Object Detection GTI Custom Model Training with PyTorch Model Development Kit

GTI 5801

User Guide

Version 1.1

Contents

About this Document

This document contains information that is proprietary and confidential to Gyrfalcon Technology Inc. and is intended for the specific use of the recipient, for the purpose of evaluating or using Gyrfalcon Technology's products and/or IP. This document is provided to the recipient with the expressed understanding that the recipient will not divulge its contents to other parties or otherwise misappropriate the information contained herein.

Contact Gyrfalcon
Silicon Valley HQ Gyrfalcon
Technology, Inc.
1900 McCarthy Boulevard, Suite 412
Milpitas, CA 95035 USA

1. Release Notes

Version	Chip	Supported Network	Changes	Date
1.0	GTI 5801	GNetDet (supported input resolution 448x448x3RGB)	Initial release	01/14/2021
1.1	GTI 5801	GNetDet (supported input resolution 448x448x3RGB)	• Updated the script to read the checkpoint from self.checkpoint_path in the config.py script instead of manually copying the checkpoint after each training step.	03/16/2021

2. Introduction

This guide provides object detection model training and conversion from a pretrained network based on example datasets. This document is a supplement to an existing PyTorch Model Development Kit (MDK) developer's guide. Users are assumed to be proficient in PyTorch and floating-point model development. The MDK is used to port part or all of the neural networks of interest onto GTI device compatible format with comparable performance to the original floating-point model. This MDK can only be used for fine-tuning from a pre-trained floating-point model since training from scratch is considerably hard.

The GTI Lightspeeur® series are low-power and high-performance AI accelerator devices. This reference model training is targeted to GTI 5801 chip. The final model is a GTI customized detection model fully accelerated by GTI 5801 chip without any CPU dependency.

2.1 The Network Structure

GTI proposes GNetDet as a custom object detection model to highly utilize GTI chip architecture and eliminate CPU dependency. GNetDet is designed with low power consumption and high speed performance in mind.

The whole GNetDet network contains 16 convolutional layers. The first 15 layers can be treated as feature extraction layers, and the 16th is the detection head. The detection head layer will output 10+num_classes channels, first 10 of which are used to determine whether there is an object: if there is an object, it will output the coordinates of the corresponding bounding box. The later num_classes channels are used as classifiers: They will predict the probability of each class. For example, assume we are using a dataset with 20 classes, the last layer will output 30 channels, the first 10 channels to identify the objects and bounding box information, and the last 20 channels to provide the probability of each class. The network structure is provided in Table 1.

Table 1. GNetDet network structure

Layer	Image size	Input Channels	Output Channels	Sub-Layer	Pooling	Upsampling
1	448	3	64	1	1	0
2	224	64	64	1	1	0
3	112	64	128	2	1	0
4	56	128	256	3	1	0
5	28	256	256	3	1	0
6	14	256	256	5	0	0
7	14	256	10+num_classes	1	0	0

3. GTI MDK Support on Linux OS

3.1 Overview

Gyrfalcon Technology Inc provides hardware and software for application development targeted to AI and machine learning.

Equipment to be supplied by users:

- Linux (Ubuntu 16.04, 64-bit version)
- Intel i5 or better processor (i7 preferred) 3.0 GHz or faster
- 8 GB or more RAM
- At least one USB 3.0 port

Third-party software:

- OpenCV 3.2 or higher
- Python 3.6.8
- PyTorch 1.2.0 or higher
- numpy 1.15
- CUDA 10.0
- cuDNN 7.6.5

There may be other Python dependencies in some of the optional sample workflow instructions. To install missing Python dependencies, simply use pip.

To verify on-chip inferencing results GTI libraries are required. There is a compatibility issue between SDK and MDK versions. SDK 5.x requires a special model structure. The default setting to run on-chip inferencing is based on the latest SDK v5.x library usage.

To switch between SDK 5.x and SDK 4.x libraries:

- Modify /gti/chip/driver.py SDK_VERSION setting
  - replace SDK_VERSION=5 with SDK_VERSION=4
- Navigate to /gti/chip and link the libGTILibary.so with SDK4.x version of the GTI library. An example is shown below:

```
ln -sf libGTILibrary.so.4.5.x  libGTILibrary.so
```

3.2 Environment Setup

For GTI PyTorch MDK installation, follow PyTorch MDK developer's guide.

3.3 File/Folder Structure

The GTI training package is delivered in two tarball files that you decompress to your local file system. Following are the unpacking instructions for Ubuntu 16.04 (Linux):

```
tar zxvf GTI5801_GNetDet448_PyTorchMDKTraining_v1.1.tar.xz
```

The resulting model development kit includes the components listed in Table 2:

Table 2. Folder structure

Location	Description
/ (project root)	contains example scripts for various tasks, e.g., training, evaluation, and model conversion
checkpoints/	reserved for saved checkpoints and logs
data/	reserved for saving text label files and test images
gti/	core GTI library package
log/	training logs
model/	reserved for model conversion output
docs/	user guide and other relevant content
net/	network script and relu_cap configuration file
LICENSE	license documents
scripts/	tool scripts folder
scripts/before_convert.py	script for preparing PyTorch checkpoint before conversion
scripts/display_by_camera_or_video.py	test on chip model using camera or video file as input
xml_2_txt.py	a script to parse a VOC style xml file to training label file
chip_eval.py	on chip mAP value testing
config.py	configuration script
convert_to_chip.py	a script to convert a checkpoint to the GTI chip compatible format
eval_voc.py	a script to verify the model accuracy in the PyTorch environment.

`predict.py`	eval_voc.py will call this function to infer each image
`train.py`	training script

4. Model Development Workflow

The following steps are recommended to properly use the PyTorch MDK:

1. Train a full floating-point model.
2. Fine-tune resulting model from (1) with GTI custom quantized convolutional layers.
3. Fine-tune resulting model from (2) with GTI custom quantized activation layers.
4. Perform model “fusion” and fine-tune resulting model from (3).
5. Convert the resulting model from (4) into GTI format and evaluate it on the chip.
6. Incrementally improve, as needed.

Layers that are accelerated on GTI devices require fine-tuning with custom quantized convolutional and activation layers.

The following is an example methodology that has worked well consistently in experiments. The approach is to leverage the quantization-aware training, the idea of simulating quantization effects during training. Note that these steps must be performed in order.

4.1 Data Preparation

The GTI reference model for object detection is trained with a combined dataset of VOC0712 dataset and MSCOCO dataset for classifiers including cars, bicycles, motorbikes, buses, people, birds and so on, total 21 classes.

All classes used for object detection are listed in Table 3:

Table 3. The classes used for object detection training

1. background	8. car	15. motorbike
2. aeroplane	9. cat	16. person
3. bicycle	10. chair	17. potted plant
4. bird	11. cow	18. sheep
5. boat	12. dining table	19. sofa
6. bottle	13. dog	20. train
7. bus	14. horse	21. TV monitor

To pre-process the dataset:

1) Download and decompress the VOC0712 dataset from the official website.
2) Make sure that the path in #xml_2_txt section of config.py points to the Annotation directory of the

dataset.

```
self.data_txt_file = <path_to_output_file>
self.label_paths = <path_to_Annotation_folder>
```

3) Make sure the "self.VOC_CLASSES" in "config.py" matches exactly all the classes that are in the training dataset.
4) Run the following command to convert xml annotation files to one single .txt label file.

```
python3 xml_2_txt.py
```

5) Repeat step 2) and 3) to generate .txt files for each sub-dataset of VOC0712 dataset.
6) Download and decompress the MSCOCO dataset from the official website. Arrange the dataset into VOC format.
7) Repeat step 2) and step 3) to generate files for the MSCOCO dataset.
8) Repeat step 2) and step 3) to generate files for the test dataset.
9) Now you should have one .txt label file for each Annotation folder from each dataset.
10) Make sure the "self.train_file" and “self.test_file” parameter in "config.py" matches all the classes that are in the training dataset. Note that "self.train_file" can be a list.
11) Copy/Move all training and testing images into one folder. Set "self.file_root" to this folder.

Example content of the generated label file:

```
006280.jpg 294 96 434 238 7
006293.jpg 163 72 244 172 19
005332.jpg 224 151 262 194 1 336 126 369 188 13
006552.jpg 150 22 436 277 14 255 58 320 211 14 225 156 257 197 14 85 160 141 255 14 1 67 122
333 14
008289.jpg 1 84 423 375 6
```

Note: Custom dataset can be used for training in similar ways, make sure that the data and folder structures are in VOC format.

4.2 Overview of the Common Parameters in the "config.py" Script

```
self.VOC_CLASSES              # The name of all classes in the dataset
self.file_root                # The path to all images of training and
                              testing dataset
self.train_file               # The list of training label files
self.test_file                # The path to testing label files
self.checkpoint_path          # The path to the checkpoint
self.learning_rate            # Initial learning rate
self.num_epochs               # The number of training epochs
self.batch_size = 5           # The batch size of training image
self.edit_gain = False        # Used during model fusion
self.batch_norm               # Currently not supported
self.quant_w = True           # Turns on/off the quantized convolution
                              training
self.quant_r = True           # Turns on/off the quantized activation
s                             training
elf.quant_31 = True           # Checks if the relu_cap is in a range of
                              [0, 31] for each activation layer
self.cal=False                # Turns off/on relu_cap calculation

self.Display                  # Camera or video file for testing
self.mp4_path                 # Testing video path


self.chip                     # Chip type, 5801 for this package
self.dat_json_448             # A file describing the hyperparameters of
                              each layer of the model
self.model_json_448           # A file describing the overall input and
                              output model of the GTI chip
self.out_model                # Path to save the converted model
self.dump_mode                # Toggle on to dump the middle layer result
                              during conversion, generally False
self.best_ckpt                # Best checkpoint path used to convert
self.image_size               # The input image size


self.data_txt_file            # The of output path of xml_2_txt.py
self.label_paths              # The Annotation file path of xml_2_txt.py
```

5. Training

The purpose of the training is to create a quantized model with almost the same accuracy as the original floating point model. The final converted model should also provide the same accuracy on the chip, as on the CPU/GPU.
An example of a JSON file ("gti/json/5801_gnetdet_model_448.json") is shown below.


```
{
	"layer": [
	{
		"data file": null,
		"data offset": null,
		"input format": "CS_BGR24_PLANAR",
		"name": "image reader",
		"operation": "IMAGEREADER",
		"output channels": 3,
		"output format": "CS_BGR24_PLANAR",
		"output height": 448,
		"output width": 448
	},
	{
		"data file": "",
		"data offset": 0,
		"device": {
		"chip": "5801",
		"name": null,
		"type": 0
		},
		"mode": 0,
		"name": "cnn",
		"operation": "GTICNN",
		"output channels": 256,
		"output height": 14,
		"output width": 14
	}
	],
	"name": "gti_5801_gnetdet"
}
```


5.1 Training a Full Floating-Point Model

Before commencing the training operations, make sure the label files have been generated, as stated in section 4.1 Data Preparation.

GTI provides reference pre-trained checkpoints at the checkpoint/448 folder. The training script will automatically load checkpoint from `checkpoint/best/best.pth`. To start the training from any of the pretrained checkpoints update the path (`self.checkpoint_path`) in the config.py script to load and use this checkpoint for further training.

Set the following parameters in config.py:

```
self.edit_gain = False, self.quant_w = False, self.quant_r = False,
self.quant_31 = False, self.cal=False
```

To train a floating point PyTorch model fine-tuning from the pre-trained checkpoint, run the following command:

```
python3 train.py
```

To evaluate the trained model:

```
python3 eval_voc.py
```

The best generated model (minimum loss) will be saved in `checkpoint/step1/best.pth`, and the last generated model will be saved in `checkpoint/step1/last.pth`.
Note: Set the learning rate by configuring `self.learning_rate`, we suggest a value between 10^{-4} to 10^{-5} for this step.
Note: Training with the next step will overwrite the checkpoints. We suggest you copy and store them to another place.

5.2 Fine-Tuning with Quantized Convolutional Layers

The training script will automatically load checkpoint from `checkpoint/best/best.pth`. Put the pre-trained checkpoint from section 5.1 Training a Full Floating-Point Model or 5.2 Fine-Tuning with Quantized Convolutional Layers under this path before training.

Set the following parameters in config.py, as follows:

```
self.edit_gain = False, self.quant_w = True, self.quant_r = False,
self.quant_31 = False, self.cal=False
```

Run the following command to start the fixed point training on GPU for fine-tuning with quantized convolutional layers:

```
python3 train.py
```

To evaluate the trained model:

```
python3 eval_voc.py
```

The best generated model (minimum loss) will be saved in `checkpoint/step2/best.pth`, and the last

generated model will be saved in `checkpoint/step2/last.pth`.
Note: Set the learning rate by configuring `self.learning_rate`, we suggest a value between 10^{-4} to 10^{-5} for this step.
Note: Training with the next step will overwrite the checkpoints. We suggest you copy and store them to another place.

5.3 Setting the Values of ReLU

The MDK provides GTI custom quantized activation layers with toggleable quantization effects built-in, which may be used as replacements for regular PyTorch layers when constructing models. Before fine-tuning with quantized activation layers, the ranges of the activation layer outputs need to be calibrated.

Set the following parameters in config.py, as follows:

```
self.edit_gain = False, self.quant_w = True, self.quant_r = False,
self.quant_31 = False, self.cal=True
self.num_epochs = 1 (suggested)
```

The training script will automatically load checkpoint from `checkpoint/best/best.pth`. Put the pre-trained checkpoint from section 5.2 Fine-Tuning with Quantized Convolutional Layers under this path before training.

Run the following command to set ReLU values:

```
python3 train.py
```

The best generated model (minimum loss) will be saved in `checkpoint/step2/best.pth`, and the last generated model will be saved in `checkpoint/step2/last.pth`.
Note: Set the learning rate by configuring `self.learning_rate`, we suggest a value between 10^{-5} to 10^{-6} for this step.
Note: Training with the next step will overwrite the checkpoints. We suggest you copy and store them to another place.

5.4 Fine-Tuning with Quantized Activation Layers

Use the pre-trained checkpoint from section 5.3 Setting the Values of ReLU or 5.4 Fine-Tuning with Quantized Activation Layers (update the path at self.checkpoint_path in config.py before training).

Set the following parameters in config.py, as follows:

```
self.edit_gain = False, self.quant_w = True, self.quant_r = True,
self.quant_31 = False, self.cal=False
```

Run the following command to start the fixed point training on GPU for fine-tuning with quantized activation

layers:

```
python3 train.py
```

To evaluate the trained model:

```
python3 eval_voc.py
```

The best generated model (minimum loss) will be saved in `checkpoint/step3/best.pth`, and the last generated model will be saved in `checkpoint/step3/last.pth`.
Note: Set the learning rate by configuring `self.learning_rate`, we suggest a value lower than 10^{-5} for this step.
Note: Training with the next step will overwrite the checkpoints. We suggest you copy and store them to another place.
Note: In this step, training will take more memory than previous steps, consider training with a lower batch_size by changing self.batch_size in config.py, if there is an OOM (Out of Memory) issue.

5.5 Model Fusion

Use the pre-trained checkpoint from 5.4 Fine-Tuning with Quantized Activation Layers or 5.5 Model Fusion (update the path at self.checkpoint_path in config.py before training).

If continue training using checkpoint from last step, set the following parameters in config.py, as follows:

```
self.edit_gain = True, self.quant_w = True, self.quant_r = True,
self.quant_31 = True, self.cal=False
```

If continue training using checkpoint from current step, set the following parameters in config.py, as follows:

```
self.edit_gain = False, self.quant_w = True, self.quant_r = True,
self.quant_31 = True, self.cal=False
```

Run the following command to train the network with batch-norm incorporated into weights and activations (model fusion):

```
python3 train.py
```

To evaluate the trained model:

```
python3 eval_voc.py
```

The best generated model (minimum loss) will be saved in `checkpoint/step4/best.pth`, and the last

generated model will be saved in `checkpoint/step4/last.pth`.
Note: Set the learning rate by configuring `self.learning_rate`, we suggest a value between 10^{-5} to 10^{-6} for this step.
Note: Training with the next step will overwrite the checkpoints. We suggest you copy and store them to another place.

5.6 Converting the Model

The JSON file is required for conversion, see section Appendix A. JSON File Format, in PyTorch MDK User's Guide, for more details.

The conversion script will automatically load the checkpoint from the `self.best_ckpt` path set in config.py.

Run the following to convert the PyTorch checkpoint into GTI chip compatible format:

```
python3 convert_to_chip.py
```

The converted *.model will be saved at `self.out_model`.

6. Inferencing

The on chip inference can be conducted with real-time camera input or an offline pre-recorded video.

6.1 Chip Inferencing

For camera inference, move to scripts/ directory, set `self.Display='camera' in config.py`, then run the following command:

```
python3 display_by_camera_or_video.py
```

For single video inference, move to scripts/ directory, set `self.Display='mp4'` and set the path to video by setting `self.mp4_path in config.py`, then run the following command:

```
python3 display_by_camera_or_video.py
```

For mAP testing, run the following command:

```
python3 chip_eval.py
```

7. Incremental Improvements

If on-chip results are unsatisfactory, there may be ways to diagnose problems and make improvements. See the section "Design Considerations" of PyTorch MDK developer’s guide, to eliminate common pitfalls that may have been missed in the model development cycle.

8. Enabling Intermediate Layer Output

The output of the model has been only from the last sublayer of the last major layer. MDK can enable or disable the output of intermediate layers by configuring the network*.json file. Intermediate layer means the layer before the last sub-layer of the last major layer. The MDK supports two different modes per major layer.

- Learning Mode : It enables/disables the output of all sublayers of this major layer.
- Last_layer_out Mode : It enables/disables the output of the last sublayer of this major layer.

These two modes cannot be used together. If learning mode is enabled, "Last_layer_out" mode cannot be enabled in any other major layers. Also, if "Last_layer_out" is enabled, learning mode cannot be enabled in any other major layers.

Consider the following, as an example:

- If learning mode is enabled in one of the major layers, "Last_layer_out" has to be false in all the major layers. However, you can enable learning mode in other major layers.
- If "Last_layer_out" mode is enabled in one of the major layers, learning mode has to be false in all the major layers. However, you can enable "Last_layer_out" mode in other major layers.

```
Snippet of *_dat.json
   "learning": true,# enable / disable output of all sublayers in this major layer
   "last_layer_out": false, # enable / disable output of last sublayer of this major layer
```