RZ/V2L, RZ/V2M, RZ/V2MA
AI IMPLEMENTATION GUIDE
Darknet YOLO
REV.7.20

2022年9月

ルネサス エレクトロニクス株式会社

R11AN0620JJ0720

はじめに

本書は、AI Implementation Guide Get Startedドキュメントの内容をDarknetフレームワークが提供する
YOLOv3、YOLOv2、Tiny YOLOv3、Tiny YOLOv2事前学習済みモデル（以降、Darknet YOLO）を使用して解説します。

事前にGet Startedドキュメントを必ずお読みください。

本書では以下のドキュメント及びファイルを使用します。

名称	ファイル名	説明
Get Startedドキュメント	r11an0616jj0720-rzv-ai-imp-getstarted.pdf	AI Implementation Guide環境構築方法及び AI開発フローが記載されたドキュメント。
Get Startedソースコード	rzv_ai-implementation-guide_ver7.20.tar.gz	AI Implementation Guide全体で使用するソースコード。
Darknet YOLO向けドキュメント	r11an0620jj0720-rzv-ai-imp-yolo.pdf	本書。 Darknet YOLOモデル向け手順が記載されたドキュメント。
Darknet YOLO向けソースコード	darknet_yolo_ver7.20.tar.gz	Darknet YOLO向けドキュメント内で使用する ソースコード及び実行例。

ガイドの流れ

Step内で作成する物
前stepの成果物

目次

Step内で作成する物

前stepの成果物

STEP-1

Introduction

STEP-2

ONNXに
変換しよう

ONNX変換用スクリプト

学習済みモデル

STEP-3

DRP-AI
Translatorを
使ってみよう

変換用設定ファイル

ONNXモデル

STEP-4

推論実行
してみよう

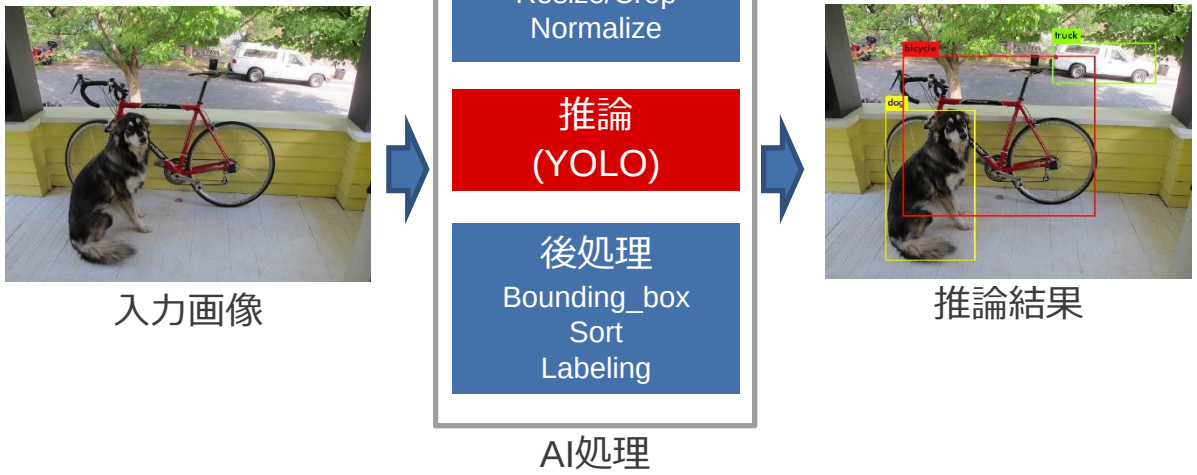
AI実行環境

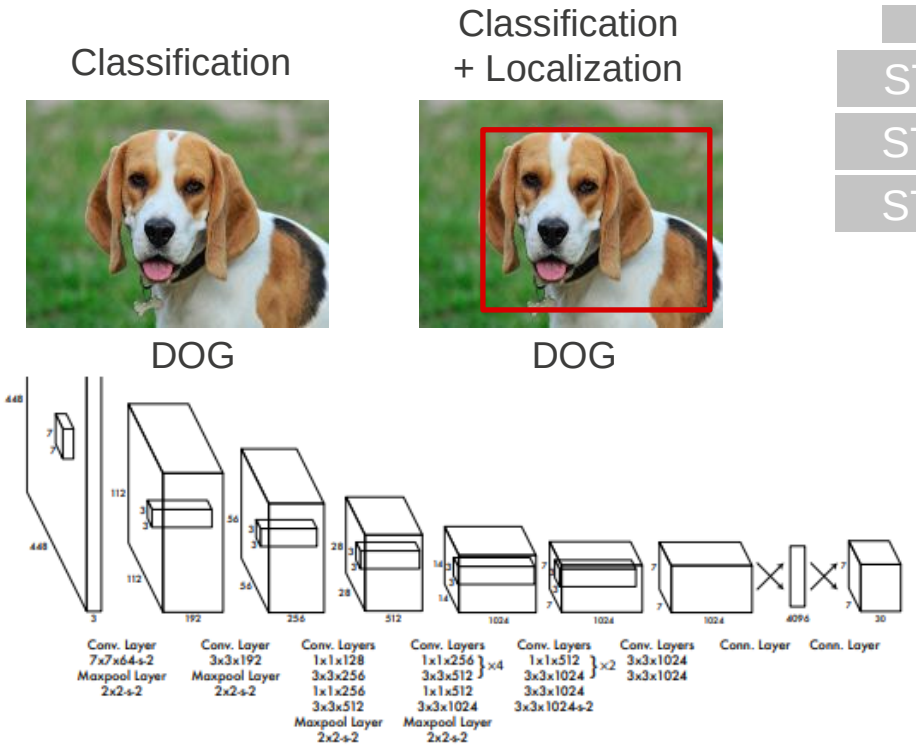
DRP-AI Object files

YOLO

YOLO (**Y**ou **o**nly **l**ook **o**nce) は、画像認識の中でも「物体検出（Object detection）」向けのNeural Networkです。
Classificationだけでなく座標情報（Localization）まで認識することができます。

本書の実装対象はYOLOv3, YOLOv2, Tiny YOLOv3, Tiny YOLOv2の4モデルです。





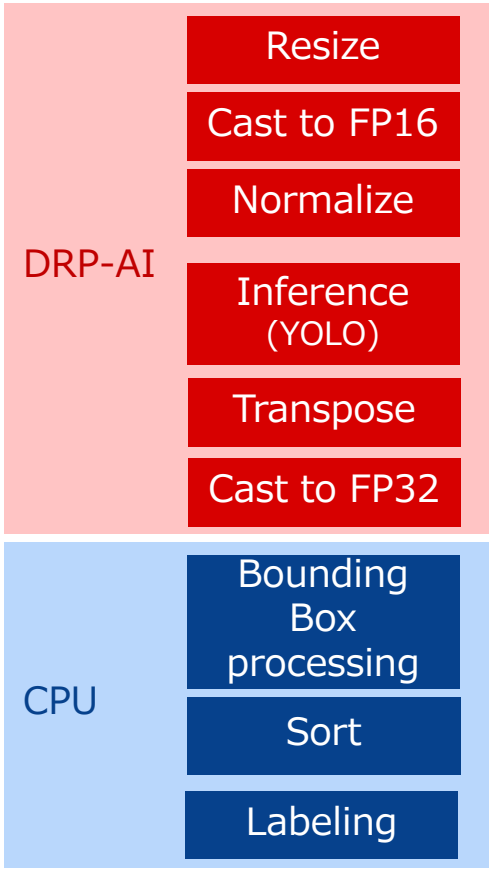
(引用) You Only Look Once: Unified, Real-Time Object Detection
https://arxiv.org/pdf/1506.02640.pdf

(論文) You Only Look Once: Unified, Real-Time Object Detection https://arxiv.org/pdf/1506.02640.pdf
(論文) YOLOv3: An Incremental Improvement https://arxiv.org/pdf/1804.02767.pdf
(論文) YOLO9000 Better, Faster, Stronger https://arxiv.org/pdf/1612.08242.pdf
(参考) YOLOv3: Real-Time Object Detection (Joseph Redmon) https://pjreddie.com/darknet/yolo/
(参考) YOLOv2: Real-Time Object Detection (Joseph Redmon) https://pjreddie.com/darknet/yolov2/

【参考】 YOLO on DRP-AI

以下はDRP-AIでYOLOモデルを実行する際の使用オペレータ例です。 入力データは640x480のBGR画像です。

DRP-AIで対応していないオペレータはCPUで実行する必要があります。



<Pre-processing>
Resize：Input image size(640x480 BGR) to HWC tensor size(416x416x3).
Cast to fp16：Cast to fp16 from int8 for DRP-AI processing.
Normalize: Normalized using 'mean' and 'std'.

<Post-processing>
Transpose: HWC to CHW.
Cast to fp32

Darknet YOLO Transpose Shape

Model	HWC	CHW
YOLOv3	[13, 13, 255] [26, 26, 255] [52, 52, 255]	[255, 13, 13] [255, 26, 26] [255, 52, 52]
YOLOv2	[13, 13, 125]	[125, 13, 13]
Tiny YOLOv3	[13, 13, 255] [26, 26, 255]	[255, 13, 13] [255, 26, 26]
Tiny YOLOv2	[13, 13, 125]	[125, 13, 13]

【参考】 Darknet

本書では、DarknetというAIフレームワークが提供する事前学習済みモデルを使用します。

Darknetは論文で発表したYOLO構造をC言語で実装しており、学習機能や推論実行機能だけではなく、事前学習済みモデルや精度計測機能を試すことができます。

Darknetが持つ様々な機能については、公式サイト（https://pjreddie.com/darknet/yolo/）をご参照ください。

学習・推論機能

精度計測機能

事前学習済みモデル

Neural Network構造

ただし、DarknetにはONNX変換機能が含まれていないため、本書では、PyTorchのONNX変換機能を使って、Darknetの事前学習済みモデルをDRP-AI Translatorの入力に必要なONNXフォーマットのモデルに変換して、使用します。

【参考】 PyTorch

PyTorchフレームワークには学習のための機能だけではなく、事前学習済みモデルやONNX変換機能(torch.onnx)が含まれており、簡単にONNX変換を試すことができます。

PyTorchが持つ様々な機能については公式ドキュメント(https://pytorch.org/docs/1.12/)をご参照ください。

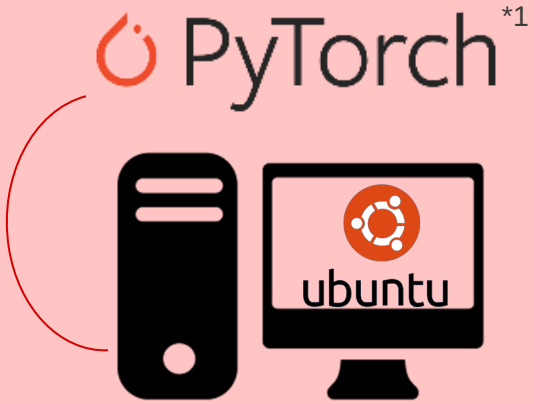
必要環境

本書の各STEPで必要な環境イメージは以下のとおりです。

環境構築方法はGet Startedドキュメントをご参照ください。

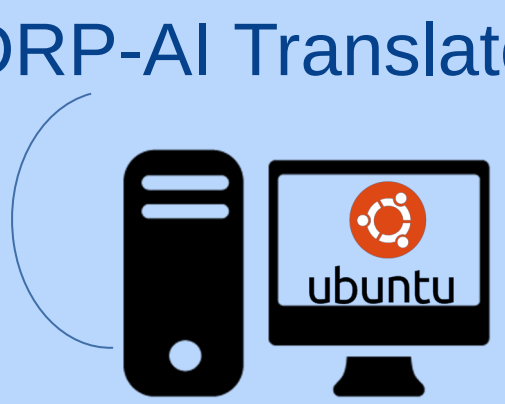
※本書ではDarknetの事前学習済みモデルを使用しますが、Darknetフレームワーク自体は使用せず、PyTorchフレームワークを使用します。

＜ONNX変換環境＞



以下のSTEPで使用します。
- STEP-2 ONNXに変換しよう

＜DRP-AI Translator環境＞



以下のSTEPで使用します。
- STEP-3 DRP-AI Translatorを使ってみよう

*1 PyTorch, the PyTorch logo and any related marks are trademarks of Facebook, Inc.

必要ファイル

本書で使用するソースコードはdarknet_yolo_ver7.20.tar.gzに含まれています。

- darknet_yolo_ver7.20
  - drpai_samples
    - onnx
    - yolov3_bmp
    - yolov2_bmp
    - tinyyolov3_bmp
    - tinyyolov2_bmp

    DRP-AI Translatorの実行に使用するファイル例です。
    STEP-3で使用します。
  - darknet
    - yolo

    ONNX変換スクリプトや後処理スクリプトです。
    STEP-2、STEP-4で使用します。

【補足】YOLOの学習環境

本ガイドではDeep Learningの学習工程については解説しません。

YOLOモデルを自作データセットで学習する場合、またはモデル構造をカスタマイズする場合は、Darknet公式サイト (https://pjreddie.com/darknet/yolo/)をご参照ください。

<モデル学習環境>

STEP-1まとめ

以上でYOLOモデルを実装するための基礎知識を学習することができました。

それでは実際にONNXモデルを作成してみましょう。

「STEP-2 ONNXに変換しよう」へ進みましょう。

Step内で作成する物
前stepの成果物

STEP-1

Introduction

STEP-2

ONNXに変換しよう

ONNX変換用スクリプト

学習済みモデル

STEP-3

DRP-AI Translatorを使ってみよう

変換用設定ファイル

ONNXモデル

STEP-4

推論実行してみよう

AI実行環境

DRP-AI Object files

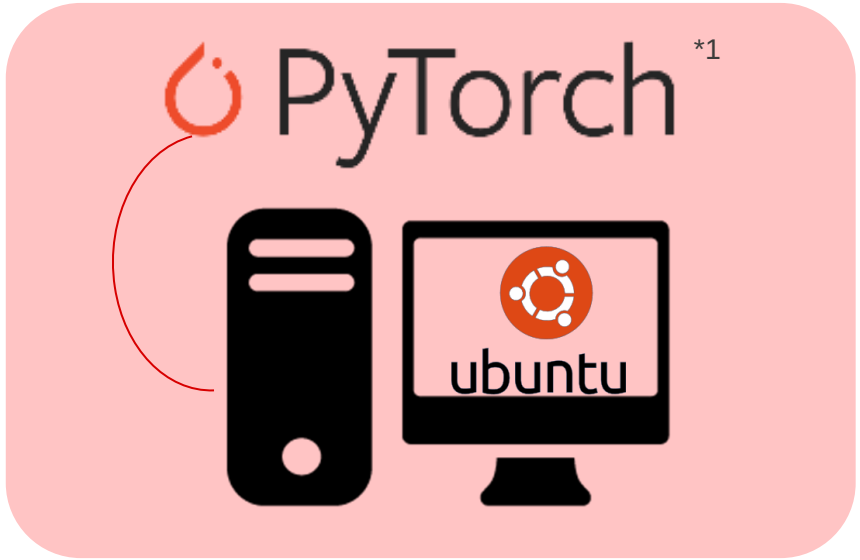
2.1: ONNXに変換するには？

Darknetフレームワークでは、事前学習済みモデルの構造とその重みパラメータを提供しています。

本章ではGet Startedドキュメントの「STEP-2 ONNXに変換しよう」を、
このDarknetのYOLOモデルを使用して解説します。使用する環境は下図の通りです。

現在、DarknetにはONNX変換機能が存在しないため、
本書ではPyTorchのフォーマットに変換後、PyTorchのONNX変換機能を使ってONNXモデルを作成します。

＜ONNX変換環境＞



事前学習済みのYOLOモデル

DarknetのYOLOモデル:
https://pjreddie.com/darknet/yolo/

PyTorch*1 ONNX

PyTorchからONNXへの変換

ONNX tutorials:
https://github.com/onnx/tutorials
PyTorch公式チュートリアル:
https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html

*1 PyTorch, the PyTorch logo and any related marks are trademarks of Facebook, Inc.

<ONNX変換環境>

2.2: ONNX変換環境を構築しよう

本章は、以下を前提としております。

- Get Startedドキュメントの「STEP-2.2: ONNX変換環境を構築しよう」を実施し、ONNX変換環境が構築済み。

以下を確認してください。

1. 環境変数が登録されているか確認してください。緑色は環境変数です。

```
$ printenv WORK
```

以下のように作業ディレクトリのパスが表示されたら、環境変数は登録できています。

```
<作業ディレクトリのパス>/rzv_ai_work
```

<ONNX変換環境>

2.3: 必要ファイルを準備しよう

本章は、以下を前提としております。

- Get Startedドキュメントの「STEP-2.3: 必要ファイルを準備しよう」を実施し、必要ファイルを展開済み。

本書で必要なファイルを準備します。

1. 作業ディレクトリへ移動します。緑色は環境変数です。

```
$ cd $WORK
```

2. tar.gzファイルを作業ディレクトリ下に解凍します。

```
$ tar xvzf <ファイルパス>/darknet_yolo_ver7.20.tar.gz -C $WORK
```

3. 作業ディレクトリ下に解凍できたか確認します。

```
$ ls $WORK
```

コマンド実行後、以下が表示されていることを確認してください。

```
drpai_samples    darknet
```

drpai_samplesディレクトリは、DRP-AI Translator用の各種サンプルコード及び実行例が格納されています。

darknetディレクトリは、Darknetモデルに対して使用するPyTorch用の各種サンプルコードが格納されています。

<ONNX変換環境>

2.3: 必要ファイルを準備しよう

以上の手順を実行後、各ディレクトリのファイル構成が以下のようになっていることを確認してください。

<ONNX変換の作業ディレクトリ($WORK)>

- rzv_ai_work
  - drpai_samples
    - onnx
      - d-yolov3.onnx
      - d-yolov2.onnx
      - d-tinyyolov3.onnx
      - d-tinyyolov2.onnx
    - yolov3_bmp
    - yolov2_bmp
    - tinyyolov3_bmp
    - tinyyolov2_bmp
    - addrmap_in_linux.yaml
  - darknet
    - yolo
      - convert_to_onnx.py
      - convert_to_pytorch.py
      - …
      - postprocess_yolo.py

<PyTorch環境>

2.4: AIモデルをONNX変換しよう

Neural Networkモデルのファイル形式はDeep Learningのフレームワークによって異なります。

現在、DarknetにはONNX変換機能がないため、

本書ではPyTorchのフォーマットに変換後、PyTorchのONNX変換機能を使ってONNXモデルを作成します。

Darknetフレームワークで作成されたNeural NetworkモデルのONNX変換は以下の流れで行います。

*1 PyTorch, the PyTorch logo and any related marks are trademarks of Facebook, Inc.

Darknetでは事前学習済みNeural Networkモデル構造と重みを提供しています。

これらを変換スクリプトを使ってPyTorch形式に変換し、ONNX形式に変換します。

<ONNX変換環境>

2.4: AIモデルをONNX変換しよう

Get Startedドキュメントの「STEP-2.4: AIモデルをONNX変換しよう」で解説している通り、

AIモデルをONNXフォーマットに変換するには、一般的にNNモデル構造とその重みパラメータが必要になります。

*1 PyTorch, the PyTorch logo and any related marks are trademarks of Facebook, Inc.

本書では、Darknetのモデル構造と重みパラメータをPyTorch形式に変換後、PyTorchの以下の関数でONNXフォーマットに変換します。

```
torch.onnx.export(model, …)
```

関数の詳細はPyTorchやONNXの公式チュートリアルをご参照ください。

ONNX tutorials: https://github.com/onnx/tutorials

PyTorch公式チュートリアル:
https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html

<ONNX変換環境>

2.4: AIモデルをONNX変換しよう

DarknetのYOLOモデルをONNXフォーマットに変換するには以下のファイルを使用します。

※すべてdarknet_yolo_ver7.20.tar.gzに含まれています。

名称	ファイル名	入手方法	用途
Darknet YOLOモデル構造＆設定	*.cfg	Darknet公式 yolov3: https://github.com/pjreddie/darknet/blob/master/cfg/yolov3.cfg yolov2: https://github.com/pjreddie/darknet/blob/master/cfg/yolov2-voc.cfg tinyyolov3: https://github.com/pjreddie/darknet/blob/master/cfg/yolov3-tiny.cfg tinyyolov2: https://github.com/pjreddie/darknet/blob/master/cfg/yolov2-tiny-voc.cfg	Darknet-PyTorch変換
Darknet YOLOモデルパラメータ (重み)	*.weights	Darknet公式 yolov3: https://pjreddie.com/media/files/yolov3.weights yolov2: https://pjreddie.com/media/files/yolov2-voc.weights tinyyolov3: https://pjreddie.com/media/files/yolov3-tiny.weights tinyyolov2: https://pjreddie.com/media/files/yolov2-tiny-voc.weights	
Darknet cfgファイルのパーサースクリプト	darknet_cfg.py	ルネサス提供	
Darknet-PyTorch変換スクリプト	convert_to_pytorch.py	ルネサス提供	
PyTorch用YOLOモデル構造	yolo.py	ルネサス提供 モデル構造参考: PyTorch-YOLOv3 https://github.com/eriklindernoren/PyTorch-YOLOv3	Darknet-PyTorch変換 PyTorch-ONNX変換
変換用設定ファイル	yolo.ini	ルネサス提供	
変換用設定ファイルのパーサースクリプト	read_ini.py	ルネサス提供	
PyTorch-ONNX変換スクリプト	convert_to_onnx.py	ルネサス提供	PyTorch-ONNX変換

Darknet-PyTorch 変換

PyTorch*1

PyTorch-ONNX 変換

ONNX

*1 PyTorch, the PyTorch logo and any related marks are trademarks of Facebook, Inc.

<ONNX変換環境>

2.4: AIモデルをONNX変換しよう

本書では変換スクリプトを実行する際に、以下の青字部分のようにモデル名を指定することで
各モデルを切り替えます。

※未指定の場合、YOLOv3モデルをデフォルト値として使用します。

```
$ python3 convert_to_*.py yolov3
```

パラメータ	モデル
yolov3	YOLOv3　（デフォルト値）
yolov2	YOLOv2
tinyyolov3	Tiny YOLOv3
tinyyolov2	Tiny YOLOv2

変換スクリプトではyolo.iniファイル内に記述されているファイル名やパラメータをロードして使用します。
以下のパラメータを右図のソースコードのように記述してご使用ください。

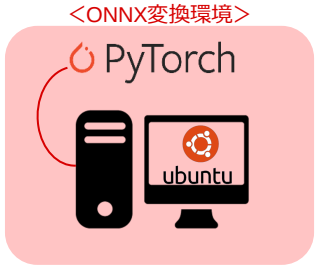
パラメータ	詳細
cfg	Darknet YOLOモデル構造＆設定ファイル名(*.cfg)
weights	Darknet YOLOモデルパラメータ (重み)ファイル名(*.weights)
pth	中間生成されるPyTorch形式重みファイル名。任意の名称を指定。
input	ONNXモデルの入力層名（STEP-3で使います）
output	ONNXモデルの出力層名（STEP-3で使います）
onnx	変換後のONNXモデルファイル名。任意の名称を指定。

<yolo.ini>

```
[yolov3]
cfg     =yolov3.cfg
weights =yolov3.weights
pth     =yolov3.pth
input   =["input1"]
output  =["output1", "output2", "output3"]
onnx    =d-yolov3.onnx

[yolov2]
cfg     =yolov2-voc.cfg
weights =yolov2-voc.weights
 :
```

2.4: AIモデルをONNX変換しよう



1. DarknetからPyTorchへの変換
    1. PyTorch変換用スクリプトを用意します。STEP-2.3を実施済みの場合、変換スクリプトの場所は以下です。
        ファイルパス： $WORK/darknet/yolo/convert_to_pytorch.py

<convert_to_pytorch.pyのソースコード>

```
    ⋮
11  def convert(darknet_cfg_path, darknet_weights_path, save_model_path, model):
12      # Load cfg file
13      cfg = DarknetConfig(darknet_cfg_path)
        ⋮
76      # Save pth file
77      torch.save(model.state_dict(), save_model_path)
78
79  if __name__ == '__main__':
80      # Load YOLO neural network parameters
81      ini = IniFile("yolo.ini")
82      model_dict = ini.model_dict
        ⋮
91      # Get the model information
92      model_params = ini.architecture[model_dict[model_name]].params
93
94      # Load YOLO neural network structure
95      model = yolo.Yolo(model_params.get('cfg'))
96      # Convert Darknet to PyTorch
97      convert(model_params.get('cfg'), model_params.get('weights'), model_params.get('pth'), model)
```

モデル構造取得

モデル構造＆設定　モデルパラメータ（重み）　変換後のpthファイル名

2.4: AIモデルをONNX変換しよう

Darknet-PyTorch

PyTorch*1

*1 PyTorch, the PyTorch logo and any related marks are trademarks of Facebook, Inc.

<ONNX変換環境>

2. ONNX変換用の作業ディレクトリへ移動します。

```
$ cd $WORK/darknet/yolo
```

3. スクリプトを実行して、DarknetのYOLOモデルをpthファイル (PyTorchフォーマット) へ変換します。

以下はYOLOv3モデルを変換する際のコマンドです。青字部分は変換対象モデル名です。

```
$ python3 convert_to_pytorch.py yolov3
```

エラーが無く、以下のように表示されていれば変換成功です。

```
...
REST 0
```

4. darknetディレクトリにpthファイルができているか確認します。

```
$ ls $WORK/darknet/yolo
```

以下のようにpthファイルがあることを確認してください。
今回はYOLOv3モデルを変換したので、yolov3.pthが生成されています。

```
convert_to_pytorch.py convert_to_onnx.py yolov3.pth
```

2.4: AIモデルをONNX変換しよう

PyTorch*1 PyTorch-ONNX ONNX

*1 PyTorch, the PyTorch logo and any related marks are trademarks of Facebook, Inc.

2. PyTorchからONNXフォーマットへの変換
    1. ONNX変換用スクリプトを作成します。STEP-2.3を実施済みの場合、変換スクリプトの場所は以下です。
        ファイルパス： $WORK/darknet/yolo/convert_to_onnx.py

<convert_to_onnx.pyのソースコード>

```
      ⋮
if __name__ == "__main__":
    # Load YOLO neural network parameters
    ini = IniFile("yolo.ini")
    model_dict = ini.model_dict
      ⋮
    # Get the model information
    model_params = ini.architecture[model_dict[model_name]].params

    # Initialise one random value filled input tensor of an image size 3x416x416 (CHW)
    dummy_input = torch.randn(1, 3, 416, 416)

    # Loads from YOLO neural network structure
    model = yolo.Yolo(model_params.get('cfg'))
    model.load_state_dict(torch.load(model_params.get('pth')))
      ⋮
    # Define the input tensor and output tensor name of the converted onnx neural network
    input_names = model_params.get('input')
    output_names = model_params.get('output')

    # Starts the pythorch to onnx conversion
    torch.onnx.export(model, dummy_input, model_params.get('onnx'), verbose=True, opset_version=12, input_names=input_names, output_names=output_names)
```

モデルへの入力サイズ (N, C, H, W)

モデル指定

モデルのパラメータ（重み）指定

モデルの先頭レイヤへの入力名（STEP-3で使います）

モデルの最終レイヤの出力名（STEP-3で使います）

変換後のONNXファイル名

2.4: AIモデルをONNX変換しよう

PyTorch*1 PyTorch-ONNX ONNX

*1 PyTorch, the PyTorch logo and any related marks are trademarks of Facebook, Inc.

2. ONNX変換用の作業ディレクトリへ移動します。

```
$ cd $WORK/darknet/yolo
```

3. スクリプトを実行して、手順１で作成したPyTorchのYOLOモデルをonnxファイル へ変換します。
以下はYOLOv3モデルを変換する際のコマンドです。青字部分は変換対象モデル名です。

```
$ python3 convert_to_onnx.py yolov3
```

エラーが無く、以下のように表示されていれば変換成功です。
※モデルによって、Warningが表示されることがありますが、動作に問題はありません。

```
 ...

 return (%output1, %output2, %output3)
```

4. darknetディレクトリにonnxファイルができているか確認します。

```
$ ls $WORK/darknet/yolo
```

以下のようにonnxファイルがあることを確認してください。
今回はYOLOv3モデルを変換したため、d-yolov3.onnxが生成されています。

```
convert_to_pytorch.py convert_to_onnx.py d-yolov3.onnx yolov3.pth
```

STEP-2まとめ

以上でDarknetが提供する事前学習済みYOLOモデルをONNXフォーマットに変換することができました。

次は作成したONNXモデルをDRP-AI Translatorを使って変換してみましょう。

「STEP-3 DRP-AI Translatorを使ってみよう」へ進みましょう。

Step内で作成する物

前stepの成果物

STEP-1

Introduction

STEP-2

ONNXに
変換しよう

ONNX変換用スクリプト

学習済みモデル

STEP-3

**DRP-AI
Translatorを
使ってみよう**

変換用設定ファイル

ONNXモデル

STEP-4

推論実行
してみよう

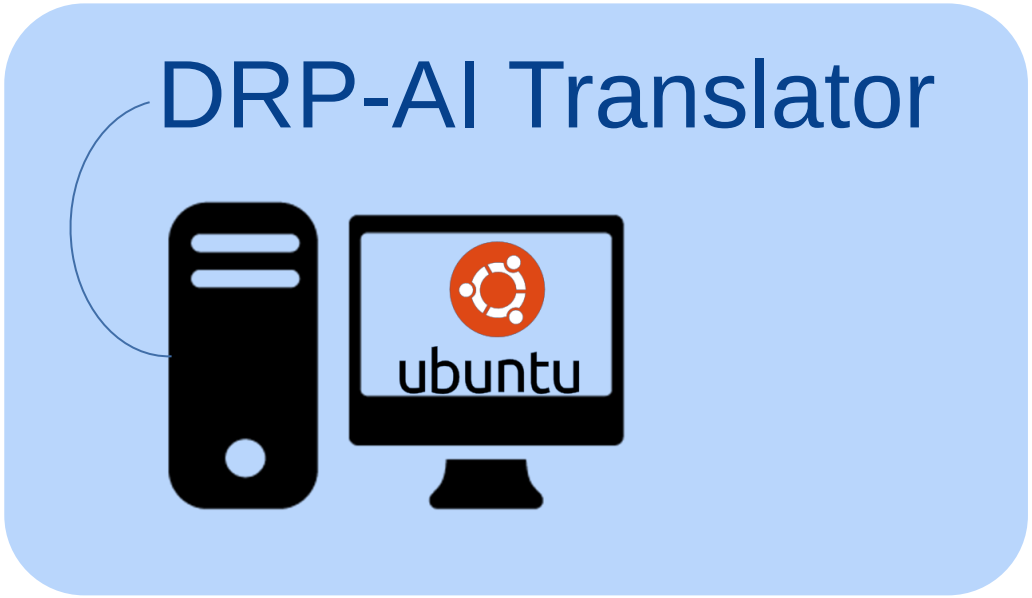
AI実行環境

DRP-AI Object files

DRP-AI Translatorを使ってみよう

STEP-2で作成したONNXフォーマットのモデルをDRP-AI Object filesへ変換してみましょう。

＜DRP-AI Translator環境＞

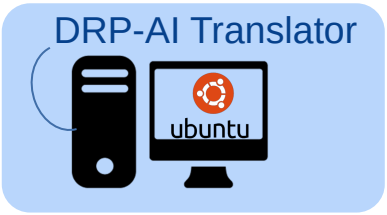


＋

ONNX

STEP-2で作成した
ONNXフォーマットのモデル

<DRP-AI Translator環境>



3.1: DRP-AI Translator環境を構築しよう

本章は、以下を前提としております。

- Get Startedドキュメントの「STEP-3.1: DRP-AI Translator環境を構築しよう」を実施し、DRP-AI Translator環境を構築済み。

DRP-AI Translator環境が構築されているかどうか、以下の手順でご確認ください

1. 作業ディレクトリの環境変数が登録されているか確認してください。

```
$ printenv WORK
```

以下のように作業ディレクトリのパスが表示されたら、環境変数は登録できています。

```
<作業ディレクトリのパス>/rzv_ai_work
```

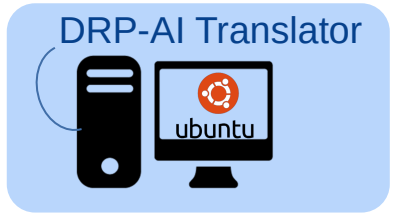
2. DRP-AI Translator作業ディレクトリの環境変数が登録されているか確認してください。

```
$ printenv DRPAI
```

以下のようにDRP-AI Translator作業ディレクトリのパスが表示されたら、環境変数として登録できています。

```
<$WORKに指定したパス>/drp-ai_translator_release
```

<DRP-AI Translator環境>



3.2: ファイル構成を確認しよう

各ディレクトリの構成が以下のようになっていることを確認してください。

<ONNX変換の作業ディレクトリ($WORK)>

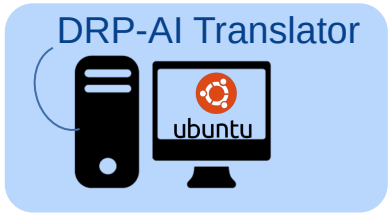
- rzv_ai_work
  - drpai_samples
    - onnx
      - d-yolov3.onnx
      - ⋮
    - yolov3_bmp
      - ⋮
    - addrmap_in_linux.yaml
  - darknet
    - yolo
  - drp-ai_translator_release

<DRP-AI Translatorの作業ディレクトリ($DRPAI)>

- drp-ai_translator_release
  - DRP-AI_translator
  - onnx
  - output
  - UserConfig
  - run_DRP-AI_translator_V2M.sh
  - run_DRP-AI_translator_V2L.sh

DRP-AI Translatorフォルダの詳細についてはGet Startedドキュメントをご参照ください。

<DRP-AI Translator環境>



3.3: ONNXファイルを用意しよう

本章では、 DRP-AI Translatorを実行するために必要なONNXファイルを用意していきます。

例として、YOLOv3モデルを使用します。適宜ONNXファイル名を読み替えて実行してください。

- drp-ai_translator_release
  - …
  - onnx
    - d-yolov3.onnx
  - UserConfig
    - addrmap_in_d-yolov3.yaml
    - prepost_d-yolov3.yaml

1. STEP-2で作成したonnxモデルをDRP-AI Translatorのonnxディレクトリへコピーします。

```
$ cp -v $WORK/darknet/yolo/d-yolov3.onnx $DRPAI/onnx/
```

2. onnxディレクトリにコピーされたか確認します。

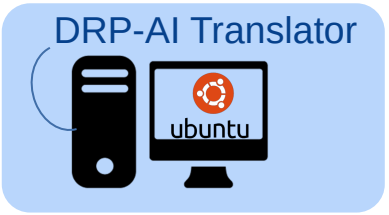
```
$ ls $DRPAI/onnx/
```

以下のようにd-yolov3.onnxがあることを確認してください。

```
d-yolov3.onnx  tiny_yolov2.onnx  yolov2.onnx
```

└これら2ファイルはもともとDRP-AI Translatorに同梱されているモデルです。

<DRP-AI Translator環境>



3.4: アドレスマップ定義ファイルを用意しよう

本章では、 DRP-AI Translatorを実行するために必要なアドレスマップ定義ファイルを用意していきます。

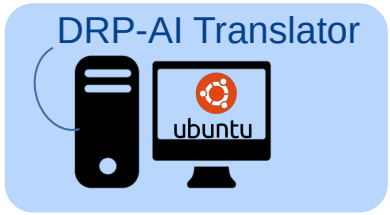
例として、YOLOv3モデルを使用します。適宜ONNXファイル名を読み替えて実行してください。

- drp-ai_translator_release
  - …
  - onnx
    - d-yolov3.onnx
  - UserConfig
    - addrmap_in_d-yolov3.yaml
    - prepost_d-yolov3.yaml

本章は実際に実行する手順のみを記載しております。

DRP-AI Object filesを格納する領域の先頭アドレスを書き換える必要があります。

詳細については、 Get Startedドキュメントの「STEP-3.4: アドレスマップ定義ファイルを用意しよう」をご参照ください。



3.4: アドレスマップ定義ファイルを用意しよう

アドレスマップ定義ファイルは、rzv_ai-implementation-guide_ver7.20.tar.gzにてご提供している addrmap_in_linux.yamlを使用します。

このファイルをアドレスマップ定義ファイルの命名ルールに従ってファイル名を変更します。

d-yolov3.onnx

addrmap_in_d-yolov3.yaml

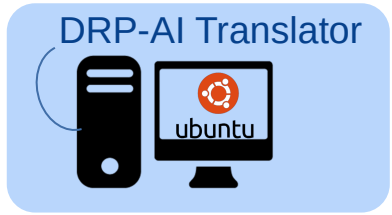
1. addrmap_in_linux.yamlをdrp-ai_translator_release/UserConfigディレクトリへコピーします。

```
$ cp -v $WORK/drpai_samples/addrmap_in_linux.yaml $DRPAI/UserConfig
```

2. addrmap_in_linux.yamlをaddrmap_in_d-yolov3.yamlに名前を変更します。

```
$ cd $DRPAI/UserConfig
```

```
$ mv -v ./addrmap_in_linux.yaml ./addrmap_in_d-yolov3.yaml
```

<＜DRP-AI Translator環境＞


3.5: 前後処理定義ファイルを用意しよう

本章では、 DRP-AI Translatorを実行するために必要な前後処理定義ファイルを用意していきます。

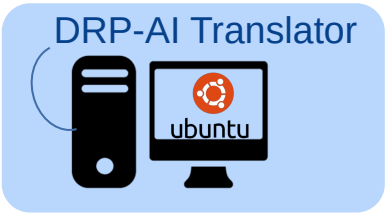
例として、YOLOv3モデルを使用します。適宜ONNXファイル名を読み替えて実行してください。

- drp-ai_translator_release
  - …
  - onnx
    - d-yolov3.onnx
  - UserConfig
    - addrmap_in_d-yolov3.yaml
    - prepost_d-yolov3.yaml

本章はYOLOモデルに関する説明及び手順のみを記載しております。

前後処理定義ファイルの詳細については、 Get Startedドキュメントの
「STEP-3.5: 前後処理定義ファイルを用意しよう」をご参照ください。

<DRP-AI Translator環境>



3.5: 前後処理定義ファイルを用意しよう

DRP-AI Translatorに同梱されているサンプルを元に、前後処理定義ファイルを作成していきましょう。

前後処理定義ファイルのサンプルは、./UserConfig/sampleディレクトリの中にあります。（下図参照）

- drp-ai_translator_release
  - DRP-AI_translator
    - …
  - UserConfig
    - sample
      - prepost_resnet50v1.yaml
      - prepost_tiny_yolov2.yaml
      - prepost_vgg16.yaml
      - prepost_yolov2.yaml

今回は、Darknet-PyTorchのYOLOを変換するので、
前後処理が似ている
prepost_tiny_yolov2.yamlをカスタマイズしていきましょう。

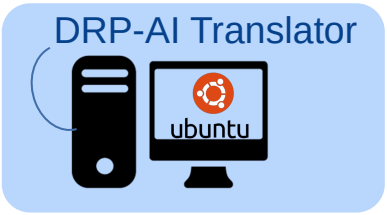
カスタマイズしたyamlファイルは、
全てUserConfigディレクトリ直下に置いてください。

1. prepost_tiny_yolov2.yamlをUserConfigディレクトリ直下へコピーします。

```
$ cd $DRPAI/UserConfig
```

```
$ cp -v ./sample/prepost_tiny_yolov2.yaml ./
```

<DRP-AI Translator環境>



3.5: 前後処理定義ファイルを用意しよう

DRP-AI TranslatorはONNXモデルファイル名をもとに前後処理定義ファイルを指定するため、
前後処理定義ファイル名を変更します。

d-yolov3.onnx

prepost_d-yolov3.yaml

2. prepost_tiny_yolov2.yamlをprepost_d-yolov3.yamlに名前を変更します。

```
$ cd $DRPAI/UserConfig
```

```
$ mv -v ./prepost_tiny_yolov2.yaml ./prepost_d-yolov3.yaml
```

3.5: 前後処理定義ファイルを用意しよう

DarknetのYOLOモデルはそれぞれ出力形式が異なるため、以下の出力形式表に合わせて適宜前後処理定義ファイルのoutput_from_body欄と関連項目をご用意ください。

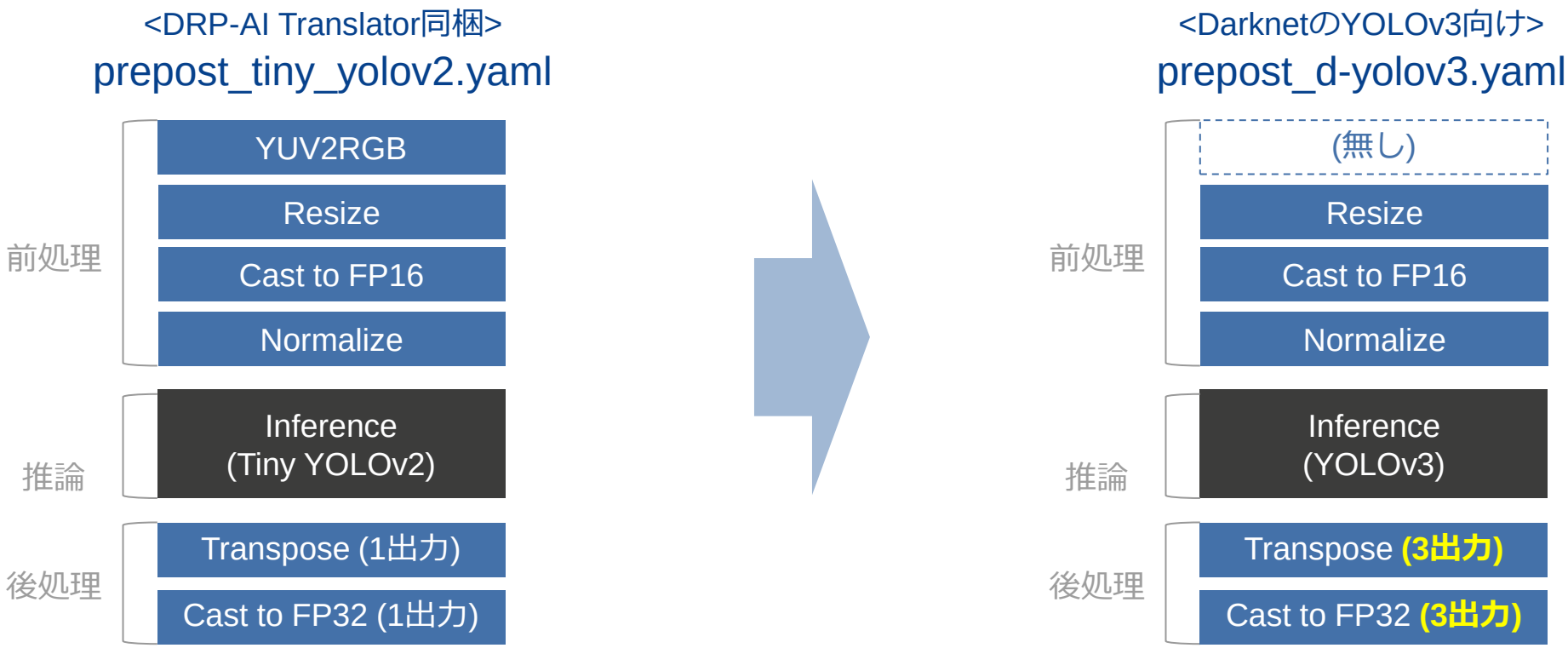
出力形式以外の項目については、本章でご説明しているYOLOv3の前後処理定義ファイルと同じように修正してください。

モデル	name	shape (HWC)
YOLOv3	output1 output2 output3	[13, 13, 255] [26, 26, 255] [52, 52, 255]
YOLOv2	output1	[13, 13, 125]
Tiny YOLOv3	output1 output2	[13, 13, 255] [26, 26, 255]
Tiny YOLOv2	output1	[13, 13, 125]

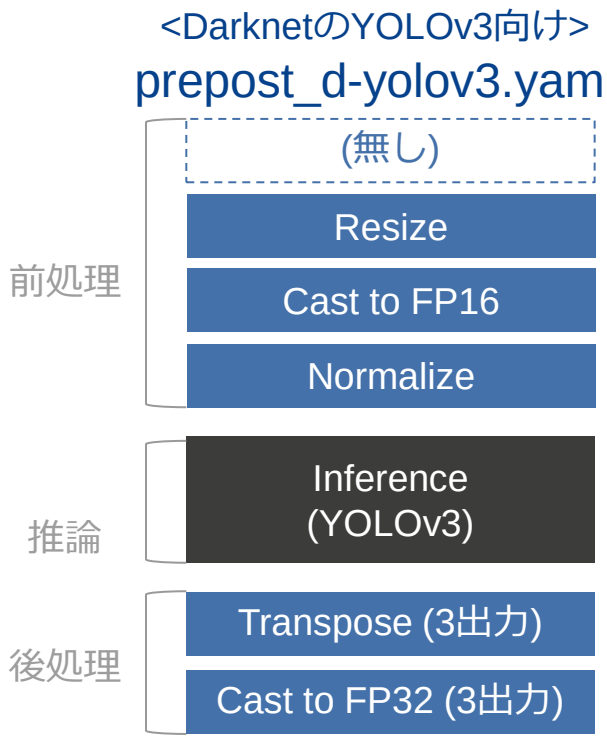
※nameはSTEP-2のONNX変換時の名称に依存します。

3.5: 前後処理定義ファイルを用意しよう

DRP-AI Translatorに同梱されているサンプルの前後処理定義ファイルは左下図のような前後処理になっています。
これを、DarknetのYOLO向けに書き換えていきましょう。今回は例としてYOLOv3を使用します。
また、サンプルでは入力データ形式がYUY2に設定されているので、BGR形式に変更してみましょう。



3.5: 前後処理定義ファイルを用意しよう



前後処理定義ファイルを左図の形にするために、4つの情報を書き換えます。

(1) 出力データの定義をモデルの出力データに合わせる。

(2) モデルの入力データ名と出力データ名を、STEP-2のONNX変換時に命名したモデルの入出力レイヤ名に合わせる

(3) 前処理への入力データフォーマットをYUY2からBGRにする

(4) 前処理のNormalizeのパラメータ値を学習時の設定に合わせる

書き換える情報と各定義の対応は下図のとおりです。

prepost_d-yolov3.yamlの構成

入力データの定義	(2), (3)
出力データの定義	(1), (2)
前処理の定義	(2), (3), (4)
後処理の定義	(1), (2)

3.5: 前後処理定義ファイルを用意しよう

まずは、「 出力データの定義をモデルの出力データに合わせる」をしましょう。

YOLOv3モデルでは出力層は３つあり、以下のように各出力サイズを計算することができます。

$[\ H\ x\ W\ x\ (3*(4\ +\ 1\ +\ N_{class}))]$

N_{class} = 検出クラス数

H, W = 各出力層の高さと幅

※モデル出力情報の詳細については以下をご参照ください。

YOLOv3: An Incremental Improvement https://arxiv.org/pdf/1804.02767.pdf

YOLOv3: Real-Time Object Detection (Joseph Redmon) https://pjreddie.com/darknet/yolo/

本書で使用しているDarknetが提供するYOLOv3モデルでは、以下のように演算結果を出力します。

出力1: [13 x 13 x (3*(4 + 1 + 80))] ⇒ [13 x 13 x 255]

出力2: [26 x 26 x (3*(4 + 1 + 80))] ⇒ [26 x 26 x 255]

出力3: [52 x 52 x (3*(4 + 1 + 80))] ⇒ [52 x 52 x 255]

これらの出力情報に合わせて前後処理定義ファイルを変更します。

※Tiny YOLOv3を使用する場合は、適宜出力数とその名称及び形式を合わせてください。

YOLOv2とTiny YOLOv2を使用する場合、出力数が１であるため、本手順は不要です。

3.5: 前後処理定義ファイルを用意しよう

1. エディタでprepost_d-yolov3.yamlを開きます。

```
$ vi $DRPAI/UserConfig/prepost_d-yolov3.yaml
```

2. 出力データの定義部分にあるモデルからの出力データを変更します。
   モデルからの出力形式を前述のYOLOv3モデル出力情報に合わせます。

```
23 output_from_body:
24   -
25     name: "grid"
26     shape: [13, 13, 125]
27     order: "HWC"
28     type: "fp16"
```

出力データをコピー

```
23 output_from_body:
24   -
25     name: "grid"
26     shape: [13, 13, 125]
27     order: "HWC"
28     type: "fp16"
29   -
30     name: "grid"
31     shape: [13, 13, 125]
32     order: "HWC"
33     type: "fp16"
34   -
35     name: "grid"
36     shape: [13, 13, 125]
37     order: "HWC"
38     type: "fp16"
```

モデルからの出力形式 (出力1)
モデルからの出力形式 (出力2)
モデルからの出力形式(出力3)

```
23 output_from_body:
24   -
25     name: "grid"
26     shape: [13, 13, 255]
27     order: "HWC"
28     type: "fp16"
29   -
30     name: "grid"
31     shape: [26, 26, 255]
32     order: "HWC"
33     type: "fp16"
34   -
35     name: "grid"
36     shape: [52, 52, 255]
37     order: "HWC"
38     type: "fp16"
```

※モデルからの出力データ名の変更方法については「モデルの入力データ名と出力データ名を、STEP-2のONNX変換時に命名したモデルの入出力レイヤ名に合わせる」で解説します。

3.5: 前後処理定義ファイルを用意しよう

3. 出力データの定義部分にある後処理からの出力データを変更します。

また、後処理からの出力形式を前述のYOLOv3モデル出力情報に合わせます。

```
40 output_from_post:
41   -
42     name: "post_out"
43     shape: [125, 13, 13]
44     order: "CHW"
45     type: "fp32"
```

出力データをコピー

```
40 output_from_post:
41   -
42     name: "post_out"
43     shape: [125, 13, 13]
44     order: "CHW"
45     type: "fp32"
46   -
47     name: "post_out"
48     shape: [125, 13, 13]
49     order: "CHW"
50     type: "fp32"
51   -
52     name: "post_out"
53     shape: [125, 13, 13]
54     order: "CHW"
55     type: "fp32"
```

後処理からの出力名と出力形式(出力1)

後処理からの出力名と出力形式(出力2)

後処理からの出力名と出力形式(出力3)

```
40 output_from_post:
41   -
42     name: "post_out1"
43     shape: [255, 13, 13]
44     order: "CHW"
45     type: "fp32"
46   -
47     name: "post_out2"
48     shape: [255, 26, 26]
49     order: "CHW"
50     type: "fp32"
51   -
52     name: "post_out3"
53     shape: [255, 52, 52]
54     order: "CHW"
55     type: "fp32"
```

※1 本項で指定した後処理からの出力名は後述の手順でも使用します。

※2 後処理からの出力名は任意の名称を使用可能ですが、各出力ごとに異なる名称をご使用ください。

3.5: 前後処理定義ファイルを用意しよう

4. 後処理の定義部分にあるモデルからの出力データを3出力に変更します。

```
94  postprocess:
95    -
96      src: ["grid"]
97
98      dest: ["post_out"]
99
100     operations:
101       -
102         op : transpose
103         param:
104           WORD_SIZE: 1     # 2Byte
105           IS_CHW2HWC: 0    # HWC to CHW
106
107       -
108         op : cast_fp16_fp32
109         param:
110           CAST_MODE: 0 # FP16 to FP32
```

3出力になるよう
後処理定義全体をコピー

```
94  postprocess:
95    -
96      src: ["grid"]
  ⋮       ⋮
107       -
108         op : cast_fp16_fp32
109         param:
110           CAST_MODE: 0 # FP16 to FP32
111
112   -
113     src: ["grid"]
  ⋮       ⋮
124       -
125         op : cast_fp16_fp32
126         param:
127           CAST_MODE: 0 # FP16 to FP32
128
129   -
130     src: ["grid"]
  ⋮       ⋮
141       -
142         op : cast_fp16_fp32
143         param:
144           CAST_MODE: 0 # FP16 to FP32
145
```

出力1
出力2
出力3

3.5: 前後処理定義ファイルを用意しよう

5. 手順4で追加した、後処理の定義部分にある後処理からの出力データ名を変更します。

```
94  postprocess:
95    -
96      src: ["grid"]
97
98      dest: ["post_out"]
      :       :
112   -
113     src: ["grid"]
114
115     dest: ["post_out"]
      :       :
129   -
130     src: ["grid"]
131
132     dest: ["post_out"]
```

後処理からの出力名(出力1)※1

後処理からの出力名(出力2)※1

後処理からの出力名(出力3)※1

```
94  postprocess:
95    -
96      src: ["grid"]
97
98      dest: ["post_out1"]
      :       :
112   -
113     src: ["grid"]
114
115     dest: ["post_out2"]
      :       :
129   -
130     src: ["grid"]
131
132     dest: ["post_out3"]
```

※1 後処理からの出力名は手順3で指定した名称を指定してください。

※2 モデルからの出力名の変更方法については「モデルの入力データ名と出力データ名を、STEP-2のONNX変換時に命名したモデルの入出力レイヤ名に合わせる」で解説します。

3.5: 前後処理定義ファイルを用意しよう

次に、「モデルの入力データ名と出力データ名を、STEP-2のONNX変換時に命名したモデルの入出力レイヤ名に合わせる」をしましょう。

「 STEP-2.4: AIモデルをONNX変換してみよう」で、convert_to_onnx.pyを使ってモデルの先頭レイヤへの入力名と最終レイヤの出力名を定義してONNX変換しました。

<convert_to_onnx.pyのソースコード>

```
      ⋮
30    # Define the input tensor and output tensor name of the converted onnx neural network
31    input_names = model_params.get('input')
32    output_names = model_params.get('output')
33
34    # Starts the pythorch to onnx conversion
35    torch.onnx.export(model, dummy_input, model_params.get('onnx'), verbose=True, opset_version=12, input_names=input_names, output_names=output_names)
```

モデルの先頭レイヤへの入力名（STEP-3で使います）

モデルの最終レイヤの出力名（STEP-3で使います）

実際の値はyolo.iniファイル内で記述されています。

この入力名（input1）と出力名（output1, output2, output3）になるように、各定義を書き換えていきます。

<yolo.ini>

```
1  [yolov3]
2  cfg     =yolov3.cfg
3  weights =yolov3.weights
4  pth     =yolov3.pth
5  input   =["input1"]
6  output  =["output1", "output2", "output3"]
7  onnx    =d-yolov3.onnx
8
9   ⋮
```

ここで
入力名：input1
出力名：output1, output2, output3
を定義しています。

3.5: 前後処理定義ファイルを用意しよう

6. 入力データの定義部分にあるモデルへの入力名を書き換えます。

```
12 input_to_body:
13   -
14     name: "image2"
15     format: "RGB"
16     order: "HWC"
17     shape: [416, 416, 3]
18     type: "fp16"
```

モデルへの入力名

```
12 input_to_body:
13   -
14     name: "input1"
15     format: "RGB"
16     order: "HWC"
17     shape: [416, 416, 3]
18     type: "fp16"
```

7. 前処理の定義部分にあるモデルへの入力名を書き換えます。

```
60 preprocess:
61   -
62     src      : ["camera_data"]
63
64     dest     : ["image2"]
```

モデルへの入力名

```
60 preprocess:
61   -
62     src      : ["camera_data"]
63
64     dest     : ["input1"]
```

3.5: 前後処理定義ファイルを用意しよう

8. 出力データの定義部分にあるモデルからの出力名を書き換えます。

```
23 output_from_body:
24   -
25     name: "grid"
26     shape: [13, 13, 255]
27     order: "HWC"
28     type: "fp16"
29   -
30     name: "grid"
31     shape: [26, 26, 255]
32     order: "HWC"
33     type: "fp16"
34   -
35     name: "grid"
36     shape: [52, 52, 255]
37     order: "HWC"
38     type: "fp16"
```

モデルからの出力名(出力1)

モデルからの出力名(出力2)

モデルからの出力名(出力3)

```
23 output_from_body:
24   -
25     name: "output1"
26     shape: [13, 13, 255]
27     order: "HWC"
28     type: "fp16"
29   -
30     name: "output2"
31     shape: [26, 26, 255]
32     order: "HWC"
33     type: "fp16"
34   -
35     name: "output3"
36     shape: [52, 52, 255]
37     order: "HWC"
38     type: "fp16"
```

3.5: 前後処理定義ファイルを用意しよう

9.後処理の定義部分にあるモデルからの出力名を書き換えます。

```
94  postprocess:
95    -
96      src: ["grid"]
97
98      dest: ["post_out1"]
        :
112   -
113     src: ["grid"]
114
115     dest: ["post_out2"]
        :
129   -
130     src: ["grid"]
131
132     dest: ["post_out3"]
```

モデルからの出力名(出力1)

モデルからの出力名(出力2)

モデルからの出力名(出力3)

```
94  postprocess:
95    -
96      src: ["output1"]
97
98      dest: ["post_out1"]
        :
112   -
113     src: ["output2"]
114
115     dest: ["post_out2"]
        :
129   -
130     src: ["output3"]
131
132     dest: ["post_out3"]
```

3.5: 前後処理定義ファイルを用意しよう

次に、「前処理への入力データフォーマットをYUY2からBGRにする」をしましょう。

10. 入力データの定義部分にある前処理への入力データフォーマットを書き換えます。

```
4  input_to_pre:
5    -
6      name: "camera_data"
7      format: "YUY2"
8      order: "HWC"
9      shape: [480, 640, 2]
10     type: "uint8"
```

入力名 → `name: "bgr_data"`
フォーマット名 → `format: "BGR"`
Channel数 → `shape: [480, 640, 3]`

```
4  input_to_pre:
5    -
6      name: "bgr_data"
7      format: "BGR"
8      order: "HWC"
9      shape: [480, 640, 3]
10     type: "uint8"
```

11. 前処理の定義部分にある前処理への入力データフォーマットを書き換えます。

```
60 preprocess:
61   -
62     src       : ["camera_data"]
```

入力名 →

```
60 preprocess:
61   -
62     src       : ["bgr_data"]
```

3.5: 前後処理定義ファイルを用意しよう

12. 前処理の定義部分にあるconv_yuv2rgbオペレーション（YUV2RGB変換処理）をコメントアウトします。

```
67  -
68      op: conv_yuv2rgb
69      param:
70        DOUT_RGB_FORMAT: 0 # "RGB"
```

```
67  # -
68  #     op: conv_yuv2rgb
69  #     param:
70  #       DOUT_RGB_FORMAT: 0 # "RGB"
```

13. 前処理の定義部分にあるnormalizeオペレーション（正規化処理）のパラメータを書き換えます。

```
84  -
85      op: normalize
86      param:
87        DOUT_RGB_ORDER: 0 # Output RGB order = Input RGB order
88        cof_add: [0.0, 0.0, 0.0]
89        cof_mul: [0.00392157, 0.00392157, 0.00392157]
```

normalize処理の出力形式

```
84  -
85      op: normalize
86      param:
87        DOUT_RGB_ORDER: 1
88        cof_add: [0.0, 0.0, 0.0]
89        cof_mul: [0.00392157, 0.00392157, 0.00392157]
```

3.5: 前後処理定義ファイルを用意しよう

次に、「前処理のNormalizeのパラメータ値を学習時の設定に合わせる」をしましょう。

Darknetが提供する学習済みモデルは、meanとstdによる正規化処理は不要ですが、
入力データを0~1の浮動小数に変換する必要があります。

そのため、今回は引数 **mean=[0, 0, 0]** と **std=[1, 1, 1]**を使って、 Normalizeパラメータを設定します。

計算式はGet Startedドキュメントの「STEP-3.5: 前後処理定義ファイルを用意しよう」に記載のものを使用します。

add = - (mean × range)

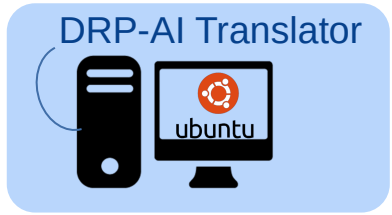
mul = 1 / (std × range)

モデルへの入力を0～1の浮動小数へ正規化するため、
rangeは入力画像のダイナミックレンジの最大値255を
使用します。

14. Normalizeオペレーションの引数であるcof_addとcof_mulを設定します。（今回はsampleと同じ数字になりました）

```
84  -
85      op: normalize
86      param:
87        DOUT_RGB_ORDER: 1
88        cof_add: [0.0, 0.0, 0.0]
89        cof_mul: [0.00392157, 0.00392157, 0.00392157]
```

```
84  -
85      op: normalize
86      param:
87        DOUT_RGB_ORDER: 1
88        cof_add: [0.0, 0.0, 0.0]
89        cof_mul: [0.00392157, 0.00392157, 0.00392157]
```



3.6: DRP-AI Translatorで変換してみよう

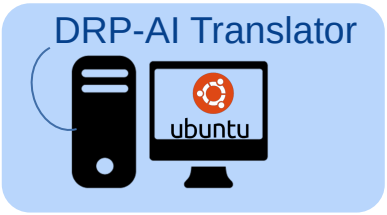
それでは実際にDRP-AI Translatorで変換してみましょう。

例として、YOLOv3モデルを使用します。適宜ONNXファイル名を読み替えて実行してください。

1. drp-ai_translator_releaseディレクトリに以下のように３つのファイルがあることを確認してください。

- drp-ai_translator_release
  - …
  - onnx
    - d-yolov3.onnx
  - UserConfig
    - addrmap_in_d-yolov3.yaml
    - prepost_d-yolov3.yaml

<DRP-AI Translator環境>



3.6: DRP-AI Translatorで変換してみよう

2. DRP-AI Translatorの作業ディレクトリへ移動します。

```
$ cd $DRPAI
```

3. 以下のコマンドでDRP-AI Translatorを実行します。（必ずdrp-ai_translator_releaseディレクトリ上で実施してください）
青字はPREFIXなので、任意の名前を指定してください。

RZ/V2M、 RZ/V2MAの場合:

```
$ ./run_DRP-AI_translator_V2M.sh yolov3 -onnx ./onnx/d-yolov3.onnx
```

RZ/V2Lの場合：

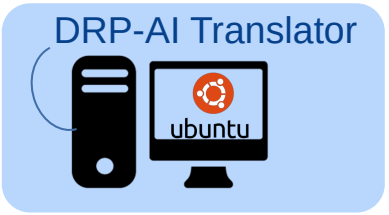
```
$ ./run_DRP-AI_translator_V2L.sh yolov3 -onnx ./onnx/d-yolov3.onnx
```

エラーが無く、以下のように表示されていれば変換成功です。

```
[Run DRP-AI Translator] Ver. 1.80
[Input file information]
   PREFIX               : yolov3
   ONNX Model           : ./onnx/d-yolov3.onnx
   Prepost file         : ./UserConfig/prepost_d-yolov3.yaml
   Address mapping file : ./UserConfig/addrmap_in_d-yolov3.yaml
      ...

-------------------------------------------
[Converter for DRP] Finish
```

<DRP-AI Translator環境>



3.7: 変換結果を確認してみよう

1. 変換結果は、outputディレクトリの中の変換時に付けたPREFIX名が付いたディレクトリに格納されています。

```
$ ls -1 $DRPAI/output/yolov3/
```

正しく変換されていれば、以下のように表示されます。

```
aimac_desc.bin
drp_desc.bin
drp_lib_info.txt
drp_param.bin
drp_param.txt
drp_param_info.txt
yolov3.json
yolov3_addrmap_intm.txt
yolov3_addrmap_intm.yaml
yolov3_data_in_list.txt
yolov3_data_out_list.txt
yolov3_drpcfg.mem
yolov3_prepost_opt.yaml
yolov3_summary.xlsx
yolov3_tbl_addr_data.txt
yolov3_tbl_addr_data_in.txt
yolov3_tbl_addr_data_out.txt
yolov3_tbl_addr_drp_config.txt
yolov3_tbl_addr_merge.txt
yolov3_tbl_addr_weight.txt
yolov3_tbl_addr_work.txt
yolov3_weight.dat
```

黄色のファイルが
実機動作時に必要な
DRP-AI Object filesです。

緑色のファイルは
処理時間の目安を見積もるときに
使用するサマリファイルです。

STEP-3まとめ

以上でONNXフォーマットからDRP-AI Object filesへ変換することができました。

DRP-AI Object filesが作成できたら、「STEP-4 推論実行してみよう」へ進みましょう。

Step内で作成する物

前stepの成果物

STEP-1

Introduction

STEP-2

ONNXに
変換しよう

ONNX変換用スクリプト

学習済みモデル

STEP-3

DRP-AI
Translatorを
使ってみよう

変換用設定ファイル

ONNXモデル

STEP-4

**推論実行
してみよう**

AI実行環境

DRP-AI Object files

推論実行してみよう

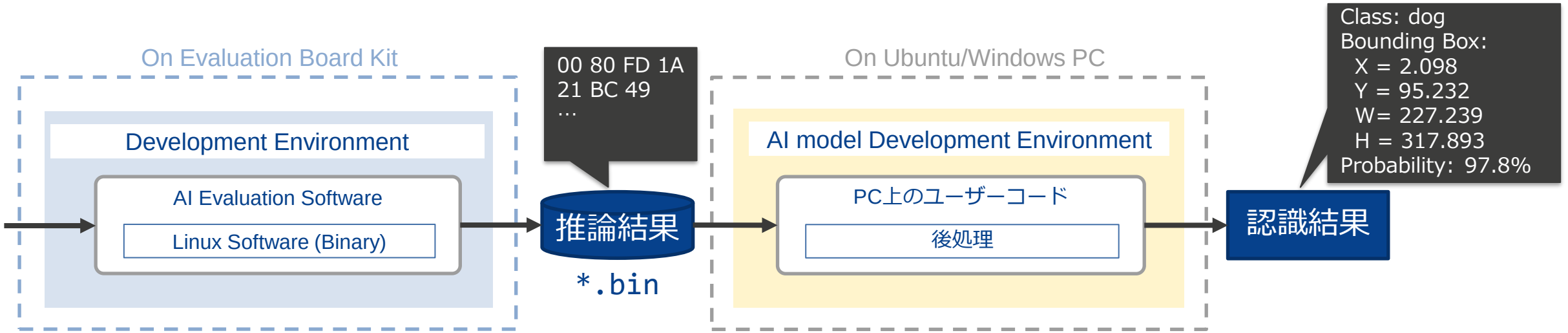
本章は、以下を前提としております。

- Get Startedドキュメントの「STEP-4 推論実行してみよう」を確認済み。

STEP-1
STEP-2
STEP-3
STEP-4

STEP-3で作成したYOLO用DRP-AI Object filesはAI Evaluation Software でそのまま動かすことができます。

AI Evaluation Softwareを実行するとDRP-AIの演算結果が書かれたバイナリファイルが出力されますが、
このファイルでは認識結果を見ることはできません。認識結果を得るにはCPUによるYOLO用の後処理が必要です。



※AI Evaluation Software の使い方はAI Evaluation Software Guideをご参照ください。

推論実行してみよう

本書では、Darknetの提供するYOLOモデル用の後処理をLinux PC上で実行するスクリプトをご用意しております。

ファイルパス： $WORK/darknet/yolo/postprocess_yolo.py

実行スクリプトの詳細は以下の通りです。

注意事項

- ✓ DRP-AIで以下の後処理を実施しているものとします。
  - transpose
  - castFP16toFP32
- ✓ 必要パッケージ（動作確認済み環境に記載のもの）は適宜インストールしてください。
- ✓ アルゴリズムの詳細はYOLOの論文（YOLOv3 https://arxiv.org/pdf/1804.02767.pdf, YOLOv2 https://arxiv.org/pdf/1612.08242.pdf）をご参照ください。
- ✓ 実行スクリプトではDRP-AI Sample Application Darknet YOLO Image version同梱サンプル入力画像のAI Evaluation Software実行結果を推論結果バイナリファイルとして指定しています。
- ✓ 実行スクリプトは入力画像に矩形を描画します。AI Evaluation Softwareの入力画像として使用した画像をご指定下さい。
- ✓ 検出クラスのラベルリストcoco-labels-2014_2017.txtは以下から入手してください。

  https://github.com/amikelive/coco-labels/blob/master/coco-labels-2014_2017.txt

動作確認済み環境

```
Ubuntu 20.04 LTS        Python        == 3.8.10
                        pip           == 22.2.2
                        torch         == 1.12.1+cpu
                        numpy         == 1.23.1
                        opencv-python == 4.6.0.66
                        Pillow        == 9.2.0
```

DRP-AI推論結果バイナリデータ

YOLOv3のDRP-AI推論結果バイナリファイル内のデータ詳細は以下です。

- データ数　：　904995
  (モデルの出力サイズに依存。YOLOv3の場合、(1x255x13x13)+(1x255x26x26)+(1x255x52x52))
- データ幅　：　4byte　(castFP16toFP32を実行したため, FP32=4byte)
- バイトオーダー　：　リトルエンディアン

推論実行してみよう

本書では後処理スクリプトを実行する際に、以下の青字部分のようにモデル名を指定することで
各モデルを切り替えます。

※未指定の場合、YOLOv3モデルをデフォルト値として使用します。

```
$ python3 postprocess_yolo.py yolov3
```

パラメータ	モデル
yolov3	YOLOv3　（デフォルト値）
yolov2	YOLOv2
tinyyolov3	Tiny YOLOv3
tinyyolov2	Tiny YOLOv2

推論実行してみよう

postprocess_yolo.pyを一部抜粋

```
261 if __name__ == '__main__':
262     if ( len(sys.argv) > 1 and sys.argv[1] in ["yolov3", "yolov2", "tinyyolov3", "tinyyolov2"]):
            :
277     if model_name=="yolov3":
278         anchors = anchors_yolov3
279         out_layer_num = 3
280         num_class = len(labels)
281         output_shape.append([1, 3*(5+num_class), 13, 13]) # [N, C, H, W]
282         output_shape.append([1, 3*(5+num_class), 26, 26]) # [N, C, H, W]
283         output_shape.append([1, 3*(5+num_class), 52, 52]) # [N, C, H, W]
284     elif model_name=="yolov2":
            :
303     result_bin = open("sample.bmp.bin", 'rb')
            :
309     # Load DRP-AI output binary
310     for n in range(out_layer_num): # Output number
311         for c in range(output_shape[n][1]): # C
312             for h in range(output_shape[n][2]): # H
313                 for w in range(output_shape[n][3]): # W
314                     a = struct.unpack('<f', result_bin.read(4))
315                     data[n][0, c, h, w] = a[0]
316
317     # Read image to draw bounding boxes
318     im = Image.open("sample.bmp")
319     im = im.resize((model_in_size, model_in_size), resample=Image.BILINEAR)
320
321     # Post-processing
322     preds = {}
323     for i in range(out_layer_num):
324         preds[i] = torch.from_numpy(data[i]).clone()
325     detections = all_post_process(model_name, preds, anchors, input_size=model_in_size,
326         grid_size=output_shape, n_classes=num_class, obj_th=0.5, nms_th=0.5)
327
328     # Draw bounding box
329     img = np.asarray(im) # RGB
330     img = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)# BGR
331     ret_im = draw_predictions(img, detections, labels) # BGR
332     cv2.imwrite("result.jpg", ret_im)
```

コマンドライン引数から対象モデルを判別

モデルごとにモデル依存パラメータを設定
①Bounding Boxアンカー
②出力層数
③検出クラス数
④出力Shape

※sample.bmp.binはDRP-AI Sample Applicationサンプル入力画像の
AI Evaluation SoftwareによるYOLOv3モデル推論実行結果

DRP-AI推論結果ファイルをロード

FP32の数値を出力サイズ分読む
リトルエンディアンを指定

※sample.bmpはDRP-AI Sample Applicationサンプル入力画像

矩形描画用画像のロード＆リサイズ

CPU後処理実行
検出結果のProbability閾値obj_thと
NMSサンプリング閾値nms_thを設定

※次ページで詳細を説明

矩形の描画

STEP-1
STEP-2
STEP-3
STEP-4

推論実行してみよう

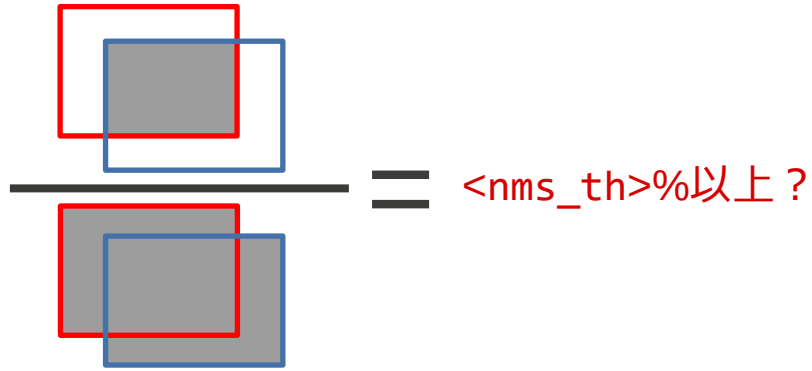
実行スクリプト内で使用されているパラメータについて説明します。

`obj_th` ： 物体検出結果のProbabilityの閾値
この値以下のProbabilityが出た検出結果は無視されます。
今回はデフォルトとして`0.5`を指定しています。

`nms_th` ： 過剰検出結果を除外するNon-Maximum Suppression (NMS) アルゴリズム用の閾値です。
今回はデフォルトとして`0.5`を指定しています。

(補足) Non-Maximum Suppression

一つの物体に対して複数の矩形が検出されることがあります。
NMSではそれらの矩形の重なった面積が全体のx%以上だった場合、Probabilityの低い矩形を除外する処理を実行します。（下図参照）
nms_thで設定する値は、このxの値になります。



STEP-1
STEP-2
STEP-3
STEP-4

以上を踏まえてpostprocess_yolo.pyを実行してみましょう。

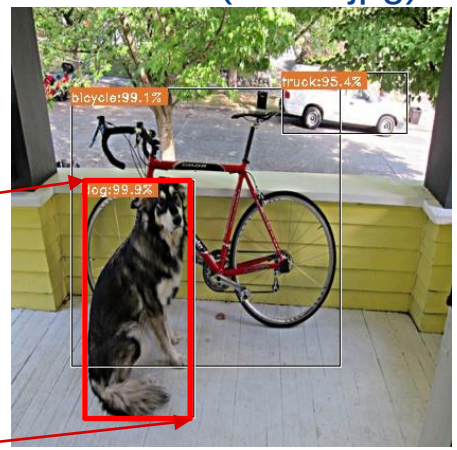
以下はDRP-AI Sample Application for Darknet YOLOの
サンプル入力画像をYOLOv3で実行した際の結果例です。

ターミナルログ

```
Class: bicycle | Probability 98.5% | [X1, Y1, X2, Y2] = [61,93,311,313]
Class: truck | Probability 95.4% | [X1, Y1, X2, Y2] = [254,62,375,121]
Class: dog | Probability 99.9% | [X1, Y1, X2, Y2] = [69,165,171,388]
```

Class ：検出された物体
Probability ：検出された物体のObjectness * Predictions(Class)

出力画像(result.jpg)



STEP-4まとめ

以上でDRP-AIを使ったYOLOモデルの推論実行についての説明は終了です。

DRP-AI Sample Applicationでは、本書でご説明したDarknetが提供するYOLOモデルを使って、推論からCPU後処理までをボード上で実行できるサンプルアプリケーションをご提供しております。

詳細はDRP-AI Sample Application Noteをご参照ください。

おわりに

STEP-1～STEP-4にかけて、Darknetの提供するYOLOモデルをRZ/V2x上で推論実行する方法をご説明させていただきました。

もし、ご不明な点がございましたらルネサスまでお問い合わせください。

最後までお読みいただきありがとうございました。

Renesas.com

変更履歴

Date	Version	Chapter	変更内容
Sep. 29, 2022	7.20	-	初版（RZ/V2L, RZ/V2M, RZ/V2MAのAI Implementation Guide を統一）