gyrfalcon technology

Model Development Kit

TensorFlow - GTI 2801 Only

User Guide

Version 2.1.3.1

Contents

1. 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 208
Milpitas, CA 95035 USA

2. Introduction

The GTI Lightspeeur® series are low-power and high-performance AI accelerator devices. The GTI Model Development Kit (MDK) is a tool for users to develop and train models that are compatible with GTI devices - GTI 2801.

This MDK is developed around TensorFlow. Users are assumed to be proficient in TensorFlow and floating-point model development. This MDK is used to port part or all of the neural networks of interest onto GTI devices with comparable performance to the original floating-point model. Although using this MDK to train a model from scratch is possible, fine-tuning from a pre-trained floating-point model is highly recommended.

3. Release Notes

Version	Chip	Supported Network	Changes	Date
V1.0	2801	VGG	Initial 2801 support	01/23/19
v1.2.1	2801	VGG	1. Added multi-chip support for 2801 2. Require --chip argument in scripts to specify which chip 3. Require target_chip argument in quantization functions to specify quantization schemes 4. "qbit" is renamed "mask bits"	02/20/19
v1.2.2	2801	VGG	1. For inference, remove dependency of GTI SDK for on-host layers (e.g. fully-connected layers). Users now have the flexibility to implement and deploy on-host layers to platforms and frameworks of their choice. An example is provided, which implements on-host layers in native TensorFlow.	03/20/19
v2.1.0.1	2801	VGG	1. Simplify command line flags for quantization 2. Add automatic ping-pong buffer address calculation and removed ping-pong address in the json templates.	05/10/19
v2.1.2.1	2801	VGG	1. Change rounding scheme in quantize.py to better emulate chip rounding. 2. Minor fix in VGG16 fusion	06/14/19
v2.1.3.1	2801	VGG	1. Support multi-gpu training 2. Add zero-padding upsampling mode 3. Updated conversion tool	08/19/19

4. GTI MDK Support on Linux OS

4.1 Overview

Gyrfalcon Technology Inc provides hardware and software for application development targeted to AI and machine learning. The software development kit (SDK) includes drivers, libraries and source code for AI learning and inferencing. The MDK package provides tools and reference examples to train and validate your models optimized for GTI Plai Plug. Once the model is successfully generated, it can then be used in a reference application included in the SDK package for verification purposes or with MDK inferencing scripts.

Note that GTI Plai Plug is not needed to perform training and conversion. Supported SDK version is 4.4.1.x.

This MDK has been tested with the versions of dependencies listed below:

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:

- CUDA 9.0.176
- CUDNN 7.3.1
- Python 3.6
- tensorflow-gpu 1.12
- numpy 1.15
- tqdm 4.28 (for training progress visualizations)

To verify on-chip inferencing results:

- GTI SDK 4.4.1.x (precompiled library included in MDK)

pip can be used to install any missing Python dependencies. Anaconda/Miniconda and/or Docker are recommended to manage packages and environments. GTI SDK 4.2 and above can be used for evaluating the model generated by this tool. However, the libraries and drivers from the SDK package are included in the MDK, to enable running the on chip inference with python scripts, such as chip_infer.py.

4.2. MDK File/Folder Structure

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

```
tar zxvf Tensorflow_MDK_Vxx_xxxxxx.tgz
```

The resulting model development kit includes the following components:

Location	**Description**
`/ (project root)`	contains example scripts for various tasks, e.g., training, evaluation, and model conversion
`gti/`	core GTI library package
`checkpoints/`	reserved for saved checkpoints
`data/`	reserved for datasets
`nets/`	folder for conversion configurations and additional files generated during model conversion process
`addon/`	addon examples
`scripts/`	scripts to perform auxiliary tasks (e.g. porting pretrained models)

5. Design Considerations

The following are important but easy-to-miss details in the model development process which may help improve your design.

1. **Chip input range specification**

GTI devices 5-bit and 10-bit quantization of activation outputs in the ranges of [0, 31] and [0, 1023] respectively. To leverage 10-bit activation, the number of channels must be reduced by half of that of the equivalent 5-bit model.

For a 5-bit activation model, if you have input ranges that are not [0, 31], e.g., [-1, 1], [0, 1], [0, 255], you may need to rescale your inputs to conform to this specification. The MDK provides an example of transforming [0, 255] inputs to [0, 31]. This is done as: `clip(((x>>2)+1)>>1), 0, 31)`, where `>>` means bitwise right shift (i.e., right shift 2 bits, add 1, right shift 1 bit).

For a 10-bit activation model, **"`--ten_bit_act`"** flag needs to be switched on when running the example command line scripts. Note that not all provided example models can support 10-bit activations due to constraints on GTI devices. An example model that uses 10-bit activation is provided in `gti/model/tenbit.py`.

2. **Imbalance of weight and bias**

GTI chip leverages quantization parameter sharing to reduce memory footprint. A common problem may arise when `max(|B|) >> max(|W|)` (i.e., maximum absolute value of biases is much larger than that of weights), or when `max(|B|) << max(|W|)` (i.e. maximum absolute value of biases is much smaller than that of weights).

When you notice that a quantized model achieves high accuracy on the PC but not on chip, this may be the problem. Quantization ranges are determined by the values of W and B, so if one is much larger than the other, then one quantization range is heavily restricted by the other, leading to reduced accuracy. One remedy is to impose constraints on one of the variables. The MDK provides an example to constrain biases to the range `[-max|W|, max|W|]`, making sure that biases will not "explode" beyond weights. Other methods (e.g., `max norm`, `L2 regularization`) may also work.

To see if your model has this issue, you may turn on logging in gti/converter.py by setting environment variable:

```
GTI_DEBUG_CONVERSION=True python convert_to_chip.py [arguments]
```

This will log each layer `max|W|` and `max|B|` to console during the conversion process.

3. Extreme activation output ranges

GTI chip supports activation in the range [0, 31] or [0, 1023]. When calibrating activation output ranges, if the range in floating-point is extremely large (e.g. 1e3) or extremely small (e.g. 1e-6), it might become a cause for concern. The vast activation range might intricately affect the weights and biases of the current and following layers. At inference time, the activation range parameters have to be mathematically transformed and "fused" into weights and biases, which might result in aforementioned imbalance of weight and bias. One possible remedy is to use normalization techniques, such as batch normalization, to normalize layer outputs. Model fusion outlined in 6.4 simulates the actual model weights and biases at inference time. To more accurately condition the weights and biases for quantized inference, the model would have to be fine-tuned after fusion. If model fine-tuning does not converge after fusion, then the same would likely happen on chip.

4. Tensor dimension orders

Default dimension orders for TensorFlow tensors are different from other popular frameworks (PyTorch/Caffe) and GTI chip dimension orders. When converting TensorFlow weights to the chip, the orders need to be changed. The following table shows the differences.

Tensor Type	TensorFlow (Default)	GTI Chip/PyTorch/Caffe
Convolutional Filters	[height, width, input channels, output channels]	[output channels, input channels, height, width]
Fully Connected Weights	[input size, output size]	[output size, input size]
Layer Inputs/Outputs	[batch, height, width, channel]	[batch, channel, height, width]

TensorFlow allows users to change default dimension orders, but due to divided and easy-to-miss APIs (e.g., some TensorFlow APIs use "channels_first", while others use "NCHW"), it's recommended to use default ordering in TensorFlow, and leave the handling logic to the GTI chip conversion tool, which assumes that models have been saved using default TensorFlow ordering.

When building hybrid models (partial model on-chip and partial model in TensorFlow on PC), if you like to feed chip output into TensorFlow layers, special care is required. One may need to

transpose and/or reshape the chip output tensor to conform with TensorFlow ordering. Vice-versa also applies when feeding TensorFlow tensors into chip—reshaping and transposing may be also required. The MDK provides examples (e.g. chip_infer.py, gti/chip/driver.py) on how to preprocess data when feeding it to chip for inference.

6. Model Development Workflow

The following steps are recommended to properly use the TensorFlow 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 resulting model from (3) into GTI format and evaluate result 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 “quantization-aware training,” the idea of simulating quantization effects during training. Note that these steps must be performed in order.

The multi-GPU training support is also available, simply specify an input to the following script.

```
python train.py \
      --num_gpus NUM_GPUS
```

6.1. Training a Full Floating-Point Model

First, construct a model and train in floating-point – either from scratch, or by leveraging a pre-trained model suitable for the end application. The gti/layers.py file contains specialized GTI layers for model construction. Be aware of GTI chip-specific data preprocessing explained in section 6. Design Considerations, when training floating-point model.

The following command can be used to train a model from scratch:

```
python train.py --net [model name] --chip [target chip]
```

This script provides default options (learning rate, batch size, and more) that can be dynamically altered throughout the training process. It’s recommended to become familiar with the options, and replace with your own defaults, if needed. For the complete list of options, use the following command:

```
python train.py --help
```

To further fine-tune a floating point model, use the following command. Note, to resume or fine-tune from the best checkpoint, e.g. from the `checkpoints/best` directory.

```
python train.py \
      --net [model name] \
      --chip [target chip] \
      --resume_from [e.g. checkpoints/best/vgg16-epoch-100]
```

To avoid out-of-memory errors, adjust the batch size to fit the allowable CPU or GPU inputs.

6.2. Fine-Tuning with Quantized Convolutional Layers

The MDK provides GTI custom convolutional and activation layers with adjustable quantization effects. These can be used as replacements for the regular TensorFlow layers when constructing models. After obtaining a good floating-point model, quantize all convolution layers that are to be accelerated by the GTI device. The following example training script can be used to fine-tune the convolution layers:

```
python train.py \
      --net [model name] \
      --chip [target chip] \
      --resume_from [floating-point checkpoint]
      --quant_w
```

With the effects added, fine-tune the model until it reaches high accuracy on the validation set. As a sanity check, this accuracy should come very close to the floating-point model accuracy, if not better.

Table 1. coef_bits in _dat.JSON per chip and convolution layer quantization

Chip	**Quantized convolution**	
2801	1 bit	`"coef_bits": 1`
	3 bit	`"coef_bits": 3`

6.3. Fine-Tuning with Quantized Activation Layers

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

1. After fine-tuning with quantized convolutional weights, calibrate the output range of each activation layer in the model obtained in the previous step. The MDK provides an example script to calibrate quantization range of activation layers by sampling a few batches (such as 5) and calculating output ranges at a certain percentile (e.g., 99.99%, ignoring outliers). If dataset is large and/or batch size is small, the number of batches may need to be larger to obtain a large enough sample.

   ```
   python set_relu_caps.py \
         --net [model name] \
         --chip [target chip] \
         --checkpoint [checkpoint with quantized weights] \
         --quant_w
   ```

The calculated output ranges will be initialized into the TensorFlow model during subsequent fine-tuning steps. Effectively, the calculated ranges will set an upper bound for each activation layer output. The bounds can then be either trained as a parameter, or remain fixed.

Note, that "**--quant_w**" must be turned on, if the model is already fine-tuned with quantized convolutional weights.

2. After calibrating the quantization range of activation layers, model needs to be fine-tuned with quantized activation layers with the following command.

```
python train.py \
      --net [model name] \
      --chip [target chip] \
      --resume_from [checkpoint with quantized weights] \
      --quant_w \
      --quant_act
```

Note that "`--quant_w`" must also be turned on, if the model has been previously fine-tuned with quantized convolutional weights. Fine-tune this model until it reaches high accuracy, which should come close to full floating-point model accuracy, if not better.

3. Before converting the checkpoint into .model format, evaluate your checkpoints for accuracy. See example:

```
python eval.py \
      --net [model name] \
      --chip [target chip] \
      --checkpoint [checkpoint with quantized weights and activation] \
      --quant_w \
      --quant_act
```

The quantization flags are dynamically toggleable to take into effects of quantization during evaluation.

6.4. Model Fusion

If model contains batch normalization or extreme activation ranges, then this fusion step might be necessary to more accurately simulate GTI chip operations. In essence, model fusion performs a mathematical transformation that merges batch normalization and activation range parameters into weights and biases. It also eliminates the batch norm layers from the model completely, and keeps activation range parameters fixed to chip supported range (i.e. activation ranges should no longer be trainable). After merging, the new weights and biases might violate other chip constraints, so they would have to be fine-tuned again. To perform model fusion and fine-tune, use the following command:

```
python train.py \
      --net [model name] \
      --chip [target chip] \
      --resume_from [checkpoint with quantized weights & activations] \
      --quant_w \
      --quant_act \
      --fuse
```

6.5. Converting the Model and Evaluating On-Chip Performance

GTI accelerator chips can accelerate parts of a neural network. For example, in a standard VGG model, the convolutional layers are accelerated by GTI chips, while the fully connected layers are processed by the host processor.

Following is the model conversion process:

1. Extract the parameters of convolutional layers from a TensorFlow checkpoint and convert them into a .DAT file (runs on the GTI chip).

2. (Optional, since v1.2.2) Extract and store the parameters of the fully connected layers into a .BIN file (runs on the host processor).

3. (Optional) Define a LABELS text file that contains: LABEL_ID LABEL_NAME entries. See examples in data/[EXAMPLE]_labels.txt. The entries must match your training labels.

4. Pack the .DAT (and optional .BIN and label files) into a .MODEL file for further evaluation either with example demo application in the SDK package (use the full model using addon/convert_full.py script) or by chip_infer.py script in native TensorFlow Framework. Since release version 1.2.2, there are two options for converting the checkpoint into .model file.

    a. Create a .model chip readable file that includes parameters for both CNN (.dat) and fully connected (.bin) layers (see Figure 1). The inferencing then can be verified using <MDK_root>/addon/infer_full.py script.

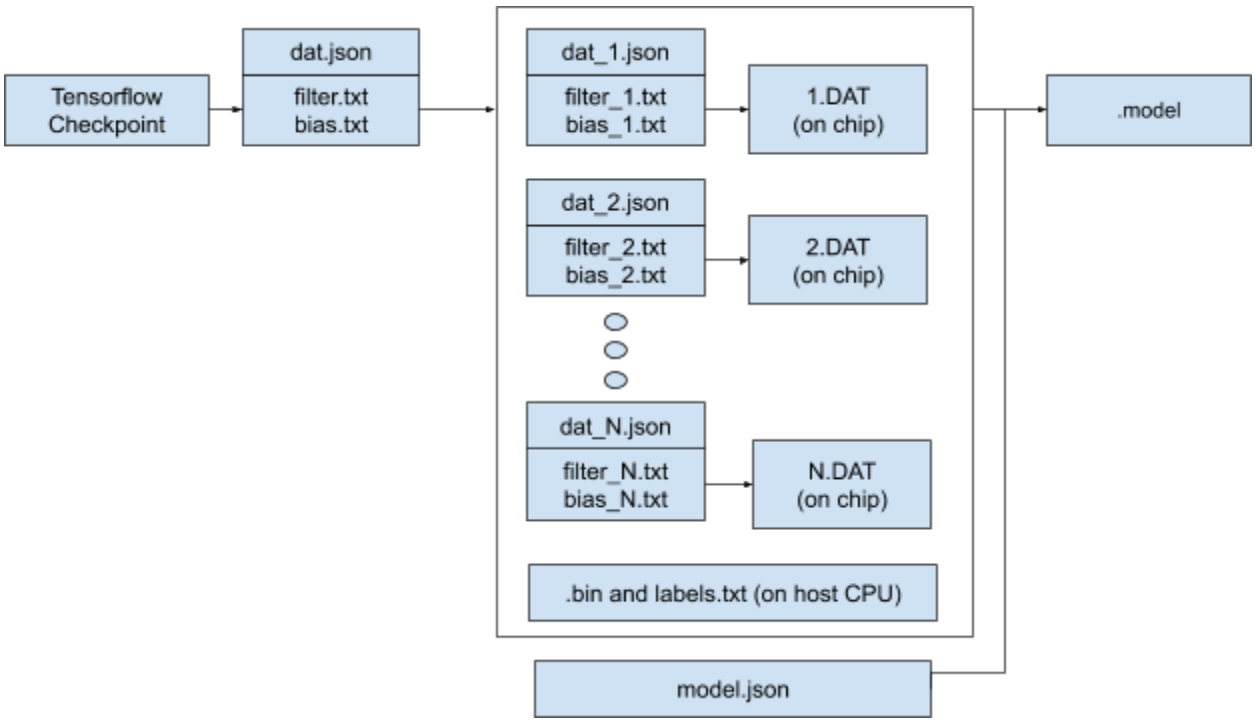

Figure 1. Combining CNN (.DAT files) and FC (.BIN files) layers into a single .MODEL file

b. Create a .model chip readable file that includes parameters for CNN layers only (.dat files). The inferencing then can be verified using chip_infer.py script, see Figure 2.

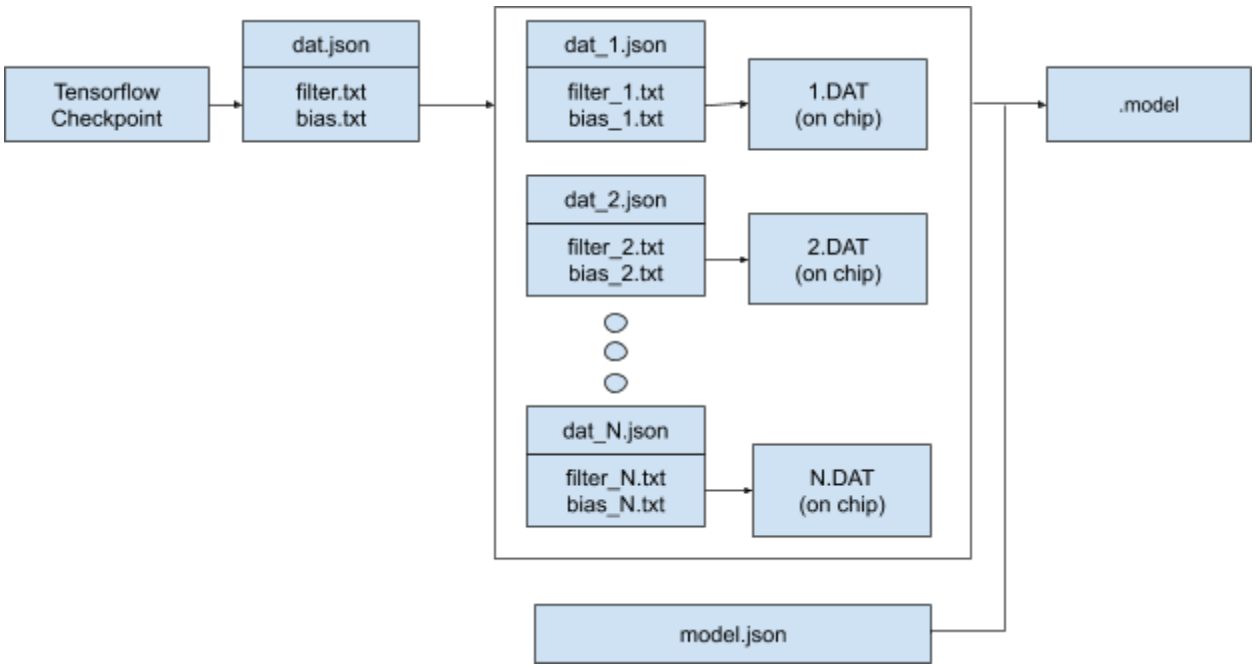

Figure 2. Combining CNN (.DAT files) layers only into a single .model file

Two JSON configuration files are used:

- Chip Definition (e.g., `nets/dat.json`)
- Full model Definition (e.g., `nets/model.json`)

The Chip Definition file describes the network configuration to be accelerated by the GTI chip and is used to generate the .DAT file, that stores the model parameters in a chip-compatible format. With the .DAT fil. Using VGG16 as an example, this .DAT file contains both network configuration information and parameters of the convolutional layers. The full model definition file is used to generate a .MODEL file, with the .DAT file for convolutional layers, and optional .BIN files for fully connected layers.

Note, that the data file (.DAT and .BIN) paths defined in the configuration files are automatically generated and overwritten each time the conversion is run, no need to modify manually.

The MDK provides an example conversion script to convert TensorFlow checkpoint to .MODEL format:

```
python convert_to_chip.py \
      --net [model name] \
      --chip [target chip] \
      --checkpoint [quantized checkpoint] \
      --fuse [if checkpoint is obtained with model fusion]
```

By default, the generated .MODEL files are saved in nets/ directory as `[chip ID]_[model name].model`. To perform chip inference after conversion, use the command:

```
python chip_infer.py \
      --net [model name] \
      --chip [target chip] \
      --checkpoint [quantized checkpoint used during conversion] \
      --last_relu_cap [variable name of last ReLU layer cap on-chip]
```

To see all options and help for chip inference, use the command:

```
python chip_infer.py --help
```

On-chip accuracy should be very close to quantized model accuracy on PC.

6.6. Incremental Improvements

If on-chip result is unsatisfactory, there may be ways to diagnose problems and make improvements. See the following section 5. Design Considerations, to eliminate common pitfalls that may have been missed in the model development cycle.

6.7. Enabling Intermediate Layer Output

Starting From MDK Tensorflow v2.0, MDK can enable or disable the output of intermediate layers by configuring the network*.json file. Intermedi ate layer means the layer before the last sub-layer of the last major layer. The MDK supports the following mode per major layers.

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

Consider the following, as an example:

- Learning mode can be enabled in a specific major layer or in all major layers.

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

The modes are described in the SDK user guide (modelTool section).

For GTI 2801, only modes 0, 1, 2, 4 are supported and it can be configured through the field of "mode" in the "fullmodel.json" file.

7. Examples

Example environment setup for training can be created with Miniconda.

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
source .bashrc
```

For CPU usage:

```
conda create -n tfcpu python=3.6 pip
conda activate tfcpu
pip install tensorflow==1.13.1 tqdm
```

For GPU usage:

```
conda create -n tensorflow_gpuenv tensorflow-gpu
conda activate tensorflow_gpuenv
pip install tensorflow==1.13.1 tqdm
```

7.1. VGG Example Walkthrough

7.1.1. Data Preparation

This MDK utilizes TFRecords (for large datasets) by default. The following example scripts may be adapted to generate TFRecords for your own data.

- https://github.com/tensorflow/models/blob/master/research/inception/inception/data/build_image_data.py
    - Place the image files, labels and converter script, as described below:

```
train/
  daisy/*.jpg
  dandelion/*.jpg
  ...
val/
  daisy/*.jpg
  dandelion/*.jpg
  ...
labels.txt
build_image_data.py
```

- Then run the script with given parameters:

```
python build_image_data.py \
      --train_directory="train" \
      --train_shards=5 \
      --validation_directory="val" \
      --validation_shards=5 \
      --num_threads=5 \
      --output_directory="output" \
      --labels_file="labels.txt"
```

- https://github.com/tensorflow/models/blob/master/research/inception/inception/data/build_imagenet_data.py

Note, both scripts create a background empty class that needs to be considered when training with MDK. For example, if your original number of classes is 4, after converting them into tfrecord format with the suggested scripts, the number of classes should be defined as 5 for the rest of the training.

You may customize your data processing pipeline, but note that input values must be transformed to [0, 31] range during training to simulate chip input ranges, see `gti/data_utils.py`, for more details.

By default, the training and validation data in TFRecord format are stored in the
- `data/[DATASET NAME]/train/`
- `data/[DATASET NAME]/val/`

folders, respectively.

Following is the structure of the `data/` folder used in this example.

```
- data/
    - dataset_name/
        - train/
            - data_00000-of-00005.tfrecord
            - data_00001-of-00005.tfrecord
            - data_00002-of-00005.tfrecord
            - data_00003-of-00005.tfrecord
            - data_00004-of-00005.tfrecord
        - val/
            - data_00000-of-00005.tfrecord
            - data_00001-of-00005.tfrecord
```

```
- data_00002-of-00005.tfrecord
- data_00003-of-00005.tfrecord
- data_00004-of-00005.tfrecord
```

7.1.2. Model Construction

Construct VGG-like model with GTI specialized layers (gti/layers.py) with toggleable quantization effects. See full example model code in gti/model/VGG.py.

7.1.3. Training

The training script (e.g., train.py) loads data from the data folder, and parses command line arguments as training parameters. The argument parser is defined in gti/param_parser.py with some defaults (e.g., --learning_rate 1e-4), but all arguments are dynamically changeable via the command line.

For example, to create and train VGG-like floating-point model from tscratch with a learning rate of 1e-3 for 10 epochs, use the following command:

```
python train.py \
      --net vgg16\
      --chip [target chip] \
      --learning_rate 1e-3 \
      --num_epochs 10
```

Users may need to explicitly provide the directories of the training and validation datasets in the above command, because the default settings in gti/param_parser.py may not be suitable. To see more available training options:

```
python train.py --help
```

The training script alternates between training and evaluation during each epoch, keeping the latest N checkpoints and the current best checkpoint on validation set.

7.1.4. Quantization

First fine-tune with quantized convolutional layers:

```
python train.py \
```


```
        --net vgg16 \
        --chip [target chip] \
        --resume_from [floating-point checkpoint] \
        --quant_w
```

After sufficient training in the previous step, calibrate the activation ranges for quantization of activation layers. Use the following script to calibrate:

```
python set_relu_caps.py \
        --net vgg16 \
        --chip [target chip] \
        --checkpoint [floating-point checkpoint]
        --quant_w
```

The script will overwrite gti/model/vgg16_settings.json with the calibrated activation ranges that will be loaded into the model instance in the following steps.

After calibration, fine-tune the model with quantized activation layers turned on:

```
python train.py \
        --net vgg16 \
        --chip [target chip] \
        --resume_from [checkpoint with weight quantized] \
        --quant_w \
        --quant_act
```

After quantization of activations, perform model fusion and fine-tune:

```
python train.py \
        --net vgg16 \
        --chip [target chip] \
        --resume_from [checkpoint with weight and activation quantized] \
        --quant_w \
        --quant_act \
        --fuse
```


7.1.5. Evaluation

At any point before chip conversion, you may evaluate your model in TensorFlow on the PC using the provided script:

```
python eval.py \
      --net vgg16 \
      --chip [target chip] \
      --checkpoint [floating-point checkpoint]
```

If your checkpoint was trained with any quantization effect, turn on the appropriate command line flags. For example:

```
python eval.py \
      --net vgg16 \
      --chip [target chip] \
      --quant_w \
      --quant_act \
      --fuse [if checkpoint has been fused] \
      --checkpoint [checkpoint with quantized weight & activation]
```

7.1.6. Conversion

Conversion is handled mainly by gti/converter.py. A script is provided to convert a quantized TensorFlow checkpoint and generate chip-compatible model files.

```
python convert_to_chip.py \
      --net vgg16 \
      --chip [target chip] \
      --checkpoint [quantized checkpoint] \
      --fuse [if checkpoint has been fused]
```

These files will be kept in the nets/ folder. The generated .DAT file and .MODEL file are described in the 6.5 of this document. The configuration files responsible for conversion are:

- `nets/[chip ID]_vgg16_dat.json` (for .DAT)
- `nets/[chip ID]_vgg16_model.json` (for .MODEL)

7.1.7. Chip Inference

After generating chip-compatible .DAT and .MODEL files, users can perform inference with GTI Lightspeeur® devices. Use this example script:

```
python chip_infer.py \
      --net vgg16 \
      --chip [target chip] \
      --checkpoint [quantized checkpoint used in conversion] \
      --last_relu_cap [variable name of last ReLU layer cap on-chip]
```

In this example, last_relu_cap is vgg16/conv5_3/relu_cap, because conv5_3 is the scope of the last convolutional layer on-chip. You may need to replace dataset directories, sizes and pipelines to suit your applications.

The accuracy of chip inference with the final quantized model should be similar to that of floating point model from step 1.

To make sure the MDK provides the highest possible accuracy, consider the following useful hints:

- Initialize the training with a better pre-trained floating point model.
- Try different learning rates. Lower learning rate by 10 times after validation accuracy has stopped improving.
- Train for more epochs, if accuracy is still improving.
- Train with larger batch sizes.
- Try a different optimizer, default optimizer in MDK is SGD with momentum, could try, for example, Adam)
- Try different values for weight decay hyperparameter (default value is 1e-4)

8.0. Reference Documentation

GTI SDK version 4.2 : This is the Software Development Kit documentation which includes release notes, installation, and APIs.

Appendix A. JSON File Format

```
"major_layer":          2,          # Major layer index
"image_size":           224,        # Input feature map size of the current major layer
"input_channels":       32,         # Input channel number of the current major layer
"output_channels":      64,         # Output channel number of the current major layer
"depth_enable":         true,       # Indicate if each sublayer is depthwise convolution or not
"sublayer_number":      3,          # Total number of sublayers of the current major layer
"scaling":              [0,0,0],    # Shift setting of each sublayer which is calculated by the model conversion tool
"one_coef":             [0,1,0],    # Indicate if each sublayer is 1x1 convolution or not
"learning":             false,      # enable / disable output of all sublayers in this major layer. Refer to section 7.6.
"last_layer_out":       false,      # enable / disable output of last sublayer of this major layer. Refer to section 7.6.
"pooling":              true,       # Pooling or not at the end of the current major layer
"coef_bits":            8           # Quantization bits of parameters of the current major layer. Refer to Table 1
                                    # Quant_convolution_param per chip and Convolution layer Quantization
```