EMBEDEEP
壹 普 智 慧

EEP-TPU Compiler User Manual

eep-ug050 (v0.6.1)

2023-02-01

www.embedeep.com

Revision history:

Version	Date	Describe	Author
0.1.0	2019-03-18	Initial version	He
0.2.0	2020-05-12	V0.2	He
0.3.0	2021-02-01	V0.3	He
0.4.0	2021-12-11	V0.4	He
0.5.0	2022-04-01	V0.5	He
0.6.0	2022-011-09	V0.6	He
0.6.1	2023-02-01	Github release	He

Contents

1. Preface....4
2. Use....5
2.1 Command line arguments....5
2.2 Description of parameters....7
2.3 Examples of use....16
2.4 Single and mixed precision....17
3. Compatibility....19
4. How to use EEP-TPU compilers for neural networks on different platforms....20
4.1 Caffe....20
4.2 Darknet....21
4.3 Pytorch....21
4.4 Keras （TensorFlow Based）....23
4.5 Tensorflow2....24
5. Compiler preprocessing aids....25
5.1 Keras_convert....25
5.2 Onnx_post....25
6. Common usage problems....26
6.1 “Hardware on-chip memory not enough” problem....26
6.2 Caffe....26
6.3 Darknet....27
6.4 ONNX....27
6.5 The data input to the compiler is user processed data....28

1. Preface

This document is a description of the EEP-TPU compiler.

Compiler version: eeptpu_compiler v2.4.1 and above.

Compiler compilation and running environment: X86 desktop version of “Ubuntu 18.04 LTS” (if you encounter problems running under other Linux systems, please install Ubuntu 18.04 and try again).

2. Use

Executed in the command line terminal on the x86 version Ubuntu system.

2.1 Command line arguments

```
./eeptpu_compiler -h
Usage:
    ../eeptpu_compiler [options]
options:
    --help(-h)              # Print this help message
    --version(-v)           # Print version
  For caffe:
    --prototxt   <path>     # Caffe prototxt file path
    --caffemodel <path>     # Caffe caffemodel file path
  For onnx:
    --onnx <path>           # Onnx file path
  For darknet:
    --darknet_cfg <path>    # Darknet cfg file path
    --darknet_weight <path> # Darknet weight file path
  For ncnn:
    --ncnn_param <path>     # Ncnn param file path
    --ncnn_bin   <path>     # Ncnn bin file path
  For keras:
    --keras_flag <n>        # Keras flag, set n=1 when the net model is from keras h5 file.
  Other params:
    --image <path>            # One typical input image for this neural network. (Support formats: jpg,pgm,bmp,png)
    --pixel_order <str>       # Input pixel order(support: RGB, BGR, RGBA, BGRA, GRAY, default). Default is BGR. Darknet will auto use RGB.
    --input_img <path>      # Same as "--image"
    --input_npy <path>      # One typical input data for this neural network. ( npy format )
    --input_shape <shape>   # Neural network input shape, string format, 4 dims. Such as '1,3,224,224'. (batch, channel, height, width)
    --input_list    <list>                  #    For    multi-inputs.    Format: 'name1;path1;[option1]#name2;path2;[option2]#...'. Use 'Netron' tool to find the input
```

```
names.
                                The 'name' and 'path' are necessary, optional paramaters can be: mean, norm, pack_type, pack_shape, del_last_channels.
                                Each input use '#' to seperate; each paramater in same input use ';' to seperate.
                                Example1:
'input1;./image1.jpg#input2;./data2.npy#input3;./data3.npy'
                                Example2:
'input1;./image1.jpg;mean:103.94,116.78,123.68;norm:0.017,0.017,0.017;pack_type:7;pack_shape:3,256,256#input2;./data2.npy;del_last_channels:1'
    --int8                  # Enable int8 quantization mode.
    --input_folder <path>   # For int8 quantize mode.
    --quant_method <n>      # 0(none),1(kl). Default is 1.
    --quantize <n>          # Quantization mode.
                              n=1: normal int8 quantization, same as '--int8';
                              n=2: improved quantization mode, use quantized model and quantize table;
                              n=3: EF8 quantization mode, use quantized model and quantize table.
    --trunc_mode <n>         # Truncation mode for 'quantize=2' mode. Default is n=0. If n=1, will do truncation before bias in 4D module.
    --qwt_mode <n>           # Quantize weight mode(for 'quantize=2'). Default is n=0, weight data use quantize range [-127,127]; if n=1, use [-128,127].
    --qtable <path>         # Quantize table for quantized model.(Each line format: "Layer_name,shift_value")
    --mean <values>          # Mean values. Format: float array string, such as "103.94,116.78,123.68"
    --norm <values>          # Normalize values. Format: float array string, such as "0.017,0.017,0.017"
    --hw_mean               # Use hareware to process mean&norm for input data
    --opt <n>                # Optimization options, default is 1; If set to 0, will not optimize network.
    --extinfo <ext>          # Extend info. Store customized format string to pass to your application.
    --output <folder>       # Output bin file to this folder
    --public_bin            # Generate 'public' mode bin file (xxx.pub.bin).
    --hybp                  # Use hybrid precision mode.
    --input_pad <l,r,t,b>   # Add padding to input data. Order: left, right, top, bottom.
    --del_last_softmax      # Auto remove last softmax layer.
    --del_last_swlayers      # Auto delete software layers that at the end of
```

```
network.
    --del_last_channels <n> # Auto remove input data's last n channels.
    --jump <n>              # Jump mode. When net1 done, will jump to net2 to run.
n=1: net1; n=2: net2.
    --tpu_threads <n>       # Use TPU multi-cores threads mode, n is TPU threads
count. (Need TPU support!)
    --pack_type <n>         # Set input raw data directly to TPU. Input data memory
layout: HWC order(default), CHW order.
                              Value n is input data format. 1:float32; 2:float16;
3:int8; 4:int16; 5:int32; 7:uint8
                              If set this param, will auto set '--hw_mean' too.
    --pack_shape <shape>    # Set pack output shape. Such as '3,256,256' (channel,
height, width).
```

2.2 Description of parameters

For the “Caffe” platform:

--prototxt: Path to the prototxt file.

--caffemodel: Path to the caffemodel file.

For the “ONNX” platform:

--onnx: The path of the model file, currently only supports ONNX exported by the pytorch framework. When exporting ONNX from pytorch, it is recommended to use:

torch.onnx.export(model, input_read, your_onnx_path, verbose=True, opset_version=11)

For the “darknet” platform:

--darknet_cfg: The path to the cfg file.

--darknet_weight: The path to the weight file.

For “ncnn” platforms:

--ncnn_param: The path to the param file.

--ncnn_bin: The path to the bin file.

For the "Keras" platform:

--keras_flag: Keras flag. The current version of the compiler has limited support for keras and requires the following transformations: Step 1, keras needs to export an "h5" file; Step 2, through "keras_ convert" tool converts the "h5" file to the "ncnn" platform format; Step 3, pass the three parameters "-- ncnn_param, -- ncnn_bin, and -- keras_flag" into the compiler. You can then generate the "eeptpu.bin" file.

Quantization mode: (requires EEP-TPU hardware support)

--int8: Enable the int8 quantization compilation mode (equivalent to "-- quantize 1", it is recommended to use "-- quantize 1").

--input_folder: When compiling int8 quantization, you need to specify multiple typical input data of the neural network for quantization, and we recommend that the number of input data is greater than 5000 (if it is only for test evaluation, the number here can be arbitrary). This parameter is used to specify the folder path for the input data.

--quant_method: Selection of quantification methods. "0-none", using the raw threshold; "1-", use KL quantification method (default selection).

--quantize<n>: Quantize option. Currently, 3 modes are supported:

When n=1: Equivalent to "-- int8", it is a traditional int8 quantization mode. Like "-- int8", it needs to be used in conjunction with "-- input_ folder".

When n=2: Shift quantization mode. You need to pass in the quantized network model and quantization table.

When n=3: EF8 shift quantization mode. You need to pass in the quantized network model and quantization table.

--qtable<path>: The path of quantization table. Example of file contents:

Conv_0,6

Conv_4,5

Conv_6,7

The format of each line is: layer name, shift value. (The two are separated by comma in English format). For example, “Conv_0” in the first line above is the name of a layer in the network, and the “6” in the first line indicates that the data calculated by the layer needs to be shifted by 6 bits to the right. After data is shifted, it is rounded to integer.

--trunc_mode<n>: It is the selection of saturation truncation mode, when the quantization mode is 2 (-- quantize 2). The default value is n=0, which means that there is no saturation truncation after the 4D calculation. n=1 means that saturation truncation is done after the 4D calculation.

--qwt_mode<n>: It is the selection of the quantization range, when the quantization mode is 2 (-- quantize 2). The default value is n=0, which means that the quantization range is -127 to 127; n=1 indicates that the quantization range is -128 to 127.

Other parameters:

--image (or ---input_img): The path to a typical image that applies to the current neural network. Supports images in formats such as jpg, bmp, png, and pgm.

--pixel_order: Used to set the pixel format of the input image. It is passed in as a string and supports pixel formats such as RGB, BGR, RGBA, BGRA, and GRAY. The default format is BGR; If using a Darknet network, the compiler will automatically set it to RGB (in the Darknet network test program, users need to read the image into RGB format by themselves).

--input_npy: A typical input data that applies to the current neural network, in npy file format. You can export “npy” files through “python numpy”. The supported data formats for “Npy” files are: F4 (float32), F2 (float16), i4 (int32), i2 (short), U1 (unsigned char), i1 (char); Fortran order is False.

--input_shape: The dimension of the input data. Usually the compiler automatically recognizes the input dimension; If the compiler cannot obtain the input dimension of the neural network, you can use this option to specify the input dimension.

--input_list: Input parameter settings for multi-input networks.

When a network has only one input, you can configure the parameters with different input parameters (e.g. --input_img, --mean, --norm, etc.). When a network has multiple inputs, it

must be configured with “--input_list”. The format is as follows:

The name of "Input 1"; The path to the data file of "Input 1"; [Optional Parameter 1] # Name of "Input 2"; The path to the data file of "Input 2"; [Optional parameter 2] ## The name of "Input N"; The path of the data file for "input N"; [Optional parameter N]

Thereinto:

Enter a name: Enter a name for the blob, which can be viewed using the Netron tool.

The input name and input path must be configured. Other optional parameters can be set based on actual requirements Optional parameters include: “mean”, “norm”, “pack_type”, “pack_shape”, “del_last_channels”.

Configurations of each input are separated by a pound sign "#"; Different parameters for the same input are separated by an English semicolon ";"; The parameter name and value of optional parameters are separated by an English colon ":". The above symbols are all English characters. Note that "#" and ";" are not allowed in the input file path.

Example 1:

--input_list 'input1;./image1.jpg#input2;./data2.npy#input3;./data3.npy’

Example 2:

--input_list 'input1;./image1.jpg;mean:103.94,116.78,123.68;norm:0.017,0.017,0.017;pack_type:7;pack_shape:3,256,256#input2;./data2.npy;del_last_channels:1'

✧ **Quantized compilation in multi-input mode**: As with single input, you need to configure "--quantize 1 --input_folder ./inputs/". The main difference is the storage of data in the input folder. In single input mode, you only need to store all input data files in the folder specified by “input_folder”. In the multi-input mode, the folder specified by “input_folder” has a specific format for storing it (take “./inputs/” as an example):

In multi-input mode, multiple input data required for a single inference is formed into a group of input data, and each group of input data contains multiple input files.

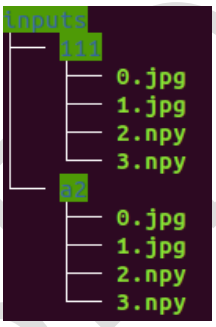
The “Inputs” folder can contain multiple subfolders for storing input data, and the name

of the subfolder can be arbitrarily named. Each subfolder must store N input data. The names of the N input data must be specified. The file name is “Input ID + suffix”, for example, 0.jpg, 1.npy.

How to get the Input ID number: The compiler precompiles the network in non-quantized mode, which prints out the information for each input data on the command line, including the input ID. For example:

```
Neural network inputs:
    InputID[0] 1,3,224,224; name: image_left
    InputID[1] 1,3,224,224; name: image_right
    InputID[2] 1,3,224,224; name: image_face
    InputID[3] 1,625,1,1; name: facegrid
```

An example of a folder structure (the network contains four inputs):



--mean, --norm: used for mean, normalization.

The formula for calculating the mean-normalization is: X '=(X - mean) * norm, where X is the input value; X 'is the value after the mean normalization calculation, which is used as the input of the neural network. The mean-normalized input should be consistent with the number of channels of the neural network input. For example, if the number of input channels is 3, mean and norm must be an array of three values, each of which corresponds to one channel. File format is not currently supported.

How to get the value of the mean-normalization: It can generally be obtained from the training file of the neural network (e.g. “train.prototxt”). Take Mobilenet V1 as an example, which can be found in its training file:

```
transform_param {
    scale: 0.017
```

```
    mirror: true
    crop_size: 224
    mean_value: [103.94, 116.78, 123.68]
    }
```

“mean” is obtained from “mean_value”: 103.94, 116.78, 123.68;

“norm” is obtained from “scale” and extended to 3 channels: 0.017, 0.017, 0.017;

Note: The mean-normalization in Python is generally named as “mean” and “std”, where “std” corresponds to the above mentioned “normal”.

Mean-normalization is equivalent to preprocessing the input data. Take OpenCV image preprocessing in Python as an example:

```
img = img / 255
img -= 0.5
img /= 0.5
```

It can be obtained from the above processing:

```
input = (( img / 255) - 0.5) / 0.5
      = (img - 0.5*255) * 2 / 255
      = (img - 127.5) * 0.007843137
```

We obtain “mean”=127.5, “norm”=0.007843137. After expanding to 3-channel data, it is: *-- mean "127.5127.5127.5" -- norm "0.007843137,0.007843137,0.007843137".*

--hw_mean: Mean-normalization is calculated using EEP-TPU hardware.

For example, by passing in the parameter “-- mean value list -- normal value list -- hw_mean”, the compiler can configure the mean-normalization for hardware calculation.

If this option is configured, the mean-normalization is calculated by the TPU, and the CPU related preprocessing program does not need to calculate the mean-normalization separately.

--opt: Neural network optimization option. It is enabled by default (n=1), which can optimize certain network structures. Set to 0 to turn off optimization.

--extinfo: A custom extension configuration string. Users can customize the written format,

read it through API functions in the user program, and parse it by themselves. Users can save some specific data from the compilation stage into a bin file and use it in applications. For example, the user can save the target name list of the target detection network in the string specified by "—extinfo", and then retrieve it in the test program through an API function.

--output: The name of the folder where the compiled bin file is saved (not the file name).

--public_bin: Used to compile and generate the public version of bin file. The default suffix is ". pub. bin". This public version of the bin file needs to be combined with "*libeeppu_pub.so*" library file. The current public version of the compiler does not yet provide support for alternate use of software and hardware layers.

When compiling a public version bin file, the following two situations can succeed:

(1) All operators can be supported by hardware, that is, all hardware reasoning.

(2) The neural network can be divided into two parts, the former part of the operator is all supported by hardware, the latter part of the operator is supported by software (library file). That is, the hardware reasoning first, then the software reasoning and get the whole network reasoning result. For example, the "detection_output" layer at the end of a "Yolo" network can be calculated by software, while the "softmax" layer at the end of a classified network such as "mobilenet" can be calculated by software.

When there is a layer in the neural network that is not supported by hardware computation, the compiler issues an error message.

--hybp: Compile using mixed precision mode. If this parameter is not added, the default is to use single precision mode.

EEP-TPU has two different modes, single precision and mixed precision. In the mixed precision mode, INT8 quantization can be used. For EEP-TPU, single precision and mixed precision bin files cannot be compatible. That is, single precision bin files compiled by the EEP-TPU compiler can only be used on EEP-TPU hardware that supports single precision; The compiled mixed precision bin file can only be used on EEP-TPU hardware that supports mixed precision.

At the end of compilation by the EEP-TPU compiler, the platform for which the bin file

is applicable will be output:

Generated 'eeptpu.bin' for platform: EEPTPU C8S1

where “S1” stands for single precision; “S2” stands for Mixed Precision.

In the sample program provided by the “EEP-TPU SDK”, you can read the configuration of the EEP-TPU hardware, for example:

EEPTPU hardware info: EEP-TPU;M1024;N1;C8;LS2;S1;

The "S1" at the end of the configuration information represents single precision; If this information is "S2", it is mixed precision.

In the application, if the bin file and the EEP-TPU precision mode do not match, an error will be reported. At this point, it is necessary to recompile the bin file to adapt to the precision mode of the EEP-TPU:

Error: EEP-TPU use S2, but bin file use S1.

--input_pad <l,r,t,b>: Padding the input data. It is only limited to the case where the first layer of the network is the convolutional layer. After this parameter is used, the input dimension of the original network model is changed, and the new input dimension plus the padding value is equal to the original input dimension. For example, the original network input dimension is 3×210×210. When compiling, set padding=5 on the left, right, up, and down sides of the new input data through “-- input_pad ‘5,5,5’”, and the new input dimension becomes 3×200×200.

--del_last_softmax: Commonly used in classified networks. If the neural network ends with a “softmax” layer, the last layer “softmax” is automatically removed at the compilation stage, which does not actually affect the classification results.

--del_last_swlayers: At the compilation stage，remove the last layer of the network that does not support TPU hardware computing. （In some cases, you can replace “del_last_softmax” above）.

--del_last_channels: Delete the last N channels of the input data. For example, the corresponding input of a network is a 3-channel image (BGR), while the input image is a 4-channel image (BGRA). Using this parameter, the last channel of the input image can be

automatically deleted to adapt it to the input dimension of the network.

--jump: Dual algorithm jump mode, supporting jump from algorithm 1 to algorithm 2. Both algorithm 1 and algorithm 2 need to add this parameter when compiling. Algorithm 1 uses "--jump 1"; Algorithm 2 uses "--jump 2". The output of algorithm 1 is used as the input of algorithm 2, and the output dimension of algorithm 1 is required to be consistent with the input dimension of algorithm 2. If algorithm 2 does mean-normalization, algorithm 2 needs to add the parameter "--hw_mean" to hardware it when compiling.

--tpu_threads<n>: Multithreaded configuration, using multithreaded federation for network reasoning. The parameter n is the number of threads. One thread corresponds to one TPU core. In general, when the EEP-TPU hardware contains multiple cores, multi-threading can be used to improve the inference speed of the network.

--pack_type <n> --pack_shape <c, h, w>: Set the parameters for "pack mode". In "Pack Mode", input data with different accuracy can be converted into the format required by TPU. The input data exists in the "layout" format of "HWC", which is the same as the data format of "opencv".

The compiler supports FP32/FP16/INT8/INT16/INT32/UINT8 precision conversion, and can use TPU to convert "pack mode" input data to EF16 data format required by TPU.

When using PACK, the input data dimension must be fixed. In particular, for classification networks, it is necessary to convert the input image into a unified dimension in advance, consistent with the dimension specified by the compiler when compiling "-- pack_type<n>-- pack_shape 'c, h, w'".

When using pack mode, the input data is one-dimensional data (e.g.1,1,196608). "Pack_shape" specifies the output dimension of the pack operator, in numerical order "C,H,W" (e.g. 3,256,256; 3×256×256=196608). If the input dimension of the neural network is inconsistent with the output dimension of the pack operator, the compiler will automatically add a resize operation after the pack operator to resize the data to the network input size. Therefore, the "pack_shape" dimension does not have to be consistent with the neural network input dimension.

"Pack_type" value:

When the neural network has the mean-normalization, the treatment of the mean-normalization is as follows:

(1) It can be processed by an application, and the value of the mean option passed to the compiler is all 0, while the value of the norm option is all 1. However, this mode is only applicable to input data of floating-point type.

(2) Or processed by the compiler.

When using precision conversion mode, both resize and mean normalization can be processed by hardware. Taking mobilenet as an example, the neural network input is 3×224×224. Pack_ Shape is set to 3×256×256, and one-dimensional 196608×N bytes of data need to be input during compilation (N values: N=4 for fp32/int32, N=2 for fp16/int16, and N=1 for int8). TPU will automatically resize the input data to 3×224×224 and perform the mean normalization process.

Input data format in precision conversion mode: one-dimensional data. Take an image as an example, and the data format is HWC arrangement. The arrangement of image data read through opencv is HWC (bgrbgrbgr....). The data size needs to be consistent with the size specified by “pack_shape” in the compilation phase. For example, if “pack_shape” specifies “3,256,256”, the input data size is 3×256×256=196608.

When the input data is an image and the “pack” mode is used, the “pack_type” is set to 7 (uint8 type, value range is 0 to 255).

2.3 Examples of use

- **Caffe Mobilenet:**

eeptpu_compiler --prototxt /path/to/prototxt --caffemodel /path/to/caffemodel --image /path/to/typical/image --mean '103.94,116.78,123.68' --norm '0.017,0.017,0.017' --output ./

- **Onnx:**

eeptpu_compiler --onnx /path/to/onnx_file --input_npy /path/to/typical/input_npy_file --output ./

- **DarkNet:**

eeptpu_compiler --darknet_cfg /path/to/darknet_cfg_file --darknet_weight /path/to/darknet_weight_file --input_img /path/to/typical/input_image_file --mean '0.0,0.0,0.0' --norm '0.003921569,0.003921569,0.003921569' --output ./

- **Ncnn Mobilenet:**

eeptpu_compiler --ncnn_param /path/to/ncnn_param --ncnn_bin /path/to/ncnn_bin --image /path/to/typical/image --mean '103.94,116.78,123.68' --norm '0.017,0.017,0.017' --output ./

- **Keras:**

eeptpu_compiler --ncnn_param /path/to/ncnn_param --ncnn_bin /path/to/ncnn_bin --image /path/to/typical/image --mean '103.94,116.78,123.68' --norm '0.017,0.017,0.017' --output ./ --keras_flag 1

2.4 Single and mixed precision

EEP-TPU processors have two architectures, single precision and mixed precision, which are incompatible with each other. The compiler specifies the architecture through the “--hybp” option. Please ensure that the EEP-TPU hardware and compiler select the same architecture.

Single precision mode

Single precision mode refers to a mode with a data format of FP16, in which EEP-TPU only supports ordinary operators. Please refer to the “EEp-ug004 EEP-TPU Operator List” for ordinary operators In this mode, the user needs to prepare a typical piece of data (such as a picture) corresponding to the algorithm, and perform a reasoning calculation through the EEP-TPU behavior model built into the compiler to preliminarily verify whether the EEP-TPU supports the algorithm. For compiler options and usage instructions, refer to Sections 2.2 and 2.3.

Mixed precision mode

The mixed precision mode refers to a mode where the data is both FP16 and INT8. In this

mode, EEP-TPU can simultaneously support ordinary operators, Quantize operators, and FQuantize operators. For details, please refer to the “EEP-ug004 EEP-TPU Operator List”.

In mixed precision mode：

(1) You can only use FP16 to calculate precision, and its usage method is consistent with single precision. Please refer to Sections 2.2 and 2.3 for compiler options and instructions.

(2) You can also use the mixed precision calculation of FP16+INT8, in which case the compiler will give preference to the “Quantize operator” and the “FQuantize” operator. To use mixed precision calculations, users need to prepare multiple input data for the compiler to perform INT8 quantization on the corresponding parameters. The recommended number of input data is greater than 5000. When performing mixed precision compilation, the following options need to be added to the compiler options: --hybp --int8 --input_folder. For example:

> INT8:

eeptpu_compiler --int8 --input_folder /path/to/images_folder/ --output ./ --mean '103.94,116.78,123.68' --norm '0.017,0.017,0.017' --prototxt /path/to/prototxt --caffemodel /path/to/caffemodel --image /path/to/typical/image

3. Compatibility

Non-INT8 version: The “eeptpu bin” file generated by the compiler needs to be used with the “EEPTPU libeeptpu.so API” version: V2.0 and above (the “public bin” version uses the “libeeptpu_pub.so API” version is V0.6.4 and above); The EEP-TPU hardware version that needs to be used with the compiler is: 0.7.2-R2 and above.

INT8 quantized version: The “eeptpu bin” file generated by the compiler needs to be used with the “EEPTPU libeeptpu.so API” version: V2.0 and above (the “public bin” version uses the “libeeptpu_pub.so API” version is V0.6.4 and above); EEP-TPU hardware supporting INT8 is required.

The EEP-TPU compiler supports a neural network model for the Cafe/Brokennet/PyTorch (onnx)/Keras/NCNN framework. Model files generated by other platforms (such as “Mxnet”) can be converted to the above format by some corresponding model conversion tools.

The EEP-TPU compiler supports hardware and software collaborative computing. For layer types that cannot be accelerated by EEP-TPU hardware for the time being, software can perform calculations.

The EEP-TPU compiler supports multiple output layers. Through the “EEPTPU API” function, multiple layers of inference result data can be read together.

4. How to use EEP-TPU compilers for neural networks on different platforms

4.1 Caffe

The EEP-TPU compiler supports the neural network model of the caffe platform.

Take the “mobilenet-v1” model as an example:

eeptpu_compiler --prototxt /path/to/prototxt --caffemodel /path/to/caffemodel --input_img /path/to/typical/image --mean '103.94,116.78,123.68' --norm '0.017,0.017,0.017' --output ./

Specify the path of the model's “prototxt” file by “—prototxt”;

Specify the path to the model's “caffemodel” file by “—caffemodel”;

Specify the path of a typical input image of the neural network by “--input_img”;

Specify the mean and normalized values of the neural network by “--mean” and “--norm”.

Under the Cafe platform, the mean value can generally be obtained from the training file of the neural network (such as “train. prototxt”). Taking “MobilenetV1” as an example:

```
transform_param {
    scale: 0.017
    mirror: true
    crop_size: 224
    mean_value: [103.94, 116.78, 123.68]
    }
```

“mean” is obtained from “mean_value”: 103.94, 116.78, 123.68;

“norm” is obtained from “scale” and expanded to 3 channels: 0.017, 0.017, 0.017.

4.2 Darknet

The EEP-TPU compiler supports the neural network model of the “darknet” platform.

Take the “darknet yolov3” model as an example:

eeptpu_compiler --darknet_cfg /path/to/darknet_cfg_file --darknet_weight /path/to/darknet_weight_file --input_img /path/to/typical/input_image_file --mean '0.0,0.0,0.0' --norm '0.003921569,0.003921569,0.003921569' --output ./

--darknet_cfg: Specify the path to the model's “cfg” file;

--darknet_weight: Specify the path to the model's “weights” file;

--input_img: Specifies the path to a typical input image of the neural network (same function as “--image”).

Mean, normalization: The normalization value of the neural network on the Darknet platform is 1/255.0 (that is, 0.003921569). So, mean is 0 and norm is 0.003921569. If the input image is 3 channels, extend “mean” and “norm” to the values of 3 channels:

--mean '0.0,0.0,0.0' --norm '0.003921569,0.003921569,0.003921569'

The order of input images for Darknet is in RGB format, and the compiler will automatically process the input images into RGB order when compiling the Darknet network. For example, when using “opencv” to read images, the sequence of images is BGR, which users need to pay attention to when writing programs. Users also need to adjust their images to RGB order during the test program.

4.3 Pytorch

The EEP-TPU compiler requires “ONNX” to support the “Pytorch” platform model. The user needs to convert the “pytorch” model file into an “onnx” file, process it with the “onnx_post” tool, and then compile it through the EEP-TPU compiler.

(1) Convert the Python network model to ONNX format

In the first step, convert the “pytorch” network model into an “onnx” file using pytorch's “onnx export” feature (torch.onnx.export). **It is important to note that when using “pytorch export”, remember to distinguish between the backbone network and the “Pre-processing & post-processing” functions. In general, EEP-TPU only supports the calculation of backbone networks. The pre-processing & post-processing (such as YOLO's post processing function Detectionout) function cannot be output through “onnx”, and must be completed through the C function.**

“Onnx export” example:

torch.onnx.export(model, input_read, your_onnx_path, verbose=True, opset_version=11)

In the second step, “onnx typically” has many useless glue operators that have no role in computation and need to be optimized. You can use the “onnx_post” tool to optimize the model of an “onnx” file to obtain a new “onnx” file. After that, it can be compiled using the eeptpu compiler.

(2) Onnx_post tools

The “Onnx_post” tool is provided by our company.

Operating environment: X86 PC, Ubuntu 18.04.

How to use: Open a command line and execute the following command:

./onnx_post input_model output_model [--input-shape size of the Input Model]

For example: *./onnx_post model.onnx new.onnx --input-shape 1,2,128,233*

Among them:

“model.onnx” is the “onnx” file exported by “pytorch”;

“new.onnx” is the new “onnx” file generated by the “onnx_post” tool;

“input-shape” is the input size of the neural network model (optional).

(3) “eeptpu_compiler” compile “onnx” files

The “new.onnx” model file generated by the “onnx_post” tool can be compiled using “eeptpu_compiler”.

Command example:

./eeptpu_compiler --output ./ --onnx /path/to/onnx_file --input_img /path/to/typical/image

Among them:

--onnx: The path to the “new onnx” file processed by the “onnx_post” tool.

--input_img: The path to a typical input image of the neural network.

4.4 Keras （TensorFlow Based）

The Keras network model needs to be converted to an “ncnn” model before being compiled through the EEP-TPU compiler.

(1) “Keras_convert” tool

The "Keras_convert" tool is provided by our company and can convert model files in “H5” format exported by the “KERS” network to “ncnn” model format.

Operating environment: X86 PC, Ubuntu 18.04.

Instruction for use: keras_convert src_h5_path dst_folder

For example: *./keras_convert ./src.h5 ./*

After execution, “output.param” and “output .bin” files will be generated in the destination folder.

(2) Compile with “eeptpu compiler”

The “output. Param” and “output. Bin” generated by the “Keras_convert” tool are in the “ncnn” model format.The model file is specified by the compilers “--ncnn_param” and “--ncnn_bin”, and the model file needs to be indicated by “--keras_flag 1” that the model file is from the KERASE framework.

Example of a compilation script:

./eeptpu_compiler --output ./ --ncnn_param ./output.param --ncnn_bin ./output.bin --input_npy ./input_data.npy --keras_flag 1

Among them:

--ncnn_param: Specify the path to the “output.param” file generated by the “keras_convert” tool;

--ncnn_bin: Specify the path of the “output.bin” file generated by the “keras_convert” tool;

--input_npy: Specifies a typical input file path for this neural network model. Take “npy” as an example: “npy” files can be generated through python. If the file is an image, you can change the parameter to “-- input_img”.

--keras_flag: It needs to be set to “1” to identify whether the current network is from the “Keras” framework.

4.5 Tensorflow2

You need to convert the “Tensorflow2” model to an “mlir” file, and then compile it through the EEP-TPU compiler.

The current version of the EEP-TPU compiler does not yet support “mlir” files, and a higher version of the compiler is under development.

5. Compiler preprocessing aids

The preprocessing auxiliary tools of the EEP-TPU compiler mainly include the conversion tools of "Keras" and "ONNX (Pytorch)", which are used to assist in the compilation of AI algorithms under the "Keras" and "Pytorch" frameworks.

5.1 Keras_convert

See Section 4.4 of this document for details.

5.2 Onnx_post

Optimized processing of "onnx" model files exported by "pytorch", see Section 4.3 of this document for details.

6. Common usage problems

6.1 “Hardware on-chip memory not enough” problem

The data in a certain layer of the neural network is too large, resulting in insufficient internal resources in the EEP-TPU.

Solution: Try to reduce the output or parameter number of layers with excessive data in the neural network. For example, when the output dimension is large, you can try to modify the network to make the output dimension smaller; When the parameter amount is too large, try to modify the network to reduce the parameter amount of this layer.

Or you can contact our technicians for assistance: herh@embedeep.com.

6.2 Caffe

(1) The compiler encountered the following error while compiling the caffe model:

Analysing caffe files...

[libprotobuf FATAL ……/protobuf_2.6.1/include/google/protobuf/repeated_field.h:886] CHECK failed: (index) < (size()):

terminate called after throwing an instance of 'google::protobuf::FatalException'

what(): CHECK failed: (index) < (size()):

Solution: Check whether the caffe model file is in the error format. If so, you can use the caffe tools “upgrade_net_proto_text” and “upgrade_net_proto_binary” to convert the caffe model, for example:

upgrade_net_proto_text ./deploy.prototxt new.prototxt

upgrade_net_proto_binary ./deploy.caffemodel new.caffemodel

The above two tools can be compiled in the caffe source code.

6.3 Darknet

(1) When compiling into eeptpu.bin, the number of target boxes from hardware is inconsistent with darknet software

Check whether the “ignore_thresh” value in the “cfg” file is consistent with that in the “darknet”. In darknet, the default value for “ignore_thresh” is 0.25, and the default value for “nms” is 0.45.

When compiling darknet, the eeptpu compiler will output the currently used “ignore_thresh” and “nms_thresh” values.

“ignore_thresh” uses the “ignore_thresh” value of the last [yolo] layer in the “cfg” file.

You can manually modify the “ignore_thresh” value of the last [yolo] layer of the “cfg” file, or add “nms_thresh” parameters to [yolo] (if not added, the default is 0.45, which is the same as the darknet default).

(2) Format of “Input Image”

The format of the input image for Darknet is RGB order.

When the compiler compiles a darknet network, it will convert the input image to RGB sequential format by default.

If “opencv” is used to read images in the test program, the image read by “opencv” is in BGR format, so it is necessary to convert the BGR format to RGB format before reasoning.

(3) Mean/normalization

The official normalized value of Darknet is 1/255. So the compiler needs to set the mean: (3-channel input as an example)

--mean '0.0,0.0,0.0' --norm '0.003921569,0.003921569,0.003921569'

6.4 ONNX

Currently, only the “onnx” model exported by the “Pytorch” framework is supported.

After exporting the “onnx” model file through “Pytorch”, you need to use the “onnx_post” tool provided by EEP to process the model file. After generating a new “onnx” file, you can compile it.

(1) When containing “interpolate” or “Upsample” operators

The interpolation mode needs to be bilinear interpolation, and you need to configure “align_corners” to “True”.

When exporting an “onnx” model file through “Pytorch”, you need to configure “opset_version” to “11”, otherwise compatibility issues may occur.

```
torch.onnx.export(model, input_read, your_onnx_path, verbose=True, opset_version=11)
```

(2) “onnx_post” tools

When using the “onnx_post” tool, if it is found that some operators are not supported; Or, after being processed by this tool, it is found that some operators are not supported during compilation. It may be that the current compiler version does not yet support this model file. You can contact us (herh@embedeep.com) for assist.

6.5 The data input to the compiler is user processed data

Users can process input data by themselves (such as custom preprocessing, subtracting the mean, or normalizing).

The data can be saved in “npy” format, and the data type supports: F4 (float32), F2 (float16), i4 (int32), i2 (short), u1 (unsigned char), i1 (char); Fortran order is False.

If the input data given by the user has been preprocessed, the compiler does not need to provide the mean options of “—mean” and “—norm” during compilation (i.e. the default mean is 0 and norm is 1).

However, in the application program, the user needs to perform the same preprocessing on the input data, and then call the “eeppu_set_input function” in the eeptpu library file:

```
int eeptpu_set_input(void* input_data, int dim1, int dim2, int dim3, int mode = 0);
```

The mode here should be set to 1, which means that the input_data is a float-type array pointer.The input_data has been preprocessed by the user, it can be directly used as the input data of the neural network.

npy files: can be saved in Python with numpy.

In an “npy” file, the stored data is a single input data.

If the “Fortran order” of “npy” is true, it can be converted to “False” through the following python script:

```
import numpy as np
data = np.load("input_fortran.npy")      # input is: 'fortran_order': True
print(np.isfortran(data))
data = data.copy(order='C')          # convert to: 'fortran_order': False
print(np.isfortran(data))
np.save("input_new.npy", data)
```