modeled during training, allowing the model to adapt and mitigate the error. During QAT, the model constructed in the training graph is modified to:

1. Replace existing nodes with nodes that support fake quantization of its weights.
2. Convert existing activations to ReLU-6 (except the output nodes).
3. Add Quantize and De-Quantize(QDQ) nodes to compute the dynamic ranges of the intermediate tensors.

The dynamic ranges computed during training, are serialized to a cache file using **tlt-export** that may then be parsed by TensorRT to create an optimized inference engine. To enable QAT during training, simply set the **enable_qat** parameter to be **True** in the **training_config** field of the corresponding spec file of each of the supported apps. The benefit of QAT training is usually a better accuracy when doing INT8 inference with TensorRT compared with traditional calibration based INT8 TensorRT inference.

The number of scales present in the cache file is less than that generated by the Post Training Quantization technique using TensorRT. This is because the QDQ nodes are added only after operations that are fused by TensorRT (in GPU) eg: operation sequences such as Conv2d -> Bias -> Relu or Conv2d -> Bias -> BatchNormalization -> Activation, whereas during PTQ, the scales are applied to all the intermediate tensors in the model. Also, the final output regression nodes are not quantized in the current training graphs. So these layers currently run in fp32.

When deploying a model with platforms that have DLA, please note that currently using Quantization cache files generated by peeling the scales from the model is not supported, since DLA requires a scale factor for all layers. Inorder to use a QAT trained model with DLA, we recommend using the post training quantization at export (see Exporting the Model). The Post Training Quantization method takes the current QAT trained model and generates scale factors for all intermediate tensors in the model since the DLA doesn't fuse operations as done by the GPU.

8.2. Automatic Mixed Precision

TLT now supports Automatic-Mixed-Precision(AMP) training. DNN training has traditionally relied on training using the IEEE-single precision format for its tensors. With mixed precision training however, one may use a mixture for FP16 and FP32 operations in the training graph to help speed up training while not compromising accuracy. There are several benefits to using AMP:

- Speed up math-intensive operations, such as linear and convolution layers.
- Speed up memory-limited operations by accessing half the bytes compared to single-precisReduce memory requirements for training models, enabling larger models or larger minibatches.

In TLT, enabling AMP is as simple as setting the environment variable **TF_ENABLE_AUTO_MIXED_PRECISION=1** when running **tlt-train**. This will help speedup the training by using FP16 tensor cores. Note that AMP is only supported on GPUs with Volta or above architecture.

8.3. Training a classification model

Use the **tlt-train** command to tune a pre-trained model:

```
tlt-train [-h] classification --gpus <num GPUs>
           -k <encoding key>
           -r <result directory>
```

```
          -e <spec file>
```

Required arguments:

- **-r, --results_dir** : Path to a folder where the experiment outputs should be written.
- **-k, --key** : User specific encoding key to save or load a .tlt model.
- **-e, --experiment_spec_file**: Path to the experiment spec file.

Optional arguments:

- **--gpus** : Number of GPUs to use and processes to launch for training. The default value is 1.

See the Specification file for classification section for more details.

Here's an example of using the tlt-train command:

```
tlt-train classification -e /workspace/tlt_drive/spec/spec.cfg -r /workspace/
output -k $YOUR_KEY
```

8.4. Training a DetectNet_v2 model

After following the steps, go here to create TFRecords ingestible by the TLT training, and setting up a spec file. You are now ready to start training an object detection network.

DetectNet_v2 training command

```
tlt-train [-h] detectnet_v2
          -k <key>
          -r <result directory>
          -e <spec_file>
          [--gpus <num GPUs>]
```

Required arguments

- **-r, --results_dir** : Path to a folder where experiment outputs should be written.
- **-k, -key** : User specific encoding key to save or load a .tlt model.
- **-e, --experiment_spec_file** : Path to spec file. Absolute path or relative to working directory. (default: spec from spec_loader.py is used).

Optional arguments

- **--gpus** : Number of GPUs to use and processes to launch for training. The default value is 1.
- -h, --help : To print help message

Sample usage

Here is an example of command for a 2 GPU training:

```
tlt-train detectnet_v2 -e <path_to_spec_file>
                                 -r <path_to_experiment_output>
                                 -k <key_to_load_the_model>
```

```
                              -n <name_string_for_the_model>
                              --gpus 2
```

The **tlt-train** tool does not support training on images of multiple resolutions, or resizing images during training. All of the images must be resized offline to the final training size and the corresponding bounding boxes must be scaled accordingly.

DetectNet_v2 now supports resuming training from intermediate checkpoints. In case a previously running training experiment is stopped prematurely, one may restart the training from the last checkpoint by simply re-running the detectnet_v2 training command with the same command line arguments as before. The trainer for detectnet_v2 finds the last saved checkpoint in the results directory and resumes the training from there. The interval at which the checkpoints are saved are defined by the `checkpoint_interval` parameter under the “training_config” for detectnet_v2.

8.5. Training a FasterRCNN model

Use this command to execute the FasterRCNN training command:

```
tlt-train [-h] faster_rcnn -e <experiment_spec>
                           [-k <enc_key>]
                           [--gpus <num_gpus>]
```

Required arguments:

- **-e, --experiment_spec_file** : Experiment specification file to set up the evaluation experiment. This should be the same as training specification file.

Optional arguments:

- **-h, --help** : Show this help message and exit.
- **-k, --enc_key**: TLT encoding key, can override the one in the spec file.
- **--gpus**: The number of GPUs to be used in the training in a multi-gpu scenario(default: 1).

Sample usage

Here's an example of using the FasterRCNN training command:

```
tlt-train faster_rcnn -e <experiment_spec>
```

Using a Pretrained Weights File

Usually, using a pretrained weights file for the initial training of FasterRCNN helps get better accuracy. NVIDIA recommends using the pretrained weights provided in NVIDIA GPU Cloud(NGC). FasterRCNN loads the pretrained weights by name. That is, layer by layer, if TLT finds a layer whose name and weights(bias) shape in the pretrained weights file matches a layer in the TLT model, it will load that layer's weights(and bias, if any) into the model. If some layer in the TLT cannot find a matching layer in the pretrained weights, then TLT will skip that layer and will use random initialization for that layer instead. An exception is that if TLT finds a matching layer in the pretrained weights(and bias, if any) but the shape of the pretrained weights(or bias, if any) in that layer does not match the shape of weights(bias) for the corresponding layer in TLT

model, it will also skip that layer. For some layers that have no weights(bias), nothing will be done for it(hence will be skipped). So, in total, there are three possible statuses to indicate how a layer's pretrained weights loading is going on. That is,**'Yes'**,**'No'** and **'None'**. **'Yes'** means a layer has weights(bias) and is loaded from the pretrained weights file successfully for initialization. **'No'** means a layer has weights(bias) but due to mismatched weights(bias) shape(or probably something else), the weights(bias) cannot be loaded successfully and will use random initialization instead. **'None'** means a layer has no weights(bias) at all and will not load any weights. In the FasterRCNN training log, there is a table that shows the pretrained weights loading status for each layer in the model.

8.6. Training an SSD model

Train the SSD model using this command:

```
tlt-train [-h] ssd -e <experiment_spec>
                   -r <output_dir>
                   -k <key>
                   -m <pretrained_model>
                   --gpus <num_gpus>
```

Required arguments:

- **-r, --results_dir**: Path to the folder where the experiment output is written.
- **-k, --key**: Provide the encryption key to decrypt the model.
- **-e, --experiment_spec_file**: Experiment specification file to set up the evaluation experiment. This should be the same as the training specification file.

Optional arguments:

- **--gpus num_gpus**: Number of GPUs to use and processes to launch for training. The default = 1.
- **-m, --resume_model_weights**: Path to a pre-trained model or model to continue training.
- **--initial_epoch**: Epoch number to resume from.
- **-h, --help**: Show this help message and exit.

Here's an example of using the train command on an SSD model:

```
tlt-train ssd --gpus 2 -e /path/to/spec.txt -r /path/to/result -k $KEY
```

8.7. Training a DSSD model

Train the DSSD model using this command:

```
tlt-train [-h] dssd -e <experiment_spec>
                    -r <output_dir>
                    -k <key>
                    -m <pretrained_model>
                    --gpus <num_gpus>
```

Required arguments:

- **-r, --results_dir**: Path to the folder where the experiment output is written.

- **-k, --key**: Provide the encryption key to decrypt the model.
- **-e, --experiment_spec_file**: Experiment specification file to set up the evaluation experiment. This should be the same as training specification file.

Optional arguments:

- **--gpus num_gpus**: Number of GPUs to use and processes to launch for training. The default = 1.
- **-m, --resume_model_weights**: Path to a pre-trained model or model to continue training.
- **--initial_epoch**: Epoch number to resume from.
- **-h, --help**: Show this help message and exit.

Here's an example of using the train command on an DSSD model:

```
tlt-train dssd --gpus 2 -e /path/to/spec.txt -r /path/to/result -k $KEY
```

8.8. Training a YOLOv3 model

Train the YOLOv3 model using this command:

```
tlt-train [-h] yolo -e <experiment_spec>
                    -r <output_dir>
                    -k <key>
                    -m <pretrained_model>
                    --gpus <num_gpus>
```

Required arguments:

- **-r, --results_dir**: Path to the folder where the experiment output is written.
- **-k, --key**: Provide the encryption key to decrypt the model.
- **-e, --experiment_spec_file**: Experiment specification file to set up the evaluation experiment. This should be the same as the training specification file.

Optional arguments:

- **--gpus num_gpus**: Number of GPUs to use and processes to launch for training. The default = 1.
- **-m, --resume_model_weights**: Path to a pre-trained model or model to continue training.
- **--initial_epoch**: Epoch number to resume from.
- **-h, --help**: Show this help message and exit.

Here's an example of using the train command on a YOLOv3 model:

```
tlt-train yolo --gpus 2 -e /path/to/spec.txt -r /path/to/result -k $KEY
```

8.9. Training a RetinaNet model

Train the RetinaNet model using this command:

```
tlt-train [-h] retinanet -e <experiment_spec>
                   -r <output_dir>
                   -k <key>
```

```
                  -m <pretrained_model>
                  --gpus <num_gpus>
```

Required arguments:

- **-r, --results_dir**: Path to the folder where the experiment output is written.
- **-k, --key**: Provide the encryption key to decrypt the model.
- **-e, --experiment_spec_file**: Experiment specification file to set up the evaluation experiment. This should be the same as the training specification file.

Optional arguments:

- **--gpus num_gpus**: Number of GPUs to use and processes to launch for training. The default = 1.
- **-m, --resume_model_weights**: Path to a pre-trained model or model to continue training.
- **--initial_epoch**: Epoch number to resume from.
- **-h, --help**: Show this help message and exit.

Here's an example of using the train command on a RetinaNet model:

```
tlt-train retinanet --gpus 2 -e /path/to/spec.txt -r /path/to/result -k $KEY
```

8.10. Training a MaskRCNN model

Train the MaskRCNN model using this command:

```
tlt-train [-h] mask_rcnn -e <experiment_spec>
                         -d <output_dir>
                         -k <key>
                         --gpus <num_gpus>
```

Required arguments:

- **-d, --model_dir**: Path to the folder where the experiment output is written.
- **-k, --key**: Provide the encryption key to decrypt the model.
- **-e, --experiment_spec_file**: Experiment specification file to set up the evaluation experiment. This should be the same as the training specification file.

Optional arguments:

- **--gpus num_gpus**: Number of GPUs to use and processes to launch for training. The default = 1.
- **-h, --help**: Show this help message and exit.

Here's an example of using the train command on a MaskRCNN model:

```
tlt-train mask_rcnn --gpus 2 -e /path/to/spec.txt -d /path/to/result -k $KEY
```

Chapter 9. EVALUATING THE MODEL

Once the model has been trained, using the experiment config file, and by following the steps to train a model, the next step would be to evaluate this model on a test set to measure the accuracy of the model. The TLT toolkit includes the **`tlt-evaluate`** command to do this.

The classification app computes evaluation loss, Top-k accuracy, precision and recall as metrics. Meanwhile, **`tlt-evaluate`** for DetectNet_v2, FasterRCNN, Retinanet, DSSD, YOLOV3, and SSD computes the Average Precision per class and the mean Average Precision metrics as defined in the Pascal VOC challenge. Both sample and integrate modes are supported to calculate average precision. The former was used in VOC challenges before 2010 while the latter was used from 2010 onwards. The SAMPLE mode uses an 11-point method to compute the AP, while the INTEGRATE mode uses a more fine-grained integration method and gets a more accurate number of AP. MaskRCNN reports COCO's detection evaluation metrics (https://cocodataset.org/#detection-eval). AP50 in COCO metrics is comparable to mAP in Pascal VOC metrics.

When training is complete, the model is stored in the output directory of your choice in $OUTPUT_DIR. Evaluate a model using the **`tlt-evaluate`** command:

```
tlt-evaluate {classification,detectnet_v2,faster_rcnn,ssd,dssd,retinanet,yolo,
 mask_rcnn} [-h] [<arguments for classification/detectnet_v2/faster_rcnn/ssd/
dssd/retinanet/yolo, mask_rcnn>]
```

Required arguments:

- ```
  {classification, detectnet_v2, faster_rcnn, ssd, dssd, retinanet, yolo,
        mask_rcnn}
  ```

Choose whether you are evaluating a **`classification`**, **`detectnet_v2`**, **`ssd`**, **`dssd`**, **`yolo`**, **`retinanet`**, **`faster_rcnn`** or **`mask_rcnn`** model.

Optional arguments: These arguments vary depending upon Classification, DetectNet_v2, SSD, DSSD, RetinaNet, YOLOv3, FasterRCNN and MaskRCNN models.
  ```

9.1. Evaluating a classification model

Execute **`tlt-evaluate`** on a classification model.

```
tlt-evaluate classification [-h] -e <experiment_spec_file> -k <key>
```

Required arguments

- **`-e, --experiment_spec_file`**: Path to the experiment spec file..
- **`-k, -key`** : Provide the encryption key to decrypt the model .

Optional arguments

- **`-h, --help`** : show this help message and exit.

If you followed the example in Training a classification model, you can run the evaluation:

```
tlt-evaluate classification -e classification_spec.cfg -k $YOUR_KEY
```

TLT evaluate for classification produces the following metrics:

- Loss
- Top-K accuracy
- Precision (P): TP / (TP + FP)
- Recall (R): TP / (TP + FN)
- Confusion Matrix

9.2. Evaluating a DetectNet_v2 model

Execute **`tlt-evaluate`** on a DetectNet_v2 model.

```
tlt-evaluate detectnet_v2 [-h] -e <experiment_spec>
                               -m <model_file>
                               -k <key>
                               [--use_training_set]
```

Required arguments:

- **`-e, --experiment_spec_file`**: Experiment spec file to set up the evaluation experiment. This should be the same as training spec file.
- **`-m, --model`**: Path to the model file to use for evaluation. This could be a **`.tlt`** model file or a **`tensorrt engine`** generated using the **`tlt-export`** tool.
- **`-k, --key`** : Provide the encryption key to decrypt the model. This is a required argument only with a **`.tlt`** model file.

Optional arguments

- **`-h, --help`** : show this help message and exit.
- **`-f, --framework`**: the framework to use when running evaluation (choices: “tlt”, “tensorrt”). By default the framework is set to TensorRT.

- **`--use_training_set`**: Set this flag to run evaluation on training + validation dataset.

If you have followed the example in Training a detection model, you may now evaluate the model using the following command:

```
tlt-evaluate detectnet_v2 -e <path to training spec file>
                          -m <path to the model>
                          -k <key to load the model>
```

This command runs evaluation on the same validation set that was used during training.

Use these steps to evaluate on a test set with ground truth labeled:

1. Create tfrecords for this training set by following the steps listed in the data input section.
2. Update the dataloader configuration part of the training spec file to include the newly generated tfrecords. For more information on the dataset config, please refer to Create an experiment spec file. You may create the tfrecords with any partition mode (sequence/random). The evaluate tool iterates through all the folds in the tfrecords patterns mentioned in the validation_data_source.

```
dataset_config {
  data_sources: {
    tfrecords_path: "<path to training tfrecords root>/<tfrecords_name*>"
    image_directory_path: "<path to training data root>"
  }
  image_extension: "jpg"
  target_class_mapping {
      key: "car"
      value: "car"
  }
  target_class_mapping {
      key: "automobile"
      value: "car"
  }
  ..
  ..
  ..
  target_class_mapping {
      key: "person"
      value: "pedestrian"
  }
  target_class_mapping {
      key: "rider"
      value: "cyclist"
  }
  validation_data_source: {
    tfrecords_path: "<path to testing tfrecords root>/<tfrecords_name*>"
    image_directory_path: "<path to testing data root>"
  }
}
```

The rest of the experiment spec file remains the same as the training spec file.

9.3. Evaluating a FasterRCNN model

To run evaluation for a faster_rcnn model use this command:

```
tlt-evaluate faster_rcnn [-h] -e <experiment_spec>
                              [-k <enc_key>]
```

Required arguments:

- **`-e, --experiment_spec_file`** : Experiment spec file to set up the evaluation experiment. This should be the same as a training spec file.

Optional arguments:

- **`-h, --help`** : show this help message and exit.
- **`-k, --enc_key`** #The encoding key, can override the one in the spec file.

Evaluation metrics:

For FasterRCNN, the evaluation will print out 4 metrics for the evaluated model: AP(average precision), precision, recall and RPN_recall for each class in the evaluation dataset. Finally, it will also print the mAP(mean average precision) as a single metric number. Two modes are supported for computing the AP, i.e., the PASCAL VOC 2007 and 2012 metrics. This can be configured in the spec file's **`evaluation_config.use_voc_11_point_metric`** parameter. If this parameter is set to **`True`**, then AP calculation will use VOC 2007 method, otherwise it will use VOC 2012 method. The RPN_recall metric indicates the recall capability of the RPN of the FasterRCNN model. The higher the RPN_recall metric, it means RPN can better detect an object as foreground(but it doesn't say anything on which class this object belongs to since that is delegated to RCNN). The RPN_recall metric is mainly used for debugging on the accuracy issue of a FasterRCNN model.

Two modes for tlt-evaluate

The **`tlt-evaluate`** command line for FasterRCNN has two modes. It can run with either TLT backend or TensorRT backend. This behavior is also controlled via the spec file. The **`evaluation_config`** in the spec file can have an optional **`trt_evaluation`** sub-field that specifies which backend the **`tlt-evaluate`** will run with. By default(if the **`trt_evaluation`** sub-field is not present in **`evaluation_config`**), **`tlt-evaluate`** will use TLT as the backend. If the **`trt_evaluation`** sub-field is present, it can specify **`tlt-evaluate`** to run at TensorRT backend. In that case, the model to do inference can be either the **`.etlt`** model from **`tlt-export`** or the TensorRT engine file from **`tlt-export`** or **`tlt-converter`**.

To use a TensorRT engine file for TensorRT backend based **`tlt-evaluate`**, the **`trt_evaluation`** sub-field should look like this:

```
trt_evaluation {
trt_engine: '/workspace/tlt-experiments/data/faster_rcnn/trt.int8.engine'
max_workspace_size_MB: 2000
}
```

To use a **.etlt** model for TensorRT backend based **tlt-evaluate**, the **trt_evaluation** sub-field should look like this:

```
trt_evaluation {
etlt_model {
model: '/workspace/tlt-experiments/data/faster_rcnn/resnet18.epoch12.etlt'
calibration_cache: '/workspace/tlt-experiments/data/faster_rcnn/cal.bin'
}
trt_data_type: 'int8'
max_workspace_size_MB: 2000
}
```

If the TensorRT inference data type is not INT8, the **calibration_cache** sub-field that provides the path to the INT8 calibration cache is not needed. In INT8 case, the calibration cache should be generated via the **tlt-export** command line in INT8 mode. See also the documentation of FasterRCNN spec file for the details of the **trt_evaluation** message structure.

9.4. Evaluating an SSD model

To run evaluation for an SSD model use this command:

```
tlt-evaluate ssd [-h] -e <experiment_spec_file> -m <model_file> -k <key>
```

Required arguments:

- **-e, --experiment_spec_file** : Experiment spec file to set up the evaluation experiment. This should be the same as the training specification file.
- **-m, --model** : Path to the model file to use for evaluation.
- **-k, --key** : Provide the key to load the model.

Optional arguments:

- **-h, --help** : show this help message and exit.

9.5. Evaluating a DSSD model

To run evaluation for an DSSD model use this command:

```
tlt-evaluate ssd [-h] -e <experiment_spec_file> -m <model_file> -k <key>
```

Required arguments:

- **-e, --experiment_spec_file** : Experiment spec file to set up the evaluation experiment. This should be the same as training spec file.
- **-m, --model** : Path to the model file to use for evaluation.
- **-k, --key** : Provide the key to load the model.

Optional arguments:

- **-h, --help** : show this help message and exit.

9.6. Evaluating a YOLOv3 model

To run evaluation for a YOLOv3 model use this command:

```
tlt-evaluate yolo [-h] -e <experiment_spec_file> -m <model_file> -k <key>
```

Required arguments:

- **`-e, --experiment_spec_file`** : Experiment spec file to set up the evaluation experiment. This should be the same as the training specification file.
- **`-m, --model`** : Path to the model file to use for evaluation.
- **`-k, --key`** : Provide the key to load the model.

Optional arguments:

- **`-h, --help`** : show this help message and exit.

9.7. Evaluating a RetinaNet model

To run evaluation for a RetinaNet model use this command:

```
tlt-evaluate retinanet [-h] -e <experiment_spec_file> -m <model_file> -k <key>
```

Required arguments:

- **`-e, --experiment_spec_file`** : Experiment spec file to set up the evaluation experiment. This should be the same as the training specification file.
- **`-m, --model`** : Path to the model file to use for evaluation.
- **`-k, --key`** : Provide the key to load the model.

Optional arguments:

- **`-h, --help`** : show this help message and exit.

9.8. Evaluating a MaskRCNN model

To run evaluation for a MaskRCNN model use this command:

```
tlt-evaluate mask_rcnn [-h] -e <experiment_spec_file> -m <model_file> -k <key>
```

Required arguments:

- **`-e, --experiment_spec_file`** : Experiment spec file to set up the evaluation experiment. This should be the same as the training spec file.
- **`-m, --model`** : Path to the model file to use for evaluation.
- **`-k, --key`** : Provide the key to load the model. This argument is not required if -m is followed by a TensorRT engine.

Optional arguments:

- **`-h, --help`** : show this help message and exit.

Chapter 10. USING INFERENCE ON A MODEL

The **tlt-infer** command runs the inference on a specified set of input images. In the classification mode, **tlt-infer** provides class label output over the command line for a single image or a csv file containing the image path and the corresponding labels for multiple images. In DetectNet_v2, SSD, RetinaNet, DSSD, YOLOV3, or FasterRCNN mode, **tlt-infer** produces output images with bounding boxes rendered on them after inference. Optionally, you can also serialize the output meta-data in kitti_format. In MaskRCNN, **tlt-infer** produces annotated images with bounding boxes and masks rendered on them after inference. TensorRT python inference can also be enabled.

10.1. Running inference on a classification model

Execute **tlt-infer** on a classification model trained on the Transfer Learning Toolkit.

```
tlt-infer classification [-h]
                          -m <model>
                          -i <image>
                          -d <image  dir>
                         [-b <batch size>]
                          -k <key>
                          -cm <classmap>
```

Here are the parameters of the **tlt-infer** tool:

Required arguments

- -m, --model : Path to the pretrained model (TLT model).
- -i, --image : A single image file for inference.
- -d, --image_dir : The directory of input images for inference.
- -k, --key : Key to load model.
- -cm, --class_map : The json file that specifies the class index and label mapping.

Optional arguments

- --batch_size : Inference batch size, default: 1

- -h, --help : show this help message and exit

The inference tool requires a cluster_params.json file to configure the post processing block.**When** executing with **-d or directory mode,**a **result.csv** file will be created and stored in the directory you specify using **-d**. The result.csv has the file path in the first column and predicted labels in the second.

In both single image and directory modes, a classmap (-cm) is required, which should be a byproduct (classmap.json) of your training process.

10.2. Running inference on a DetectNet_v2 model

The tlt-infer tool for object detection networks which may be used to visualize bboxes, or generate frame by frame kitti format labels on a single image or a directory of images. An example of the command for this tool is shown here:

```
tlt-infer detectnet_v2 [-h] -e </path/to/inference/spec/file> \
          -i </path/to/inference/input> \
          -o </path/to/inference/output> \
          -k <model key>
```

Required parameters

- -e, --inference_spec: Path to an inference spec file.
- -i, --inference_input: The directory of input images or a single image for inference.
- -o, --inference_output: The directory to the output images and labels. The annotated images are in inference_output/images_annotated and labels are in inference_output/labels
- -k, --enc_key: Key to load model

The tool automatically generates bbox rendered images in **output_path/images_annotated**. In order to get the bbox labels in KITTI format, please configure the bbox_handler_config spec file using the kitti_dump parameter as mentioned here. This will generate the output in **output_path/labels**.

10.3. Running inference on a FasterRCNN model

The tlt-infer tool for FasterRCNN networks can be used to visualize bboxes, or generate frame by frame KITTI format labels on a directory of images. You can execute this tool from the command line as shown here:

```
tlt-infer faster_rcnn [-h] -e <experiment_spec> [-k <enc_key>]
```

Required arguments:

- **-e, --experiment_spec_file**: Path to the experiment specification file for FasterRCNN training.

Optional arguments:

- **-h, --help**: Print help log and exit.
- **-k, --enc_key**: The encoding key, can override the one in the spec file.

Two modes for tlt-infer

The **tlt-infer** command line for FasterRCNN has two modes. It can run with either TLT backend or TensorRT backend. This behavior is also controlled via the spec file. The **inference_config** in the spec file can have an optional **trt_inference** sub-field that specifies which backend the **tlt-infer** will run with. By default(if the **trt_inference** sub-field is not present in **inference_config**), **tlt-infer** will use TLT as the backend. If the **trt_inference** sub-field is present, it can specify **tlt-infer** to run at TensorRT backend. In that case, the model to do inference can be either the **.etlt** model from **tlt-export** or the TensorRT engine file from **tlt-export** or **tlt-converter**.

To use a TensorRT engine file for TensorRT backend based **tlt-infer**, the **trt_inference** sub-field should look like this:

```
trt_inference {
trt_engine: '/workspace/tlt-experiments/data/faster_rcnn/trt.int8.engine'
}
```

To use a **.etlt** model for TensorRT backend based **tlt-infer**, the **trt_inference** sub-field should look like this:

```
trt_inference {
etlt_model {
model: '/workspace/tlt-experiments/data/faster_rcnn/resnet18.epoch12.etlt'
calibration_cache: '/workspace/tlt-experiments/data/faster_rcnn/cal.bin'
}
trt_data_type: 'int8'
}
```

If the TensorRT inference data type is not INT8, the **calibration_cache** sub-field that provides the path to the INT8 calibration cache is not needed. In INT8 case, the calibration cache should be generated via the **tlt-export** command line in INT8 mode. See also the documentation of FasterRCNN spec file for the details of the **trt_inference** message structure.

10.4. Running inference on an SSD model

The tlt-infer tool for SSD networks can be used to visualize bboxes, or generate frame by frame KITTI format labels on a directory of images. Here's an example of using this tool:

```
tlt-infer ssd  -i <input directory>
               -o <output annotated image directory>
               -e <experiment spec file>
               -m <model file>
               [-l <output label directory>]
               [-t <visualization threshold>]
               -k <key>
```

Required arguments

- -m, --model : Path to the pretrained model (TLT model).
- -i, --in_image_dir : The directory of input images for inference.

- -o, --out_image_dir : The directory path to output annotated images.
- -k, --key : Key to load model.
- -e, --config_path : Path to an experiment spec file for training.

Optional arguments

- -t, --draw_conf_thres : Threshold for drawing a bbox. default: 0.3
- -h, --help : Show this help message and exit
- -l, --out_label_dir : The directory to output KITTI labels.

10.5. Running inference on a DSSD model

The tlt-infer tool for DSSD networks can be used to visualize bboxes, or generate frame by frame KITTI format labels on a directory of images. Here's an example of using this tool:

```
tlt-infer dssd  -i <input directory>
               -o <output annotated image directory>
               -e <experiment spec file>
               -m <model file>
               [-l <output label directory>]
               [-t <visualization threshold>]
               -k <key>
```

Required arguments

- -m, --model : Path to the pretrained model (TLT model).
- -i, --in_image_dir : The directory of input images for inference.
- -o, --out_image_dir : The directory path to output annotated images.
- -k, --key : Key to load model.
- -e, --config_path : Path to an experiment spec file for training.

Optional arguments

- -t, --draw_conf_thres : Threshold for drawing a bbox. default: 0.3
- -h, --help : Show this help message and exit
- -l, --out_label_dir : The directory to output KITTI labels.

10.6. Running inference on a YOLOv3 model

The tlt-infer tool for YOLOv3 networks can be used to visualize bboxes, or generate frame by frame KITTI format labels on a directory of images. Here's an example of using this tool:

```
tlt-infer yolo -i <input directory>
               -o <output annotated image directory>
               -e <experiment spec file>
               -m <model file>
               [-l <output label directory>]
               [-t <visualization threshold>]
               -k <key>
```

Required arguments

- -m, --model : Path to the pretrained model (TLT model).
- -i, --in_image_dir : The directory of input images for inference.
- -o, --out_image_dir : The directory path to output annotated images.
- -k, --key : Key to load model.
- -e, --config_path : Path to an experiment spec file for training.

Optional arguments

- -t, --draw_conf_thres : Threshold for drawing a bbox. default: 0.3
- -h, --help : Show this help message and exit
- -l, --out_label_dir : The directory to output KITTI labels.

10.7. Running inference on a RetinaNet model

The tlt-infer tool for RetinaNet networks can be used to visualize bboxes, or generate frame by frame KITTI format labels on a directory of images. Two modes are supported, namely TLT model model and TensorRT engine mode. You can execute the TLT model mode using the following command:

```
tlt-infer retinanet -i <input directory>
               -o <output annotated image directory>
               -e <experiment spec file>
               -m <model file>
               [-l <output label directory>]
               [-t <visualization threshold>]
               -k <key>
```

Required arguments

- -m, --model : Path to the pretrained model (TLT model).
- -i, --in_image_dir : The directory of input images for inference.
- -o, --out_image_dir : The directory path to output annotated images.
- -k, --key : Key to load model.
- -e, --config_path : Path to an experiment spec file for training.

Optional arguments

- -t, --draw_conf_thres : Threshold for drawing a bbox. default: 0.3
- -h, --help : Show this help message and exit
- -l, --out_label_dir : The directory to output KITTI labels.

Alternatively, you can execute the TensorRT engine mode as follows:

```
tlt-infer retinanet -i <input directory>
               -o <output annotated image directory>
               -e <experiment spec file>
               -p <engine path>
               [-t <visualization threshold>]
               -k <key>
```

Required arguments

- -p, --engine_path : Path to the TensorRT (TLT exported).

- -i, --in_image_dir : The directory of input images for inference.
- -o, --out_image_dir : The directory path to output annotated images.
- -k, --key : Key to load model.
- -e, --config_path : Path to an experiment spec file for training.

Optional arguments

- -t, --draw_conf_thres : Threshold for drawing a bbox. default: 0.3
- -h, --help : Show this help message and exit
- -l, --out_label_dir : The directory to output KITTI labels.

Alternatively, you can execute the TensorRT engine mode as follows:

```
tlt-infer retinanet -i <input directory>
               -o <output annotated image directory>
               -e <experiment spec file>
               -p <engine path>
               [-t <visualization threshold>]
               -k <key>
```

Required arguments

- -p, --engine_path : Path to the TensorRT (TLT exported).
- -i, --in_image_dir : The directory of input images for inference.
- -o, --out_image_dir : The directory path to output annotated images.
- -k, --key : Key to load model.
- -e, --config_path : Path to an experiment spec file for training.

Optional arguments

- -t, --draw_conf_thres : Threshold for drawing a bbox. default: 0.3
- -h, --help : Show this help message and exit

10.8. Running inference on a MaskRCNN model

The tlt-infer tool for MaskRCNN networks can be used to visualize bboxes, or generate frame by frame COCO format labels on a directory of images. Here's an example of using this tool:

```
tlt-infer mask_rcnn -i <input directory>
               -o <output annotated image directory>
               -e <experiment spec file>
               -m <model file>
               [-l <label file>]
               [-b <batch size>]
               [-t <visualization threshold>]
               [--include_mask]
               -k <key>
```

Required arguments

- **`-m, --model`** : Path to the trained model (TLT model).
- **`-i, --input_dir`** : The directory of input images for inference.
- **`-k, --key`** : Key to load model.
- **`-e, --config_path`** : Path to an experiment spec file for training.

- **`-o, --out_dir`** : The directory path to output annotated images.

Optional arguments

- **`-t, --threshold`** : Threshold for drawing a bbox. default: 0.3.
- **`-h, --help`** : Show this help message and exit.
- **`-l, --label_file`** : The label txt file containing groundtruth class labels.
- **`--include_mask`** : Whether to draw masks on the annotated output.

When calling tlt-infer with --trt, the command expects a TensorRT engine as input:

```
tlt-infer mask_rcnn --trt
               -i <input image>
               -o <output annotated image>
               -e <experiment spec file>
               -m <TensorRT engine file>
               [-l <output label file>]
               [-c <class label file>]
               [-t <visualization threshold>]
               [-mt <mask_threshold>]
               [--include_mask]
```

Required arguments

- **`-m, --model`** : Path to the trained model (TLT model).
- **`-i, --in_image_path`**: A directory of input images or a single image file for inference.
- **`-k, --key`** : Key to load model.
- **`-e, --config_path`** : Path to an experiment spec file for training.

Optional arguments

- **`-t, --threshold`** : Confidence threshold for drawing a bbox. Default: 0.6.
- **`-mt, --mask_threshold`**: Confidence threshold for drawing a mask. Default: 0.4.
- **`-o, --out_image_path`** : The output directory of annotated images or a single annotated image file.
- **`-c, --class_label`** : The path to groundtruth label file. If used, the annotated image will display label names.
- -l, --out_label_file : The output directory of predicted labels in json format or a single json file.
- **`--include_mask`** : Whether to draw masks on the annotated output.

Chapter 11. PRUNING THE MODEL

Pruning removes parameters from the model to reduce the model size without compromising the integrity of the model itself using the **tlt-prune** command. Currently tlt-prune **doesn’t** support MaskRCNN models.

The **tlt-prune** command includes these parameters:

```
tlt-prune [-h] -pm <pretrained_model>
               -o <output_file> -k <key>
               [-n <normalizer>]
               [-eq <equalization_criterion>]
               [-pg <pruning_granularity>]
               [-pth <pruning threshold>]
               [-nf <min_num_filters>]
               [-el [<excluded_list>]
```

Required arguments:

- **-pm, --pretrained_model** : Path to pretrained model.
- **-o, --output_file** : Path to output checkpoints.
- **-k, --key** : Key to load a .tlt model

Optional arguments

- **-h, --help**: Show this help message and exit.
- **-n, –normalizer** : `max` to normalize by dividing each norm by the maximum norm within a layer; `L2` to normalize by dividing by the L2 norm of the vector comprising all kernel norms. (default: `max`)
- **-eq, --equalization_criterion** : Criteria to equalize the stats of inputs to an element wise op layer, or depth-wise convolutional layer. This parameter is useful for resnets and mobilenets. Options are [arithmetic_mean, geometric_mean, union, intersection]. (default: `union`)
- **-pg, -pruning_granularity**: Number of filters to remove at a time. (default:8).
- **-pth** : Threshold to compare normalized norm against. (default:0.1)

NVIDIA recommends changing the threshold to keep the number of parameters in the model to within 10-20% of the original unpruned model.

- **-nf, --min_num_filters** : Minimum number of filters to keep per layer. (default:16)
- **-el, --excluded_layers**: List of excluded_layers. Examples: -i item1 item2 (default: [])

After pruning, the model needs to be retrained. See Re-training the pruned model.

Using the Prune command

Here's an example of using the **tlt-prune** command:

```
tlt-prune -m /workspace/output/weights/resnet_003.tlt \
                     -o /workspace/output/weights/resnet_003_pruned.tlt \
                     -eq union \
                     -pth 0.7 -k $KEY
```

Re-training the pruned model

Once the model has been pruned, there might be a slight decrease in accuracy. This happens because some previously useful weights may have been removed. In order to regain the accuracy, NVIDIA recommends that you retrain this pruned model over the same dataset. To do this, use the **tlt-train** command as documented in Training the model, with an updated spec file that points to the newly pruned model as the pretrained model file.

Users are advised to turn off the regularizer in the training_config for detectnet to recover the accuracy when retraining a pruned model. You may do this by setting the regularizer type to NO_REG as mentioned here. All the other parameters may be retained in the spec file from the previous training.

For detectnet_v2, it is important to set the **load_graph** under **model_config** to **true** to import the pruned graph.

Chapter 12. EXPORTING THE MODEL

The Transfer Learning Toolkit includes the **`tlt-export`** command to export and prepare TLT models for Deploying to DeepStream. The **`tlt-export`** command optionally generates the calibration cache for TensorRT INT8 engine calibration.

Exporting the model decouples the training process from inference and allows conversion to TensorRT engines outside the TLT environment. TensorRT engines are specific to each hardware configuration and should be generated for each unique inference environment. This may be interchangeably referred to as .trt or .engine file. The same exported TLT model may be used universally across training and deployment hardware. This is referred to as the .etlt file or encrypted TLT file. During model export TLT model is encrypted with a private key. This key is required when you deploy this model for inference.

INT8 mode overview

TensorRT engines can be generated in INT8 mode to improve performance, but require a calibration cache at engine creation-time. The calibration cache is generated using a calibration tensor file, if **`tlt-export`** is run with the **`--data_type`** flag set to **`int8`**. Pre-generating the calibration information and caching it removes the need for calibrating the model on the inference machine. Moving the calibration cache is usually much more convenient than moving the calibration tensorfile, since it is a much smaller file and can be moved with the exported model. Using the calibration cache also speeds up engine creation as building the cache can take several minutes to generate depending on the size of the Tensorfile and the model itself.

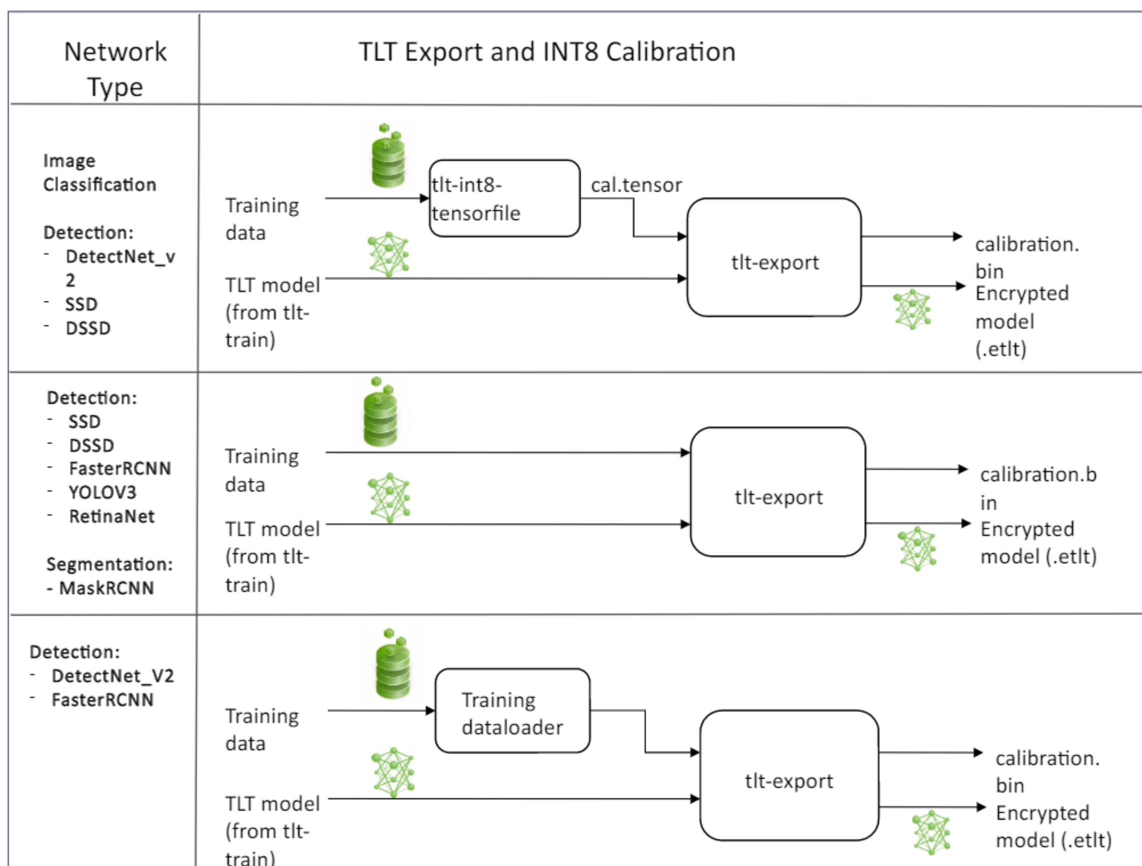
The export tool can generate INT8 calibration cache by ingesting training data using either of these options:

- Option 1: Providing a calibration tensorfile generated using the **`tlt-int8-tensorfile`** command. For image classification, and detection using Detectnet_v2, SSD and DSSD, the recommendation is to use this option, because the **`tlt-int8-tensorfile`** command uses the data generators to produce the training data. This helps easily generate a representative subsample of the training dataset.
- Option 2: Pointing the tool to a directory of images that you want to use to calibrate the model. For this option, make sure to create a sub-sampled directory of random

images that best represent your training dataset. For FasterRCNN, YOLOV3 and RetinaNet detection architecture, only option 2 is supported.

- Option 3: Using the training data loader to load the training images for INT8 calibration.This option is supported for DetectNet_v2 and FasterRCNN. This option is now the recommended approach to support multiple image directories by leveraging the training dataset loader. This also ensures 2 important aspects of data during calibration:
  - Data pre=processing in the INT8 calibration step is the same as in the training process and
  - The data batches are sampled randomly across the entire training dataset, thereby improving the accuracy of the int8 model.

NVIDIA plans to eventually deprecate the Option 1 and only support Option 2 and 3.



FP16/FP32 model

The **`calibration.bin`** is only required if you need to run inference at INT8 precision. For FP16/FP32 based inference, the export step is much simpler. All that is required is to provide a model from the **`tlt-train`** step to **`tlt-export`** to convert into an encrypted tlt model.

Generating an INT8 tensorfile using the tlt-int8-tensorfile command

The INT8 tensorfile is a binary file that contains the preprocessed training samples, which maybe used to calibrate the model. In this release, TLT only supports calibration tensorfile generation for SSD, DSSD, DetectNet_v2 and classification models.

The sample usage for the **`tlt-int8-tensorfile`** command to generate a calibration tensorfile is defined as below:

```
tlt-int8-tensorfile {classification, detectnet_v2} [-h]
                       -e <path to training experiment spec file>
                       -o <path to output tensorfile>
                       -m <maximum number of batches to serialize>
                       [--use_validation_set]
```

Positional arguments:

`classification`, **`detectnet_v2`**, **`ssd`** or **`dssd`**

Required arguments:

- **`-e, --experiment_spec_file`**: Path to the experiment spec file. (Only required for SSD and FasterRCNN.)
- **`-o, --output_path`**: Path to the output tensorfile that will be created.
- **`-m, --max_batches`**: Number of batches of input data to be serialized.

Optional argument

- **`--use_validation_set`**: Flag to use validation dataset instead of training set.

Here's a sample command to invoke the **`tlt-int8-tensorfile`** command for a classification model.

```
tlt-int8-tensorfile classification -e $SPECS_DIR/classification_retrain_spec.cfg
 \
                                   -m 10 \
                                   -o $USER_EXPERIMENT_DIR/export/
calibration.tensor
```

Exporting the model using tlt-export

Here's an example of the command line arguments of the tlt-export command:

```
tlt-export [-h] {classification, detectnet_v2, ssd, dssd, faster_rcnn, yolo,
 retinanet}
                -m <path to the .tlt model file generated by tlt train>
                -k <key>
                [-o <path to output file>]
                [--cal_data_file <path to tensor file>]
                [--cal_image_dir <path to the directory images to calibrate the
 model]
                [--cal_cache_file <path to output calibration file>]
                [--data_type <Data type for the TensorRT backend during export>]
                [--batches <Number of batches to calibrate over>]
                [--max_batch_size <maximum trt batch size>]
                [--max_workspace_size <maximum workspace size]
```

```
                [--batch_size <batch size to TensorRT engine>]
                [--experiment_spec <path to experiment spec file>]
                [--engine_file <path to the TensorRT engine file>]
                [--verbose Verbosity of the logger]
                [--force_ptq Flag to force PTQ]
```

Required arguments:

- **`export_module`**: Which model to export, can be **classification**, **detectnet_v2**, **faster_rcnn**, **ssd**, **dssd**, **yolo**, **retinanet**. This is a positional argument.
- **`-m, --model`**: Path to the .tlt model file to be exported using **`tlt-export`**.
- **`-k, --key`**: Key used to save the .tlt model file.

Optional arguments:

- **`-o, --output_file`** : Path to save the exported model to. The default is **`./<input_file>.etlt`**.
- **`--data_type`**: Desired engine data type, generates calibration cache if in INT8 mode. The options are: {fp32, fp16, int8} The default value is fp32. If using int8, following INT8 arguments are required.
- **`-s, --strict_type_constraints`**: A Boolean flag to indicate whether or not to apply the TensorRT strict_type_constraints when building the TensorRT engine. Note this is only for applying the strict type of INT8 mode.

INT8 export mode required arguments:

- **`--cal_data_file`**: tensorfile generated from **`tlt-int8-tensorfile`** for calibrating the engine. This can also be an output file if used with **`--cal_image_dir`**.
- **`--cal_image_dir`**: Directory of images to use for calibration.

`--cal_image_dir` parameter for images and applies the necessary preprocessing to generate a tensorfile at the path mentioned in the **`--cal_data_file`** parameter, which is in turn used for calibration. The number of batches in the tensorfile generated is obtained from the value set to the **`--batches`** parameter, and the batch_size is obtained from the value set to the **`--batch_size`** parameter. Be sure that the directory mentioned in **`--cal_image_dir`** has at least **`batch_size * batches`** number of images in it. The valid image extensions are jpg, jpeg and png. In this case, the input_dimensions of the calibration tensors are derived from the input layer of the .tlt model.

INT8 export optional arguments:

- **`--cal_cache_file`**: Path to save the calibration cache file. The default value is **`./cal.bin`**.
- **`--batches`**: Number of batches to use for calibration and inference testing.The default value is 10.
- **`--batch_size`**: Batch size to use for calibration. The default value is 8.
- **`--max_batch_size`**: Maximum batch size of TensorRT engine. The default value is 16.
- **`--max_workspace_size`** : Maximum workspace size of TensorRT engine. The default value is: 1073741824 = 1<<30)

- **--experiment_spec**: The experiment_spec for training/inference/evaluation. This is used to generate the graphsurgeon config script for FasterRCNN from the experiment_spec, only useful for FasterRCNN. This when used with DetectNet_v2 and FasterRCNN also sets up the dataloader based calibrator to leverage the training dataloader to calibrate the model.
- **--engine_file**: Path to the serialized TensorRT engine file. Note that this file is hardware specific, and cannot be generalized across GPUs. Useful to quickly test your model accuracy using TensorRT on the host. As TensorRT engine file is hardware specific, you cannot use this engine file for deployment unless the deployment GPU is identical to training GPU.
- **--force_ptq**: A boolean flag to force post training quantization on the exported etlt model.

When exporting a model trained with QAT enabled, the tensor scale factors to calibrate the activations are peeled out of the model and serialized to a TensorRT readable cache file defined by the **cal_cache_file** argument. However, do note that the current version of QAT doesn't natively support DLA int8 deployment in the Jetson. Inorder to deploy this model on a Jetson with DLA **int8**, please use the **--force_ptq** flag to use TensorRT post training quantization to generate the calibration cache file.

Exporting a model

Here's a sample command to export a DetectNet_v2 model in INT8 mode. This command shows option 1; uses **--cal_data_file** option with the **calibration.tensor** generated using **tlt-int8-tensorfile** command.

```
tlt-export  detectnet_v2 \
            -m $USER_EXPERIMENT_DIR/experiment_dir_retrain/weights/
resnet18_detector_pruned.tlt \
           -o $USER_EXPERIMENT_DIR/experiment_dir_final/resnet18_detector.etlt \
           -k $KEY \
           --cal_data_file $USER_EXPERIMENT_DIR/experiment_dir_final/
calibration.tensor \
           --data_type int8 \
           --batches 10 \
           --cal_cache_file $USER_EXPERIMENT_DIR/experiment_dir_final/
calibration.bin
           --engine_file $USER_EXPERIMENT_DIR/experiment_dir_final/
resnet_18.engine
```

Here's an example log of a successful export:

```
Using TensorFlow backend.
2018-11-02 18:59:43,347 [INFO] iva.common.tlt-export: Loading model from
 resnet10_kitti_multiclass_v1.tlt
..
2018-11-02 18:59:47,572 [INFO] tensorflow: Restoring parameters from /tmp/
tmp8crUBp.ckpt
INFO:tensorflow:Froze 82 variables.
2018-11-02 18:59:47,701 [INFO] tensorflow: Froze 82 variables.
Converted 82 variables to const ops.
2018-11-02 18:59:48,123 [INFO] iva.common.tlt-export: Converted model was saved
 into resnet10_kitti_multiclass_v1.etlt
2018-11-02 18:59:48,123 [INFO] iva.common.tlt-export: Input node: input_1
2018-11-02 18:59:48,124 [INFO] iva.common.tlt-export: Output node(s):
 ['output_bbox/BiasAdd', 'output_cov/Sigmoid']
```

Here's a sample command using the **`--cal_image_dir`** option for a FasterRCNN model using option 2.

```
tlt-export faster_rcnn \
          -m $USER_EXPERIMENT_DIR/data/faster_rcnn/
frcnn_kitti_retrain.epoch12.tlt \
          -o $USER_EXPERIMENT_DIR/data/faster_rcnn/frcnn_kitti_retrain.int8.etlt
 \
          -e $SPECS_DIR/frcnn_kitti_retrain_spec.txt \
          --key $KEY \
          --cal_image_dir  $USER_EXPERIMENT_DIR/data/KITTI/val/image_2 \
          --data_type int8 \
          --batch_size 8 \
          --batches 10 \
          --cal_data_file $USER_EXPERIMENT_DIR/data/faster_rcnn/cal.tensorfile \
          --cal_cache_file $USER_EXPERIMENT_DIR/data/faster_rcnn/cal.bin \
          --engine_file $USER_EXPERIMENT_DIR/data/faster_rcnn/detection.trt
```

Chapter 13. DEPLOYING TO DEEPSTREAM

The deep learning and computer vision models that you trained can be deployed on edge devices, such as a Jetson Xavier, Jetson Nano or a Tesla or in the cloud with NVIDIA GPUs. TLT has been designed to integrate with DeepStream SDK, so models trained with TLT will work out of the box with DeepStream SDK.

DeepStream SDK is a streaming analytic toolkit to accelerate building AI-based video analytic applications. DeepStream supports direct integration of Classification and DetectNet_v2 exported models into the deepstream sample app. The documentation for the DeepStream SDK is provided here [https://docs.nvidia.com/metropolis/deepstream/dev-guide/index.html]. For other models such as YOLOv3, FasterRCNN, SSD, DSSD, RetinaNet, and MaskRCNN there are few extra steps that are required which are covered in this chapter.

To deploy a model trained by TLT to DeepStream you can run multiple options:

- Option 1: Integrate the model (.etlt) with the encrypted key directly in the DeepStream app. The model file is generated by **`tlt-export`**.
- Option 2: Generate a device specific optimized TensorRT engine, using **`tlt-converter.`** The TensorRT engine file can also be ingested by DeepStream.

Machine specific optimizations are done as part of the engine creation process, so a distinct engine should be generated for each environment and hardware configuration. If the inference environment's TensorRT or CUDA libraries are updated – including minor version updates or if a new model is generated– new engines need to be generated. Running an engine that was generated with a different version of TensorRT and CUDA is not supported and will cause unknown behavior that affects inference speed, accuracy, and stability, or it may fail to run altogether.

This image shows DeepStream deployment method for all the models plus the two deployment options. Option 1 is very straightforward. The .etlt file and calibration cache are directly used by DeepStream. DeepStream will automatically generate TensorRT engine file and then run inference. The generation of TensorRT engine can take some time depending on size of the model and type of Hardware. The generation of TensorRT engine can be done ahead of time with Option 2. With option 2, use tlt-converter to convert the .etlt file to TensorRT engine and then provide the engine file directly to DeepStream.

Network Type	DeepStream Deployment Option 1	DeepStream Deployment Option 2
Image Classification **Detection:** - **DetectNet_v2**	calibration.bin Encrypted model (.etlt) DeepStream 5.0	calibration.bin Encrypted model (.etlt) tlt-converter TRT engine DeepStream 5.0
Detection: - **SSD** - **DSSD** - **FasterRCNN** - **YOLOV3** - **RetinaNet** **Segmentation:** - **MaskRCNN**	calibration.bin Encrypted model (.etlt) DeepStream 5.0 Build Custom Parser Build TensorRT 7+ OSS	Build Custom Parser calibration.bin Encrypted model (.etlt) tlt-converter TRT engine DeepStream 5.0 Build TensorRT 7+ OSS

Running TLT models on DeepStream for DetectNet_v2 based detection and image classification, shown on the top half of the table is very straightforward. All that is required is the encrypted tlt model (.etlt), optional INT8 calibration cache and DeepStream config file. Go to Integrating a DetectNet_v2 model to see the DeepStream config file.

For other detection models such as FasterRCNN, YOLOv3, RetinaNet, SSD, and DSSD, and segmentation model such as MaskRCNN there are extra steps that need to be completed before the models will work with DeepStream. Here are the steps with detailed instructions in the following sections.

Step 1: Build TensorRT Open source software (OSS). This is required because several TensorRT plugins that are required by these models are only available in TensorRT open source repo and not in the general TensorRT release. For more information and instructions, see the TensorRT Open Source Software section.

Step 2: Build custom parsers for DeepStream. The parsers are required to convert the raw Tensor data from the inference to (x,y) location of bounding boxes around the detected object. This post-processing algorithm will vary based on the detection architecture. For DetectNet_v2, the custom parsers are not required because the parsers are built-in with DeepStream SDK. For other detectors, DeepStream provides flexibility to add your own custom bounding box parser and that will be used for these 5 models.

13.1. TensorRT Open Source Software (OSS)

TensorRT OSS build is required for FasterRCNN, SSD, DSSD, YOLOv3, RetinaNet, and MaskRCNN models. This is required because several TensorRT plugins that are required

by these models are only available in TensorRT open source repo and not in the general TensorRT release. The table below shows the plugins that are required by each network.

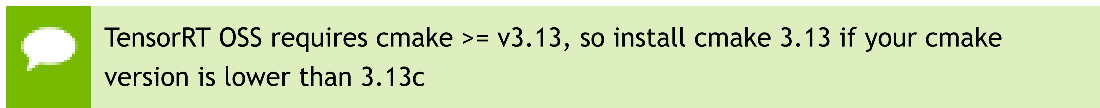
Network	Plugins required
SSD	batchTilePlugin and NMSPlugin
FasterRCNN	cropAndResizePlugin and proposalPlugin
YOLOV3	batchTilePlugin, resizeNearestPlugin and batchedNMSPlugin
DSSD	batchTilePlugin and NMSPlugin
RetinaNet	batchTilePlugin and NMSPlugin
MaskRCNN	generateDetectionPlugin, multilevelProposeROI, multilevelCropAndResizePlugin, resizeNearestPlugin

If the deployment platform is x86 with NVIDIA GPU, follow instructions for x86 and if your deployment is on NVIDIA Jetson platform, follow instructions for Jetson.

TensorRT OSS on x86

Building TensorRT OSS on x86:

1. Install Cmake (>=3.13)



TensorRT OSS requires cmake >= v3.13, so install cmake 3.13 if your cmake version is lower than 3.13c

```
sudo apt remove --purge --auto-remove cmake
wget https://github.com/Kitware/CMake/releases/download/v3.13.5/
cmake-3.13.5.tar.gz
tar xvf cmake-3.13.5.tar.gz
cd cmake-3.13.5/
./configure
make -j$(nproc)
sudo make install
sudo ln -s /usr/local/bin/cmake /usr/bin/cmake
```

2. Get GPU Arch

   GPU_ARCHS value can be retrieved by the **deviceQuery** CUDA sample

```
cd /usr/local/cuda/samples/1_Utilities/deviceQuery
sudo make
./deviceQuery
```

   If the **"/usr/local/cuda/samples"** doesn't exist in your system, you could download **deviceQuery.cpp** from this repo. Compile and run **deviceQuery**.

```
nvcc deviceQuery.cpp -o deviceQuery
./deviceQuery
```

   This command will output something like this, which indicates the "GPU_ARCHS" is 75 based on CUDA Capability major/minor version.

```
Detected 2 CUDA Capable device(s)

Device 0: "Tesla T4"
```

```
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    7.5
```

3. Build TensorRT OSS

```
git clone -b release/7.0 https://github.com/nvidia/TensorRT
cd TensorRT/
git submodule update --init --recursive
export TRT_SOURCE=`pwd`
cd $TRT_SOURCE
mkdir -p build && cd build
```

Make sure your GPU_ARCHS from step 2 is in TensorRT OSS **`CMakeLists.txt`**. If GPU_ARCHS is not in TensorRT OSS CMakeLists.txt, add **`-DGPU_ARCHS=<VER>`** as below, where **`<VER>`** represents GPU_ARCHS from step 2.

```
/usr/local/bin/cmake .. -DGPU_ARCHS=xy  -DTRT_LIB_DIR=/usr/lib/aarch64-
linux-gnu/ -DCMAKE_C_COMPILER=/usr/bin/gcc -DTRT_BIN_DIR=`pwd`/out
make nvinfer_plugin -j$(nproc)
```

After building ends successfully, **`libnvinfer_plugin.so*`** will be generated under **`` `pwd`/out/ ``**.

4. Replace the original "**`libnvinfer_plugin.so*`**"

```
sudo mv /usr/lib/x86_64-linux-gnu/libnvinfer_plugin.so.7.x.y ${HOME}/
libnvinfer_plugin.so.7.x.y.bak   // backup original libnvinfer_plugin.so.x.y
sudo cp $TRT_SOURCE/`pwd`/out/libnvinfer_plugin.so.7.m.n  /usr/lib/x86_64-
linux-gnu/libnvinfer_plugin.so.7.x.y
sudo ldconfig
```

TensorRT OSS on Jetson (ARM64)

1. Install Cmake (>=3.13)

TensorRT OSS requires cmake >= v3.13, while the default cmake on Jetson/ UBuntu 18.04 is cmake 3.10.2.

Upgrade TensorRT OSS using:

```
sudo apt remove --purge --auto-remove cmake
wget https://github.com/Kitware/CMake/releases/download/v3.13.5/
cmake-3.13.5.tar.gz
tar xvf cmake-3.13.5.tar.gz
cd cmake-3.13.5/
./configure
make -j$(nproc)
sudo make install
sudo ln -s /usr/local/bin/cmake /usr/bin/cmake
```

2. Get GPU Arch based on your platform. The GPU_ARCHS for different Jetson platform are given in the following table.

Jetson Platform	GPU_ARCHS
Nano/Tx1	53
Tx2	62

AGX Xavier/Xavier NX	72

3. Build TensorRT OSS

```
git clone -b release/7.0 https://github.com/nvidia/TensorRT
cd TensorRT/
git submodule update --init --recursive
export TRT_SOURCE=`pwd`
cd $TRT_SOURCE
mkdir -p build && cd build
```

The **-DGPU_ARCHS=72** below is for Xavier or NX, for other Jetson platform, please change "72" referring to "GPU_ARCH" from step 2.

```
/usr/local/bin/cmake .. -DGPU_ARCHS=72  -DTRT_LIB_DIR=/usr/lib/aarch64-
linux-gnu/ -DCMAKE_C_COMPILER=/usr/bin/gcc -DTRT_BIN_DIR=`pwd`/out
make nvinfer_plugin -j$(nproc)
```

After building ends successfully, **libnvinfer_plugin.so*** will be generated under **'pwd'/out/**.

4. Replace "**libnvinfer_plugin.so***" with the newly generated.

```
sudo mv /usr/lib/aarch64-linux-gnu/libnvinfer_plugin.so.7.x.y ${HOME}/
libnvinfer_plugin.so.7.x.y.bak   // backup original libnvinfer_plugin.so.x.y
sudo cp `pwd`/out/libnvinfer_plugin.so.7.m.n  /usr/lib/aarch64-linux-gnu/
libnvinfer_plugin.so.7.x.y
sudo ldconfig
```

13.2. Generating an engine using tlt-converter

Setup and Execution

This is part of option 2 from the DeepStream deployment table above. The **tlt-converter** is a tool that is provided with the Transfer Learning Toolkit to facilitate the deployment of TLT trained models on TensorRT and/or Deepstream. For deployment platforms with an x86 based CPU and discrete GPU's, the **tlt-converter** is distributed within the TLT docker. Therefore, it is suggested to use the docker to generate the engine. However, this requires that the user adhere to the same minor version of TensorRT as distributed with the docker. The TLT docker includes TensorRT version 5.1 for JetPack 4.2.2 and TensorRT version 6.0.1 for JetPack 4.2.3 / 4.3. In order to use the engine with a different minor version of TensorRT, copy the converter from **/opt/nvidia/tools/tlt-converter** to the target machine and follow the instructions for x86 to run it and generate a TensorRT engine.

Instructions for x86

1. Copy **/opt/nvidia/tools/tlt-converter** to the target machine.
2. Install TensorRT 7.0+ for the respective target machine from here.
3. If you are deploying FasterRCNN, SSD, DSSD, YOLOv3, RetinaNet, or MaskRCNN model, you need to build TensorRT Open source software on the machine. If you are using DetectNet_v2 or image classification, you can skip this step. Instructions to build TensorRT OSS on x86 can be found in TensorRT OSS on x86 section above or in this GitHub repo.

4. Run **tlt-converter** using the sample command below and generate the engine.

Instructions for Jetson

For the Jetson platform, the **tlt-converter** is available to download in the dev zone here. Once the **tlt-converter** is downloaded, please follow the instructions below to generate a TensorRT engine.

1. Unzip **tlt-converter-trt7.1.zip** on the target machine.
2. Install the open ssl package using the command:

   sudo apt-get install libssl-dev
3. Export the following environment variables:

```
$ export TRT_LIB_PATH="/usr/lib/aarch64-linux-gnu"
$ export TRT_INC_PATH="/usr/include/aarch64-linux-gnu"
```
4. For Jetson devices, TensorRT 7.1 comes pre-installed with https://developer.nvidia.com/embedded/jetpack. If you are using older JetPack, upgrade to JetPack 4.4.
5. If you are deploying FasterRCNN, SSD, DSSD, YOLOv3, or RetinaNet model, you need to build TensorRT Open source software on the machine. If you are using DetectNet_v2 or image classification, you can skip this step. Instructions to build TensorRT OSS on Jetson can be found in TensorRT OSS on Jetson (ARM64) section above or in this GitHub repo.
6. Run the **tlt-converter** using the sample command below and generate the engine.

Make sure to follow the output node names as mentioned in Exporting the model.

Using the tlt-converter

```
tlt-converter [-h] -k <encryption_key>
              -d <input_dimensions>
              -o <comma separated output nodes>
              [-c <path to calibration cache file>]
              [-e <path to output engine>]
              [-b <calibration batch size>]
              [-m <maximum batch size of the TRT engine>]
              [-t <engine datatype>]
              [-w <maximum workspace size of the TRT Engine>]
              [-i <input dimension ordering>]
              input_file
```

Required arguments:

- **input_file**: Path to the model exported using **tlt-export**.
- **-k**: The API key used to configure the ngc cli to download the models.
- **-d**: Comma-separated list of input dimensions that should match the dimensions used for **tlt-export**. Unlike **tlt-export** this cannot be inferred from calibration data.
- **-o**: Comma-separated list of output blob names that should match the output configuration used for **tlt-export**.
  - For classification use: predictions/Softmax.

- For DetectNet_v2: output_bbox/BiasAdd,output_cov/Sigmoid
- For FasterRCNN: dense_class_td/Softmax,dense_regress_td/BiasAdd, proposal
- For SSD, DSSD, RetinaNet: NMS
- For YOLOv3: BatchedNMS
- For MaskRCNN: generate_detections, mask_head/mask_fcn_logits/BiasAdd

Optional arguments:

- **`-e`**: Path to save the engine to. (default: ./saved.engine)
- **`-t`**: Desired engine data type, generates calibration cache if in INT8 mode. The default value is fp32.The options are {fp32, fp16, int8}
- **`-w`**: Maximum workspace size for the TensorRT engine. The default value is **`1<<30`**.
- **`-i`**: Input dimension ordering, all other tlt commands use NCHW. The default value is nchw. The options are {nchw, nhwc, nc}.

INT8 Mode Arguments:

- **`-c`**: Path to calibration cache file, only used in INT8 mode. The default value is **`./cal.bin`**.
- **`-b`**: Batch size used during the tlt-export step for INT8 calibration cache generation. (default: 8).
- **`-m`**: Maximum batch size of TensorRT engine. The default value is 16.

Sample output log

Sample log for exporting a resnet10 detectnet_v2 model.

Here's a sample:

```
export API_KEY=<NGC API key used to download the original model>
export OUTPUT_NODES=output_bbox/BiasAdd,output_cov/Sigmoid
export INPUT_DIMS=3,384,124
export D_TYPE=fp32
export ENGINE_PATH=resnet10_kitti_multiclass_v1.engine
export MODEL_PATH=resnet10_kitti_multiclass_v1.etlt

tlt-converter -k $API_KEY \
              -o $OUTPUT_NODES \
              -d $INPUT_DIMS \
              -e $ENGINE_PATH \
              $MODEL_PATH

[INFO] UFFParser: parsing input_1
[INFO] UFFParser: parsing conv1/kernel
[INFO] UFFParser: parsing conv1/convolution
[INFO] UFFParser: parsing conv1/bias
[INFO] UFFParser: parsing conv1/BiasAdd
[INFO] UFFParser: parsing bn_conv1/moving_variance
..
..
..
[INFO] Tactic 4 scratch requested: 1908801536, available: 16
[INFO] Tactic 5 scratch requested: 55567168, available: 16
[INFO] --------------- Chose 1 (0)
[INFO] Formats and tactics selection completed in 5.0141 seconds.
[INFO] After reformat layers: 16 layers
[INFO] Block size 490733568
[INFO] Block size 122683392
[INFO] Block size 122683392
```

```
[INFO] Block size 30670848
[INFO] Block size 16
[INFO] Total Activation Memory: 766771216
[INFO] Data initialization and engine generation completed in 0.0412826 seconds
```

13.3. Integrating the model to DeepStream

There are 2 options to integrate models from TLT with DeepStream:

- Option 1: Integrate the model (.etlt) with the encrypted key directly in the DeepStream app. The model file is generated by **`tlt-export`**.
- Option 2: Generate a device specific optimized TensorRT engine, using **`tlt-converter`**. The TensorRT engine file can also be ingested by DeepStream.

Network Type	DeepStream Deployment Option 1	DeepStream Deployment Option 2
Image Classification **Detection:** - **DetectNet_v2**	calibration.bin Encrypted model (.etlt) DeepStream 5.0	calibration.bin Encrypted model (.etlt) tlt-converter TRT engine DeepStream 5.0
Detection: - **SSD** - **DSSD** - **FasterRCNN** - **YOLOV3** - **RetinaNet** **Segmentation:** **- MaskRCNN**	calibration.bin Encrypted model (.etlt) DeepStream 5.0 Build Custom Parser Build TensorRT 7+ OSS	Build Custom Parser calibration.bin Encrypted model (.etlt) tlt-converter TRT engine DeepStream 5.0 Build TensorRT 7+ OSS

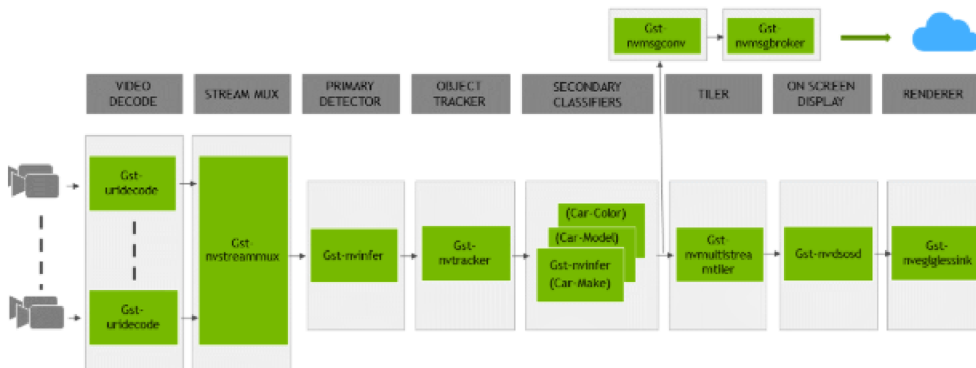
As shown in the lower half of the table, for models such as YOLOv3, FasterRCNN, SSD, DSSD, RetinaNet, and MaskRCNN, you will need to build TensorRT Open source plugins and custom bounding box parsing. The instructions are provided below in the TensorRT OSS section above and the required code can be found in this GitHub repo.

In order to integrate the models with DeepStream, you need the following:

1. Download and install DeepStream SDK. The installation instructions for DeepStream are provided in DeepStream development guide .
2. An **`exported .etlt`** model file and optional calibration cache for INT8 precision.
3. TensorRT 7+ OSS Plugins (Required for FasterRCNN, SSD, DSSD, YOLOv3, RetinaNet, MaskRCNN).
4. A **`labels.txt`** file containing the labels for classes in the order in which the networks produces outputs.

5. A sample **config_infer_*.txt** file to configure the nvinfer element in DeepStream. The nvinfer element handles everything related to TensorRT optimization and engine creation in DeepStream.

DeepStream SDK ships with an end-to-end reference application which is fully configurable. Users can configure input sources, inference model and output sinks. The app requires a primary object detection model, followed by an optional secondary classification model. The reference application is installed as **deepstream-app**. The graphic below shows the architecture of the reference application.



There are typically 2 or more configuration files that are used with this app. In the install directory, the config files are located in '**samples/configs/deepstream-app**' or '**sample/configs/tlt_pretrained_models**'. The main config file configures all the high level parameters in the pipeline above. This would set input source and resolution, number of inferences, tracker and output sinks. The other supporting config files are for each individual inference engine. The inference specific config files are used to specify models, inference resolution, batch size, number of classes and other customization. The main config file will call all the supporting config files. Here are some config files in '**samples/configs/deepstream-app**' for your reference.

source4_1080p_dec_infer-resnet_tracker_sgie_tiled_display_int8.txt - Main config file

config_infer_primary.txt - Supporting config file for primary detector in the pipeline above

config_infer_secondary_*.txt - Supporting config file for secondary classifier in the pipeline above

The **deepstream-app** will only work with the main config file. This file will most likely remain the same for all models and can be used directly from the DeepStream SDK will little to no change. User will only have to modify or create **config_infer_primary.txt** and **config_infer_secondary_*.txt**

13.3.1. Integrating a Classification model

See Exporting the model for more details on how to export a TLT model. Once the model has been generated two extra files are required:

1. Label file
2. DeepStream configuration file

Label file

The label file is a text file, containing the names of the classes that the TLT model is trained to classify against. The order in which the classes are listed must match the order in which the model predicts the output. This order may be deduced from the **classmap.json** file that is generated by TLT. This file is a simple dictionary containing the 'class_name' to 'index map'. For example, in the sample classification sample notebook file included with the tlt-docker, the **classmap.json** file generated for pascal voc would look like this:

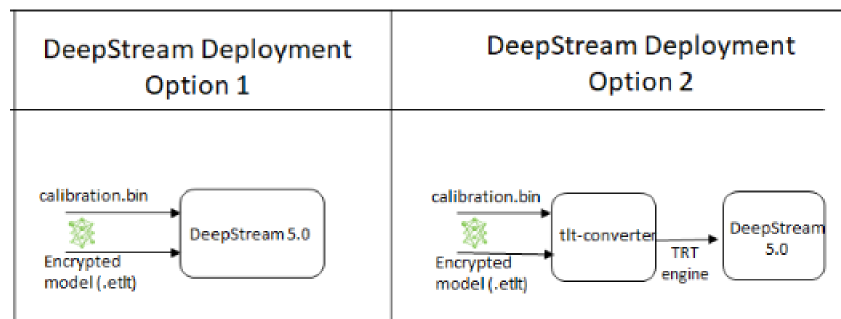
```
{"sheep": 16,"horse": 12,"bicycle": 1, "aeroplane": 0, "cow": 9,
 "sofa": 17, "bus": 5, "dog": 11, "cat": 7, "person": 14, "train": 18,
 "diningtable": 10, "bottle": 4, "car": 6, "pottedplant": 15,
 "tvmonitor": 19, "chair": 8, "bird": 2, "boat": 3, "motorbike": 13}
```

The 0th index corresponds to **aeroplane**, the 1st index corresponds to **bicycle**, etc. up to 19 which corresponds to **tvmonitor**. Here is a sample **label.txt** file, **classification_labels.txt**, arranged in the order of index.

```
aeroplane
bicycle
bird
boat
bottle
bus
..
..
tvmonitor
```

DeepStream configuration file

A typical use case for video analytic is first to do an object detection and then crop the detected object and send it further for classification. This is supported by **'deepstream-app'** and the app architecture can be seen above. For example, to classify models of cars on the road, first you will need to detect all the cars in a frame. Once you do detection, you do classification on the cropped image of the car. So in the sample DeepStream app, the classifier is configured as a secondary inference engine after the primary detection. If configured appropriately, **deepstream-app** will automatically crop the detected image and send the frame to the secondary classifier. The **config_infer_secondary_*.txt** is used to configure the classification model.



Option 1: Integrate the model (.etlt) directly in the DeepStream app. For this option, users will need to add the following parameters in the configuration file. The **'int8-calib-file'** is only required for INT8 precision.

```
tlt-encoded-model=<TLT exported .etlt>
tlt-model-key=<Model export key>
```

```
int8-calib-file=<Calibration cache file>
```

Option 2: Integrate TensorRT engine file with DeepStream app.

Step 1: Generate TensorRT engine using tlt-converter. Detail instructions are provided in the Generating an engine using tlt-converter section.

Step 2: Once the engine file is generated successfully, modify the following parameters to use this engine with DeepStream.

```
model-engine-file=<PATH to generated TensorRT engine>
```

All other parameters are common between the 2 approaches. Add the label file generated above using:

labelfile-path=<Classification labels>

For all the options, see the configuration file below. To learn about what all the parameters are used for, refer to DeepStream Development Guide .

```
[property]
gpu-id=0
# preprocessing parameters: These are the same for all classification models
 generated by TLT.
net-scale-factor=1.0
offsets=123.67;116.28;103.53
model-color-format=1
batch-size=30

# Model specific paths. These need to be updated for every classification model.
int8-calib-file=<Path to optional INT8 calibration cache>
labelfile-path=<Path to classification_labels.txt>
tlt-encoded-model=<Path to Classification TLT model>
tlt-model-key=<Key to decrypt model>
input-dims=c;h;w;0 # where c = number of channels, h = height of the model
 input, w = width of model input, 0: implies CHW format.
uff-input-blob-name=input_1
output-blob-names=predictions/Softmax #output node name for classification

## 0=FP32, 1=INT8, 2=FP16 mode
network-mode=0
# process-mode: 2 - inferences on crops from primary detector, 1 - inferences on
 whole frame
process-mode=2
interval=0
network-type=1 # defines that the model is a classifier.
gie-unique-id=1
classifier-threshold=0.2
```

13.3.2. Integrating a DetectNet_v2 model

See Exporting the model for more details on how to export a TLT model. Once the model has been generated two extra files are required:

1. Label file
2. DS configuration file

Label file

The label file is a text file, containing the names of the classes that the DetectNet_v2 model is trained to detect. The order in which the classes are listed here must match the order in which the model predicts the output. This order is derived from the order

the objects are instantiated in the **`cost_function_config`** field of the DetectNet_v2 experiment config file. Here's an example, of the DetectNet_v2 sample notebook file included with the TLT docker, the **`cost_function_config`** parameter looks like this:

```
cost_function_config {
  target_classes {
    name: "sheep"
    class_weight: 1.0
    coverage_foreground_weight: 0.05
    objectives {
      name: "cov"
      initial_weight: 1.0
      weight_target: 1.0
    }
    objectives {
      name: "bbox"
      initial_weight: 10.0
      weight_target: 1.0
    }
  }
  target_classes {
    name: "bottle"
    class_weight: 1.0
    coverage_foreground_weight: 0.05
    objectives {
      name: "cov"
      initial_weight: 1.0
      weight_target: 1.0
    }
    objectives {
      name: "bbox"
      initial_weight: 10.0
      weight_target: 1.0
    }
  }
  target_classes {
    name: "horse"
    class_weight: 1.0
    coverage_foreground_weight: 0.05
    objectives {
      name: "cov"
      initial_weight: 1.0
      weight_target: 1.0
    }
    objectives {
      name: "bbox"
      initial_weight: 10.0
      weight_target: 1.0
    }
  }
  ..
  ..
  target_classes {
    name: "boat"
    class_weight: 1.0
    coverage_foreground_weight: 0.05
    objectives {
      name: "cov"
      initial_weight: 1.0
      weight_target: 1.0
    }
    objectives {
      name: "bbox"
      initial_weight: 10.0
      weight_target: 1.0
    }
```

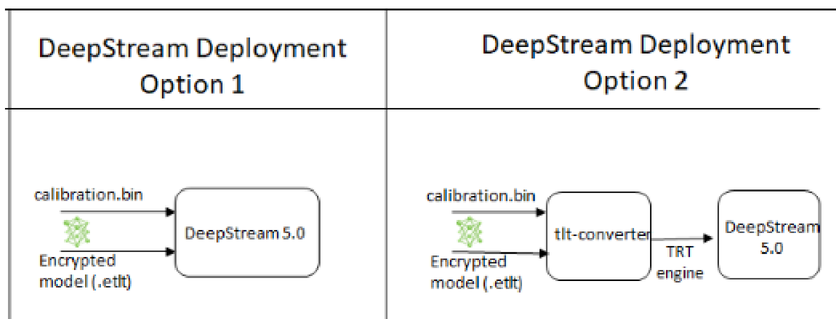
```
  }
  target_classes {
    name: "car"
    class_weight: 1.0
    coverage_foreground_weight: 0.05
    objectives {
      name: "cov"
      initial_weight: 1.0
      weight_target: 1.0
    }
    objectives {
      name: "bbox"
      initial_weight: 10.0
      weight_target: 1.0
    }
  }
  enable_autoweighting: False
  max_objective_weight: 0.9999
  min_objective_weight: 0.0001
}
```

Here's an example of the corresponding, detectnet_v2_labels.txt. The order in the labels.txt should match the order in the **`cost_function_config`**:

```
sheep
bottle
horse
..
..
boat
car
```

DeepStream configuration file

The detection model is typically used as a primary inference engine. It can also be used as a secondary inference engine. To run this model in the sample **`deepstream-app`**, you must modify the existing **`config_infer_primary.txt`** file to point to this model.



Option 1: Integrate the model (.etlt) directly in the DeepStream app.

For this option, users will need to add the following parameters in the configuration file. The **`int8-calib-file`** is only required for INT8 precision.

```
tlt-encoded-model=<TLT exported .etlt>
tlt-model-key=<Model export key>
int8-calib-file=<Calibration cache file>
```

The **`'tlt-encoded-model'`** parameter points to the exported model (.etlt) from TLT. The **`tlt-model-key`** is the encryption key used during model export.

Option 2: Integrate TensorRT engine file with DeepStream app.

Step 1: Generate TensorRT engine using tlt-converter. Detail instructions are provided in the Generating an engine using tlt-converter section above.

Step 2: Once the engine file is generated successfully, modify the following parameters to use this engine with DeepStream.

```
model-engine-file=<PATH to generated TensorRT engine>
```

All other parameters are common between the 2 approaches. Add the label file generated above using:

```
labelfile-path=<Classification labels>
```

For all the options, see the configuration file below. To learn about what all the parameters are used for, refer to DeepStream Development Guide .

```
[property]
gpu-id=0
# preprocessing parameters.
net-scale-factor=0.0039215697906911373
model-color-format=0

# model paths.
int8-calib-file=<Path to optional INT8 calibration cache>
labelfile-path=<Path to detectNet_v2_labels.txt>
tlt-encoded-model=<Path to DetectNet_v2 TLT model>
tlt-model-key=<Key to decrypt the model>
input-dims=c;h;w;0 # where c = number of channels, h = height of the model
 input, w = width of model input, 0: implies CHW format.
uff-input-blob-name=input_1
batch-size=4
## 0=FP32, 1=INT8, 2=FP16 mode
network-mode=0
num-detected-classes=3
interval=0
gie-unique-id=1
is-classifier=0
output-blob-names=output_cov/Sigmoid;output_bbox/BiasAdd
#enable_dbscan=0

[class-attrs-all]
threshold=0.2
group-threshold=1
## Set eps=0.7 and minBoxes for enable-dbscan=1
eps=0.2
#minBoxes=3
roi-top-offset=0
roi-bottom-offset=0
detected-min-w=0
detected-min-h=0
detected-max-w=0
detected-max-h=0
```

13.3.3. Integrating an SSD model

To run an SSD model in DeepStream, you need a label file and a DeepStream configuration file. In addition, you need to compile the TensorRT 7+ Open source software and SSD bounding box parser for DeepStream.

A DeepStream sample with documentation on how to run inference using the trained SSD models from TLT is provided on github at: https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps.

Prerequisites for SSD model

1. SSD requires batchTilePlugin. This plugin is available in the TensorRT open source repo, but not in TensorRT 7.0. Detailed instructions to build TensorRT OSS can be found in TensorRT Open Source Software (OSS).
2. SSD requires custom bounding box parsers that are not built-in inside the DeepStream SDK. The source code to build custom bounding box parsers for SSD is available in: https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps. The following instructions can be used to build bounding box parser:

   Step1: Install git-lfs (git >= 1.8.2)

   git-lfs are needed to support downloading model files >5MB.

   ```
   curl -s https://packagecloud.io/install/repositories/github/git-lfs/
   script.deb.sh | sudo bash
   sudo apt-get install git-lfs
   git lfs install
   ```

   Step 2: Download Source Code with HTTPS

   ```
   git clone -b release/tlt2.0
   https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps
   ```

   Step 3: Build

   ```
   export DS_SRC_PATH=/opt/nvidia/deepstream/deepstream-5.0
   // or Path for DS installation
   export CUDA_VER=10.2         // CUDA version, e.g. 10.2
   cd nvdsinfer_customparser_ssd_tlt
   make
   ```

   This generates **`libnvds_infercustomparser_ssd_tlt.so`** in the directory.

Label file

The label file is a text file, containing the names of the classes that the SSD model is trained to detect. The order in which the classes are listed here must match the order in which the model predicts the output. This order is derived from the order the objects are instantiated in the **`dataset_config`** field of the SSD experiment config file. For example, if the **`dataset_config`** is:

```
dataset_config {
  data_sources: {
    tfrecords_path: "/workspace/tlt-experiments/tfrecords/pascal_voc/
pascal_voc*"
    image_directory_path: "/workspace/tlt-experiments/data/VOCdevkit/VOC2012"
  }
  image_extension: "jpg"
  target_class_mapping {
    key: "car"
    value: "car"
  }
  target_class_mapping {
    key: "person"
    value: "person"
  }
  target_class_mapping {
    key: "bicycle"
    value: "bicycle"
  }
  validation_fold: 0
```

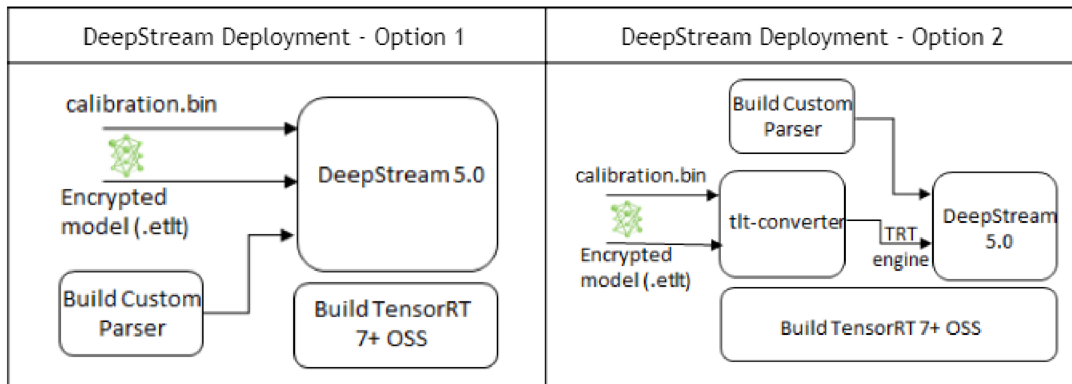
```
}
```

Here's an example of the corresponding classification_lables.txt file:

```
car
person
bicycle
```

DeepStream configuration file

The detection model is typically used as a primary inference engine. It can also be used as a secondary inference engine. To run this model in the sample **deepstream-app**, you must modify the existing **config_infer_primary.txt** file to point to this model as well as the custom parser.



Option 1: Integrate the model (.etlt) directly in the DeepStream app.

For this option, users will need to add the following parameters in the configuration file. The **int8-calib-file** is only required for INT8 precision.

```
tlt-encoded-model=<TLT exported .etlt>
tlt-model-key=<Model export key>
int8-calib-file=<Calibration cache file>
```

The **tlt-encoded-model** parameter points to the exported model (.etlt) from TLT. The **tlt-model-key** is the encryption key used during model export.

Option 2: Integrate TensorRT engine file with DeepStream app.

Step 1: Generate TensorRT engine using tlt-converter. See the Generating an engine using tlt-converter for detailed instructions.

Step 2: Once the engine file is generated successfully, modify the following parameters to use this engine with DeepStream. **model-engine-file=<PATH to generated TensorRT engine>**

All other parameters are common between the 2 approaches. Add the label file generated above using:

```
labelfile-path=<Classification labels>
```

For all the options, see the configuration file below. To learn about what all the parameters are used for, refer to DeepStream Development Guide .

```
[property]
gpu-id=0
net-scale-factor=1.0
offsets=103.939;116.779;123.68
model-color-format=1
```

```
labelfile-path=<Path to ssd_labels.txt>
tlt-encoded-model=<Path to SSD TLT model>
tlt-model-key=<Key to decrypt model>
uff-input-dims=3;384;1248;0
uff-input-blob-name=Input
batch-size=1
## 0=FP32, 1=INT8, 2=FP16 mode
network-mode=0
num-detected-classes=3
interval=0
gie-unique-id=1
is-classifier=0
#network-type=0
output-blob-names=BatchedNMS
parse-bbox-func-name=NvDsInferParseCustomSSDTLT
custom-lib-path=<Path to libnvds_infercustomparser_ssd_tlt.so>

[class-attrs-all]
threshold=0.3
roi-top-offset=0
roi-bottom-offset=0
detected-min-w=0
detected-min-h=0
detected-max-w=0
detected-max-h=0
```

13.3.4. Integrating a FasterRCNN model

To run a FasterRCNN model in DeepStream, you need a label file and a DeepStream configuration file. In addition, you need to compile the TensorRT 7+ Open source software and FasterRCNN bounding box parser for DeepStream.

A DeepStream sample with documentation on how to run inference using the trained FasterRCNN models from TLT is provided on github at: https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps.

Prerequisite for FasterRCNN model

1. FasterRCNN requires the cropAndResizePlugin and the proposalPlugin. This plugin is available in the TensorRT open source repo, but not in TensorRT 7.0. Detailed instructions to build TensorRT OSS can be found in TensorRT Open Source Software (OSS).
2. FasterRCNN requires custom bounding box parsers that are not built-in inside the DeepStream SDK. The source code to build custom bounding box parsers for FasterRCNN is available in https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps. The following instructions can be used to build bounding box parser:

Step1: Install git-lfs (git >= 1.8.2)

```
curl -s https://packagecloud.io/install/repositories/github/git-lfs/
script.deb.sh | sudo bash
sudo apt-get install git-lfs
git lfs install
```

Step 2: Download Source Code with SSH or HTTPS

```
git clone -b release/tlt2.0
https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps
```

Step 3: Build

```
export DS_SRC_PATH=/opt/nvidia/deepstream/deepstream-5.0
// or Path for DS installation
export CUDA_VER=10.2         // CUDA version, e.g. 10.2
cd nvdsinfer_customparser_frcnn_tlt
make
```

This generates **`libnvds_infercustomparser_frcnn_tlt.so`** in the directory.

Label file

The label file is a text file, containing the names of the classes that the FasterRCNN model is trained to detect. The order in which the classes are listed here must match the order in which the model predicts the output. This order is derived from the order the objects are instantiated in the **`target_class_mapping`** field of the FasterRCNN experiment specification file. During the training, TLT FasterRCNN will make all the class names in lower case and sort them in alphabetical order. For example, if the **`target_class_mapping`** label file is:

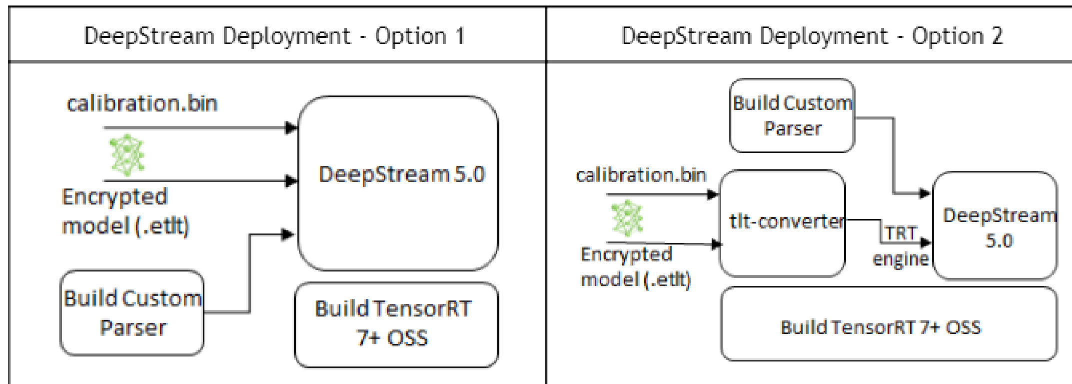
```
target_class_mapping {
    key: "car"
    value: "car"
  }
  target_class_mapping {
    key: "person"
    value: "person"
  }
  target_class_mapping {
    key: "bicycle"
    value: "bicycle"
  }
```

The actual class name list is **`bicycle, car, person`**. The example of the corresponding **`label_file_frcnn.txt`** file is:

```
bicycle
car
person
```

DeepStream configuration file

The detection model is typically used as a primary inference engine. It can also be used as a secondary inference engine. To run this model in the sample **`deepstream-app`**, you must modify the existing **`config_infer_primary.txt`** file to point to this model as well as the custom parser.



Option 1: Integrate the model (.etlt) directly in the DeepStream app.

For this option, users will need to add the following parameters in the configuration file. The **`int8-calib-file`** is only required for INT8 precision.

```
tlt-encoded-model=<TLT exported .etlt>
tlt-model-key=<Model export key>
int8-calib-file=<Calibration cache file>
```

The **`tlt-encoded-model`** parameter points to the exported model (.etlt) from TLT. The **`tlt-model-key`** is the encryption key used during model export.

Option 2: Integrate TensorRT engine file with DeepStream app.

Step 1: Generate TensorRT engine using tlt-converter. See the Generating an engine using tlt-converter section above for detailed instructions.

Step 2: Once the engine file is generated successfully, modify the following parameters to use this engine with DeepStream.

`model-engine-file=<PATH to generated TensorRT engine>`

All other parameters are common between the 2 approaches. To use the custom bounding box parser instead of the default parsers in DeepStream, modify the following parameters in [property] section of primary infer configuration file:

```
parse-bbox-func-name=NvDsInferParseCustomFrcnnUff
custom-lib-path=<PATH to libnvds_infercustomparser_frcnn_tlt.so>
```

Add the label file generated above using:

```
labelfile-path=<Classification labels>
```

For all the options, see the configuration file below. To learn about what all the parameters are used for, refer to DeepStream Development Guide .

Here's a sample config file, config_infer_primary.txt:

```
[property]
gpu-id=0
net-scale-factor=1.0
offsets=<image mean values as in the training spec file> # e.g.:
 103.939;116.779;123.68
model-color-format=1
labelfile-path=<Path to frcnn_labels.txt>
tlt-encoded-model=<Path to FasterRCNN model>
tlt-model-key=<Key to decrypt the model>
```

```
uff-input-dims=<c;h;w;0> # 3;272;480;0. Where c = number of channels, h = height
 of the model input, w = width of model input, 0: implies CHW format
uff-input-blob-name=<input_blob_name> # e.g.: input_1
batch-size=<batch size> e.g.: 1
## 0=FP32, 1=INT8, 2=FP16 mode
network-mode=0
num-detected-classes=<number of classes to detect(including background)> #
e.g.: 5
interval=0
gie-unique-id=1
is-classifier=0
#network-type=0
output-blob-names=<output_blob_names> e.g.:
dense_class_td/Softmax,dense_regress_td/BiasAdd, proposal
parse-bbox-func-name=NvDsInferParseCustomFrcnnTLT
custom-lib-path=<PATH to libnvds_infercustomparser_frcnn_tlt.so>

[class-attrs-all]
roi-top-offset=0
roi-bottom-offset=0
detected-min-w=0
detected-min-h=0
detected-max-w=0
detected-max-h=0
```

13.3.5. Integrating a YOLOv3 model

To run a YOLOv3 model in DeepStream, you need a label file and a DeepStream configuration file. In addition, you need to compile the TensorRT 7+ Open source software and YOLOv3 bounding box parser for DeepStream.

A DeepStream sample with documentation on how to run inference using the trained YOLOv3 models from TLT is provided on github at: https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps.

Prerequisite for YOLOv3 model

1. YOLOv3 requires batchTilePlugin, resizeNearestPlugin and batchedNMSPlugin. This plugin is available in the TensorRT open source repo, but not in TensorRT 7.0. Detailed instructions to build TensorRT OSS can be found in TensorRT Open Source Software (OSS).
2. YOLOv3 requires custom bounding box parsers that are not built-in inside the DeepStream SDK. The source code to build custom bounding box parsers for YOLOv3 is available in https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps. The following instructions can be used to build bounding box parser:

   Step1: Install git-lfs (git >= 1.8.2)

   git-lfs are needed to support downloading model files >5MB.

   ```
   curl -s
   https://packagecloud.io/install/repositories/github/git-lfs/
   script.deb.sh | sudo bash
   sudo apt-get install git-lfs
   git lfs install
   ```

   Step 2: Download Source Code with HTTPS

   ```
   git clone -b release/tlt2.0
   ```

```
https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps
```

Step 3: Build

```
export DS_SRC_PATH=/opt/nvidia/deepstream/deepstream-5.0
// or Path for DS installation
export CUDA_VER=10.2         // CUDA version, e.g. 10.2
cd nvdsinfer_customparser_yolov3_tlt
make
```

This will generate **`libnvds_infercustomparser_yolov3_tlt.so`** in the directory.

Label file

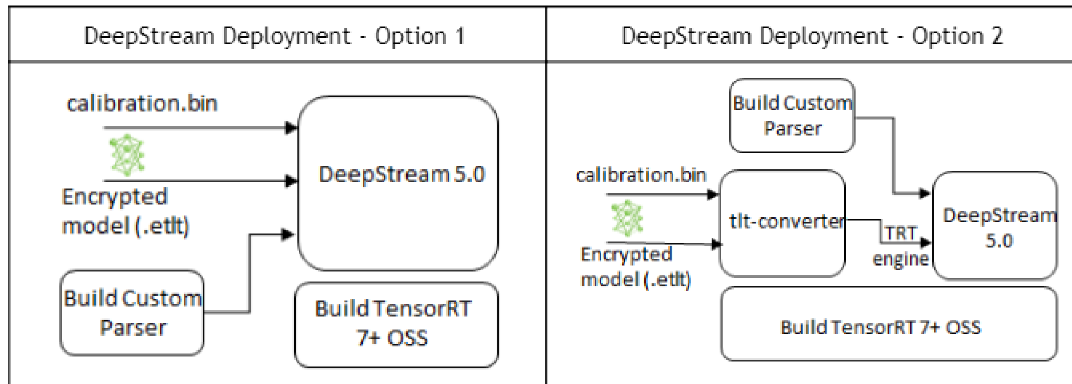
The label file is a text file, containing the names of the classes that the YOLOv3 model is trained to detect. The order in which the classes are listed here must match the order in which the model predicts the output. This order is derived from the order the objects are instantiated in the **`dataset_config`** field of the YOLOv3 experiment config file. For example, if the **`dataset_config`** is:

```
dataset_config {
  data_sources: {
    tfrecords_path: "/workspace/tlt-experiments/tfrecords/pascal_voc/
pascal_voc*"
    image_directory_path: "/workspace/tlt-experiments/data/VOCdevkit/VOC2012"
  }
  image_extension: "jpg"
  target_class_mapping {
    key: "car"
    value: "car"
  }
  target_class_mapping {
    key: "person"
    value: "person"
  }
  target_class_mapping {
    key: "bicycle"
    value: "bicycle"
  }
  validation_fold: 0
}
```

Here's an example of the corresponding yolov3_labels.txt file:

DeepStream configuration file

The detection model is typically used as a primary inference engine. It can also be used as a secondary inference engine. To run this model in the sample **`deepstream-app`**, you must modify the existing **`config_infer_primary.txt`** file to point to this model as well as the custom parser.



Option 1: Integrate the model (.etlt) directly in the DeepStream app.

For this option, users will need to add the following parameters in the configuration file. The **`int8-calib-file`** is only required for INT8 precision.

```
tlt-encoded-model=<TLT exported .etlt>
tlt-model-key=<Model export key>
int8-calib-file=<Calibration cache file>
```

The **`tlt-encoded-model`** parameter points to the exported model (.etlt) from TLT. The **`tlt-model-key`** is the encryption key used during model export.

Option 2: Integrate TensorRT engine file with DeepStream app.

Step 1: Generate TensorRT engine using tlt-converter. See the **Generating an engine using tlt-converter** section above for detailed instructions.

Step 2: Once the engine file is generated successfully, modify the following parameters to use this engine with DeepStream.

`model-engine-file=<PATH to generated TensorRT engine>`

All other parameters are common between the 2 approaches. To use the custom bounding box parser instead of the default parsers in DeepStream, modify the following parameters in [property] section of primary infer configuration file:

```
parse-bbox-func-name=NvDsInferParseCustomYOLO3TLT
custom-lib-path=<PATH to libnvds_infercustomparser_yolov3_tlt.so>
```

Add the label file generated above using:

```
labelfile-path=<Classification labels>
```

For all the options, see the configuration file below. To learn about what all the parameters are used for, refer to DeepStream Development Guide .

Here's a sample config file, **`pgie_yolov3_config.txt`**:

```
[property]
gpu-id=0
net-scale-factor=1.0
offsets=103.939;116.779;123.68
model-color-format=1
labelfile-path=<Path to yolov3_labels.txt>
tlt-encoded-model=<Path to YOLOV3 etlt model>
tlt-model-key=<Key to decrypt model>
uff-input-dims=3;384;1248;0
uff-input-blob-name=Input
batch-size=1
```

```
## 0=FP32, 1=INT8, 2=FP16 mode
network-mode=0
num-detected-classes=3
interval=0
gie-unique-id=1
is-classifier=0
#network-type=0
output-blob-names=BatchedNMS
parse-bbox-func-name=NvDsInferParseCustomYOLOV3TLT
custom-lib-path=<Path to libnvds_infercustomparser_yolov3_tlt.so>

[class-attrs-all]
threshold=0.3
roi-top-offset=0
roi-bottom-offset=0
detected-min-w=0
detected-min-h=0
detected-max-w=0
detected-max-h=0
```

13.3.6. Integrating a DSSD model

To run a DSSD model in DeepStream, you need a label file and a DeepStream configuration file. In addition, you need to compile the TensorRT 7+ Open source software and DSSD bounding box parser for DeepStream.

A DeepStream sample with documentation on how to run inference using the trained DSSD models from TLT is provided on github at: https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps.

Prerequisite for DSSD model

1. DSSD requires batchTilePlugin and NMS_TRT. This plugin is available in the TensorRT open source repo, but not in TensorRT 7.0. Detailed instructions to build TensorRT OSS can be found in TensorRT Open Source Software (OSS).
2. DSSD requires custom bounding box parsers that are not built-in inside the DeepStream SDK. The source code to build custom bounding box parsers for DSSD is available in https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps. The following instructions can be used to build bounding box parser:

   Step1: Install git-lfs (git >= 1.8.2)

   git-lfs are needed to support downloading model files >5MB.

   ```
   curl -s
   https://packagecloud.io/install/repositories/github/git-lfs/
   script.deb.sh | sudo bash
   sudo apt-get install git-lfs
   git lfs install
   ```

   Step 2: Download Source Code with HTTPS

   ```
   git clone -b release/tlt2.0
   https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps
   ```

   Step 3: Build

   ```
   export DS_SRC_PATH=/opt/nvidia/deepstream/deepstream-5.0
   // or Path for DS installation
   export CUDA_VER=10.2         // CUDA version, e.g. 10.2
   ```

```
cd nvdsinfer_customparser_dssd_tlt
make
```

This will generate **libnvds_infercustomparser_dssd_tlt.so** in the directory.

Label file

The label file is a text file, containing the names of the classes that the DSSD model is trained to detect. The order in which the classes are listed here must match the order in which the model predicts the output. This order is derived from the order the objects are instantiated in the **dataset_config** field of the DSSD experiment config file. For example, if the **dataset_config** is:

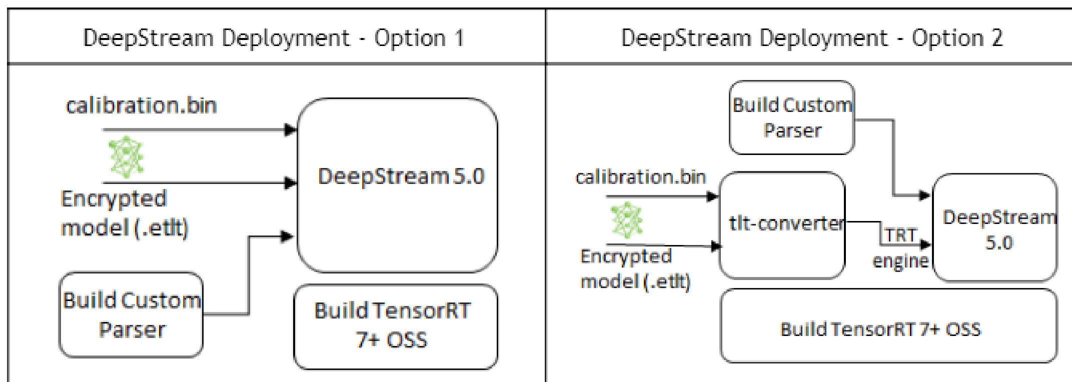
```
dataset_config {
  data_sources: {
    tfrecords_path: "/workspace/tlt-experiments/tfrecords/pascal_voc/
pascal_voc*"
    image_directory_path: "/workspace/tlt-experiments/data/VOCdevkit/VOC2012"
  }
  image_extension: "jpg"
  target_class_mapping {
    key: "car"
    value: "car"
  }
  target_class_mapping {
    key: "person"
    value: "person"
  }
  target_class_mapping {
    key: "bicycle"
    value: "bicycle"
  }
  validation_fold: 0
}
```

Here's an example of the corresponding dssd_labels.txt file:

```
car
person
bicycle
```

DeepStream configuration file

The detection model is typically used as a primary inference engine. It can also be used as a secondary inference engine. To run this model in the sample **deepstream-app**, you must modify the existing **config_infer_primary.txt** file to point to this model as well as the custom parser.



Option 1: Integrate the model (.etlt) directly in the DeepStream app.

For this option, users will need to add the following parameters in the configuration file. The **int8-calib-file** is only required for INT8 precision.

```
tlt-encoded-model=<TLT exported .etlt>
tlt-model-key=<Model export key>
int8-calib-file=<Calibration cache file>
```

The **tlt-encoded-model** parameter points to the exported model (.etlt) from TLT. The **tlt-model-key** is the encryption key used during model export.

Option 2: Integrate TensorRT engine file with DeepStream app.

Step 1: Generate TensorRT engine using tlt-converter. See the Generating an engine using tlt-converter section above for detailed instructions.

Step 2: Once the engine file is generated successfully, modify the following parameters to use this engine with DeepStream. **model-engine-file=<PATH to generated TensorRT engine>**

All other parameters are common between the 2 approaches. To use the custom bounding box parser instead of the default parsers in DeepStream, modify the following parameters in [property] section of primary infer configuration file:

```
parse-bbox-func-name=NvDsInferParseCustomDSSDTLT
custom-lib-path=<PATH to libnvds_infercustomparser_dssd_tlt.so>
```

Add the label file generated above using:

```
labelfile-path=<Classification labels>
```

For all the options, see the configuration file below. To learn about what all the parameters are used for, refer to DeepStream Development Guide .

```
[property]
gpu-id=0
net-scale-factor=1.0
offsets=103.939;116.779;123.68
model-color-format=1
labelfile-path=<Path to ssd_labels.txt>
tlt-encoded-model=<Path to DSSD TLT model>
tlt-model-key=<Key to decrypt model>
uff-input-dims=3;384;1248;0
uff-input-blob-name=Input
batch-size=1
## 0=FP32, 1=INT8, 2=FP16 mode
network-mode=0
num-detected-classes=3
interval=0
gie-unique-id=1
is-classifier=0
#network-type=0
output-blob-names=BatchedNMS
parse-bbox-func-name=NvDsInferParseCustomSSDTLT
custom-lib-path=<Path to libnvds_infercustomparser_dssd_tlt.so>

[class-attrs-all]
threshold=0.3
roi-top-offset=0
roi-bottom-offset=0
detected-min-w=0
detected-min-h=0
detected-max-w=0
detected-max-h=0
```

13.3.7. Integrating a RetinaNet model

To run a RetinaNet model in DeepStream, you need a label file and a DeepStream configuration file. In addition, you need to compile the TensorRT 7+ Open source software and RetinaNet bounding box parser for DeepStream.

A DeepStream sample with documentation on how to run inference using the trained DSSD models from TLT is provided on github at: https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps.

Prerequisite for RetinaNet model

1. RetinaNet requires batchTilePlugin and NMS_TRT. This plugin is available in the TensorRT open source repo, but not in TensorRT 7.0. Detailed instructions to build TensorRT OSS can be found in TensorRT Open Source Software (OSS).
2. RetinaNet requires custom bounding box parsers that are not built-in inside the DeepStream SDK. The source code to build custom bounding box parsers for DSSD is available in https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps. The following instructions can be used to build bounding box parser:

   Step1: Install git-lfs (git >= 1.8.2)

   git-lfs are needed to support downloading model files >5MB.

   ```
   curl -s
   https://packagecloud.io/install/repositories/github/git-lfs/
   script.deb.sh | sudo bash
   sudo apt-get install git-lfs
   git lfs install
   ```

   Step 2: Download Source Code with HTTPS

   ```
   git clone -b release/tlt2.0
   https://github.com/NVIDIA-AI-IOT/deepstream_tlt_apps
   ```

   Step 3: Build

   ```
   export DS_SRC_PATH=/opt/nvidia/deepstream/deepstream-5.0
   // or Path for DS installation
   export CUDA_VER=10.2         // CUDA version, e.g. 10.2
   cd nvdsinfer_customparser_retinanet_tlt
   make
   ```

   This will generate **`libnvds_infercustomparser_retinanet_tlt.so`** in the directory.

Label file

The label file is a text file, containing the names of the classes that the RetinaNet model is trained to detect. The order in which the classes are listed here must match the order in which the model predicts the output. This order is derived from the order the objects are instantiated in the **`dataset_config`** field of the RetinaNet experiment config file. For example, if the **`dataset_config`** is:

```
dataset_config {
  data_sources: {
```

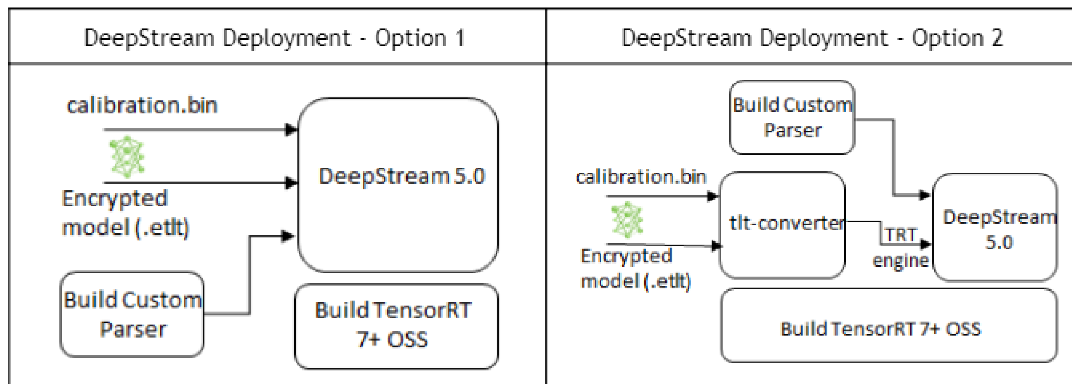
```
    tfrecords_path: "/workspace/tlt-experiments/tfrecords/pascal_voc/
pascal_voc*"
    image_directory_path: "/workspace/tlt-experiments/data/VOCdevkit/VOC2012"
  }
  image_extension: "jpg"
  target_class_mapping {
    key: "car"
    value: "car"
  }
  target_class_mapping {
    key: "person"
    value: "person"
  }
  target_class_mapping {
    key: "bicycle"
    value: "bicycle"
  }
  validation_fold: 0
}
```

Here's an example of the corresponding retinanet_labels.txt file:

```
car
person
bicycle
```

DeepStream configuration file

The detection model is typically used as a primary inference engine. It can also be used as a secondary inference engine. To run this model in the sample **deepstream-app**, you must modify the existing **config_infer_primary.txt** file to point to this model as well as the custom parser.



Option 1: Integrate the model (.etlt) directly in the DeepStream app.

For this option, users will need to add the following parameters in the configuration file. The **int8-calib-file** is only required for INT8 precision.

```
tlt-encoded-model=<TLT exported .etlt>
tlt-model-key=<Model export key>
int8-calib-file=<Calibration cache file>
```

The **tlt-encoded-model** parameter points to the exported model (.etlt) from TLT. The **tlt-model-key** is the encryption key used during model export.

Option 2: Integrate TensorRT engine file with DeepStream app.

Step 1: Generate TensorRT engine using tlt-converter. See the Generating an engine using tlt-converter section above for detailed instructions.

Step 2: Once the engine file is generated successfully, modify the following parameters to use this engine with DeepStream.

model-engine-file=<PATH to generated TensorRT engine>

All other parameters are common between the 2 approaches. To use the custom bounding box parser instead of the default parsers in DeepStream, modify the following parameters in [property] section of primary infer configuration file:

```
parse-bbox-func-name=NvDsInferParseCustomSSDTLT
custom-lib-path=<PATH to libnvds_infercustomparser_retinanet_tlt.so>
```

Add the label file generated above using:

```
labelfile-path=<Classification labels>
```

For all the options, see the configuration file below. To learn about what all the parameters are used for, refer to DeepStream Development Guide.

```
[property]
gpu-id=0
net-scale-factor=1.0
offsets=103.939;116.779;123.68
model-color-format=1
labelfile-path=<Path to retinanet_labels.txt>
tlt-encoded-model=<Path to RetinaNet TLT model>
tlt-model-key=<Key to decrypt model>
uff-input-dims=3;384;1248;0
uff-input-blob-name=Input
batch-size=1
## 0=FP32, 1=INT8, 2=FP16 mode
network-mode=0
num-detected-classes=3
interval=0
gie-unique-id=1
is-classifier=0
#network-type=0
output-blob-names=BatchedNMS
parse-bbox-func-name=NvDsInferParseCustomYOLOV3Uff
custom-lib-path=<Path to libnvds_infercustomparser_retinanet_tlt.so>

[class-attrs-all]
threshold=0.3
roi-top-offset=0
roi-bottom-offset=0
detected-min-w=0
detected-min-h=0
detected-max-w=0
detected-max-h=0
```

13.3.8. Integrating Purpose-built models

Integrating purpose-built models is very straightforward in DeepStream. The configuration file and label file for these models are provided in the SDK. These files can be used with the provided pruned model as well as your own trained model. For the provided pruned models, the config and label file should work out of the box. For your custom model, minor modification might be required.

Download and install DeepStream SDK. The installation instructions for DeepStream are provided in DeepStream development guide. The config files for the purpose-built models are located in:

```
/opt/nvidia/deepstream/deepstream-5:0/samples/configs/tlt_pretrained_models
```

'/opt/nvidia/deepstream' is the default DeepStream installation directory. This path will be different if you are installing in a different directory.

There are two sets of config files: main config files and inference config files. Main config file can call one or multiple inference config files depending on number of inferences. The table below shows the models being deployed by each config file.

Model(s)	Main DeepStream configuration	Inference configuration(s)	Label file(s)
TrafficCamNet	deepstream _app _source1 _trafficcamnet.txt	config_infer _primary _trafficcamnet.txt	labels_trafficnet.txt
PeopleNet	deepstream_app _source1 _peoplenet.txt	config_infer _primaryn _peoplenet.txt	labels_peoplenet.txt
DashCamNetVehicleMake NetVechicleTypeNet	deepstream_app_source1 _dashcamnet _vehiclemakenet _vehicletypenet.txt	config_infer _primary _dashcamnet.txt config_infer _secondary _vehiclemakenet.txt config_infer _secondary _vehicletypenet.txt	labels_dashcamnet.txt labels_vehiclemakenet.txt labels_vehicletypenet.txt
FaceDetect-IR	deepstream_app _source1_faceirnet.txt	config_infer _primary _faceirnet.txt	labels_faceirnet.txt

The main configuration file is to be used with **deepstream-app**, DeepStream reference application. In the **deepstream-app**, the primary detector will detect the objects and send the cropped frame to secondary classifiers. For more information on DeepStream reference application, refer to documentation.

The **deepstream_app_source1_dashcamnet_vehiclemakenet_vehicletypenet.txt** configures 3 models: DashCamNet as primary detector, and VehicleMakeNet and VehicleTypeNet as secondary classifiers. The classifier models are typically used after initial object detection. The other configuration files use single detection models.

Key Parameters in **config_infer_*.txt**:

```
tlt-model-key=<tlt_encode or TLT Key used during model export>
tlt-encoded-model=<Path to TLT model>
labelfile-path=<Path to label file>
int8-calib-file=<Path to optional INT8 calibration cache>
input-dims=<Inference resolution if different than provided>
num-detected-classes=<# of classes if different than default>
```

Run deepstream-app:

```
deepstream-app -c <DS config file>
```

13.3.9. Integrating a MaskRCNN model

Integrating a MaskRCNN model is very straightforward in DeepStream since DS 5.0 can support instance segmentation network type out of the box. The configuration file and label file for the model are provided in the SDK. These files can be used with the provided model as well as your own trained model. For the provided MaskRCNN model, the config and label file should work out of the box. For your custom model, minor modification might be required.

Download and install DeepStream SDK. The installation instructions for DeepStream are provided in DeepStream development guide. You need to follow the README under **/opt/nvidia/deepstream/deepstream-5.0/samples/configs/tlt_pretrained_models** to download the model and int8 calibration file. The config files for the Mask RCNN model are located in:

```
/opt/nvidia/deepstream/deepstream-5.0/samples/configs/tlt_pretrained_models
```

/opt/nvidia/deepstreamis the default DeepStream installation directory. This path will be different if you are installing in a different directory.

deepstream-app config file

deepstream-app config file is used by deepstream-app, see the deepstream-app config guide for more details, you need to enable the display-mask under osd group to see the mask visual view:

```
[osd]
enable=1
gpu-id=0
border-width=3
text-size=15
text-color=1;1;1;1;
text-bg-color=0.3;0.3;0.3;1
font=Serif
display-mask=1
display-bbox=0
display-text=0
```

Nvinfer config file

Nvinfer configure file is used in nvinfer plugin, see the Deepstream plugin manual for more details, following is key parameters to run the MaskRCNN model:

```
tlt-model-key=<tlt_encode or TLT Key used during model export>
tlt-encoded-model=<Path to TLT model>
parse-bbox-instance-mask-func-name=<post process parser name>
custom-lib-path=<path to post process parser lib>
network-type=3 ## 3 is for instance segmentation network
output-instance-mask=1
labelfile-path=<Path to label file>
int8-calib-file=<Path to optional INT8 calibration cache>
infer-dims=<Inference resolution if different than provided>
num-detected-classes=<# of classes if different than default>
```

Here's an example:

```
[property]
gpu-id=0
net-scale-factor=0.017507
offsets=123.675;116.280;103.53
model-color-format=0
tlt-model-key=<tlt_encode or TLT Key used during model export>
```

```
tlt-encoded-model=<Path to TLT model>
parse-bbox-instance-mask-func-name=<post process parser name>
custom-lib-path=<path to post process parser lib>
network-type=3 ## 3 is for instance segmentation network
labelfile-path=<Path to label file>
int8-calib-file=<Path to optional INT8 calibration cache>
infer-dims=<Inference resolution if different than provided>
num-detected-classes=<# of classes if different than default>
uff-input-blob-name=Input
batch-size=1
## 0=FP32, 1=INT8, 2=FP16 mode
network-mode=2
interval=0
gie-unique-id=1
#no cluster
## 0=Group Rectangles, 1=DBSCAN, 2=NMS, 3= DBSCAN+NMS Hybrid, 4 = None(No
 clustering)
## MRCNN supports only cluster-mode=4; Clustering is done by the model itself
cluster-mode=4
output-instance-mask=1

[class-attrs-all]
pre-cluster-threshold=0.8
```

Label file

If the COCO annotation file has the following in “categories”:

```
[{'supercategory': 'person', 'id': 1, 'name': 'person'},
 {'supercategory': 'car', 'id': 2, 'name': 'car'}]
```

Then, the corresponding maskrcnn_labels.txt file is:

```
BG
person
car
```

Run deepstream-app:

```
deepstream-app -c <deepstream-app config file>
```

Also you can use deepstream-mrcnn-test to run the Mask RCNN model, see the README under: **`$DS_TOP/source/apps/sample_apps/deepstream-mrcnn-test/`**

Notice

THE INFORMATION IN THIS GUIDE AND ALL OTHER INFORMATION CONTAINED IN NVIDIA DOCUMENTATION REFERENCED IN THIS GUIDE IS PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE INFORMATION FOR THE PRODUCT, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

THE NVIDIA PRODUCT DESCRIBED IN THIS GUIDE IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN CONNECTION WITH THE DESIGN, CONSTRUCTION, MAINTENANCE, AND/OR OPERATION OF ANY SYSTEM WHERE THE USE OR A FAILURE OF SUCH SYSTEM COULD RESULT IN A SITUATION THAT THREATENS THE SAFETY OF HUMAN LIFE OR SEVERE PHYSICAL HARM OR PROPERTY DAMAGE (INCLUDING, FOR EXAMPLE, USE IN CONNECTION WITH ANY NUCLEAR, AVIONICS, LIFE SUPPORT OR OTHER LIFE CRITICAL APPLICATION). NVIDIA EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR SUCH HIGH RISK USES. NVIDIA SHALL NOT BE LIABLE TO CUSTOMER OR ANY THIRD PARTY, IN WHOLE OR IN PART, FOR ANY CLAIMS OR DAMAGES ARISING FROM SUCH HIGH RISK USES.

NVIDIA makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/ or requirements beyond those contained in this guide. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this guide, or (ii) customer product designs.

Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by NVIDIA under this guide. Reproduction of information in this guide is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, cuDNN, cuFFT, cuSPARSE, DIGITS, DGX, DGX-1, DGX Station, GRID, Jetson, Kepler, NVIDIA GPU Cloud, Maxwell, NCCL, NVLink, Pascal, Tegra, TensorRT, Tesla and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2020 NVIDIA Corporation. All rights reserved.

www.nvidia.com