Machine Vision: Barcode Detection

Reference Design

FPGA-RD-02280-1.0

December 2023

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS, with all faults, and all associated risk is the responsibility entirely of the Buyer. The information provided herein is for informational purposes only and may contain technical inaccuracies or omissions, and may be otherwise rendered inaccurate for many reasons, and Lattice assumes no obligation to update or otherwise correct or revise this information. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. LATTICE PRODUCTS AND SERVICES ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS, OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING ANY APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, "HIGH-RISK USES"). FURTHER, BUYER MUST TAKE PRUDENT STEPS TO PROTECT AGAINST PRODUCT AND SERVICE FAILURES, INCLUDING PROVIDING APPROPRIATE REDUDANCIES, FAIL-SAFE FEATURES, AND/OR SHUT-DOWN MECHANISMS. LATTICE EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Contents

Figures

Tables

Acronyms in This Document

A list of acronyms used in this document.

Acronym	Definition
CKPT	Checkpoint
CNN	Convolutional Neural Network
CPNX	CertusPro-NX
CSP	Cross Stage Partial network
EVDK	Embedded Vision Development Kit
FPGA	Field Programmable Gate Array
LED	Light-Emitting Diode
MLE	Machine Learning Engine
NN	Neural Network
NNC	Neural Network Compiler
SD	Secure Digital
SDHC	Secure Digital High Capacity
SDXC	Secure Digital extended Capacity
SPI	Serial Peripheral Interface
USB	Universal Serial Bus
VIP	Video Interface Platform
VnV	Voice and Vision
VVML	Voice and Vision Machine Learning Board

1. Introduction

The Barcode Detection design process uses the CertusPro™-NX Voice and Vision Machine Learning Board. This document describes the barcode detection reference design.

1.1. Design Process Overview

The design process involves the following steps:

1. Training the model.
   - Setting up the basic environment.
   - Preparing the dataset.
   - Training the machine.
   - Training the machine and creating the checkpoint data.
   - Creating the frozen file (*.pb).
2. Compiling the neural network: creating the filter and firmware binary files with Lattice sensAI™ 6.0 program.
3. FPGA design: creating the FPGA Bit stream file.
4. FPGA bitstream and Quantized Weights and Instructions: flashing the binary and bitstream files to the CertusPro-NX VVML hardware.

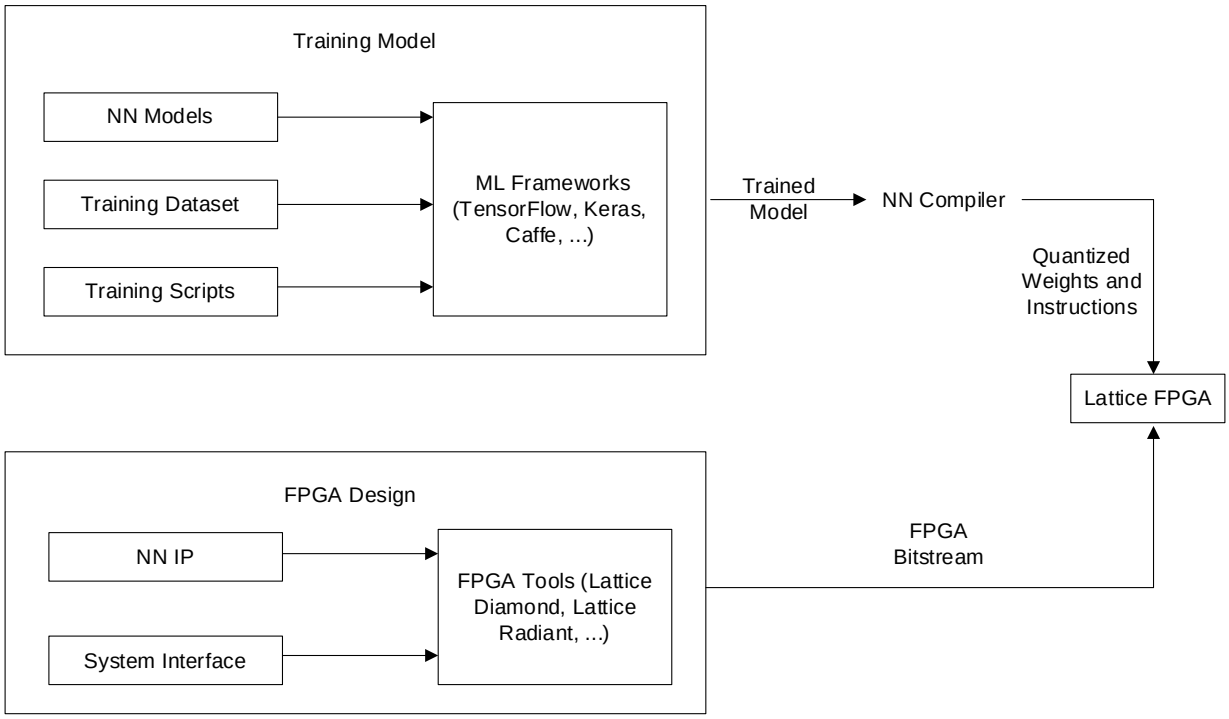

Figure 1.1. Lattice Machine Learning Design Flow

2. Setting Up the Basic Environment

2.1. Tools and Hardware Requirements

This section describes the required tools and environment setup for training and model freezing.

2.1.1. Lattice Tools

- Lattice Radiant™ software v2022.1: refer to http://www.latticesemi.com/latticeradiant
- Lattice Radiant Programmer v2022.1: refer to http://www.latticesemi.com/latticeradiant
- Lattice Neural Network Compiler v 6.1: refer to https://www.latticesemi.com/Products/DesignSoftwareAndIP/AIML/NeuralNetworkCompiler

2.1.2. Hardware

This design uses the CertusPro-NX Voice and Vision Machine Learning (VVML) Board, Rev A Board, as shown in Figure 2.1.

Figure 2.1. Lattice CertusPro-NX Voice and Vision Machine Learning (VVML) Board, Rev A

2.2. Setting Up the Linux Environment for Machine Training

This section describes the steps for NVIDIA GPU drivers and/or libraries for 64-bit Ubuntu 18.04 Operation System. The NVIDIA library and TensorFlow version is dependent on PC and Ubuntu/Windows version.

2.2.1. Installing the NVIDIA CUDA and cuDNN Library for Machine Learning Training on GPU

2.2.1.1. Installing the CUDA Toolkit

To install CUDA toolkit-11.1, go to https://developer.nvidia.com/cuda-toolkit-archive and click on specific required version (CUDA 10.1) from the list. Refer to Figure 2.2.

Figure 2.2. CUDA Archive Page

Select the appropriate combination of as highlighted in Figure 2.3. to get the cuda toolkit download option.

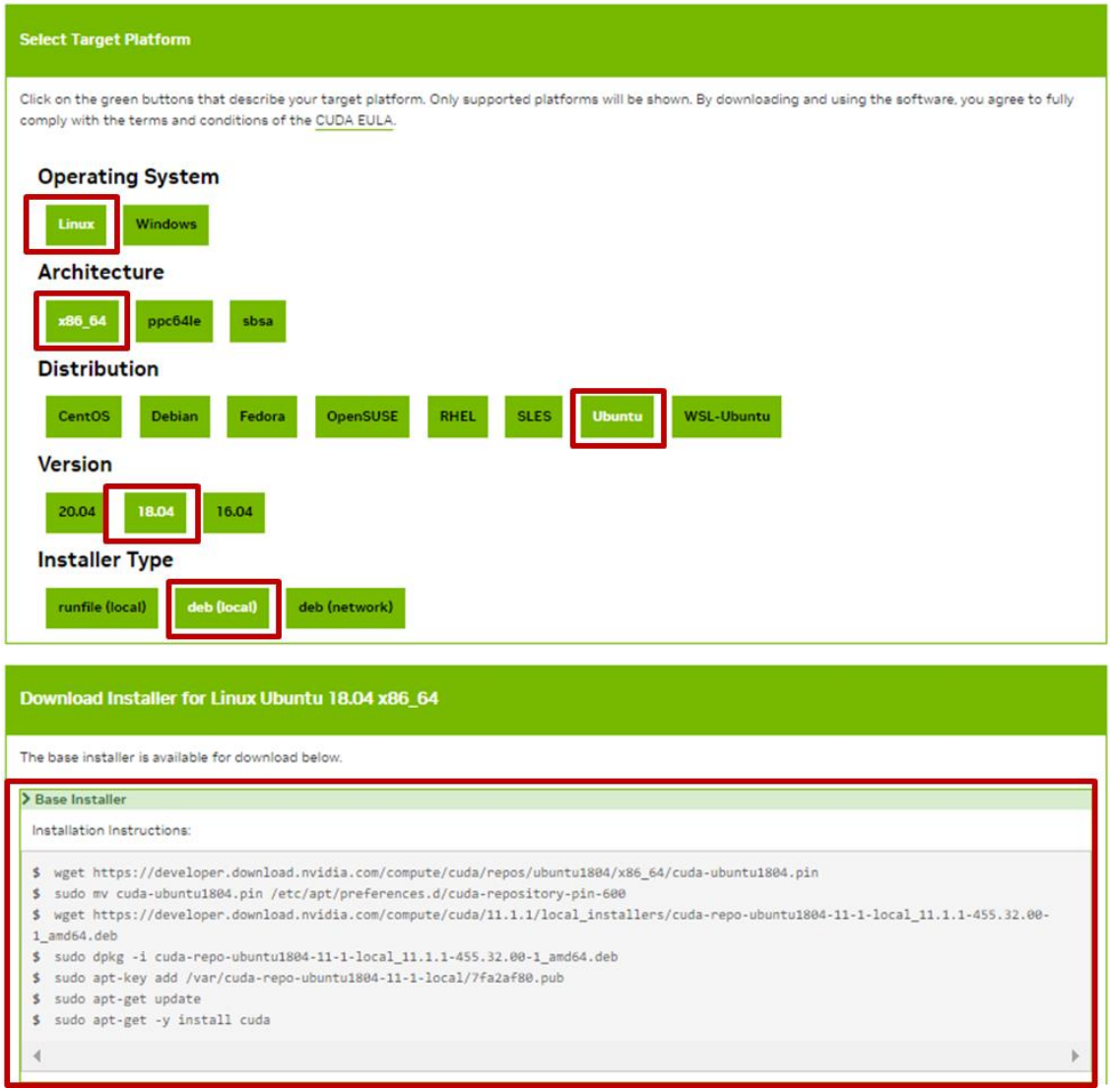

Figure 2.3. Download CUDA Configuration

Once downloaded, follow the Installation Instructions given in the download block of the webpage.

2.2.1.2. Installing the cuDNN

To install the cuDNN:

1. Create an NVIDIA developer account: https://developer.nvidia.com.
2. Download cuDNN lib: https://developer.nvidia.com/compute/machine-learning/cudnn/secure/v7.1.4/prod/9.0_20180516/cudnn-9.0-linux-x64-v7.1
3. Execute the following commands to install cuDNN:

```
$ tar xvfcudnn-9.0-linux-x64-v7.1.tgz
$ sudo cp cuda/include/cudnn.h/usr/local/cuda/include
$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
$ sudochmoda+r/usr/local/cuda/include/cudnn.h/usr/local/cuda/lib64/libcudnn*
```

```
$ tar xcf cudnn-9.0-linux-x64-v7.1.tgz
cuda/include/cudnn.h
cuda/NVIDIA_SLA_cuDNN_Support.txt
cuda/lib64/libcudnn.so
cuda/lib64/libcudnn.so.7
cuda/lib64/libcudnn.so.7.1.4
cuda/lib64/libcudnn_static.a

$ sudo cp cuda/include/cudnn.h /usr/local/cuda/include
$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
$ sudo chmod a+r /usr/local/cuda/include/cudnn.h /usr/local/cuda/lib64/libcudnn*
```

Figure 2.4. cuDNN Library Installation

2.2.2. Setting Up the Environment for Training and Model Freezing Scripts

This section describes the environment setup information for training and model freezing scripts for 64-bit Ubuntu 18.04. Anaconda provides one of the easiest ways to perform machine learning development and training on Linux.

Installing Anaconda Python

To install the Anaconda and Python 3:

1. Go to https://www.anaconda.com/products/distribution
2. Scroll down for Anaconda Additional Installers.
3. Download Python3 version of Anaconda for Linux.
4. Run the command below to install the Anaconda environment:

```
$  sh  Anaconda3-2022.10-Linux-x86_64.sh
```

Note: Anaconda3-**<version>**-Linux-x86_64.sh, version may vary based on the release.

```
(base) $ sh Anaconda3-2020.07-Linux-x86_64.sh

Welcome to Anaconda3 2020.07

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>>
```

Figure 2.5. Anaconda Installation

5. Accept the license.

```
Do you accept the license terms? [yes|no]
[no] >>> yes
```

Figure 2.6. Accept License Terms

6. Confirm the installation path. Follow the instruction on screen if you want to change the default path.

```
[no] >>> yes


Anaconda3 will now be installed into this location:
/home/user/anaconda3

  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below

[/home/user/anaconda3] >>> /home/user/anaconda3
```

Figure 2.7. Confirm/Edit Installation Location

7. After installation, enter No, as shown in Figure 2.8.

```
Preparing transaction: done
Executing transaction: done
installation finished.
Do you wish the installer to initialize Anaconda3
by running conda init? [yes|no]
[no] >>>  no
```

Figure 2.8. Launch/Initialize Anaconda Environment on Installation Completion

2.2.3. Creating New Environment with the yml File Provided

The project structure is provided with the corresponding yml file *env_barcode.yml* which directly creates the required virtual environment.

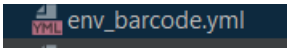

Figure 2.9. env_barcode.yml (1)

Prior to executing this file, you need to edit certain information in yml file. Follow the steps below:

1. After successful installation of Ananconda, navigate to its directory */home/user/anaconda3/* to find env folder. Copy path to this env folder.
2. Open *env_barcode.yml* and make changes below:

   Set your environment name in first line of the file as shown below.

env_barcode.yml

```yaml
name: <Set your name>
channels:
  - conda-forge
  - defaults
dependencies:
  - _libgcc_mutex=0.1=main
  - _openmp_mutex=5.1=1_gnu
  - ca-certificates=2023.7.22=hbcca054_0
  - certifi=2023.7.22=pyhd8ed1ab_0
  - cudatoolkit=11.1.1=ha002fc5_10
  - cudnn=8.9.2.26=cuda11_0
  - ld_impl_linux-64=2.38=h1181459_1
  - libffi=3.4.2=h6a678d5_6
  - libgcc-ng=11.2.0=h1234567_1
  - libgomp=11.2.0=h1234567_1
  - libstdcxx-ng=11.2.0=h1234567_1
  - ncurses=6.3=h5eee18b_3
  - openssl=1.1.1w=h7f8727e_0
  - pip=22.2.2=py39h06a4308_0
  - python=3.9.15=h7a1cb2a_2
  - readline=8.2=h5eee18b_0
  - setuptools=65.5.0=py39h06a4308_0
  - sqlite=3.40.0=h5082296_0
  - tk=8.6.12=h1ccaba5_0
  - tzdata=2022g=h04d1e81_0
  - wheel=0.37.1=pyhd3eb1b0_0
  - xz=5.2.8=h5eee18b_0
  - zlib=1.2.13=h5eee18b_0
  - pip:
      - absl-py==1.3.0
```

Figure 2.10. env_barcode.yml (2)

3. Modify the last line in the yml file with the new path and environment name.

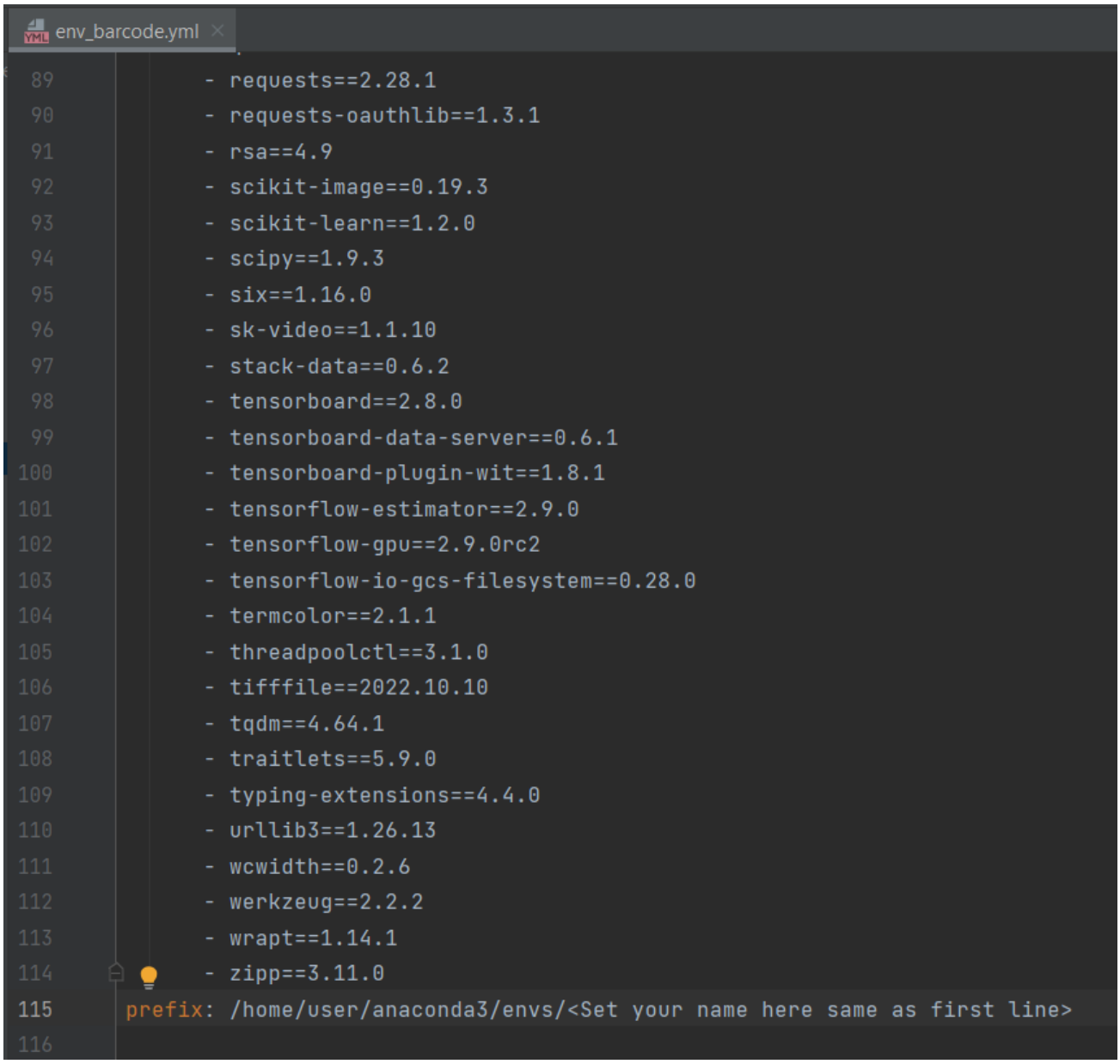

Figure 2.11. env_barcode.yml Prefix Line Edit

4. Activate your conda environment using the command below:

```
$ source anaconda3/bin/activate
```

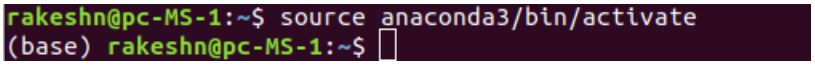

Figure 2.12. Anaconda Activate

5. Create the environment from the env_barcode.yml file using the command below:

```
$ conda env create -f env_barcode.yml
```

```
(tf2.9) softnautics@gpuserver:~/Rakesh_Nakod$ conda env create -f env_barcode.yml
Collecting package metadata (repodata.json): done
Solving environment: done


==> WARNING: A newer version of conda exists. <==
  current version: 22.11.1
  latest version: 23.9.0

Please update conda by running

    $ conda update -n base -c defaults conda

Or to minimize the number of packages updated during conda update use

     conda install conda=23.9.0



Downloading and Extracting Packages

Preparing transaction: done
Verifying transaction: done
Executing transaction: | By downloading and using the CUDA Toolkit conda packages, you accept the te
rms and conditions of the CUDA End User License Agreement (EULA): https://docs.nvidia.com/cuda/eula/
index.html

done
Installing pip dependencies: | Ran pip subprocess with arguments:
['/home/softnautics/anaconda3/envs/tf2.9_test/bin/python', '-m', 'pip', 'install', '-U', '-r', '/hom
e/softnautics/Rakesh_Nakod/condaenv.w4plkw52.requirements.txt', '--exists-action=b']
Pip subprocess output:
Collecting absl-py==1.3.0
  Using cached absl_py-1.3.0-py3-none-any.whl (124 kB)
```

Figure 2.13. env_barcode.yml Execution

```
ts, pyasn1-modules, protobuf, prompt-toolkit, pillow, pexpect, parso, packaging, oauthlib, numpy, ne
tworkx, markupsafe, kiwisolver, joblib, idna, grpcio, gast, fonttools, decorator, cycler, charset-no
rmalizer, cachetools, absl-py, werkzeug, tifffile, scipy, requests, pywavelets, python-dateutil, opt
-einsum, opencv-python-headless, opencv-python, matplotlib-inline, keras-preprocessing, jedi, import
lib-metadata, imageio, h5py, google-pasta, google-auth, contourpy, astunparse, asttokens, stack-data
, sk-video, scikit-learn, scikit-image, requests-oauthlib, matplotlib, markdown, qudida, ipython, go
ogle-auth-oauthlib, tensorboard, albumentations, tensorflow-gpu
Successfully installed absl-py-1.3.0 albumentations-1.3.0 asttokens-2.2.1 astunparse-1.6.3 backcall-
0.2.0 cachetools-5.2.0 charset-normalizer-2.1.1 check-digit-ean13-0.2 contourpy-1.0.6 cycler-0.11.0
decorator-5.1.1 easydict-1.10 executing-1.2.0 flatbuffers-1.12 fonttools-4.38.0 gast-0.4.0 google-au
th-2.15.0 google-auth-oauthlib-0.4.6 google-pasta-0.2.0 grpcio-1.51.1 h5py-3.7.0 idna-3.4 imageio-2.
23.0 importlib-metadata-5.1.0 ipython-8.13.2 jedi-0.18.2 joblib-1.2.0 keras-2.9.0 keras-preprocessin
g-1.1.2 kiwisolver-1.4.4 libclang-14.0.6 markdown-3.4.1 markupsafe-2.1.1 matplotlib-3.6.2 matplotlib
-inline-0.1.6 networkx-2.8.8 numpy-1.23.5 oauthlib-3.2.2 opencv-python-4.6.0.66 opencv-python-headle
ss-4.6.0.66 opt-einsum-3.3.0 packaging-22.0 parso-0.8.3 pexpect-4.8.0 pickleshare-0.7.5 pillow-9.3.0
 prompt-toolkit-3.0.38 protobuf-3.19.6 ptyprocess-0.7.0 pure-eval-0.2.2 pyasn1-0.4.8 pyasn1-modules-
0.2.8 pygments-2.15.1 pyparsing-3.0.9 python-barcode-0.14.0 python-dateutil-2.8.2 pywavelets-1.4.1 p
yyaml-6.0 qudida-0.0.4 requests-2.28.1 requests-oauthlib-1.3.1 rsa-4.9 scikit-image-0.19.3 scikit-le
arn-1.2.0 scipy-1.9.3 six-1.16.0 sk-video-1.1.10 stack-data-0.6.2 tensorboard-2.8.0 tensorboard-data
-server-0.6.1 tensorboard-plugin-wit-1.8.1 tensorflow-estimator-2.9.0 tensorflow-gpu-2.9.0rc2 tensor
flow-io-gcs-filesystem-0.28.0 termcolor-2.1.1 threadpoolctl-3.1.0 tifffile-2022.10.10 tqdm-4.64.1 tr
aitlets-5.9.0 typing-extensions-4.4.0 urllib3-1.26.13 wcwidth-0.2.6 werkzeug-2.2.2 wrapt-1.14.1 zipp
-3.11.0

done
#
# To activate this environment, use
#
#     $ conda activate tf2.9_test
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

Figure 2.14. env_barcode.yml Execution Completion

6. Hit the following command to activate your newly created environment:

```
$ conda activate <your env name>
```

```
(tf2.9) softnautics@gpuserver:~/Rakesh_Nakod$ conda activate tf2.9_test
(tf2.9_test) softnautics@gpuserver:~/Rakesh_Nakod$
```

Figure 2.15. Activating the Environment

3. Code Structure

Download the Lattice Barcode Detection demo training code. The reference design comes with two algorithms for barcode detection, which are MobileNet-v1 and Yolov5. The directory structures are shown in Figure 3.1. and Figure 3.2.

3.1. Code structure for MobileNet-v1 based Barcode Detection

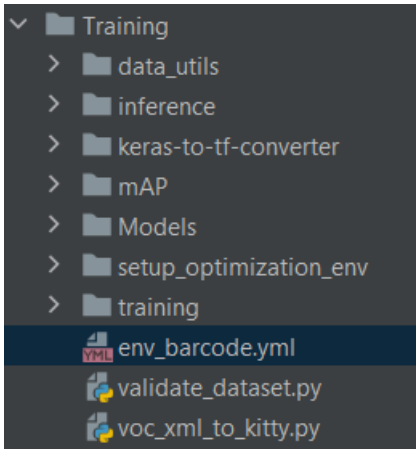

Figure 3.1. MobileNet-v1 Training Code Directory Structure

3.2. Code Structure for Yolov5 – Focus Layer based Barcode Detection

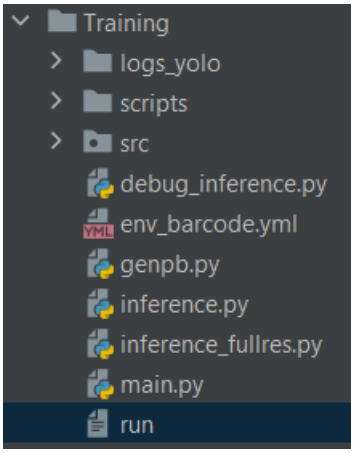

Figure 3.2. Yolov5: Training Code Directory Structure

4. Dataset Preparation

This section describes steps and guidelines used to prepare the dataset to train the barcode detection demo for CPNX-VNV.

Note: This section is for the example reference. Lattice Semiconductor only provides the guidelines and/or examples which can be used as reference for preparing the dataset for given use cases. Lattice Semiconductor does not recommend and/or endorse any dataset(s). Lattice Semiconductor strongly recommends you gather and prepare your own datasets for your end applications.

4.1. Downloading the Dataset

In this document, we use Artelabs 1-D barcode Dataset (http://artelab.dista.uninsubria.it/downloads/datasets/barcode/medium_barcode_1d/medium_barcode_1d.html) as reference. This dataset is under the Creative Commons Attribution 3.0. License. Download the Dataset and extract the data.

Figure 4.1. shows the Arte-lab dataset directory structure and Figure 4.2. shows the Arte-lab sample image.

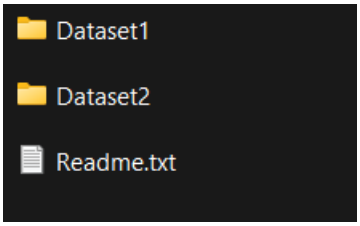

Figure 4.1. Arte-lab Dataset Directory Structure

Figure 4.2. Arte-lab Sample Image

4.2. Labelling Artelab Dataset Using the LabelImg Tool

Activate your environment and install the LabelImg tool using the command below. Figure 4.3. shows the output of the installation.

```
$ python pip install labelImg
```

```
(base) C:\Users\Rakesh Nakod>activate tf1_14_py_36

(tf1_14_py_36) C:\Users\Rakesh Nakod>pip install labelImg
Collecting labelImg
  Using cached labelImg-1.8.6.tar.gz (247 kB)
Collecting pyqt5
  Downloading PyQt5-5.15.6-cp36-abi3-win_amd64.whl (6.7 MB)
     |████████████████████████████████| 6.7 MB 2.2 MB/s
Collecting lxml
  Downloading lxml-4.9.2-cp36-cp36m-win_amd64.whl (3.8 MB)
     |████████████████████████████████| 3.8 MB 2.2 MB/s
Collecting PyQt5-Qt5>=5.15.2
  Using cached PyQt5_Qt5-5.15.2-py3-none-win_amd64.whl (50.1 MB)
Collecting PyQt5-sip<13,>=12.8
  Downloading PyQt5_sip-12.9.1-cp36-cp36m-win_amd64.whl (83 kB)
     |████████████████████████████████| 83 kB 713 kB/s
Building wheels for collected packages: labelImg
  Building wheel for labelImg (setup.py) ... done
  Created wheel for labelImg: filename=labelImg-1.8.6-py2.py3-none-any.whl size=261544 sha256=90ee214b84cdcbd2e4f7f1708f
1f1e20d2b4dc99803c098bbe011bfde93c9909
  Stored in directory: c:\users\rakesh nakod\appdata\local\pip\cache\wheels\35\6f\d6\f96de77faaf1eab00bfe865ebe8806d3f79
0d609e4d7a15f50
Successfully built labelImg
Installing collected packages: PyQt5-sip, PyQt5-Qt5, pyqt5, lxml, labelImg
Successfully installed PyQt5-Qt5-5.15.2 PyQt5-sip-12.9.1 labelImg-1.8.6 lxml-4.9.2 pyqt5-5.15.6
```

Figure 4.3. LabelImg Tool Installation

4.3. Annotate Images

4.3.1. Open LabelImg Tool

After installation of the LabelImg tool, use the command below to launch the tool as shown in Figure 4.4.

```
$ labelImg
```

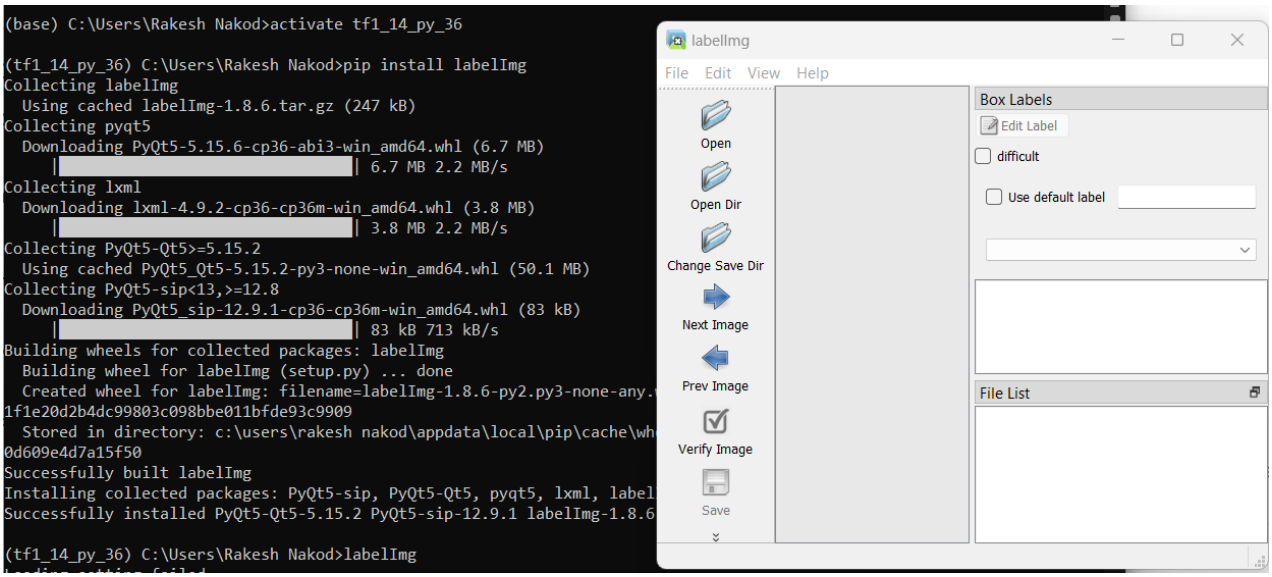

Figure 4.4. LabelImg Tool

1. Click on the **Open Dir** menu option on left of the LabelImg tool GUI and select your Dataset directory that contains the images as shown in Figure 4.5.

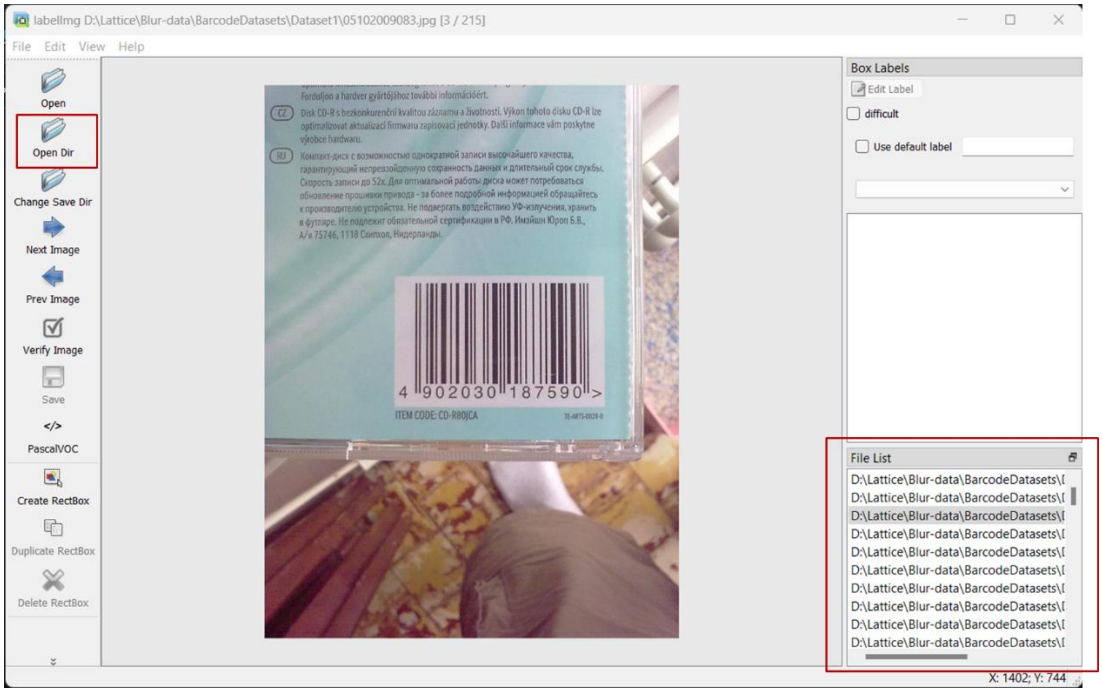

Figure 4.5. Open Dataset Directory

2. Click on the **Change Save Dir** menu option to select a folder where you want to save your annotation file as shown in Figure 4.5.

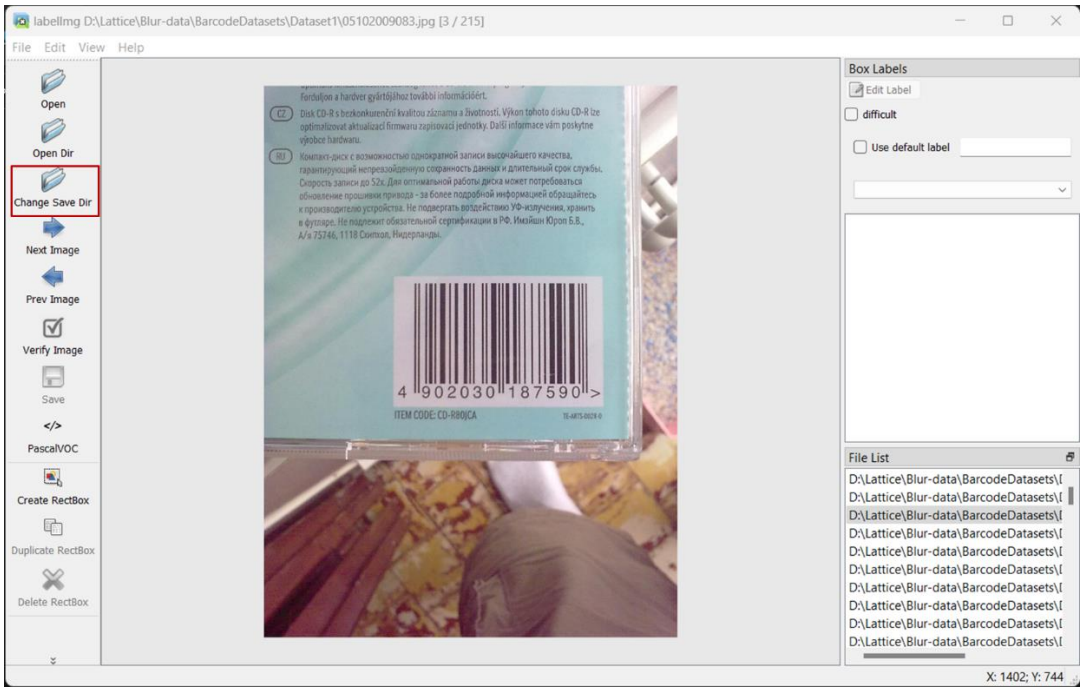

Figure 4.6. Saving Output Directory

3. Click on the **Create RectBox** menu option as shown in Figure 4.7 to draw bounding box around the barcode.

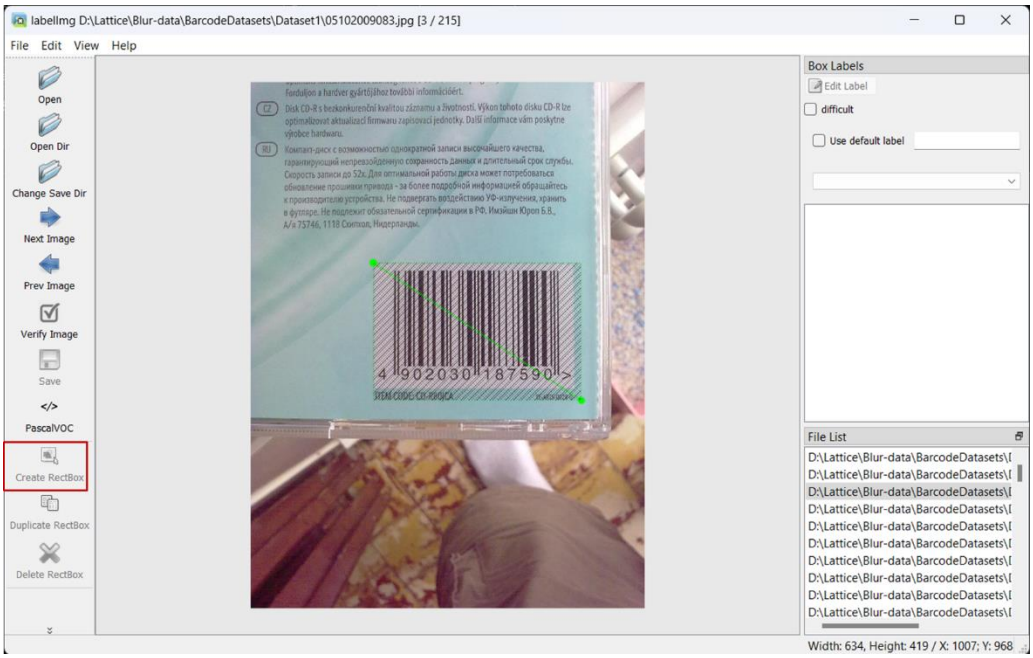

Figure 4.7. Drawing Bounding Box

4. Label them as *barcode* and click on **OK**. After that, the **Save** menu option becomes active. You can now save the label for the image. Click **Save** as shown in Figure 4.8.

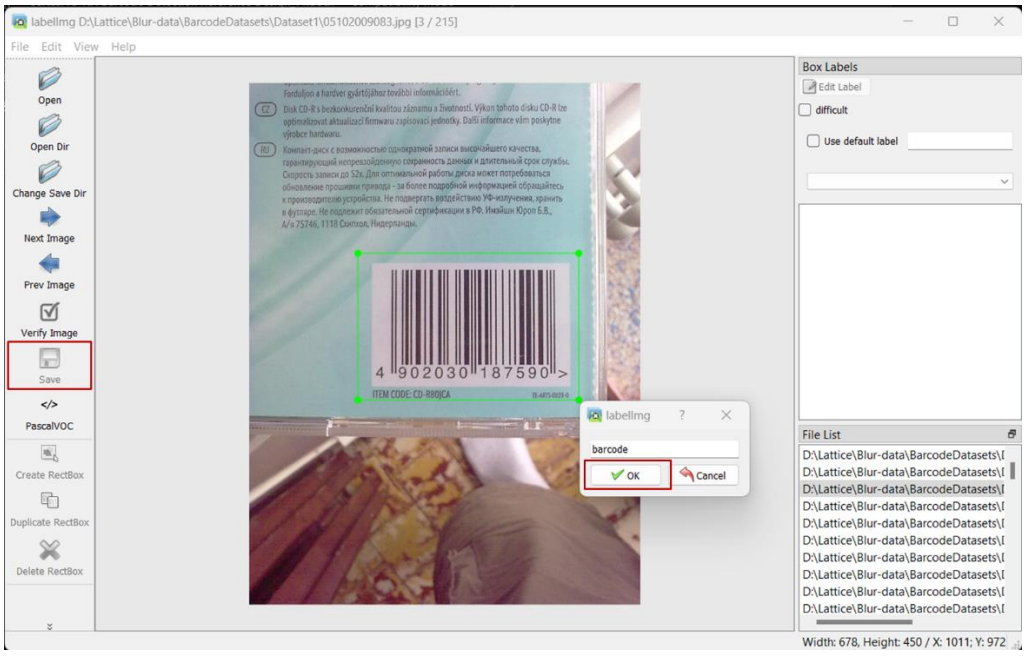

Figure 4.8. Saving Label and Image

5. You can review all saved images. If required, the **Delete Box** menu option enables you to remove or edit the existing bounding box as shown in Figure 5.9.

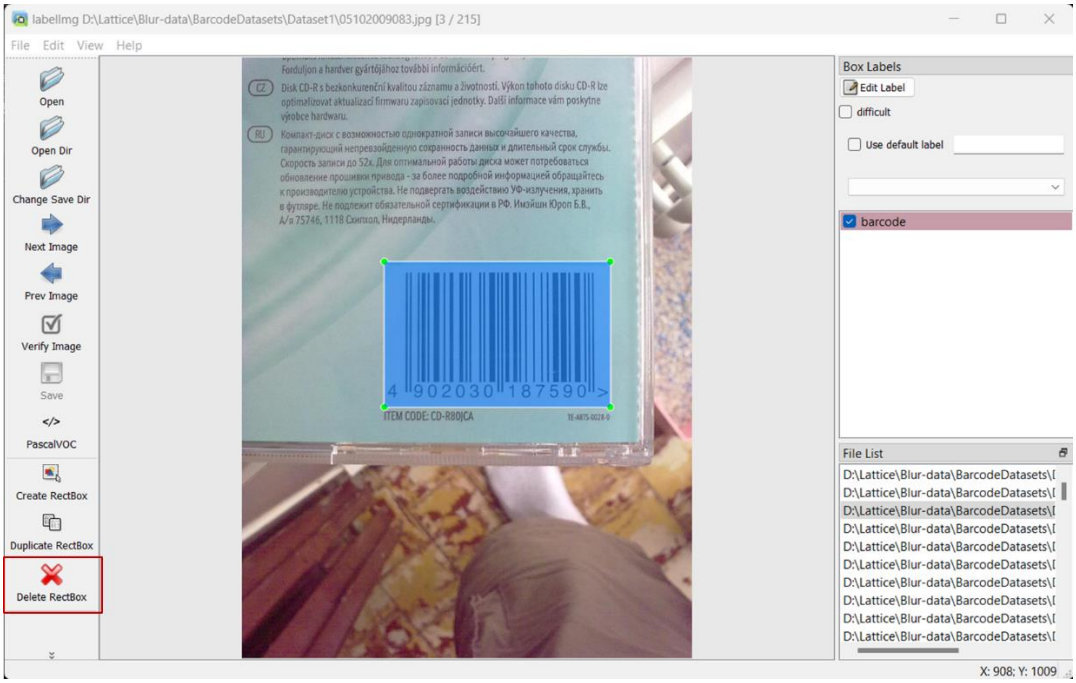

Figure 4.9. LabelImg Tool

6. All the labelled images have corresponding voc format .xml files created in the Change save directory as shown in Figure 4.10.

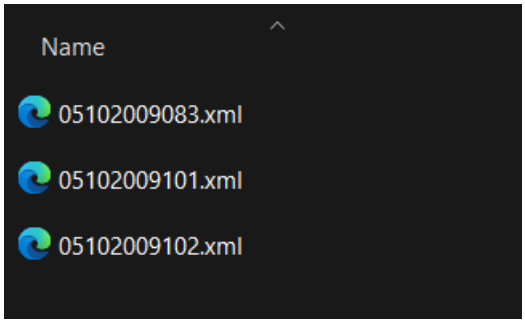

Figure 4.10. Output Label Folder

4.4. Convert LabelImg Tools VOC XML Label Format to Kitti Format

- Download the Lattice Barcode Detection and Reading demo training code.
- There is *"voc_xml_to_kitty.py"* file in Training directory.
- Run command below to convert "VOC XML for images" format to "Kitti" format.

```
$ python voc_xml_to_kitty.py --input _dir < path to output folder of labelImg tool
containing xml files> --output <Output folder for saving kitti format files>
```

Figure 4.11. and Figure 4.12. show the output label folder and the label format, respectively.

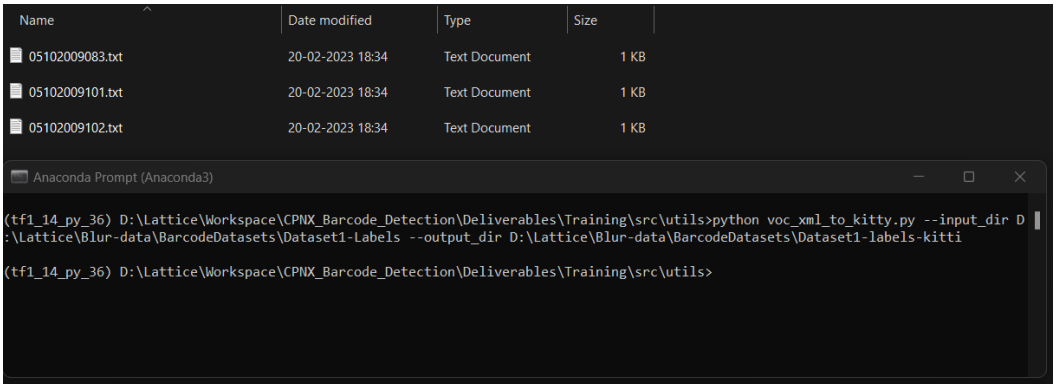

Figure 4.11. Output Kitti Label Folder

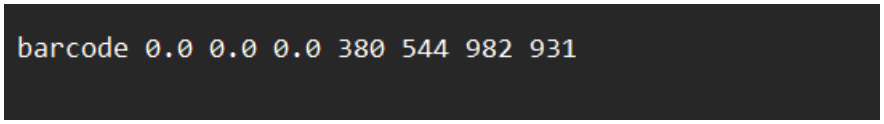

Figure 4.12. Kitti Label Format

5. Training Code Preparation

Note: Refer to Appendix A for Training Code Preparation for *Yolov5 Architecture* based Barcode Detection Design.

5.1. Neural Network Architecture

This section provides information on the Convolution Network Configuration of the Barcode Detection design. The Neural Network model of the Barcode Detection design uses MobileNetV1 NN base model and the detection layer of the SqueezeDet model.

<table>
<tr><td colspan="3">Image Input (320 x 240 x 1)</td></tr>
<tr><td rowspan="8">Fire 1</td><td>DWConv3 - 32</td><td rowspan="32">Conv3 - # where:
• Conv3 = 3 x 3 Convolution filter Kernel size
• # = The number of filter
DWConv3 - 32- # where:
• DWConv3 = Depth wise convolution filter with 3x3 size
• # = The number of filter
Conv1 - 32- # where:
• Conv1 = 1 x 1 Convolution filter Kernel size
• # = The number of filter

For example, Conv3 - 16 = 16 3 x 3 convolution filters

BN – Batch Normalization</td></tr>
<tr><td>BN</td></tr>
<tr><td>Relu</td></tr>
<tr><td>Maxpool</td></tr>
<tr><td>Conv1 – 32</td></tr>
<tr><td>BN</td></tr>
<tr><td>Relu</td></tr>
<tr><td>Maxpool</td></tr>
<tr><td rowspan="8">Fire 2</td><td>DWConv3 - 32</td></tr>
<tr><td>BN</td></tr>
<tr><td>Relu</td></tr>
<tr><td>Conv1 – 32</td></tr>
<tr><td>BN</td></tr>
<tr><td>Relu</td></tr>
<tr><td rowspan="8">Fire 3</td><td>DWConv3 – 32</td></tr>
<tr><td>BN</td></tr>
<tr><td>Relu</td></tr>
<tr><td>Maxpool</td></tr>
<tr><td>Conv1 – 32</td></tr>
<tr><td>BN</td></tr>
<tr><td>Relu</td></tr>
<tr><td rowspan="8">Fire 4</td><td>DWConv3 – 64</td></tr>
<tr><td>BN</td></tr>
<tr><td>Relu</td></tr>
<tr><td>Maxpool</td></tr>
<tr><td>Conv1 – 64</td></tr>
<tr><td>BN</td></tr>
<tr><td>Relu</td></tr>
<tr><td rowspan="6">Fire 5</td><td>DWConv3 – 64</td></tr>
<tr><td>BN</td></tr>
<tr><td>Relu</td></tr>
<tr><td>Conv1 – 64</td></tr>
<tr><td>BN</td></tr>
</table>

	Relu	
Fire 6	DWConv3 – 128	
	BN	
	Relu	
	Conv1 – 128	
	BN	
	Relu	
Fire 7	DWConv3 – 128	
	BN	
	Relu	
	Conv1 – 128	
	BN	
	Relu	
Conv12	Conv3 – 42	

Figure 5.1. MobileNet-v1 Architecture

- As shown in Figure 5.1, this model consists of seven fire layers followed by one convolution layer. Fire layer contains convolution, depth wise convolution, batch normalization and relu layers with pooling layer only in fire1, fire 3, and fire 4. Layers fire 2, fire 5, fire 6, and fire 7 do not contain pooling.
- Note that fire 1 contains two Maxpooling operations to reduce calculation complexity and model size.
- Layer contains Convolution (conv), batch normalization (bn) and relu layers.
- Layer information:
  - Convolutional Layer

    In general, the first layer in a CNN is always a convolutional layer. Each layer consists of number of filters (sometimes referred as kernels) which convolves with input layer/image and generates activation map (i.e., feature map). This filter is an array of numbers (the numbers are called weights or parameters). Each of these filters can be thought of as feature identifiers, like straight edges, simple colors, and curves and other high-level features. For example, the filters on the first layer convolve around the input image and “activate” (or compute high values) when the specific feature (say curve) it is looking for is in the input volume.

  - Relu (Activation layer)

    After each convolutional layer, it is a convention to apply a nonlinear layer (or activation layer) immediately afterward. The purpose of this layer is to introduce nonlinearity to a system that computes linear operations in the convolutional layers (element wise multiplications and summations). ReLU layers work better than nonlinear functions such as tanh and sigmoid because the network is able to faster (because of computational efficiency) without impacting accuracy. The ReLU layer applies the function f(x) = max (0, x) to all the values in the input volume. This layer just changes all the negative activations to 0. This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the convolutional layer.

  - Pooling Layer

    After some ReLU layers, programmers may choose to apply a pooling layer. It is also referred to as a down sampling layer. In this category, there are also several layer options, with Maxpooling being the most popular. This layer takes a filter (normally of size 2x2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every sub region that the filter convolves around.

    The intuitive reasoning behind this layer is that once we know that a specific feature is in the original input volume (there will be a high activation value), its exact location is not as important as its relative location to

the other features. This layer drastically reduces the spatial dimension (the length and the width change but not the depth) of the input volume. This serves two main purposes.

The first is that the number of parameters or weights is reduced by 75%, thus lessening the computation cost. The second is that it will control over fitting. This term refers to when a model is so tuned to the training examples that it is not able to generalize well for the validation and test sets. A symptom of over fitting is having a model that gets 100% or 99% on the training set, but only 50% on the test data.

- Batch Normalization Layer
  The batch normalization layer reduces the internal covariance shift. To train a neural network, you need to preprocess the input data. For example, you can normalize all data so that it resembles a normal distribution (that is, zero mean and a unitary variance) to prevent the early saturation of non-linear activation functions, such as the sigmoid function, and assuring that all input data is in the same range of values.

  However, the problem appears in the intermediate layers because the distribution of the activations is constantly changing during training. This slows down the training process because each layer must learn to adapt them to a new distribution in every training step. This problem is known as internal covariate shift.

  Batch normalization layer forces the input of every layer to have approximately the same distribution in every training step by following the process below when training:
  - Calculate the mean and variance of the layers input.
  - Normalize the layer inputs using the previously calculated batch statistics.
  - Scale and shift to obtain the output of the layer.

  This makes the learning of layers in the network more independent of each other and allows you to be more flexible with weight initialization, works as regularization in place of dropout and other regularization techniques.

- Depth wise Convolution and 1 x 1 Convolution Layer
  Depth wise convolutions are used to apply a single filter per each input channel (input depth). Pointwise convolution, a simple 1 × 1 convolution, is then used to create a linear combination of the output of the depth wise layer.
  Depth wise convolution is extremely efficient relative to standard convolution. However, it only filters input channels and does not combine them to create new features. An additional layer that computes a linear combination of the output of depth wise convolution via 1 × 1 convolution is needed to generate these new features.
  The 1 × 1 convolutional layer compresses an input tensor with large channel size to one with the same batch and spatial dimension, but smaller channel size. Given a 4D input tensor and a filter tensor of shape [filter_height, filter_width, in_channels, channel_multiplier], containing in_channels, convolutional filters of depth 1, depthwise_conv2d applies a different filter to each input channel, the layer concatenates the results together. The output has in_channels * channel_multiplier channels.

The architecture above provides nonlinearities and preservation of dimension that help to improve the robustness of the network and control over fitting.

5.1.1. Barcode Detection Network Output

Note: Refer to Appendix B for the network output for *Yolov5 Architecture* based Barcode Detection Design.

For MobileNet-V1 architecture, the input image model first extracts feature maps, overlays them with a W x H grid and at each cell computes K pre-computed bounding boxes called anchors. Each bounding box has the following:

- Four scalars (x, y, w, h)
- A confidence score (Pr(Object)x IOU)
- C conditional classes

Hence current model architecture has a fixed output of W x H x K x (4 + 1 + C). Where:

- W, H = Grid Size
- K = Number of anchor boxes
- C = Number of classes for detection

Based on the description above, the model has a total of 12600 output values. It is derived from following:

- 20 x 15 grid
  - 20 anchor boxes per grid
    - 6 values per anchor box consisting of:
      - 4 bounding box coordinates (x, y, w, h)
      - 1 class probability
      - 1 confidence score

  So, total 15 x 20 x 20 x 6 = 36000 output values.

Note: Smaller images do not work as well with the resulting spacer grid. If your images are smaller, stretch the devices to default size. You can also up-sample them beforehand.

If your images are bigger and you are not satisfied with the results of the default image size, you can try using a denser grid, as details might get lost during the downscaling.

5.1.2. Training Code Overview

Note: Refer to Appendix C for training code overview for *Yolov5 Architecture* based Barcode Detection Design.

Figure 5.2. shows the training code flow diagram.

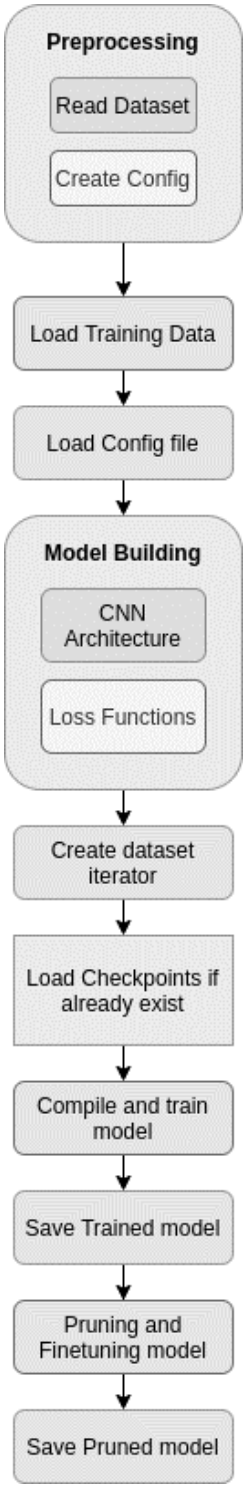

Figure 5.2. Training Code Flow Diagram

Training code can be divided into the following parts:

- Model config
- Model building
- Model freezing
- Data preparation
- Training for overall execution flow

Details of each can be found in the subsequent sections.

Model Config

Demo uses Kitti dataset and SqueezeDet main/config/create_config.py maintains all the configurable parameters for the model. A summary of configurable parameters is shown below:

- Image size
  - Change mc.IMAGE_WIDTH and mc.IMAGE_HEIGHT to configure Image size (width and height) in *src/config/kitti_squeezeDet_config.py*

```
# image properties
cfg.IMAGE_WIDTH = 320
cfg.IMAGE_HEIGHT = 240
if gray:
    cfg.N_CHANNELS = 1
else:
    cfg.N_CHANNELS = 3
```

Figure 5.3. Code Snippet: Snippet: Input Image Size Config

  - Grid dimensions are H = 15 and W = 20. Update them based on anchors per grid size changes.

```
cfg.ANCHORS_HEIGHT = 15
cfg.ANCHORS_WIDTH = 20
```

Figure 5.4. Code Snippet: Anchors Per Grid Config #1 (grid sizes)

- Batch size

  Change *mc.BATCH_SIZE* in *training/main/config/create_config.py* to configure batch size.

- Output classes

  Edit classes in src/config/config.py as shown below.

```
cfg.CLASS_NAMES = ("barcode",)
```

Figure 5.5. Code Snippet: Classes

- Anchors per grid
  - Change cfg.ANCHOR_PER_GRID in *training/main/config/create_config.py* to configure anchors per grid.

```
cfg.ANCHOR_PER_GRID = len(cfg.ANCHOR_SEED)
```

Figure 5.6. Code Snippet: Anchors Per Grid Config #2

- If you want to run network on your own dataset, you need to adjust the anchor sizes. Anchors are prior distribution over what shapes your boxes should have. The better this fits to the true distribution of boxes, the faster and easier your training can be.
- To determine anchor shapes, first load all ground truth boxes and pictures. If your images do not have all the same size, normalize their height and width by the image's height and width. All images need be normalized before being fed to the network. You need to do the same to the bounding boxes and consequently, the anchors.
- Second, perform a clustering on these normalized boxes. You can use k-means without feature whitening to determine the number of clusters either by eyeballing or by using the elbow method.
- Check for boxes that extend beyond the image or have a zero to negative width or height.

```
cfg.ANCHOR_SEED = np.array([[185, 178],
                            [ 24,  46],
                            [ 70,  82],
                            [292, 232],
                            [ 53,  51],
                            [ 65,  27],
                            [136, 113],
                            [ 96, 178],
                            [ 71, 125],
                            [257, 119],
                            [ 97,  15],
                            [110,  60],
                            [139,  24],
                            [108,  42],
                            [ 34,  15],
                            [172,  61],
                            [ 35, 108],
                            [124,  84],
                            [177, 105],
                            [187,  37]])


cfg.ANCHOR_PER_GRID = len(cfg.ANCHOR_SEED)
```

Figure 5.7. Code Snippet: Anchors per Grid Config #3

- Training parameters

  Other training related parameters such as learning rate, loss parameters, and different thresholds can be configured in *src/config/kitti_yolov5_config.py*.

```
# batch sizes
cfg.BATCH_SIZE = 20
cfg.VISUALIZATION_BATCH_SIZE = 20

# SGD + Momentum parameters
cfg.WEIGHT_DECAY = 0.0001
cfg.LEARNING_RATE = 0.005
cfg.MAX_GRAD_NORM = 1.0
cfg.MOMENTUM = 0.9

# coefficients of loss function
cfg.LOSS_COEF_BBOX = 5.0
cfg.LOSS_COEF_CONF_POS = 75.0
cfg.LOSS_COEF_CONF_NEG = 100.0
cfg.LOSS_COEF_CLASS = 1.0

# thresholds for evaluation
cfg.NMS_THRESH = 0.4
cfg.PROB_THRESH = 0.005
cfg.TOP_N_DETECTION = 10
cfg.IOU_THRESHOLD = 0.5
cfg.FINAL_THRESHOLD = 0.0
div_scale_h = 2.0 * (224 / cfg.IMAGE_HEIGHT)
div_scale_w = 2.0 * (224 / cfg.IMAGE_WIDTH)
```

Figure 5.8. Code Snippet: Training Parameters

The SqueezeDet class constructor builds a model which can be divided in below sections:

- Forward graph:
  - File path: main/model/SqueezeDet.py -> _create_model()
  - CNN architecture consists of convolution, batch normalization, relu, maxpool and 1 x 1 depth wise convolution layers.
  - Default forward graph consists of seven fire layers as described in Figure 5.9.
  - The length of network is generated based on the argument depth which consists of the number of filters for each layer.
    Note: The minimum length supported is 5. The maximum length supported is 10.

```
        self.input_layer = Input(shape=(self.config.IMAGE_HEIGHT, self.config.IMAGE_WIDTH, self.config.N_CHANNELS),
                                 name="input")
        prev_layer = self.input_layer
        if self.useconv3:
            self.fire1 = self._fire_layer(name="fire1", input=self.input_layer, out_channels=self.depth[0], stdd=0.012,
                                          wt_quant=True, act_quant=True, bn_eps=1e-3, pool=self.pooling[0], fpq = self.fpq)
        else:
            self.fire1 = self._mobile_layer(name="fire1", input=prev_layer, out_channels=self.depth[0], stdd=0.012,
                                            wt_quant=True, act_quant=True, bn_eps=1e-3, pool=self.pooling[0],
                                            pool2=True, input_channels = 1, fpq=self.fpq)
        prev_layer = self.fire1
        for layer_count in range(len(self.depth) - 1):
            prev_layer = self._mobile_layer(name="fire{}".format(str(layer_count + 2)), input=prev_layer,
                                            out_channels=self.depth[layer_count + 1],
                                            stdd=0.012, wt_quant=True, act_quant=True,
                                            bn_eps=1e-3, pool=self.pooling[layer_count + 1],
                                            input_channels = self.depth[layer_count], fpq = self.fpq)
```

Figure 5.9. Code Snippet: Forward Graph Fire Layers

Note: Layers have depth wise 2D convolution.

```
if self.fpq:
    self.preds = Conv2D(
    name='conv12', filters=num_output, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="SAME",
    use_bias=False, kernel_initializer=TruncatedNormal(stddev=0.01),
    kernel_constraint=bo.MyConstraints('quant_' + 'conv12'))(prev_layer)
    # reshape
    self.pred_reshaped = Reshape((self.config.ANCHORS, -1))(self.preds)
    # pad for loss function so y_pred and y_true have the same dimensions, wrap in lambda layer
    self.pred_padded = Lambda(self._pad)(self.pred_reshaped)
    model = Model(inputs=self.input_layer, outputs=self.pred_reshaped)

else:
    self.preds = Quantize_Conv(trainable=True, name='conv12', k_size=3, input_channels=self.depth[-1],
                  output_channels=num_output, strides=1, padding="SAME", dilation=1, is_depthwise=False, bits=8,
                  isLSQ=True, isSigned=True, fsq_min_rng=0, fsq_max_rng=1)(prev_layer)

    act_quant3 = Quantize(trainable=True, name="q3", bits=16, isLSQ=True,
                      isSigned=True, fsq_min_rng=0, fsq_max_rng=1)
    self.preds = act_quant3(self.preds)
# reshape
    self.pred_reshaped = Reshape((self.config.ANCHORS, -1))(self.preds)
    model = Model(inputs=self.input_layer, outputs=self.pred_reshaped)
```

Figure 5.10. Code Snippet: Forward Graph Last Convolution Layer

- Interpretation graph

  This block interprets output from the network and extracts predicted class probability, predicated confidence scores, and bounding box values.

  Output of the convnet is a 15 x 20 x 120 tensor. There are 120 channels of data for each of the cells in the grid that is overlaid on the image and contains the bounding boxes and class predictions. The 120 channels are not stored consecutively. Figure 5.11 shows the details.

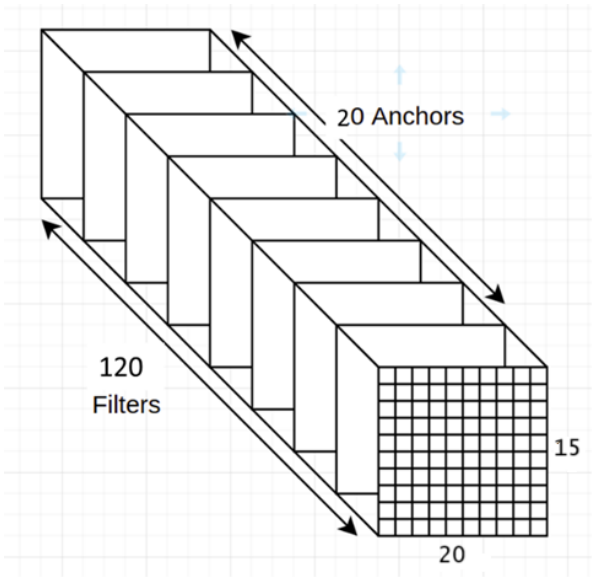

Figure 5.11. Grid Output Visualization #1

For each grid, the cell values are aligned as shown in Figure 5.12:

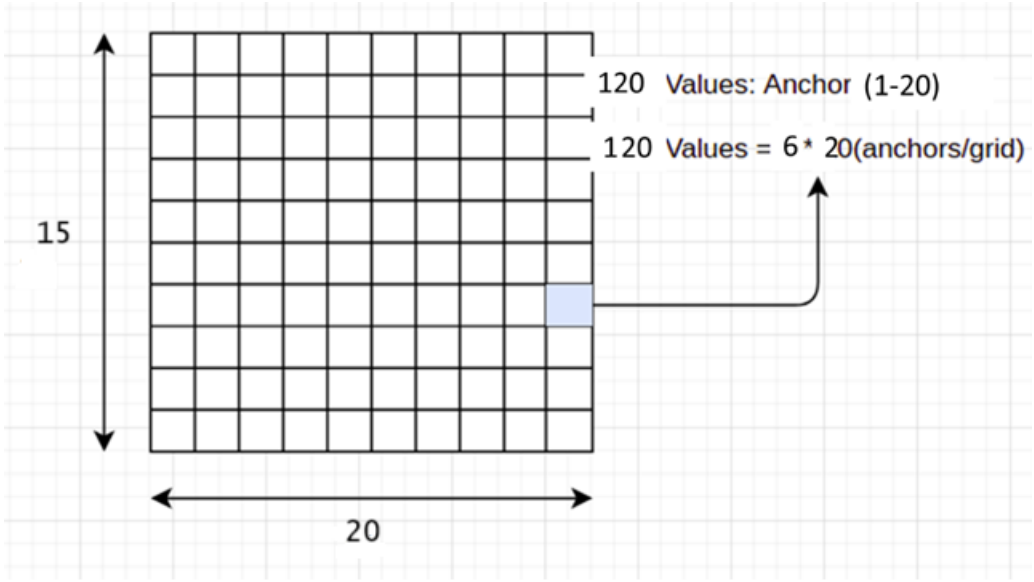

Figure 5.12. Grid Output Visualization #2

As shown in Figure 5.13, output from conv12 layer (4D array of batch size x 15 x 20 x 120) needs to be sliced with proper index to get all values of probability, confidence, and coordinates.

Code snippet Interpret Output: main/utils/utils.py -> slice_predictions()

```
# slice pred tensor to extract class pred scores and then normalize them
pred_class_probs = K.reshape(
    K.softmax(
        K.reshape(
            y_pred[:, :, :, :num_class_probs],
            [-1, config.CLASSES]
        )
    ),
    [config.BATCH_SIZE, config.ANCHORS, config.CLASSES],
)
# number of confidence scores, one for each anchor + class probs
num_confidence_scores = config.ANCHOR_PER_GRID + num_class_probs

# slice the confidence scores and put them trough a sigmoid for probabilities
pred_conf = K.sigmoid(
    K.reshape(
        y_pred[:, :, :, num_class_probs:num_confidence_scores],
        [config.BATCH_SIZE, config.ANCHORS]
    )
)
# slice remaining bounding box_deltas
pred_box_delta = K.reshape(
    y_pred[:, :, :, num_confidence_scores:],
    [config.BATCH_SIZE, config.ANCHORS, 4]
)
```

Figure 5.13. Code Snippet: Interpret Output Graph

For confidence score (between 0 and 1), sigmoid is used.

To predict the class probabilities, there is a vector of NUM_CLASS values at each bounding box. Apply a softmax layer for probability distribution.

- Bboxes: bboxes_from_deltas()

This block calculates bounding boxes based on anchor box and predicated bounding boxes.

- IOU: tensor_iou()

This block calculates Intersection over Union for detected bounding boxes and actual bounding boxes.

- Loss graph

  File Location: main/model/SqueezeDet.py -> loss()

  This block calculates different types of losses that need to be minimized. To learn detection, localization, and classification, the model defines a multi-task loss function. There are three types of losses that are considered for calculation:

  - Bounding box

    This loss is regression of the scalars for the anchors.

```
# bounding box loss
bbox_loss = (K.sum(mc.LOSS_COEF_BBOX * K.square(input_mask * (pred_box_delta - box_delta_input))) / num_objects)
```

Figure 5.14. Code Snippet: Bbox Loss

- Confidence score
  - To obtain a meaningful confidence score, each box's predicted value is regressed against the Intersection over Union of the real and the predicted box. During training, we compare ground truth bounding boxes with all anchors and assign them to the anchors that have the largest overlap (IOU) with each of them.
  - The "closest" anchor is selected to match the ground truth box such that the transformation needed is reduced to minimum. Equation evaluates to 1 if the kth anchor at position-(i, j) has the largest overlap with a ground truth box, and to 0 if no ground truth is assigned. This way, we only include the loss generated by the "responsible" anchors.
  - As there can be multiple objects per image, we normalize the loss by dividing it by the number of objects.

```
# confidence score loss
conf_loss = K.mean(
    K.sum(
        K.square((ious - pred_conf))
        * (input_mask * mc.LOSS_COEF_CONF_POS / num_objects
           + (1 - input_mask) * mc.LOSS_COEF_CONF_NEG / (mc.ANCHORS - num_objects)),
        axis=[1]
    ),
)
```

Figure 5.15. Code Snippet: Confidence Loss

- Class

  The last part of the loss function is cross-entropy loss for classification for each box to perform classification, as we would for image classification.

```
# compute class loss,add a small value into log to prevent blowing up
class_loss = K.sum(labels * (-K.log(pred_class_probs + mc.EPSILON))
                   + (1 - labels) * (-K.log(1 - pred_class_probs + mc.EPSILON))
                   * input_mask * mc.LOSS_COEF_CLASS) / num_objects
```

Figure 5.16. Code Snippet: Class Loss

In one model architecture, you obtain the bounding box prediction, the classification, and the confidence score.

5.1.2.1. Training

- Training data generator

  The main/model/datageberator.py script reads dataset and creates iterator that feeds data to model in a given batch size.

```
# create train generator
train_generator = generator_from_data_path(self.img_names, self.gt_names, config=self.cfg)
val_generator = generator_from_data_path(self.val_img_names, self.val_gt_names, config=self.cfg)
```

Figure 5.17. Code Snippet: dataset iterator

- Data generator scales image pixel values from [0, 255] to [0, 2] as shown in Figure 5.18. It also converts images to gray scale if the model is trained with the –gray flag.
- Current human count training code uses mean = 0 and scale = 1/128 (0.0078125) in the pre-processing step.

```
if gray:
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    orig_h, orig_w = [float(v) for v in img.shape]
else:
    orig_h, orig_w, _ = [float(v) for v in img.shape]

img /= 128.0
```

Figure 5.18. Code Snippet: Image Scale

- Training Callbacks
  - Reduce learning Rate on Plateau:

```
# if flag was given, add reducelronplateu callback
if self.REDUCELRONPLATEAU:
    reduce_lr = ReduceLROnPlateau(monitor='loss', factor=0.1, verbose=1,
                                  patience=4, min_lr=0.0)
    cb.append(reduce_lr)
```

Figure 5.19. Code Snippet: Reduce Learning Rate on Plateau

  - Save Checkpoint:

```
# add a checkpoint saver
ckp_saver = ModelCheckpoint(self.checkpoint_dir + "/model.{epoch:02d}-{loss:.2f}.hdf5", monitor='loss',
                            verbose=0,
                            save_best_only=False,
                            save_weights_only=True, mode='auto', period=1)
cb.append(ckp_saver)
```

Figure 5.20. Code Snippet: Save

  - Transfer Learning

    You can pass model checkpoint or saved keras model as argument in –init .

    Note: Checkpoint architecture and model should match.

    Checkpoints are restored if you are using log directory with existing training.

```
initial_epoch = 0
if self.init_file is not None:
    print("Weights initialized by name from {}".format(self.init_file))
    squeeze.model.load_weights(self.init_file)
    initial_epoch = int(os.path.basename(self.init_file).split('-')[0].split('.')[1])
elif len(os.listdir(self.checkpoint_dir)) > 0:
    ckpt_list = os.listdir(self.checkpoint_dir)
    sorted_list = sorted(ckpt_list, key=self.ckpt_sorting)
    ckpt_path = os.path.join(self.checkpoint_dir, sorted_list[-1])
    squeeze.model.load_weights(ckpt_path)
    print("Weights initialized by name from {}".format(ckpt_path))
    initial_epoch = self.ckpt_sorting(sorted_list[-1])
```

Figure 5.21. Code Snippet: Transfer Learning

- Freezing Layers

  If you are using a pre-trained checkpoint and want to the freeze model until some layer, you can provide a flag:

  –freeze_landmark sub set of layername.
  For example: --freeze_landmark = fire5

```
if self.freeze_landmark is not None:
    # Freezing some of the layers as per user input
    model_layer_names = []
    max_matched_index = -1
    for layer in squeeze.model.layers:
        model_layer_names.append(layer.name)
        if self.freeze_landmark in model_layer_names[-1]:
            layerindex = squeeze.model.layers.index(layer)
            if layerindex > max_matched_index:
                max_matched_index = layerindex
    for layer in squeeze.model.layers[:max_matched_index]:
        layer.trainable = False
    print("layers frozen till {}".format(squeeze.model.layers[max_matched_index].name))
```

Figure 5.22. Code Snippet: Freezing Layers

5.2. Training

To train the machine:

1. Modify training script.

   Training script at @train.sh is used to trigger training. Figure 5.23 shows the input parameters that can be configured.

```
python train_and_prune.py --dataset_path='/home/user/datasets/qvga_dataset' \
                --logdir='log' \
                --val_set_size=10 \
                --validation_freq=10 \
                --gray \
                --early_pooling \
                --filterdepths='32,32,32,64,64,128,128' \
                --sparsity='0.25,0.25,0.35,0.35,0.45,0.45' \
                --epochs='50,5,50' \
                --gpuid=0 \
                --configfile='squeezedet.config' \
                --runpruning \
                # --usecov3 \
                # --usedefaultvalset \
                # --freeze_landmark='fire1' \
                # --init=<path to weights>
```

Figure 5.23. Training Input Parameter

- --dataset_path: dataset directory path. Example: /home/dataset/qvga_dataset.
- --logdir: log directory where checkpoint files are generated while the model is training.
- --val_set_size: validation split percentage.
- --validation_freq: validation frequency in terms of number of epochs.
- --gray: add flag to train model with grayscale images.
- --early_pooling: add flag to use early-pooling.
- --filterdepths: comma separated list of number of features for each layer.
  You can use depth length of 5 to 10 (default: 7).
- --sparsity: list of fraction to prune layer channels.
  Note: First fire layer is not pruned so length of sparsity list should be one less than length of filter depths.
- --epochs: comma separated epoch list for training, pruning, and fine-tuning.
- –gpuid: if the system has more than one gpu, it indicates the one to use.
- --configfile: config file name. If the file exist in logdir, the code will reuse it, otherwise it creates a new file.
- --runpruning: add flag to run pruning after training is completed.
- --usecov3: use normal convolution as first layer instead of depthwise 1 x 1 convolutional layer.
- --usedefaultvalset: add this flag if you want to reuse validation set images from val.txt. If flag is not present, the code creates a new validation set.
- --freeze_landmark: if you want to freeze some layers in network, you can provide a subpart or layer name as argument and the code freezes weights until that layer. Example: fire1, fire6.
- --init: if you want to specify a pre-trained model to load weights.

2. Execute the train.sh script to start training as shown in Figure 5.24., Figure 5.25., and Figure 5.26.

```
$ ./train.sh
Creating Direcory: /home/user/qvga/log
Creating Direcory: /home/user/qvga/log/data
Creating Direcory: /home/user/qvga/log/config
Number of images: 168979
Number of epochs: 111
Number of batches: 8448
Batch size: 20
Epoch 1/110
8448/8448 [==============] - 954s 113ms/step - loss: 0.9749 - loss_without_regularization: 0.9749
- bbox_loss: 0.5738 - class_loss: -1.1921e-07 - conf_loss: 0.4011
Epoch 2/110
8448/8448 [==============] - 944s 112ms/step - loss: 0.8804 - loss_without_regularization: 0.8804
 - bbox_loss: 0.5091 - class_loss: -1.1921e-07 - conf_loss: 0.3713
```

Figure 5.24. Execute Training Script

```
$ ./train.sh
Creating Direcory: /home/user/qvga/log
Creating Direcory: /home/user/qvga/log/data
Creating Direcory: /home/user/qvga/log/config
Config File Already Exist at /home/user/qvga/log/config/squeezedet.config
Target directory already Exists : /home/user/qvga/log/train/checkpoints
Target directory already Exists : /home/user/qvga/log/train/tensorboard
Number of images: 168979
Number of epochs: 111
Number of batches: 8448
Batch size: 20
Weights initialized by name from /home/user/qvga/log/train/checkpoints/model.110-0.74.hdf5
```

Figure 5.25. Execute Training with Transfer Learning

```
$ ./train.sh
Creating Direcory: /home/user/qvga/log
Creating Direcory: /home/user/qvga/log/data
Creating Direcory: /home/user/qvga/log/config
Config File Already Exist at /home/user/qvga/log/config/squeezedet.config
Target directory already Exists : /home/user/qvga/log/train/checkpoints
Target directory already Exists : /home/user/qvga/log/train/tensorboard
Number of images: 168979
Number of epochs: 111
Number of batches: 8448
Batch size: 20
Weights initialized by name from /home/user/qvga/log/train/checkpoints/model.110-0.74.hdf5
layers frozen till fire5_PW
```

Figure 5.26. Execute Training with Transfer Learning + Frozen Layers

Note: If the model is not converging in pruning, you can reduce sparsity and rerun.

3. Start **TensorBoard.**

```
$ tensorboard –logdir=<log directory of training>
```

For example: tensorboard –logdir='./logs/train/tensorboard'

4. Open the local host port on your web browser.

```
$ tensorboard --logdir log/train/tensorboard/
TensorBoard 1.15.0 at http://gpu-server:6006/ (Press CTRL+C to quit)
```

Figure 5.27. TensorBoard: Generated Link

5. Check the training status on **TensorBoard.**

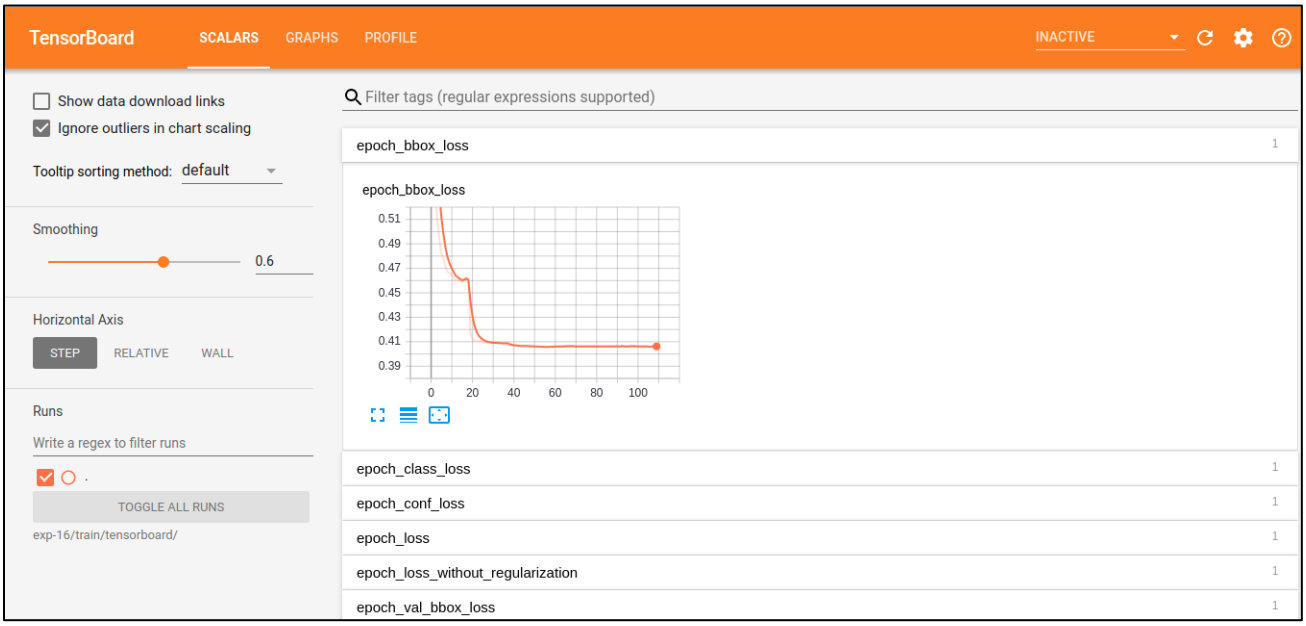

Figure 5.28. TensorBoard

Figure 5.29 shows the backbone graph, Figure 5.30 shows the example files in the log folder, and Figure 5.31 shows example checkpoints and training model.

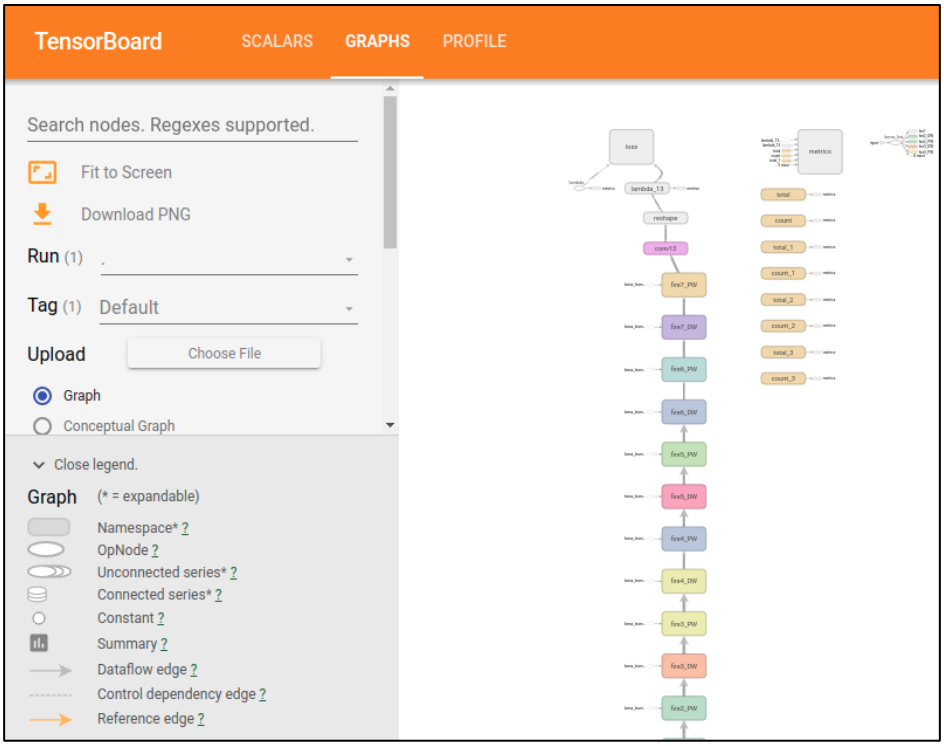

Figure 5.29. Backbone Graph

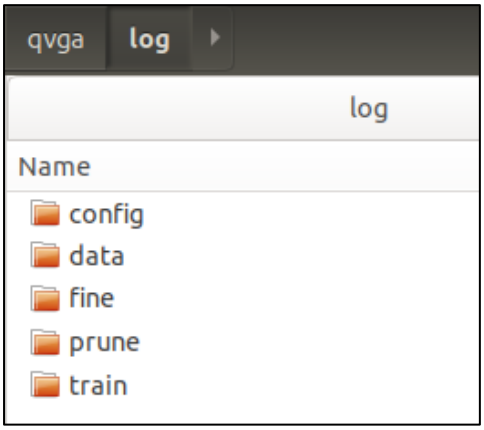

Figure 5.30. Example Files in Log Folder

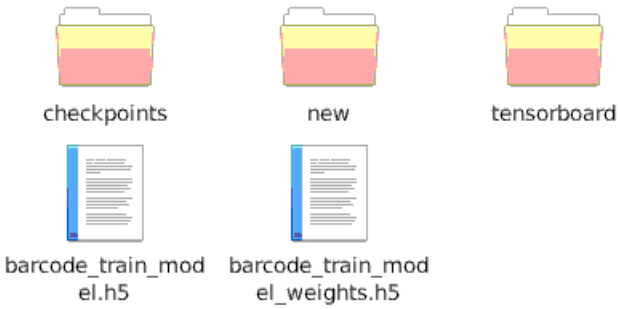

Figure 5.31. Example Checkpoints and Training Model

6. Creating a Frozen File

Note: Refer to Appendix D for creating frozen files for *Yolov5 Architecture* based Barcode Detection Design.

This section describes the procedure for freezing the MobileNet-v1 model, which is aligned with the Lattice sensAI tool. Perform the steps below to generate the frozen *protobuf* file.

6.1. Convert Keras model to tensorflow pb

Barcode-QVGA code contains keras2tf.py under keras-to-tf-converter directory as shown in Figure 6.1.

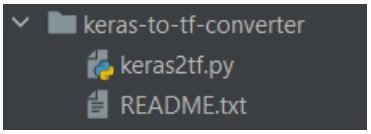

Figure 6.1. Keras to tf converter directory

Note: If you did any quantization change in the training code binary_ops.py, replicate the changes in binary_ops.py.

Run the command below to generate pb in the same path as the h5 file.

```
$ python keras2tf.py –kerasmodel <h5 model path>
```

The script saves the corresponding. pb file in the given input kerasmodel directory.

Note: For models trained on tensorflow version < 2.x, you need to upgrade your environment and create another environment with tensorflow version 2.5x and beyond for generating pb files. You can compile with the latest NNcompiler v6.0.

7. Model Evaluation

Note: Refer to Appendix E for Model Evaluation and mAP calculation for *Yolov5 Architecture* based Barcode Detection Design.

This section describes steps to calculate model performance in terms of MAP.

7.1. Run Inference on test set.

Barcode Detection QVGA code contains qvga_inference_320x240.py under inference directory as shown in Figure 7.1.

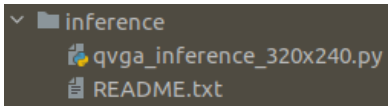

Figure 7.1. Inference directory

Note: If you did any change in the training code regarding image size, number of anchors, or grid size, you need to replicate the changes in the inference script.

Run the command below to run inference on test set.

```
$ python qvga_inference_320x240.py –pb <converted pb path> --input_image <test set images path>
```

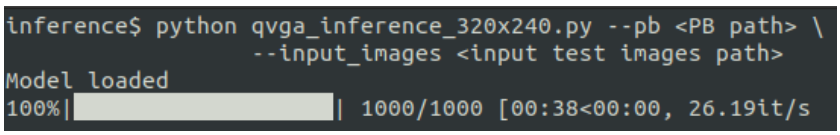

Figure 7.2. Run Inference

The command above saves images with bbox drawn in inference_output/image_output and produce kitti output in inference_output/predictions as shown in Figure 7.3.

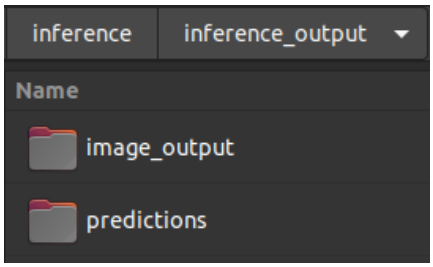

Figure 7.3. Inference Output

7.2. Calculate MAP

The Barcode Detection QVGA code contains qvga_inference_320x240.py in the inference directory as shown in Figure 7.4.

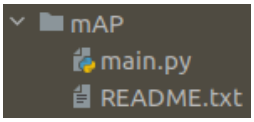

Figure 7.4. mAP Directory Structure

Run the command below to calculate mAP using predictions generated from inference and groundtruth from test set.

```
$ python main.py –input_images <input test set images path> --ground_truth <input test set labels path> --predictions <path to prediction generated from inference> --no-animation –no-plot
```

```
mAP$ python main.py --input_images <input test images path> \
                    --ground_truth <input testset groundtruth path> \
                    --predictions <pridictions path> \
                    --no-animation  --no-plot
XX.XX% = person AP
mAP = XX.XX%
```

Figure 7.5. mAP Calculation

After a successful run, the script shows the mAP for each class and the total mAP, as shown in Figure 7.5.

8. Creating a Binary File with the Lattice sensAI Software

Note: Refer to Appendix F for Binary File Creation for *Yolov5 Architecture* based Barcode Detection Design.

This section describes how to generate a binary file using the Lattice sensAI version 6.0 software.

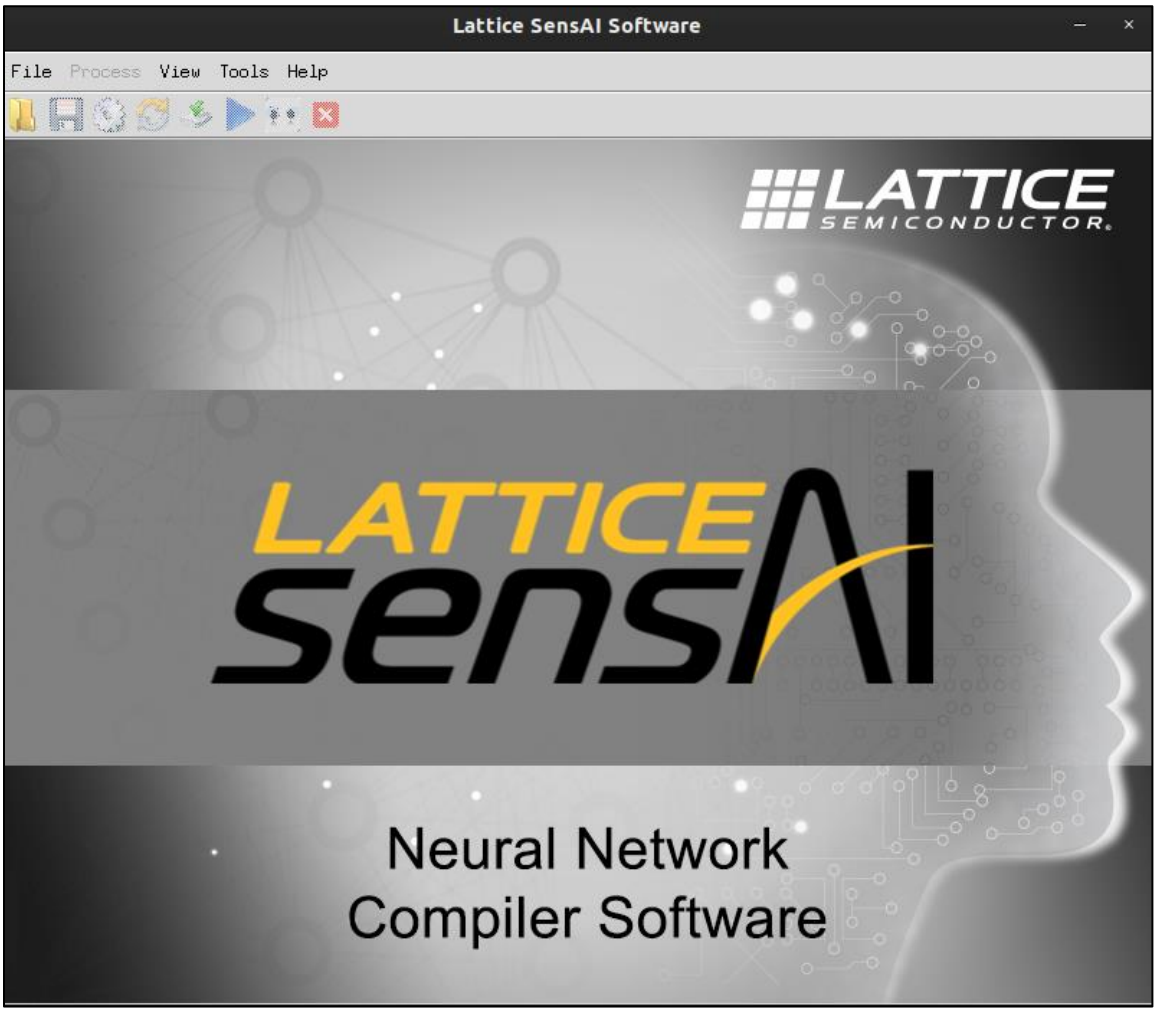

Figure 8.1. sensAI – Home Screen

To create the project in theLattice sensAI software:

1. Click **File > New**.
2. Enter the following settings:
   - Project name
   - Framework – TensorFlow
   - Class – CNN
   - Device – CertusPro-NX
   - IP – Advanced_CNN
3. Click **Network File** and select the network (PB) file.

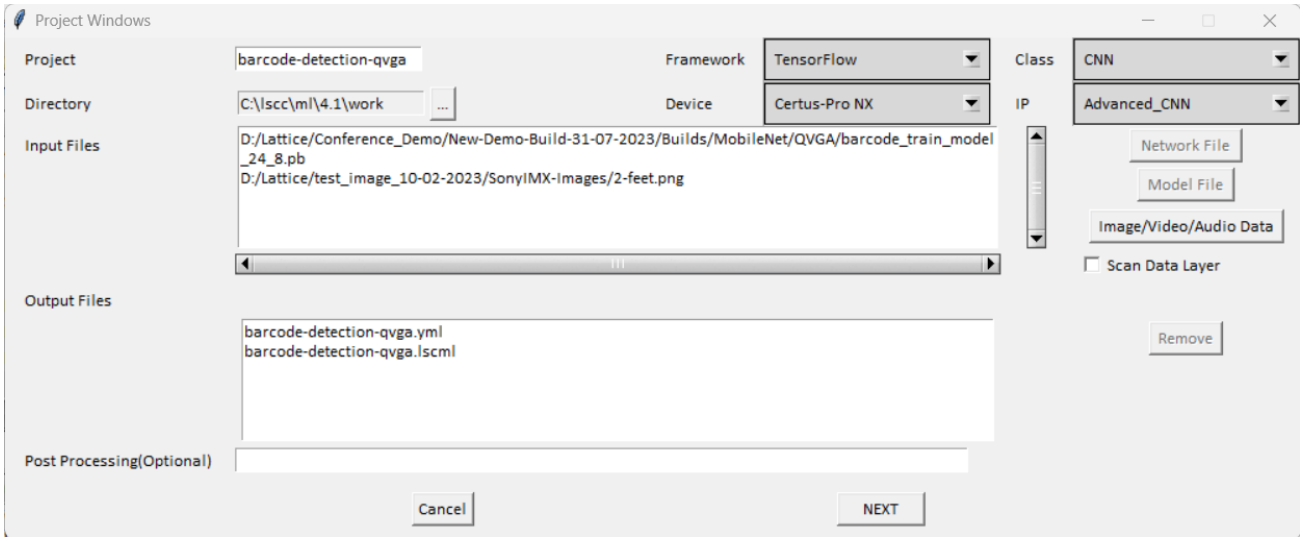

Figure 8.2. sensAI – Select Framework, Device, IP, Network File, and Image/Video Data

4. Click the **Image/Video/Audio Data** button and select the input image file.

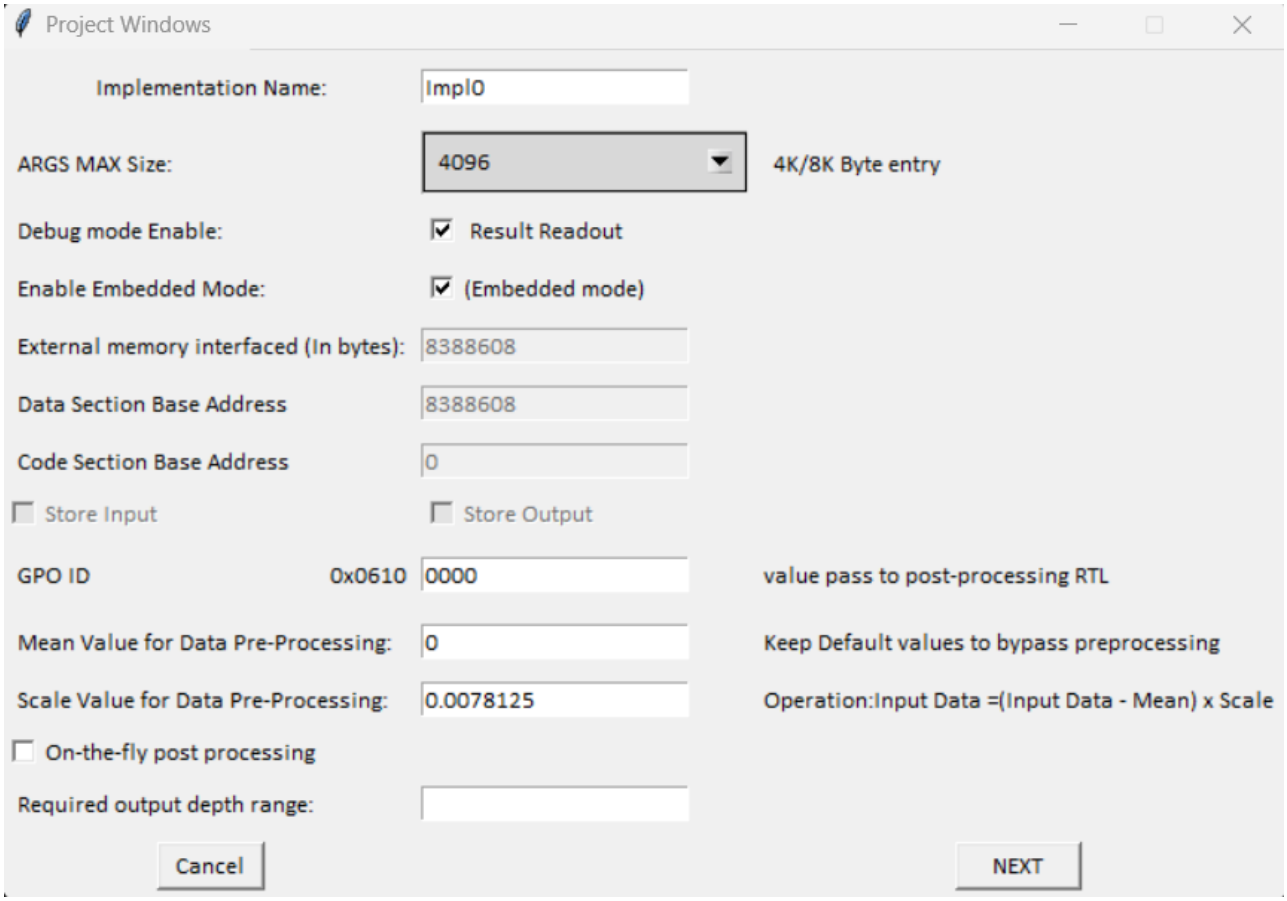

Figure 8.3. sensAI – Select Debug Mode and Enable Embedded Mode

5. Click **Next**.
6. Set the following attributes:
   - **Mean Value for Data Pre-Processing**: 0
   - **Scratch Pad Memory Block Size**: 8192
   - **ARGS MAX Size**: 4096
   - **External Memory Interfaced**: 8388608
   - **Scale Value for Data Pre-Processing**: 0.0078125
   - **Number of VE SPD**: 4
   - **Multi-port parallel**: 2

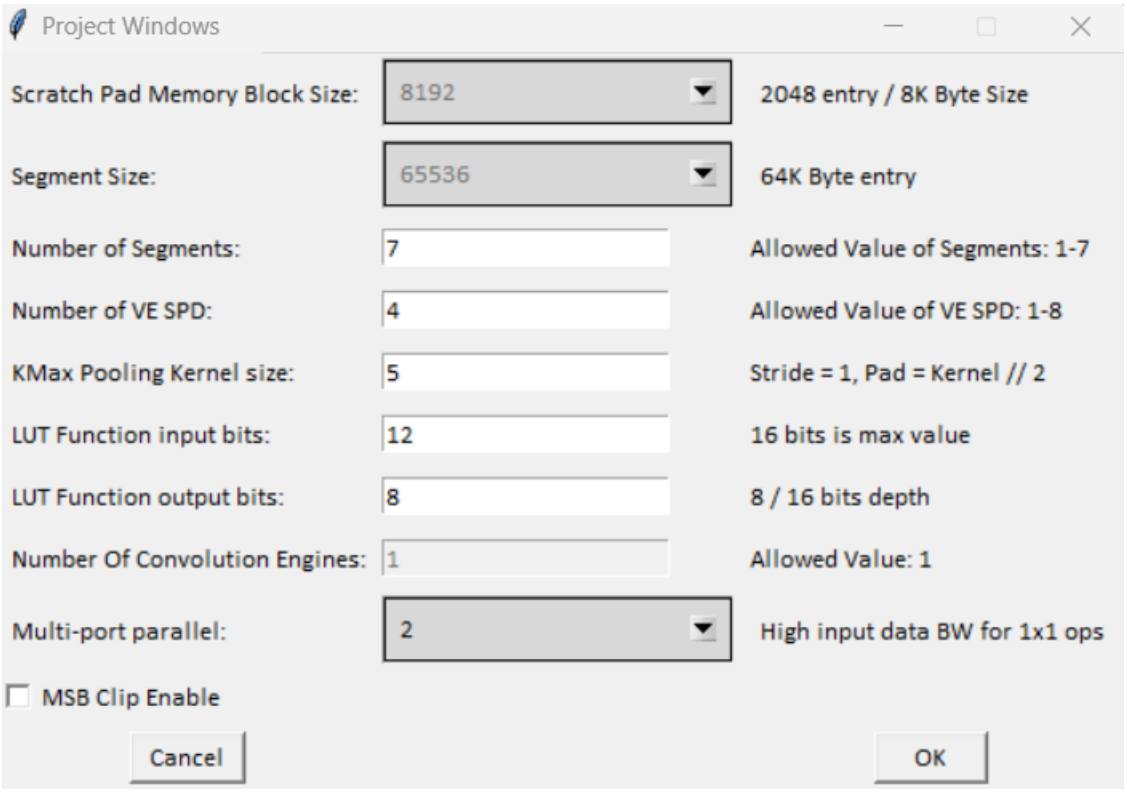

Figure 8.4. sensAI – Update Project Settings

7. Click **Ok** to create project.
8. Double-click on **Analyze** as shown in Figure 8.5.

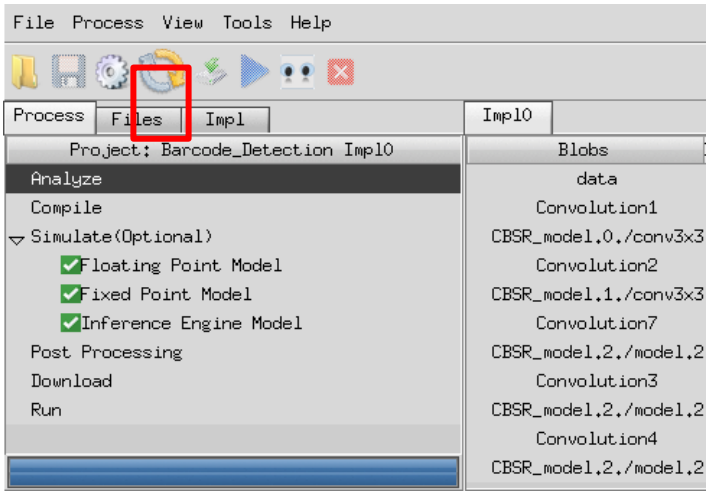

Figure 8.5. Analyze and Compile Project

9. Confirm the Q format of each layer as shown in Figure 8.6.

Blobs	Data Format (Analyzed)	Stored Data Format	Internal(External) Mem Bytes
data	8.7	1.7	None
Convolution1	5.10	5.10	None
CBSR_fire1_DW/depthwise	8.7	1.7	None
Convolution2	5.10	5.10	None
CBSR_fire1_PW/Conv2D	8.7	1.7	None
Convolution3	5.10	5.10	None
CBSR_fire2_DW/depthwise	8.7	1.7	None
Convolution4	5.10	5.10	None
CBSR_fire2_PW/Conv2D	8.7	1.7	None
Convolution5	5.10	5.10	None
CBSR_fire3_DW/depthwise	8.7	1.7	None
Convolution6	5.10	5.10	None
CBSR_fire3_PW/Conv2D	8.7	1.7	None
Convolution7	5.10	5.10	None
CBSR_fire4_DW/depthwise	8.7	1.7	None
Convolution8	5.10	5.10	None
CBSR_fire4_PW/Conv2D	8.7	1.7	None
Convolution9	5.10	5.10	None
CBSR_fire5_DW/depthwise	8.7	1.7	None
Convolution10	5.10	5.10	None
CBSR_fire5_PW/Conv2D	8.7	1.7	None
Convolution11	5.10	5.10	None
CBSR_fire6_DW/depthwise	8.7	1.7	None
Convolution12	5.10	5.10	None
CBSR_fire6_PW/Conv2D	8.7	1.7	None
conv12/Conv2D	5.10	5.10	None
CBSR_conv12/Conv2D	8.7	5.10	None

Figure 8.6. Q Format Settings for Each Layer (1)

10. Double-click **Compile** to generate the firmware and filter binary file as shown in Figure 8.7.

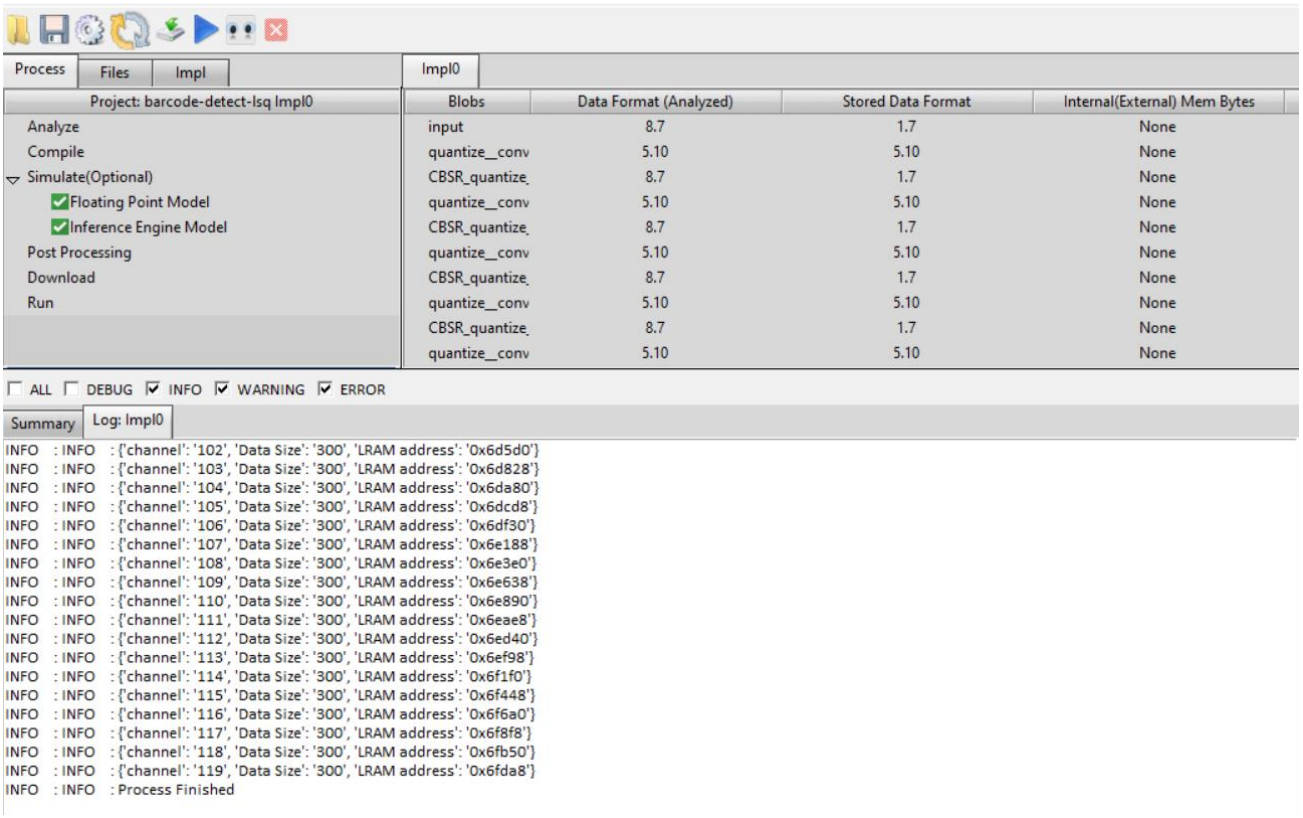

Figure 8.7. Compiled Project

The firmware bin file location is displayed in the compilation log. Use the generated firmware bin on hardware for testing.

9. Hardware (RTL) Implementation

Note: Refer to Appendix G for Hardware (RTL) Implementation for *Yolov5 Architecture* based Barcode Detection Design.

9.1. Top Level Information

9.1.1. Block Diagram

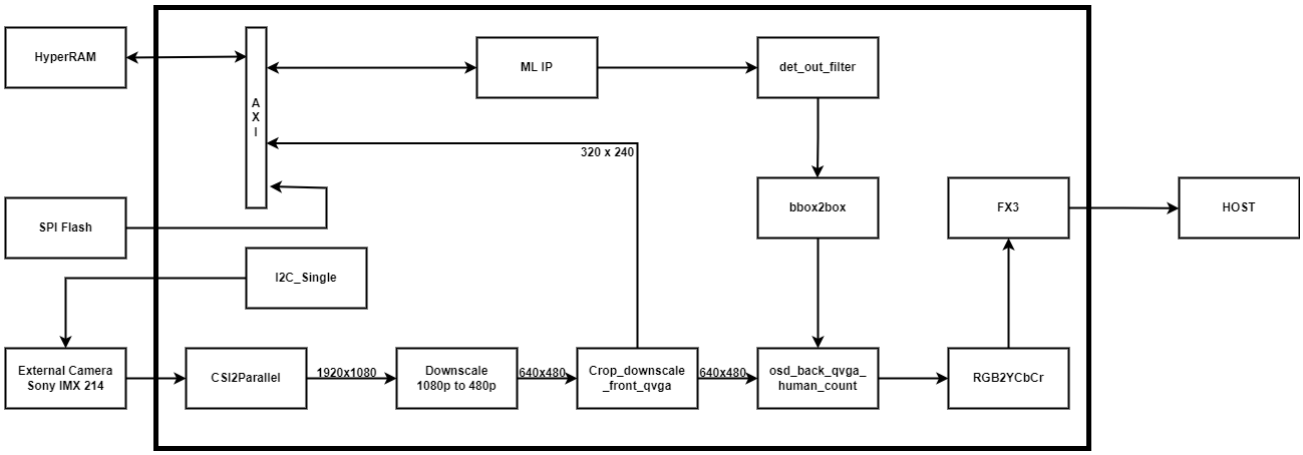

Figure 9.1. Top Block Diagram of Barcode Detection with CertusPro-NX Voice and Vision ML (Rev A) Board

9.1.2. Operational Flow

- The external camera Sony IMX214 is configured using I^2C Master Block i2c_single.v.
- The real time input image data is received by the video path. The RAW10 data from (csi2_to_parallel.v) is sent to downscale_1080p_to480p.v, which downscales the 1080p (1920x1080) image data received from the camera to 480p (640x480). Downscaling is performed by selecting 4 lines out of every 9 lines and 1 pixel out of 3 pixels. Overall, 4 Valid pixels are passed from 27 pixels (9 x 3 window).

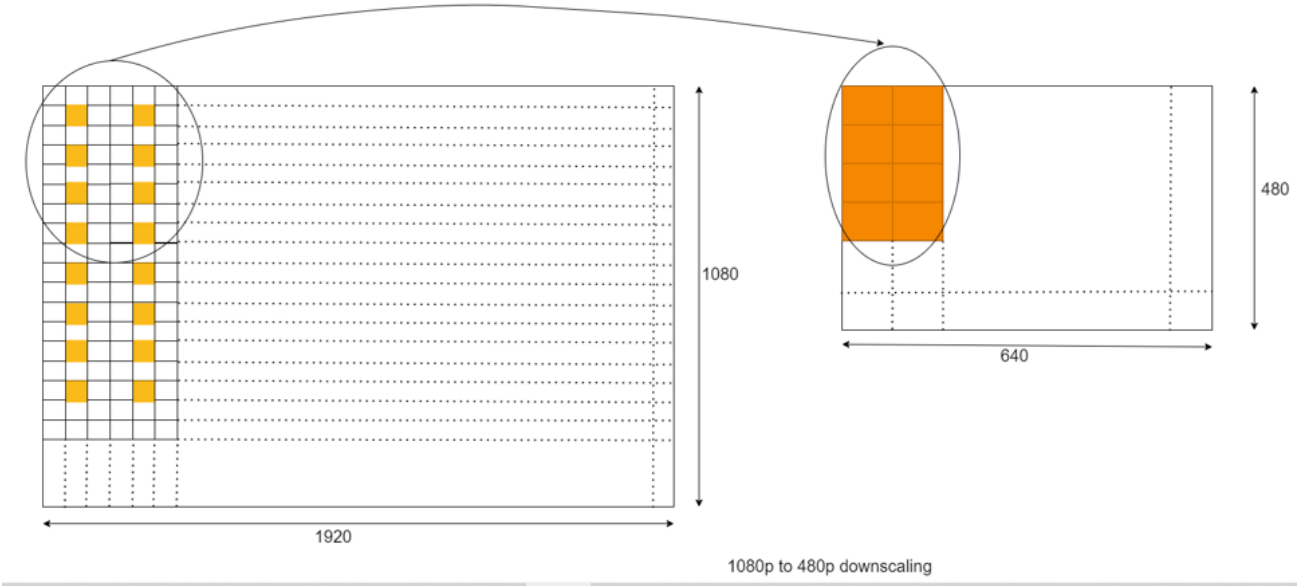

Figure 9.2. Downscaling from 1080p to 480p (Obtaining 4x2 Pixels in 480p from 18 x 6 Pixels in 1080p)

- After the image is downscaled to 480p, it is passed to pre-processing crop_downscale_front_qvga.v, which performs crop and downscale operations to provide a compatible input image resolution of 320 x 240 to the ML engine.
- The 6 Mb firmware BIN file (.mcs) is loaded to the SPI Flash module *spi_loader_spram.v* configured with starting address 24'h300000 to end address 24'h800000.
- ML Engine receives the downscaled image data from crop_downscale_front_qvga.*v* through the axi interface and the firmware file through the AXI hyperbus interface to provide an inference result output.
- ML inference output is stored in a FIFO register and passed to det_out_filter.v for post processing.
- For final output display, the *osd_back_qvga_human_count.v* module performs barcode detection on the image received from the video path and the downscaled image obtained from the crop and downscale module using the ML object detection pixel information obtained from the det_out_filter.*v* block.
- Output of osd_back_qvga_human_count.v is passed to the rgb2ycbcr module, which converts RGB data of pixels to YCbCr data which is compatible for FX3. FX3 is passed to the host for display.

9.1.3. Core Customization

Table 9.1 and Table 9.2 show the available parameters.

Table 9.1. Core Parameters

Parameter	Value	Description
USE_ML	1'b1	Indicates 1: Enable/0: Disable to use CNN engine.
EN_UART	1'b0	Indicates 1: Enable/0:Disable UART for video output.
FLASH_START_ADDR	24'h300000	Indicates starting address to load Firmware in external SPI flash.
FLASH_END_ADDR	24'h800000	Indicates ending address to load Firmware in external SPI flash.
CNN IP Attributes	**Default Value**	**Description**
Number of LRAMs	7	Number of LRAMs in the ML SPD.
Number of VE SPD Packs	8	1 VE SPD Pack = 4 VE SPDs. Number of EBRs in a VE SPD is the attribute below.
Number of EBRs in VE SPD	4	1 VE SPD Pack = 4 VE SPDs. This attribute specifies the number of EBRs in a VE SPD.
Max AXI4 external memory DMA Burst	31	Max burst limit on AXI4 bus during DMA transactions with external memory.
LMMI read mode	BYTE	Access width mode for LMMI read interface (reading data out of the IP) BYTE: byte mode; HWORD: 16 bit half word mode; WORD: 32 bit word mode.
LMMI Write mode	BYTE	Access width mode for LMMI write interface (writing data into the IP) BYTE: byte mode; HWORD: 16-bit half word mode; WORD: 32 bit word mode.
No. of Convolution engines	1	Number of parallel convolution engines. Higher compute throughput can be achieved with a greater number of engines, at the expense of higher utilization of DSP resources.
Enable 4 parallel ports for 1x1 convolution	Unchecked	Enable four parallel ports internally for high bandwidth data accesses for 1 × 1 convolutions.
Enable Vector ALU	Checked	Enables the Vector Engine for pixelwise ALU operations.
Enable 5x5 convolution	Checked	Enables 5 × 5 convolutions inside the Conv EU.
Enable 7x7 convolution	Checked	Enables 7 × 7 convolutions inside the Conv EU.
Enable Argmax pool	Checked	Enables the Argmax pool compute module.
Enable Maxpool stride=1 module	Checked	Enables the 1-D stride=1 maxpool compute engine.
Avant Mode	Unchecked	**IMPORTANT:** This must be enabled for LAV-AT devices and disabled for CertusPro-NX devices.

Table 9.2. Camera Parameters

Module	Parameter/ Port	Sony IMX258	Sony IMX214
vvml_barcode_detection_top.v	SENSOR_SLAVE_ADDR	7'h1A	7'h10
vvml_barcode_detection_top.v	NUM_OF_TRANS_I2C	d80	d81
vvml_barcode_detection_top.v (downscale_1080p_to480p)	USE_IMX214	1'b0	1'b1
i2c_m_ctrl.v	Instance	rom_sonyimx258 u_rom_sonyimx258	rom_sonyimx214 u_rom_sonyimx214
ROM IP (rom_sonyimx258/rom_sonyimx214)	Config ROM Read Address	9 Bit	9 Bit
ROM IP (rom_sonyimx258/rom_sonyimx214)	ROM Address Depth	320	324

9.2. Architectural Details

9.2.1. Pre-processing Operation

- The *crop_downscale_front_qvga.v* video processing block is used to crop and downscale the image data to make it compatible with the ML engine.
- Masking values for incoming image data from the camera are set to capture the image data of resolution 640 × 480.
- As shown in Figure 9.3, a 640 × 480 image is downscaled to 320 × 240 image resolution using block size 2.

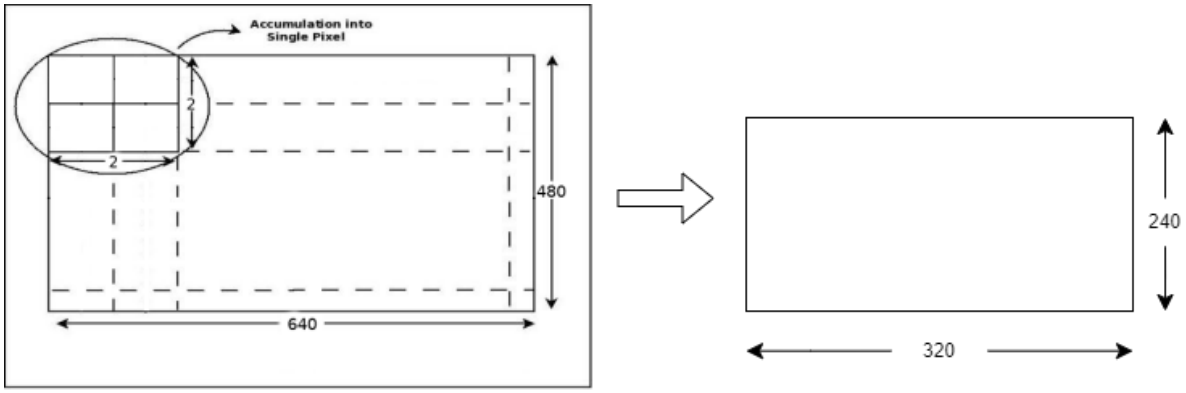

Figure 9.3. Downscaling Image

- The accumulated pixel values are written to the accumulation buffer while the data from the buffer is sent to the ML engine for inference through the line buffer.

9.2.2. Post-processing operation

The post-processing operation is explained as follows:

- After the image is scaled down to 320 x 240, it is passed to the ML engine through the axi_ws2m1.v module.
- When the ML engine completes processing the input image, it asserts the enable signal and starts transmitting processed data at its output port which is stored in FIFO. This data is then used in calculating bounding boxes in det_out_filter.v.
- When bounding box co-ordinates are calculated, it is passed to osd_back_qvga_human_count.v.

- The *osd_back_qvga_human_count.v* module receives mainly the 320 x 240 downscaled image from the pre-processing module and the ML engine result data from the post processing block.
- Using the pixel information of barcode detection received from the post-processing block, the OSD module performs text and graphics addition over the 320 x 240 downscaled image.
- In the final segmentation output display, the barcode can be observed in grayscale with bounding box.
- Upon completion of data processing in OSD, it is sent to FX3 via rgb2ycbcr.

The pre-processing and post-processing cycle goes on until the board is powered on and the real time image is captured by the camera.

10. Creating the FPGA Bitstream File

This section provides the procedure for creating your FPGA bitstream file using the Lattice Radiant Software.

To create the FPGA bit stream file, follow the steps below.

1. Open the Lattice Radiant software, as shown in Figure 10.1.

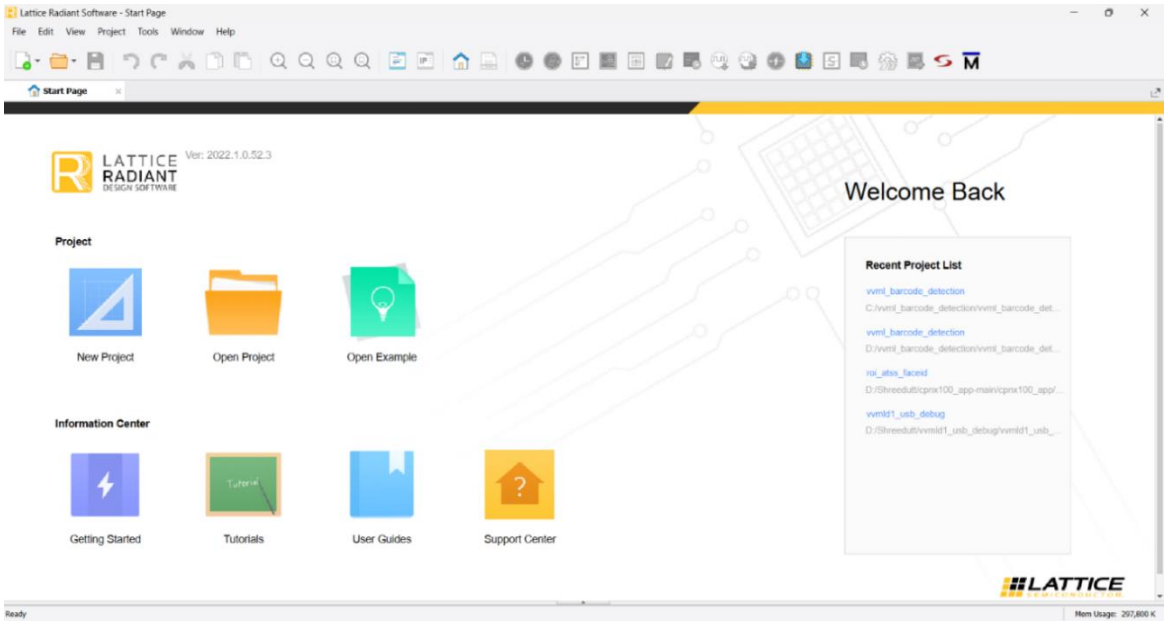

Figure 10.1. Radiant Software

2. Click **File > Open Project** and from project database, open the Radiant project file (.rdf) from the *vvml_barcode_detection* folder, as shown in Figure 10.2.

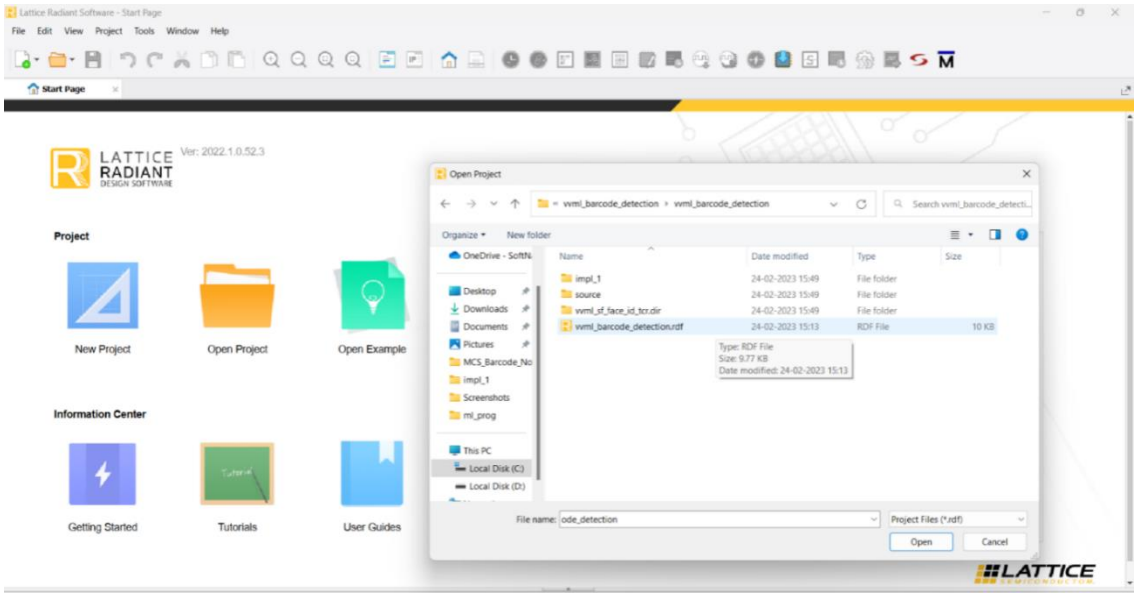

Figure 10.2. Radiant Software – Open Project

3. Click **Export Files** to generate the bit file. View the log message in Export Reports that indicates the generated bitstream. Find this bit file under */vvml_barcode_detection/impl_1,* as shown in Figure 10.3.

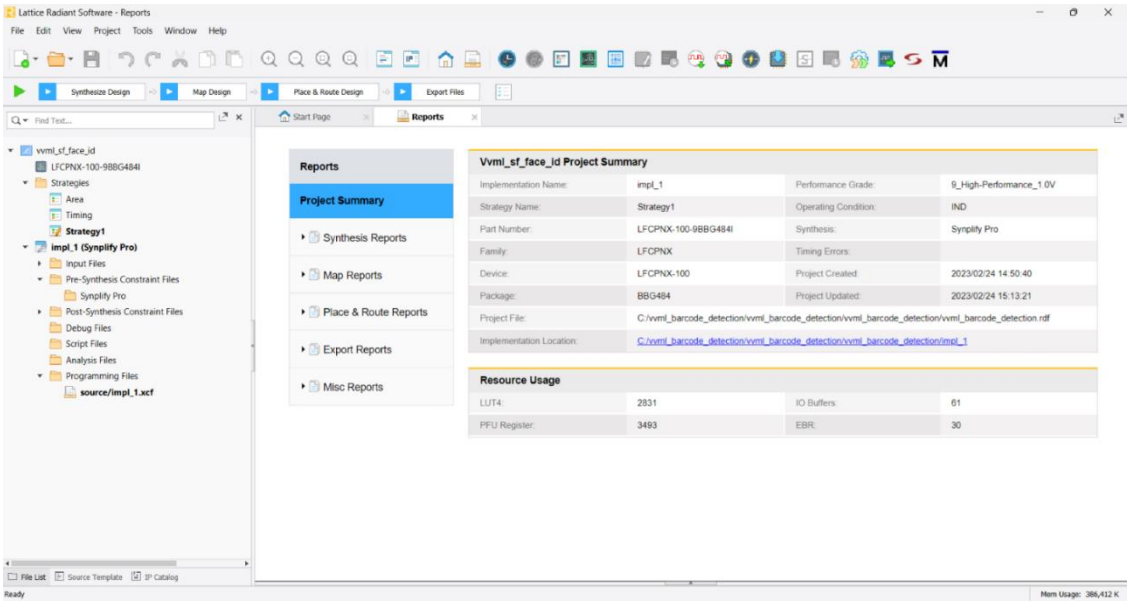

Figure 10.3. Radiant Software – Bitstream Generation Export Report

Appendix A. Yolov5 Training Code Preparation

This section provides information on the Convolution Neural Network configuration of the Yolov5 Barcode Detection design.

1 x 160 x 160 x 1	
Focus Layer	ConvBNReLU - # where: • Conv3x3-BatchNorm-ReLU • # - The number of filters For example, ConvBNReLU - 8 = 8 3 x 3 convolution filter followed by BatchNorm and ReLU BottleNeckCSP - # where: # - The number of outputs
ConvBNReLU-16	
MaxPool	
ConvBNReLU-32	
MaxPool	
BottleneckCSP-32	
ConvBNReLU-64	
BottleneckCSP-64	
ConvBNReLU-128	
BottleneckCSP-128	
ConvBNReLU-256	
BottleneckCSP-256	
ConvBNReLU-256	
Conv1x1-20	

Figure A.1. Yolov5 Architecture

As shown in Figure A.1, this model contains convolution (Conv), batch normalization (BN), Relu and Bottleneck CSP.

- Layer information
  - Focus Layer
    Focus layer acts like a transformation from space to depth. In YOLOv5, we reduce the cost of Conv2d computation using tensor reshaping to reduce space (resolution) and increase the depth (number of channels). The input is transformed as such: [b, c, h, w] -> [b, c*2, h//2, w//2].
  - Convolutional Layer

    In general, the first layer in a CNN is always a convolutional layer**.** Each layer consists of several filters (sometimes referred to as kernels), which convolves with input layer/image and generates an activation map (that is the feature map). This filter is an array of numbers. The numbers are called weights or parameters. Each of these filters can be thought of as feature identifiers, such as straight edges, simple colors, and curves and other high-level features. For example, the filters on the first layer convolve around the input image and activate (or compute high values) when the specific feature (such as curve) it is looking for is in the input volume.
  - Relu (Activation layer)

    After each convolutional layer, it is a convention to apply a nonlinear layer (or activation layer) immediately afterward. The purpose of this layer is to introduce nonlinearity to a system that has just been computing linear operations during the convolutional layers (just element wise multiplications and summations). The Relu layer applies the function f(x) = max (0, x) to all of the values in the input volume. This layer changes all negative activations to 0. This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the convolutional layer.

- Bottleneck CSP

  CSP stands for cross stage partial network. YOLO is a deep network. YOLO uses residual and dense blocks to enable the flow of information to the deepest layers and to overcome the vanishing gradient problem. However, a downside of using dense and residual blocks is redundant gradients. CSP helps reduce this issue by truncating the gradient flow.
  YOLOv5 employs CSP strategy to partition the feature map of the base layer into two parts and then merges them through a cross-stage hierarchy as shown in Figure A.2.

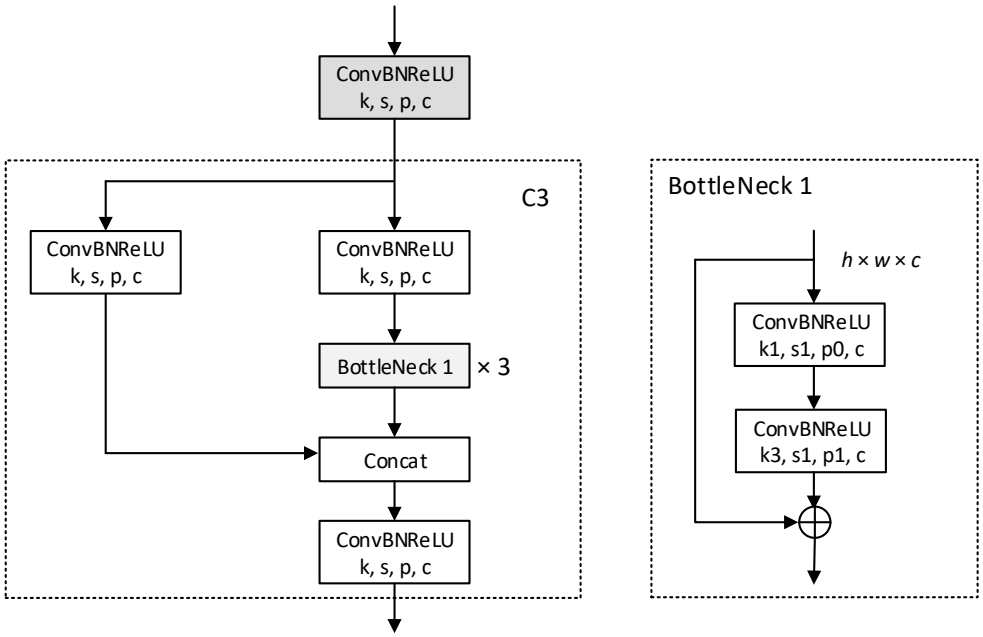

Figure A.2. Yolov5: Bottleneck CSP Architecture

  Applying this strategy comes with big advantages to YOLOv5 as it helps reduce the number of parameters and a great amount of computation (less FLOPS), which leads to increasing the inference speed that is crucial in real-time object detection models.

- Pooling Layer

  After some Relu layers, you may choose to apply a pooling layer. It is also referred to as a down sampling layer. In this category, there are several layer options, with Maxpooling being the most popular. This takes a filter (normally of size 2 × 2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every sub region that the filter convolves around.

  The intuitive reasoning behind this layer is that once you know that a specific feature is in the original input volume (there is to be a high activation value), its exact location is not as important as its relative location to the other features. This layer drastically reduces the spatial dimension (the length and the width change but not the depth) of the input volume. This serves two main purposes. The first is that the number of parameters or weights is reduced by 75%, thus lessening the computation cost. The second is that it can control over fitting. This term refers to when a model is so tuned to the training examples that it is not able to generalize well for the validation and test sets. A symptom of over fitting is having a model that gets 100% or 99% on the training set, but only 50% on the test data.

- Batch Normalization Layer

  The batch normalization layer reduces the internal covariance shift. To train a neural network, the input data is preprocessed. For example, you can normalize all data so that it resembles a normal distribution, which means, zero mean and a unitary variance, to prevent the early saturation of non-linear activation functions like the sigmoid function, assuring that all input data is in the same range of values.

But the problem appears in the intermediate layers because the distribution of the activations is constantly changing during training. This slows down the training process because each layer must learn to adapt them to a new distribution in every training step. This problem is known as internal covariate shift.

The batch normalization layer forces the input of every layer to have approximately the same distribution in every training step by following the process below during training.

Calculate the mean and variance of the layers input:

- Normalize the layer inputs using the previously calculated batch statistics.
- Scales and shifts to obtain the output of the layer.

This makes the learning of layers in the network more independent of each other and allows you to be more flexible about weight initialization, works as regularization in place of dropout and other regularization techniques.

- Drop-out Layer

  Drop-out layers have a very specific function in neural networks. After training, the weights of the network are tuned to the training examples they are given such that the network does not perform well when given new examples. This layer “drops out” a random set of activations in that layer by setting them to zero. It forces the network to be redundant. This means the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out. It makes sure that the network is not getting too “fitted” to the training data and thus helps alleviate the over fitting problem.

  Note: This layer is only used during training, but not during test time.

- Fully connected Layer

  This layer takes an input volume (whatever the output is of the conv or Relu or pool layer preceding it) and outputs an N dimensional vector where N is the number of classes that the program must choose from.

- Quantization

  Quantization is a method to bring the neural network to a reasonable size, while also achieving high performance accuracy. This is especially important for on-device applications, where the memory size and number of computations are necessarily limited. Quantization for deep learning is the process of approximating a neural network that uses floating-point numbers by a neural network of low bit width numbers. This dramatically reduces both the memory requirement and computational cost of using neural networks.

The architecture above provides nonlinearities and preservation of dimension that help to improve the robustness of the network and control over fitting.

Appendix B. Yolov5 Barcode Detection Network Output

The Barcode Detection network generates the output tensor of dimension (BATCH_SIZE, Anchor Width, Anchor Height, 60). This can be interpreted by Yolov5 detection.

Barcode Detection Demo has one class as shown below:

- barcode

From the input image model, the network first extracts feature maps, overlays them with a W × H grid and at each cell computes K pre-computed bounding boxes called anchors. Each bounding box has the following:

- Four scalars (x, y, w, h)
- A confidence score (Pr(Obj)xIOU)
- C conditional classes

Hence the current model architecture has a fixed output of W × H × K (4 + 1 + C). Where:

- W, H = Grid Size
- K = Number of Anchor boxes
- C = Number of classes for which we want detection

Based on the description above, the model has a total of 12000 output values. It is derived from following:

```
10 × 10 grid
     20 anchor boxes per grid
        6 values per anchor box. It consists of:
          4 bounding box coordinates (x, y, w, h)
          1 class probability
          1 confidence score
```

Total 10 × 10 × 20 × 6 = 12000 output values.

Note: Smaller images do not work as well with the resulting spacer grid. If your images are smaller, stretch the images to default size. You can also up-sample them beforehand.

If your images are bigger and you are not satisfied with the results of the default image size, you can try using a denser grid, as details might get lost during the downscaling.

Appendix C. Yolov5 Training Code Overview

The training code can be divided into the following parts:

- Model config
- Model building
- Model freezing
- Data preparation
- Training for overall execution flow

Details of each can be found in the subsequent sections.

Model Config

The demo uses Kitti dataset and Yolov5 model. *kitti_yolov5_config()* maintains all the configurable parameters for the model. A summary of configurable parameters is shown below:

- Image size
  - Change mc.IMAGE_WIDTH and mc.IMAGE_HEIGHT to configure Image size (width and height) in *src/config/kitti_squeezeDet_config.py*

```
mc.IMAGE_WIDTH = 160
mc.IMAGE_HEIGHT = 160
```

Figure C.1. Yolov5 Code Snippet: Input Image Size Config

  - Grid dimension would be H = 10 and W = 10. *anchor_shapes* variable of *set_anchors()* in *src/config/kitti_yolov5_config.py* indicates anchors width and heights. Update based on anchors per grid size changes.

```
def set_anchors(mc):
    H, W, B = 10, 10, 20
```

Figure C.2. Yolov5 Code Snippet: Anchors Per Grid Config #1 (Grid Sizes)

- Batch size

  Change *mc.BATCH_SIZE* in *src/config/kitti_yolov5_config.py* to configure batch size.

- Output classes

  Edit classes in *src/config/config.py* as shown below.

```
cfg.CLASS_NAMES = ("barcode",)
```

Figure C.3. Yolov5 Code Snippet: Classes

- Anchors per grid
  - Change mc.ANCHOR_PER_GRID in *src/config/kitti_yolov5_config.py* to configure anchors per grid.

```
mc.ANCHOR_BOX = set_anchors(mc)
mc.ANCHORS = len(mc.ANCHOR_BOX)
mc.ANCHOR_PER_GRID = 20
```

Figure C.4. Yolov5 Code Snippet: Anchors Per Grid Config #2

  - Change hard coded anchors per grid in *set_anchors()* in *src/config/kitti_yolov5_config.py*. Here B (value 20) indicates anchors per grid.

- If you want to run network on your own dataset, you need to adjust the anchor sizes. Anchors are prior distribution over what shapes your boxes should have. The better these fit the true distribution of boxes, the faster and easier your training can be.
- To determine anchor shapes, first load all ground truth boxes and pictures. If your images do not have all the same size, normalize the height and width by the image's height and width. All images need to be normalized before being fed to the network. You need to do the same to the bounding boxes and consequently, the anchors.
- Second, perform a clustering on these normalized boxes. You can use k-means without feature whitening and determine the number of clusters either by eyeballing or by using the elbow method.
- Check for boxes that extend beyond the image or have a zero to negative width or height.

```
anchor_shapes = np.reshape(
    [np.array(
        [[ 22,  52],
        [ 19,  23],
        [ 52,  77],
        [ 69, 113],
        [ 35,  54],
        [ 62,  98],
        [ 23,  62],
        [ 56,  87],
        [ 38,  61],
        [ 40,  93],
        [ 14,  37],
        [ 46, 70],
        [ 16,  79],
        [ 43,  63],
        [ 15,  54],
        [ 81, 131],
        [ 20,  45],
        [ 12,  15],
        [ 28,  47],
        [ 32,  72]]
                    )] * H * W,(H, W, B, 2))
```

Figure C.5. Yolov5 Code Snippet: Anchors per Grid Config #3

- Training parameters

  Other training related parameters like learning rate, loss parameters and different thresholds can be configured in *src/config/kitti_yolov5_config.py*.

```
mc.WEIGHT_DECAY = 0.0001
mc.LEARNING_RATE = 0.00001
mc.DECAY_STEPS = 20000
mc.MAX_GRAD_NORM = 1.0
mc.MOMENTUM = 0.9
mc.LR_DECAY_FACTOR = 0.9

mc.LOSS_COEF_BBOX = 10.0
mc.LOSS_COEF_CONF_POS = 75.0
mc.LOSS_COEF_CONF_NEG = 100.0
mc.LOSS_COEF_CLASS = 1.0

mc.PLOT_PROB_THRESH = 0.4
mc.NMS_THRESH = 0.4
mc.PROB_THRESH = 0.005
mc.TOP_N_DETECTION = 8

mc.DATA_AUGMENTATION = True
mc.DRIFT_X = 90
mc.DRIFT_Y = 60
mc.EXCLUDE_HARD_EXAMPLES = False
```

Figure C.6. Yolov5 Code Snippet: Training Parameters

Model Building

The Yolov5 class constructor builds models, which can be divided into the following sections:

- Forward graph
- Interpretation graph
- Loss graph
- Training graph
- Visualization graph

Details of each graph are explained as follows:

- **Forward graph**
  - CNN architecture consists of Focus Layer, Convolution, Batch normalization, Relu, and Maxpool layers.
  - Forward graph consists of one focus layer, one fire layer, and 17 bsconv layers.

```
fl_w_bin = 8
fl_a_bin = 8
ml_w_bin = 8
ml_a_bin = 8
ll_w_bin = 8
ll_a_bin = 16

min_rng = 0.0
max_rng = 2.0

bias_on = False

depth = [16, 32, 32, 64, 64, 128, 128, 256, 256]  # another thick version #12
```

Figure C.7. Yolov5 Code Snippet: Filter Values

```
def _focus_layer(self, inputs):
    B, H, W, C = inputs.shape
    output = []
    output.append(tf.strided_slice(inputs, [0, 0, 0, 0], [B, H, W, C], [1, 2, 2, 1]))
    output.append(tf.strided_slice(inputs, [0, 1, 0, 0], [B, H, W, C], [1, 2, 2, 1]))
    output.append(tf.strided_slice(inputs, [0, 0, 1, 0], [B, H, W, C], [1, 2, 2, 1]))
    output.append(tf.strided_slice(inputs, [0, 1, 1, 0], [B, H, W, C], [1, 2, 2, 1]))
    return tf.concat(output, -1)
```

Figure C.8. Yolov5 Code Snippet: Focus Layer

```
num_output = mc.ANCHOR_PER_GRID * (mc.CLASSES + 1 + 4)
self.preds, _ = self._conv_layer('fire_o', fire_o, filters=num_output, size=3, stride=1,
                                 padding='SAME', xavier=False, relu=False, stddev=0.0001, w_bin=ll_w_bin,
                                 bias_on=bias_on, mul_f=mul_f)
```

Figure C.9. Yolov5 Code Snippet: Forward Graph Last Convolution Layer

- **Interpretation graph**

  This graph consists of the following sub-blocks:

  - Interpret Output

    This block interprets output from network and extracts predicted class probability, predicated confidence scores and bounding box values.

    Output of the convnet is a 10 × 10 × 120 tensor. There are 120 channels of data for each of the cells in the grid that is overlaid on the image, and contains the bounding boxes and class predictions, which means the 120 channels are not stored consecutively and need to be sorted. Figure C.10 and Figure C.11Figure 5.11 show the details.

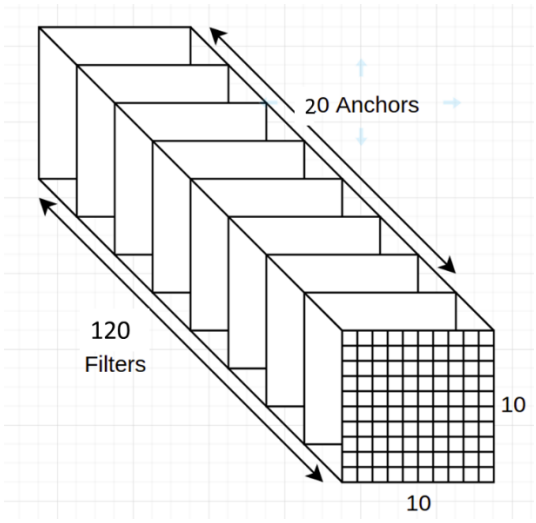

Figure C.10. Yolov5 Grid Output Visualization #1

For each grid, cell values are aligned as shown in Figure C.11:

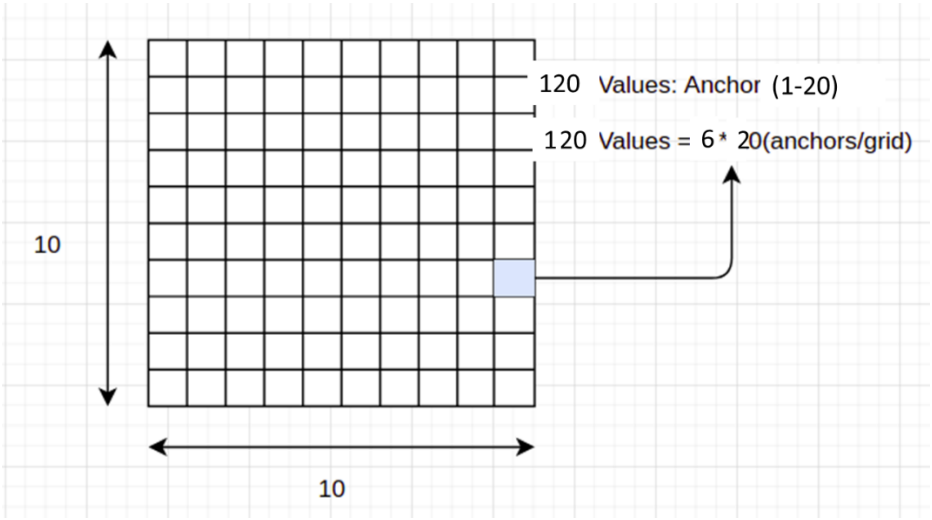

Figure C.11. Yolov5 Grid Output Visualization #2

As shown in the code below, output from fire_o layer (4Dd array of batch size × 10 × 10 × 120) needs to be sliced with a proper index to get all values of probability, confidence, and coordinates.

```
self.pred_class_probs = tf.reshape(
    tf.nn.softmax(
        tf.reshape(
            preds[:, :, :, num_confidence_scores:num_class_probs],
            [-1, mc.CLASSES]
        )
    ),
    [mc.BATCH_SIZE, mc.ANCHORS, mc.CLASSES],
    name='pred_class_probs'
)
# bbox_delta
self.pred_box_delta = tf.reshape(
    preds[:, :, :, num_class_probs:],
    [mc.BATCH_SIZE, mc.ANCHORS, 4],
    name='bbox_delta'
```

```
# confidence
num_confidence_scores = mc.ANCHOR_PER_GRID
self.pred_conf = tf.sigmoid(
    tf.reshape(
        preds[:, :, :, :num_confidence_scores],
        [mc.BATCH_SIZE, mc.ANCHORS]
    ),
    name='pred_confidence_score'
)
```

Figure C.12. Yolov5 Code Snippet: Interpret Output Graph

For confidence score between 0 and 1, sigmoid is used.
To predict the class probabilities, there is a vector of NUM_CLASS values at each bounding box. Apply a softmax layer for a nice probability distribution.
Bbox: this block calculates bounding boxes based on anchor box and predicated bounding boxes.
IOU: this block calculates Intersection over Union for detected bounding boxes and actual bounding boxes.
Probability: this block calculates detection probability and object class.

- Loss graph
  This block calculates different types of losses which need to be minimized. To learn detection, localization and classification, the model defines a multi-task loss function. There are three types of losses, which are considered for calculation:
  - Bounding box
    This loss is regression of the scalars for the anchors.

```
with tf.variable_scope('bounding_box_regression') as scope:
    self.bbox_loss = tf.truediv(
        tf.reduce_sum(
            mc.LOSS_COEF_BBOX * tf.square(
                self.input_mask * (self.pred_box_delta - self.box_delta_input))),
        self.num_objects,
        name='bbox_loss'
    )
    tf.add_to_collection('losses', self.bbox_loss)
```

Figure C.13. Yolov5 Code Snippet: Bbox Loss

  - Confidence score
    - To obtain meaningful confidence score, the predicted value of each box is regressed against the Intersection over Union of the real and the predicted box. During training, compare the ground truth

bounding boxes with all anchors and assign them to the anchors that have the largest overlap (IOU) with each of them.

- The reason being, to select the closest anchor to match the ground truth box such that the transformation needed is reduced to minimum. Equation evaluates to 1 if the k-th anchor at position-(i, j) has the largest overlap with a ground truth box, and to 0 if no ground truth is assigned to it. This way, we only include the loss generated by *responsible* anchors.
- As there can be multiple objects per image, normalize the loss by dividing it by the number of objects (self.num_objects).

```
with tf.variable_scope('confidence_score_regression') as scope:
    input_mask = tf.reshape(self.input_mask, [mc.BATCH_SIZE, mc.ANCHORS])
    self.conf_loss = tf.reduce_mean(
        tf.reduce_sum(
            tf.square((self.ious - self.pred_conf))
            * (input_mask * mc.LOSS_COEF_CONF_POS / self.num_objects
               + (1 - input_mask) * mc.LOSS_COEF_CONF_NEG / (mc.ANCHORS - self.num_objects)),
            reduction_indices=[1]
        ),
        name='confidence_loss'
    )
```

Figure C.14. Yolov5 Code Snippet: Confidence Loss

- Class
  The last part of the loss function is cross-entropy loss for classification for each box to perform image classification.

```
with tf.variable_scope('class_regression') as scope:
    self.class_loss = tf.truediv(
        tf.reduce_sum(
            (self.labels * (-tf.log(self.pred_class_probs + mc.EPSILON))
             + (1 - self.labels) * (-tf.log(1 - self.pred_class_probs + mc.EPSILON)))
            * self.input_mask * mc.LOSS_COEF_CLASS),
        self.num_objects,
        name='class_loss'
    )
```

Figure C.15. Yolov5 Code Snippet: Class Loss

In one model architecture, you obtain the bounding box prediction, the classification, as well as the confidence score.

- Optimizer
  This block is responsible for training the model with Momentum optimizer to reduce all losses.

```
opt = tf.train.MomentumOptimizer(learning_rate=lr, momentum=mc.MOMENTUM)
grads_vars = opt.compute_gradients(self.loss, tf.trainable_variables())

with tf.variable_scope('clip_gradient') as scope:
    for i, (grad, var) in enumerate(grads_vars):
        grads_vars[i] = (tf.clip_by_norm(grad, mc.MAX_GRAD_NORM), var)

apply_gradient_op = opt.apply_gradients(grads_vars, global_step=self.global_step)
```

Figure C.16. Yolov5 Code Snippet: Optimizer

- Training

```
feed_dict, org_img_per_batch, image_per_batch, label_per_batch, bbox_per_batch = \
    _load_data(load_to_placeholder=False)
op_list = [
    model.train_op, model.loss, summary_op, model.det_boxes,
    model.det_probs, model.det_class, model.conf_loss,
    model.bbox_loss, model.class_loss
]
_, loss_value, summary_str, det_boxes, det_probs, det_class, conf_loss, bbox_loss, class_loss = sess.run(
    op_list, feed_dict=feed_dict)
```

Figure C.17. Yolov5 Code Snippet: Training

sess.run feeds the data and labels batches to network and optimizes the weights and biases.

C.1. Training Yolov5

To train the machine:

1. Modify the training script.

   The training script at @train.sh is used to trigger training. Figure C.18 shows the input parameters which can be configured.

```
python ./src/train.py \
  --dataset=KITTI \
  --pretrained_model_path=$PRETRAINED_MODEL_PATH \
  --data_path=$TRAIN_DATA_DIR \
  --image_set=train \
  --train_dir="merged_dataset/train" \
  --net="yolov5" \
  --summary_step=100 \
  --checkpoint_step=10000 \
  --max_steps=300000 \
  --gpu=$GPUID
  --checkpoint_step=1000
```

Figure C.18. Yolov5 Code Snippet: Training

2. Execute the train.sh script to start training.

```
$ bash train.sh
100%|                                                                                | 172125/172125 [14:07<00:00, 203.07it/s]
========= ENet Class Weights =========
 [1.6701622214015108, 5.025416365192917]
2022-05-16 11:44:07.422311: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcuda.so.1
2022-05-16 11:44:07.481669: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1618] Found device 0 with properties:
name: NVIDIA GeForce RTX 2080 Ti major: 7 minor: 5 memoryClockRate(GHz): 1.65
pciBusID: 0000:01:00.0
2022-05-16 11:44:08.003479: I tensorflow/core/platform/profile_utils/cpu_utils.cc:94] CPU Frequency: 3491835000 Hz
2022-05-16 11:44:08.004207: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Host, Default Version
2022-05-16 11:44:08.004715: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1618] Found device 0 with properties:
name: NVIDIA GeForce RTX 2080 Ti major: 7 minor: 5 memoryClockRate(GHz): 1.65
2022-05-16 11:44:08.409419: I tensorflow/compiler/xla/service/service.cc:176]
StreamExecutor device (0): NVIDIA GeForce RTX 2080 Ti, Compute Capability 7.5
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Starting standard services.
INFO:tensorflow:Saving checkpoint to path ./log/Enet/model.ckpt
INFO:tensorflow:Starting queue runners.
INFO:tensorflow:global_step/sec: 0
INFO:tensorflow:Epoch 1.0/300
2022-05-16 11:44:45.263388: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcublas.so.10.0
INFO:tensorflow:Current Learning Rate: [0.0005]
2022-05-16 11:44:59.804109: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcudnn.so.7
2022-05-16 11:45:05.888458: W tensorflow/stream_executor/cuda/redzone_allocator.cc:312] Not found: ./bin/ptxas not found
Relying on driver to perform ptx compilation. This message will be only logged once.
INFO:tensorflow:global step 0: loss: 0.6934 (15.45 sec/step)  Current Streaming Accuracy: 0.0000  Current Mean IOU: 0.0000
INFO:tensorflow:---VALIDATION--- Validation Accuracy: 0.0000    Validation Mean IOU: 0.0000    (3.44 sec/step)
INFO:tensorflow:---VALIDATION--- Validation Accuracy: 0.7666    Validation Mean IOU: 0.3833    (0.04 sec/step)
```

Figure C.19. Yolov5: Execute Run Script

3. Start TensorBoard.

```
$ tensorboard –-logdir=<log directory of training>
For example: tensorboard –logdir=’./logs/’
```

4. Open the local host port on your web browser.

```
$ tensorboard --logdir logs_yolo/
TensorBoard 1.15.0 at http://pc-MS-1:6066/ (Press CTRL+C to quit)
```

Figure C.20. Yolov5 TensorBoard: Generated Link

5. Check the training status on TensorBoard.

Figure C.21. Yolov5 TensorBoard

Figure C.22 shows the image menu of TensorBoard.

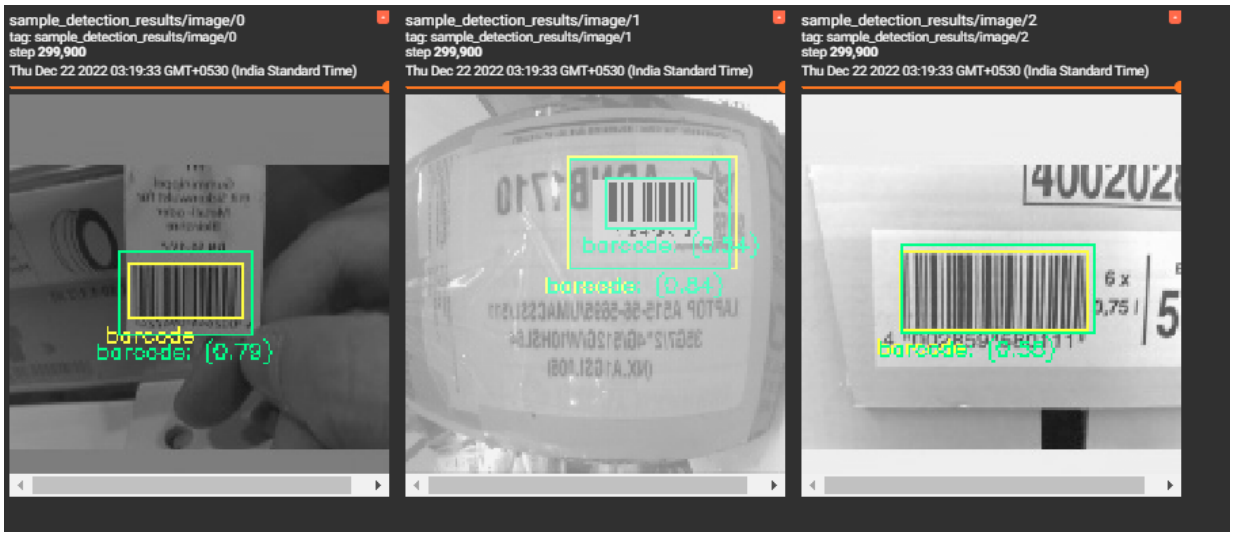

Figure C.22. Yolov5: Image Menu of TensorBoard

6. Check if the checkpoint, data, meta, index, and events (if using TensorBoard) files are created at the log directory. These files are used for creating the frozen file (*.pb).

```
model.ckpt-290000.data-00000-of-00001
model.ckpt-290000.index
model.ckpt-290000.meta
model.ckpt-299999.data-00000-of-00001
model.ckpt-299999.index
model.ckpt-299999.meta
model_metrics.txt
```

Figure C.23. Yolov5: Example of Checkpoint Data Files at Log Folder

Appendix D. Yolov5 Frozen File Creation

This section describes the procedure for freezing the model, which is aligned with the Lattice sensAI software. Perform the steps below to generate the frozen protobuf file.

Command to generate the Frozen (.pb) File

```
$ python genpb.py --ckpt_path <COMPLETE_PATH_TO_LOG_DIRECTORY>/model.ckpt-<ckpt number> --input_node_name image_input –output_node_name fire_o/convolution
Example: python genpb.py –ckpt_path logs/yolov5/train/model.ckpt-499999. --input_node_name batch –output_node_name fire_o/convolution
```

```
~/atss$ python genpb.py --input_node_name batch --output_node_name fire_o/convolution  --ckpt_path logs/atss/train/model.ckpt-499999
2021-09-22 22:34:54.017347: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this TensorFlow binary was not co
mpiled to use: SSE4.1 SSE4.2 AVX AVX2 AVX512F FMA
2021-09-22 22:34:54.039518: I tensorflow/core/platform/profile_utils/cpu_utils.cc:94] CPU Frequency: 3499910000 Hz
2021-09-22 22:34:54.040393: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x555c92b411e0 initialized for platform Host (this does not
guarantee that XLA will be used). Devices:
2021-09-22 22:34:54.040423: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Host, Default Version
2021-09-22 22:34:54.041351: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcuda.so.1
2021-09-22 22:34:54.047020: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:169] retrieving CUDA diagnostic information for host: gpu-server
2021-09-22 22:34:54.047032: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:176] hostname: gpu-server
2021-09-22 22:34:54.047129: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:200] libcuda reported version is: 450.119.3
2021-09-22 22:34:54.047158: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:204] kernel reported version is: 450.119.3
2021-09-22 22:34:54.047169: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:310] kernel version seems to match DSO: 450.119.3i
```

Figure D.1. Yolov5: Run genpb.py to Generate Inference .pb

- *genpb.py* uses the latest checkpoint in train directory to generate frozen '.pb' file.
- Once the genpb.py is executed successfully <ckpt-prefix>_frozenforInference.pb file can be found in the log directory, as shown in Figure D.1.

```
model.ckpt-299999.data-00000-of-00001
model.ckpt-299999.index
model.ckpt-299999.meta
model.pbtxt
model_frozenforInference.pb
model_metrics.txt
```

Figure D.2. Yolov5: Frozen Inference. pb Output

Appendix E. Yolov5 Model Evaluation and mAP calculation

This section contains guide to calculate model performance in terms of MAP.

E.1. Run Inference on Test Set

The Barcode code contains 'inference.py' under the inference directory as shown in Figure E.1.

Note: If you make any change in the training code regarding image size, number of anchors or grid size, you must replicate those changes in the inference script.

Run the command below to run inference on test set.

```
$ python inference.py –pb <converted pb path> --input_image <test set images path>
```

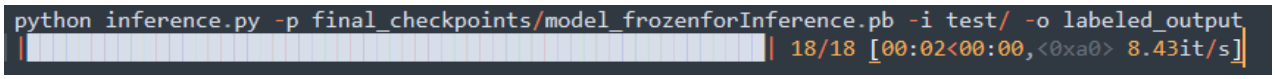

Figure E.1. Yolov5: Run Inference

The command above can save images with bbox drawn in *inference_output/image_output* and resultant kitti output in *inference_output/predictions*, as shown in Figure E.2.

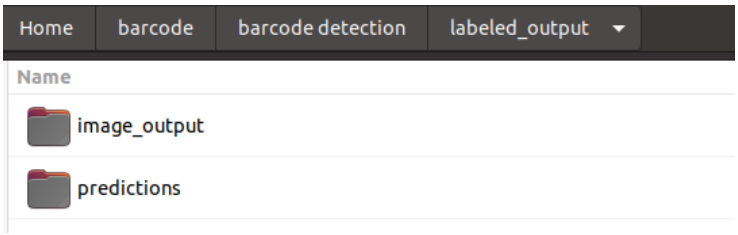

Figure E.2. Yolov5: Inference Output

E.2. Calculate MAP

Barcode Detection code contains main.py under the Training directory as shown in Figure E.3.

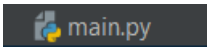

Figure E.3. Yolov5: mAP File

Run the command below to calculate mAP using predictions generated from inference and groundtruth from test set.

```
$ python .\main.py --input_images ..\training\datasets\test\images\ --ground_truth
..\training\datasets\t
est\labels\ --predictions ..\training\datasets\test\images_out\predictions\ --no-animation
--no-plot
```

```
mAP$ python main.py --input_images <input test images path> \
                    --ground_truth <input testset groundtruth path> \
                    --predictions <pridictions path> \
                    --no-animation  --no-plot
XX.XX% = person AP
mAP = XX.XX%
```

Figure E.4. Yolov5: mAP Calculation

After the successful run of the script, mAP for each class and the total mAP are shown.

Appendix F. Yolov5 Binary File creation using sensAI NN Compiler

This section describes how to generate binary file using the Lattice sensAI version 6.0 software.

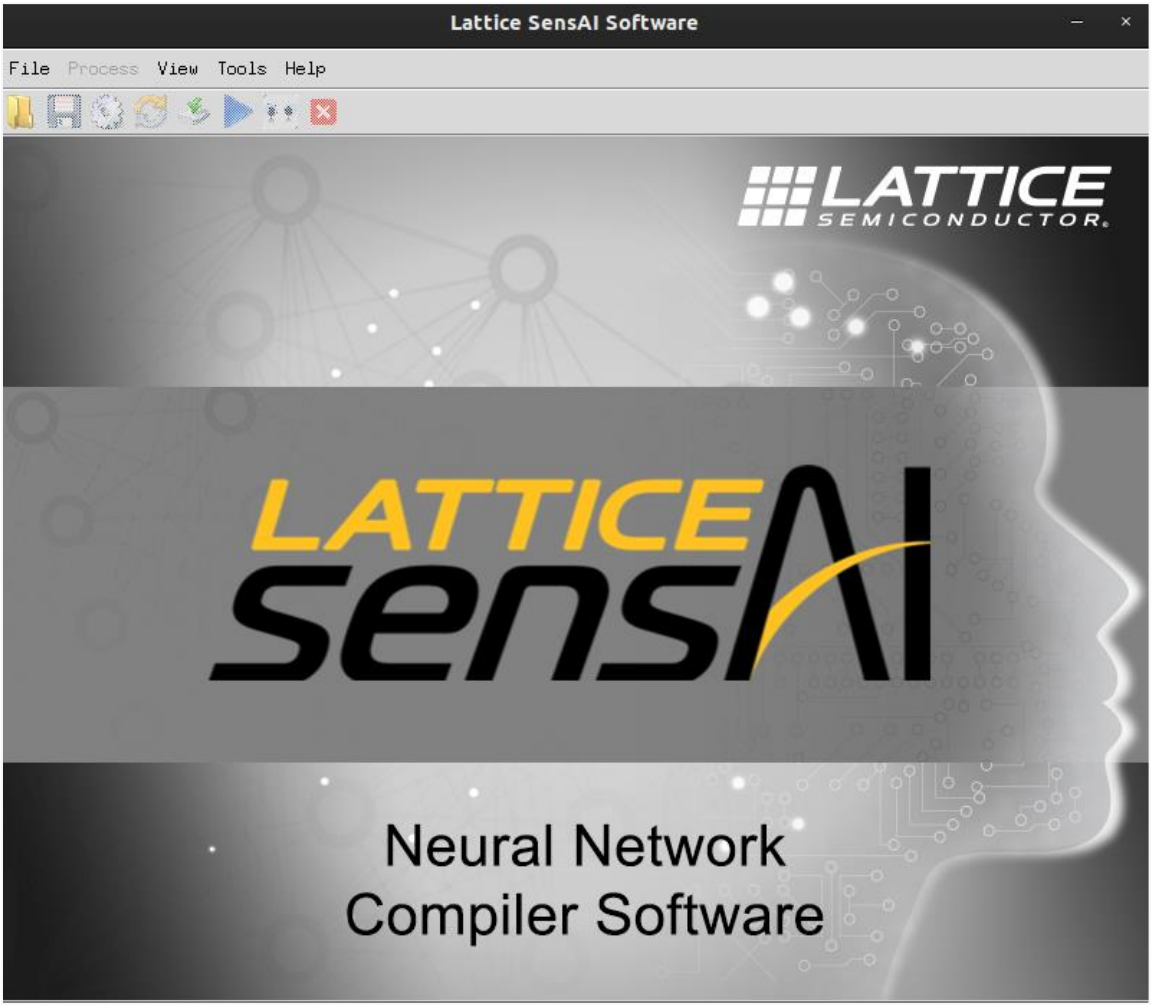

Figure F.1. Yolov5: sensAI – Home Screen

To create the project in sensAI tool:

1. Click **File > New**.
2. Enter the following settings:
   - Project name
   - Framework – TensorFlow
   - Class – CNN
   - Device – CertusPro-NX
   - IP – Advanced_CNN
3. Click **Network File** and select the network (PB) file.

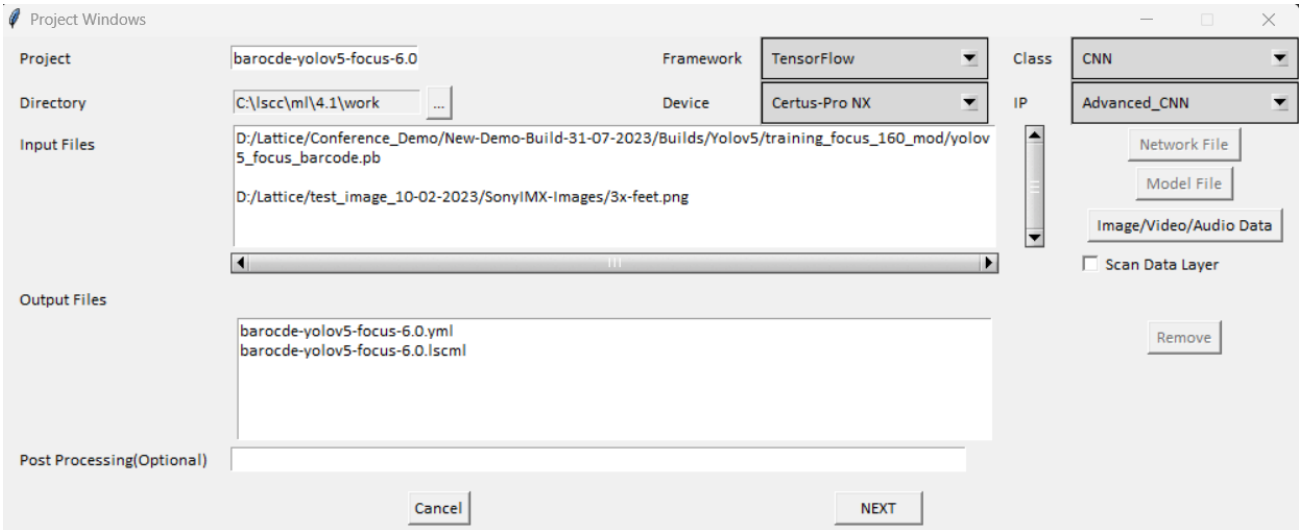

Figure F.2. Yolov5: sensAI – Select Framework, Device, IP, Network File, and Image

4. Click the **Image/Video/Audio Data** button and select the input image file.

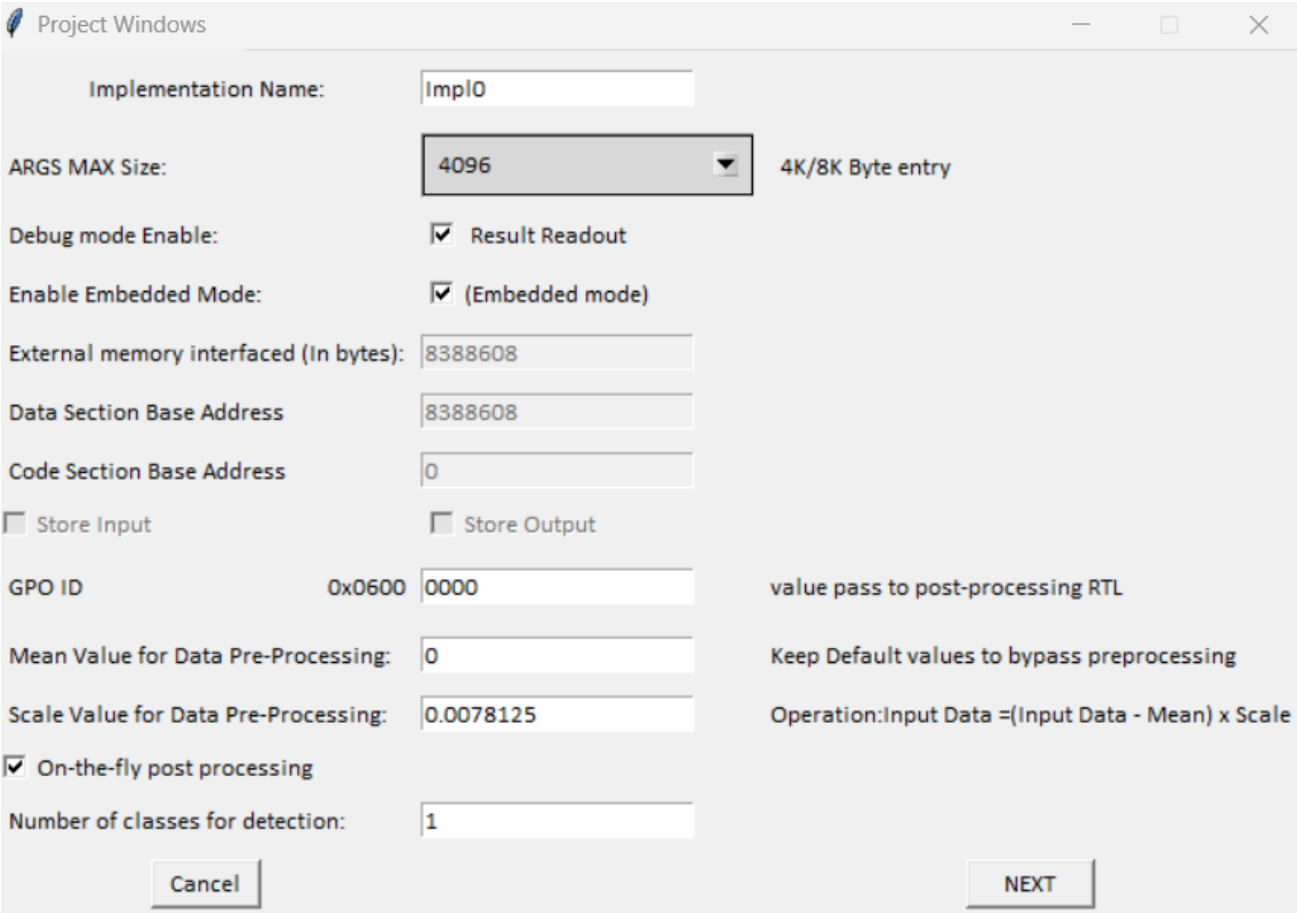

Figure F.3. Yolov5: sensAI – Select Project Setting

5. Click **Next**.

6. Set the following attributes:
   - **Mean Value for Data Pre-Processing**: 0
   - **Scratch Pad Memory Block Size**: 8192
   - **ARGS MAX Size**: 4096
   - **External Memory Interfaced**: 8388608
   - **Scale Value for Data Pre-Processing**: 0.0078125

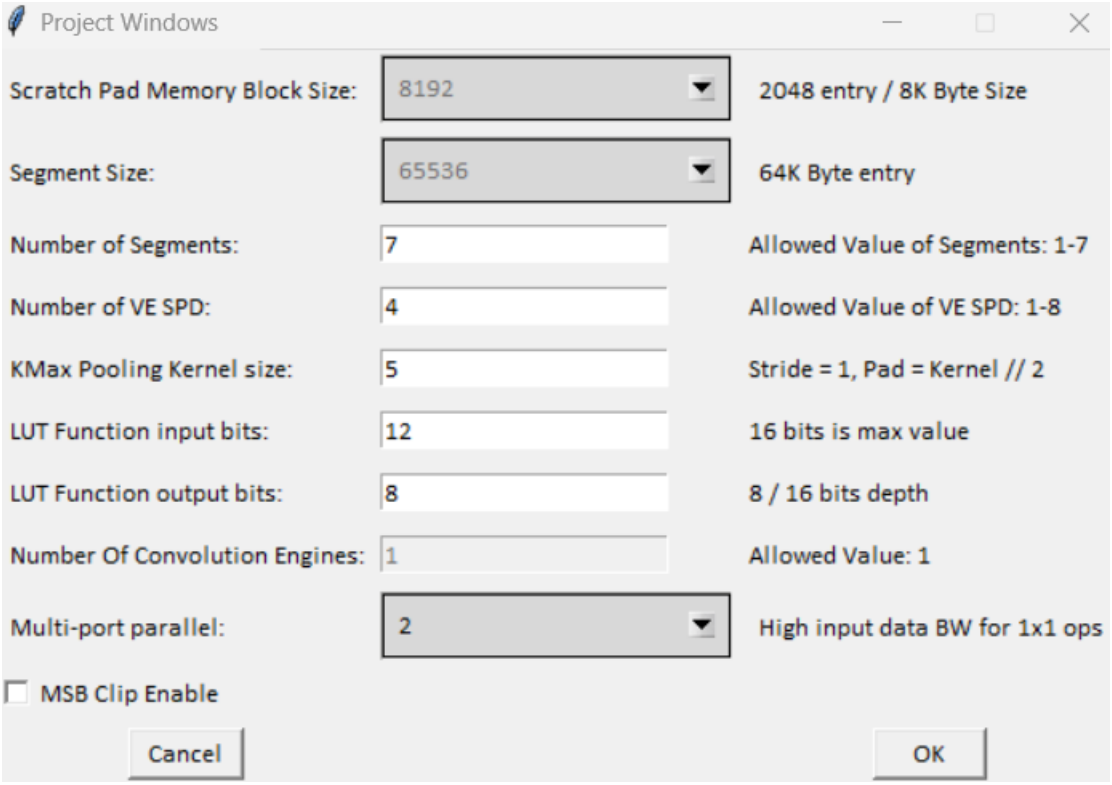

Figure F.4. Yolov5: sensAI – Update Project Settings

7. Clock **Ok** to create project.
8. Double click on **Analyze**.

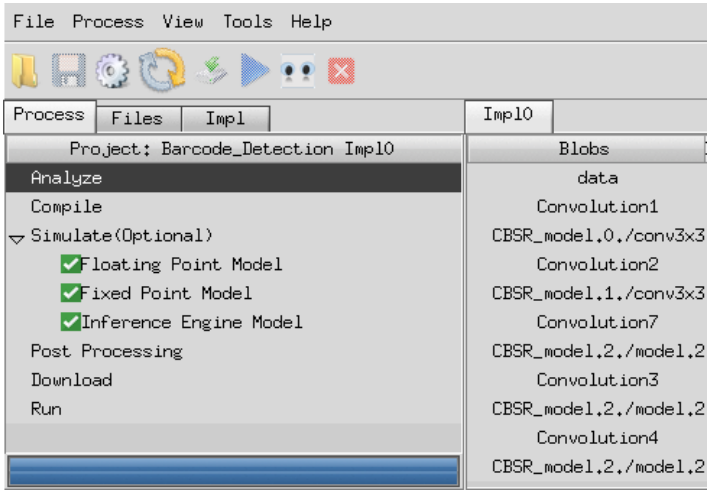

Figure F.5. Yolov5: Analyze Project

9. Confirm the Q format of each layer as shown below.

Impl0

Blobs	Data Format (Analyzed)	Stored Data Format	Internal(External) Mem Bytes
data	8.7	1.7	None
Convolution1	5.10	5.10	None
CBSR_model.0./conv3x3/convolution	8.7	1.7	None
Convolution2	5.10	5.10	None
CBSR_model.1./conv3x3/convolution	8.7	1.7	None
Convolution7	5.10	5.10	None
CBSR_model.2./model.2.bottleneck_csp_y2/conv3:	8.7	1.7	None
Convolution3	5.10	5.10	None
CBSR_model.2./model.2.bottleneck_csp_x1/conv3:	8.7	1.7	None
Convolution4	5.10	5.10	None
CBSR_model.2./model.2.bottleneck_csp_x10/mode.	8.7	1.7	None
Convolution5	5.10	5.10	None
CBSR_model.2./model.2.bottleneck_csp_x10/mode.	8.7	1.7	None
Eltwise1	5.10	5.10	None
Elt_quant_model.2./model.2.bottleneck_csp_x10.	8.7	1.7	None
Convolution6	5.10	5.10	None
CBSR_model.2./model.2.bottleneck_csp_y1/conv3:	8.7	1.7	None
Concat_model.2./concat_Placeholder	5.10	5.10	None
Concat_model.2./concat	8.7	1.7	None
Convolution8	5.10	5.10	None
CBSR_model.2./model.2.bottleneck_csp_out/conv:	8.7	1.7	None
Convolution9	5.10	5.10	None
CBSR_model.3./conv3x3/convolution	8.7	1.7	None
Convolution18	5.10	5.10	None
CBSR_model.4./model.4.bottleneck_csp_y2/conv3:	8.7	1.7	None
Convolution10	5.10	5.10	None
CBSR_model.4./model.4.bottleneck_csp_x1/conv3:	8.7	1.7	None
Convolution11	5.10	5.10	None
CBSR_model.4./model.4.bottleneck_csp_x10/mode.	8.7	1.7	None
Convolution12	5.10	5.10	None
CBSR_model.4./model.4.bottleneck_csp_x10/mode.	8.7	1.7	None
Eltwise2	5.10	5.10	None

Figure F.6. Yolov5: Q Format Settings for Each Layer (1)

Impl0

Blobs	Data Format (Analyzed)	Stored Data Format	Internal(External) Mem Bytes
CBSR_model.4./model.4.bottleneck_csp_x10/mode.	8.7	1.7	None
Eltwise2	5.10	5.10	None
Elt_quant_model.4./model.4.bottleneck_csp_x10.	8.7	1.7	None
Convolution13	5.10	5.10	None
CBSR_model.4./model.4.bottleneck_csp_x11/mode.	8.7	1.7	None
Convolution14	5.10	5.10	None
CBSR_model.4./model.4.bottleneck_csp_x11/mode.	8.7	1.7	None
Eltwise3	5.10	5.10	None
Elt_quant_model.4./model.4.bottleneck_csp_x11.	8.7	1.7	None
Convolution15	5.10	5.10	None
CBSR_model.4./model.4.bottleneck_csp_x12/mode.	8.7	1.7	None
Convolution16	5.10	5.10	None
CBSR_model.4./model.4.bottleneck_csp_x12/mode.	8.7	1.7	None
Eltwise4	5.10	5.10	None
Elt_quant_model.4./model.4.bottleneck_csp_x12.	8.7	1.7	None
Convolution17	5.10	5.10	None
CBSR_model.4./model.4.bottleneck_csp_y1/conv3:	8.7	1.7	None
Concat_model.4./concat_Placeholder	5.10	5.10	None
Concat_model.4./concat	8.7	1.7	None
Convolution19	5.10	5.10	None
CBSR_model.4./model.4.bottleneck_csp_out/conv:	8.7	1.7	None
Convolution20	5.10	5.10	None
CBSR_model.5./conv3x3/convolution	8.7	1.7	None
Convolution29	5.10	5.10	None
CBSR_model.6./model.6.bottleneck_csp_y2/conv3:	8.7	1.7	None
Convolution21	5.10	5.10	None
CBSR_model.6./model.6.bottleneck_csp_x1/conv3:	8.7	1.7	None
Convolution22	5.10	5.10	None
CBSR_model.6./model.6.bottleneck_csp_x10/mode.	8.7	1.7	None
Convolution23	5.10	5.10	None
CBSR_model.6./model.6.bottleneck_csp_x10/mode.	8.7	1.7	None
Eltwise5	5.10	5.10	None
Elt_quant_model.6./model.6.bottleneck_csp_x10.	8.7	1.7	None
Convolution24	5.10	5.10	None
CBSR_model.6./model.6.bottleneck_csp_x11/mode.	8.7	1.7	None
Convolution25	5.10	5.10	None
CBSR_model.6./model.6.bottleneck_csp_x11/mode.	8.7	1.7	None
Eltwise6	5.10	5.10	None
Elt_quant_model.6./model.6.bottleneck_csp_x11.	8.7	1.7	None
Convolution26	5.10	5.10	None
CBSR_model.6./model.6.bottleneck_csp_x12/mode.	8.7	1.7	None
Convolution27	5.10	5.10	None
CBSR_model.6./model.6.bottleneck_csp_x12/mode.	8.7	1.7	None

Figure F.7. Yolov5: Q Format Settings for Each Layer (2)

Impl0

Blobs	Data Format (Analyzed)	Stored Data Format	Internal(External) Mem Bytes
CBSR_model.6./model.6.bottleneck_csp_x12/mode.	8.7	1.7	None
Convolution27	5.10	5.10	None
CBSR_model.6./model.6.bottleneck_csp_x12/mode.	8.7	1.7	None
Eltwise7	5.10	5.10	None
Elt_quant_model.6./model.6.bottleneck_csp_x12.	8.7	1.7	None
Convolution28	5.10	5.10	None
CBSR_model.6./model.6.bottleneck_csp_y1/conv3:	8.7	1.7	None
Concat_model.6./concat_Placeholder	5.10	5.10	None
Concat_model.6./concat	8.7	1.7	None
Convolution30	5.10	5.10	None
CBSR_model.6./model.6.bottleneck_csp_out/conv:	8.7	1.7	None
Convolution31	5.10	5.10	None
CBSR_model.7./conv3x3/convolution	8.7	1.7	None
Convolution40	5.10	5.10	None
CBSR_model.8./model.8.bottleneck_csp_y2/conv3:	8.7	1.7	None
Convolution32	5.10	5.10	None
CBSR_model.8./model.8.bottleneck_csp_x1/conv3:	8.7	1.7	None
Convolution33	5.10	5.10	None
CBSR_model.8./model.8.bottleneck_csp_x10/mode.	8.7	1.7	None
Convolution34	5.10	5.10	None
CBSR_model.8./model.8.bottleneck_csp_x10/mode.	8.7	1.7	None
Eltwise8	5.10	5.10	None
Elt_quant_model.8./model.8.bottleneck_csp_x10.	8.7	1.7	None
Convolution35	5.10	5.10	None
CBSR_model.8./model.8.bottleneck_csp_x11/mode.	8.7	1.7	None
Convolution36	5.10	5.10	None
CBSR_model.8./model.8.bottleneck_csp_x11/mode.	8.7	1.7	None
Eltwise9	5.10	5.10	None
Elt_quant_model.8./model.8.bottleneck_csp_x11.	8.7	1.7	None
Convolution37	5.10	5.10	None
CBSR_model.8./model.8.bottleneck_csp_x12/mode.	8.7	1.7	None
Convolution38	5.10	5.10	None
CBSR_model.8./model.8.bottleneck_csp_x12/mode.	8.7	1.7	None
Eltwise10	5.10	5.10	None
Elt_quant_model.8./model.8.bottleneck_csp_x12.	8.7	1.7	None
Convolution39	5.10	5.10	None
CBSR_model.8./model.8.bottleneck_csp_y1/conv3:	8.7	1.7	None
Concat_model.8./concat_Placeholder	5.10	5.10	None
Concat_model.8./concat	8.7	1.7	None
Convolution41	5.10	5.10	None
CBSR_model.8./model.8.bottleneck_csp_out/conv:	8.7	1.7	None
fire_o/convolution	5.10	5.10	None
CBSR_fire_o/convolution	8.7	5.10	None

Figure F.8. Yolov5: Q Format Settings for Each Layer (3)

10. Double-click on **Compile** to generate the firmware and filter binary file.

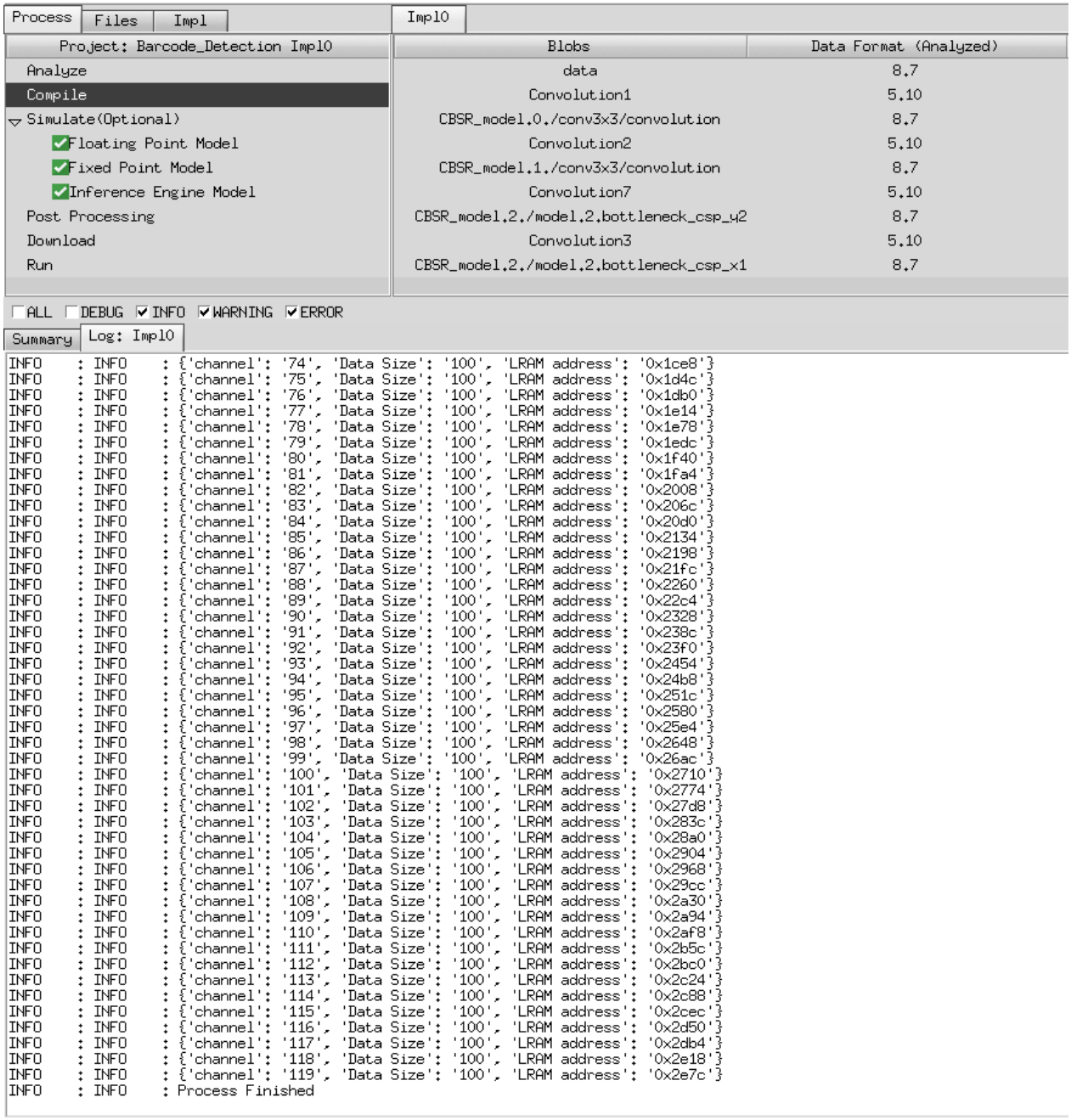

Figure F.9. Yolov5: Compile Project

The firmware bin file location is displayed in the compilation log. Use the generated firmware bin on hardware for testing.

Appendix G. Yolov5 Hardware RTL Implementation

G.1. Top Level Information

Block Diagram

Figure G.1. shows the top-level block diagram of barcode detection with the CertusPro-NX Voice and Vision ML (Rev A) board.

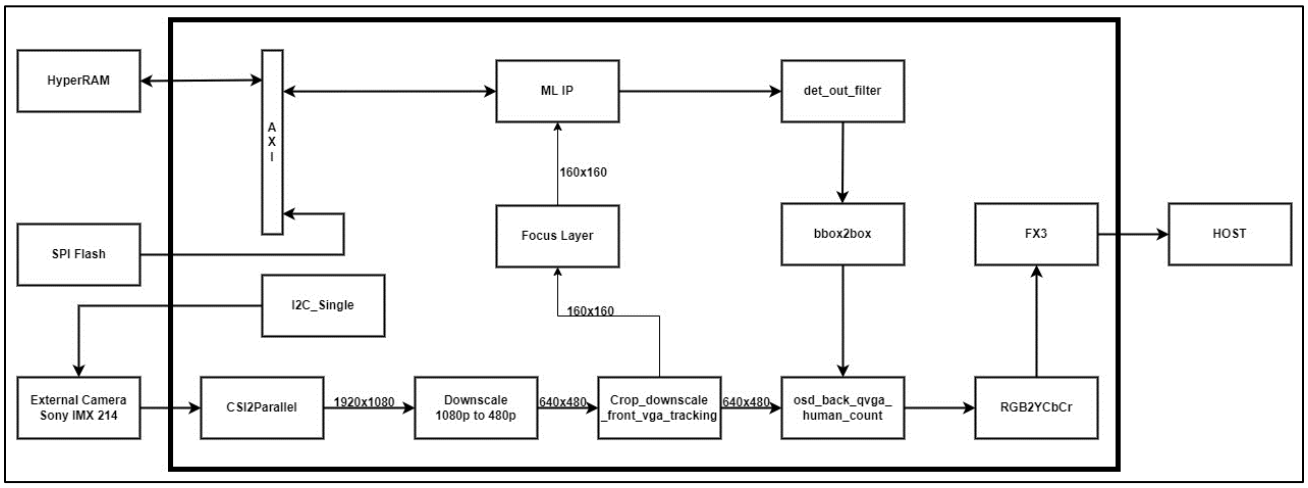

Figure G.1. Yolov5: Top Level Block Diagram of Barcode Detection with CertusPro-NX Voice and Vision ML (Rev A) Board

Operational Flow

Figure G.2. shows image downscaling from 1080p to 480p.

- The external camera Sony IMX214 is configured using I^2C Master Block i2c_single.v.
- The real time input image data is received by Video path. The RAW10 data from *csi2_to_parallel.v* is sent to downscale_1080p_to480p.v which downscales the 1080p (1920 x 1080) image data received from the camera to 480p (640 x 480). Downscaling is performed by selecting four lines out of every nine lines and one pixel out of three pixels. Overall, four valid pixels are passed from 27 pixels (9 x 3 window).

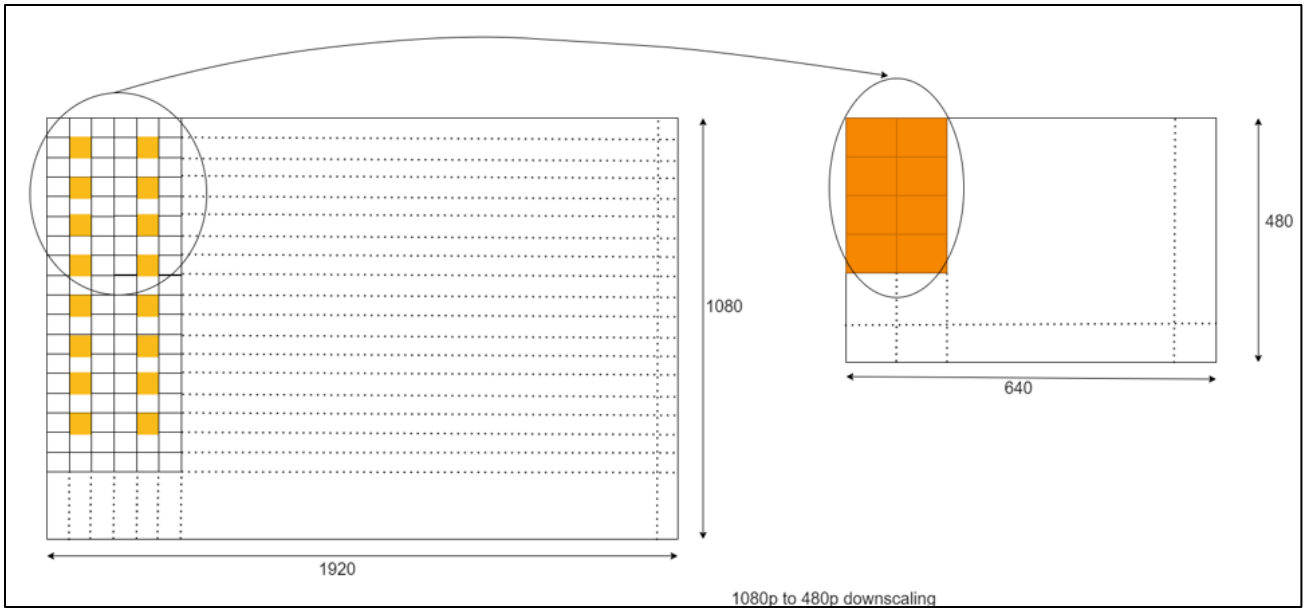

Figure G.2. Yolov5: Downscaling from 1080p to 480p (Obtaining 4 x 2 pixels in 480p from 18 x 6 pixels in 1080p)

- After the image is downscaled to 480p, it is passed to pre-processing *crop_downscale_front_vga_tracking.v* which performs crop and downscale operation to provide a 160 x 160 (with padding of 40 blank rows) image to the focus layer. The focus layer starts generating an address, 32-bit data, and valid write enable signals to the ML engine through the LMMI interface.
- The 6 Mb firmware BIN file (.mcs) is loaded to the SPI Flash module *spi_loader_spram.v* configured with starting address 24'h300000 to end address 24'h800000.
- The ML Engine receives the downscaled image data from *focus_layer.v* through LMMI interface and the firmware file through the AXI hyperbus interface to provide an inference result output.
- The ML inference output is stored in FIFO and passed to *det_out_filter.v* for post processing.
- For final output display, the *osd_back_qvga_human_count.v* module performs barcode detection on image received from the video path and the downscaled image obtained from the Crop and Downscale module using the ML object detection pixel information obtained from the *det_out_filter.v* block.
- Output of *osd_back_qvga_human_count.v* is passed to the rgb2ycbcr module, which converts RGB data of pixels to YCbCr data which is compatible for FX3. FX3 passes this to the host for display.

Core Customization

Table G.1. and Table G.2. show the available parameters.

Table G.1. Core Parameters - Yolov5

Parameter	Value	Description
USE_ML	1'b1	Indicates 1: Enable/0: Disable to use CNN engine.
EN_UART	1'b0	Indicates 1: Enable/0: Disable UART for video output.
FLASH_START_ADDR	24'h300000	Indicates starting address to load Firmware in external SPI flash.
FLASH_END_ADDR	24'h800000	Indicates ending address to load Firmware in external SPI flash.
FL_OFFSET (crop_downscale_front_vga_tracking.v)	433152	Indicates Focus Layer starting address offset in ML IP LRAM. It should be obtained from the SensAI compiler log while generating mcs.
CNN IP Attributes	**Default Value**	**Description**
Number of LRAMs	7	Number of LRAMs in the ML SPD.
Number of VE SPD Packs	8	1 VE SPD Pack = 4 VE SPDs. Number of EBRs in a VE SPD is the attribute below.
Number of EBRs in VE SPD	4	1 VE SPD Pack = 4 VE SPDs. This attribute specifies the number of EBRs in a VE SPD.
Max AXI4 external memory DMA Burst	31	Max burst limit on AXI4 bus during DMA transactions with external memory.
LMMI read mode	WORD	Access width mode for LMMI read interface (reading data out of the IP) BYTE: byte mode; HWORD: 16-bit half word mode; WORD: 32-bit word mode.
LMMI Write mode	WORD	Access width mode for LMMI write interface (writing data into the IP) BYTE: byte mode; HWORD: 16-bit half word mode; WORD: 32-bit word mode.
No. of Convolution engines	1	Number of parallel convolution engines. Higher compute throughput can be achieved with more number of engines, at the expense of higher utilization of DSP resources.
Enable 4 parallel ports for 1x1 convolution	Unchecked	Enable four parallel ports internally for high bandwidth data accesses for 1 × 1 convolutions.
Enable Vector ALU	Checked	Enables the Vector Engine for pixelwise ALU operations.
Enable 5x5 convolution	Checked	Enables 5 × 5 convolutions inside the Conv EU.
Enable 7x7 convolution	Checked	Enables 7 × 7 convolutions inside the Conv EU.
Enable Argmax pool	Checked	Enables the Argmax pool compute module.
Enable Maxpool stride=1 module	Checked	Enable the 1-D stride=1 maxpool compute engine.
Avant Mode	Unchecked	**IMPORTANT:** This must be enabled for the Avant devices and disabled for the CertusPro-NX devices.

Table G.2. Camera Parameters - Yolov5

Module	Parameter/ Port	Sony IMX258	Sony IMX214
vvml_barcode_detection_top.v	SENSOR_SLAVE_ADDR	7'h1A	7'h10
vvml_barcode_detection_top.v	NUM_OF_TRANS_I2C	d80	d81
vvml_barcode_detection_top.v (downscale_1080p_to480p)	USE_IMX214	1'b0	1'b1
i2c_m_ctrl.v	Instance	rom_sonyimx258 u_rom_sonyimx258	rom_sonyimx214 u_rom_sonyimx214
ROM IP (rom_sonyimx258/rom_sonyimx214)	Config ROM Read Address	9 bits	9 bits
ROM IP (rom_sonyimx258/rom_sonyimx214)	ROM Address Depth	320	324

G.2. Architectural Details

Pre-processing Operation

- The *crop_downscale_front_vga_tracking.v* video processing block is used to crop and downscale the image data to make it compatible with the ML engine.
- Masking values for incoming image data from the camera are set to capture the image data at 640 × 480.
- As shown below, the 640 × 480 image is downscaled into 160 x 160 image resolution using block size 4.

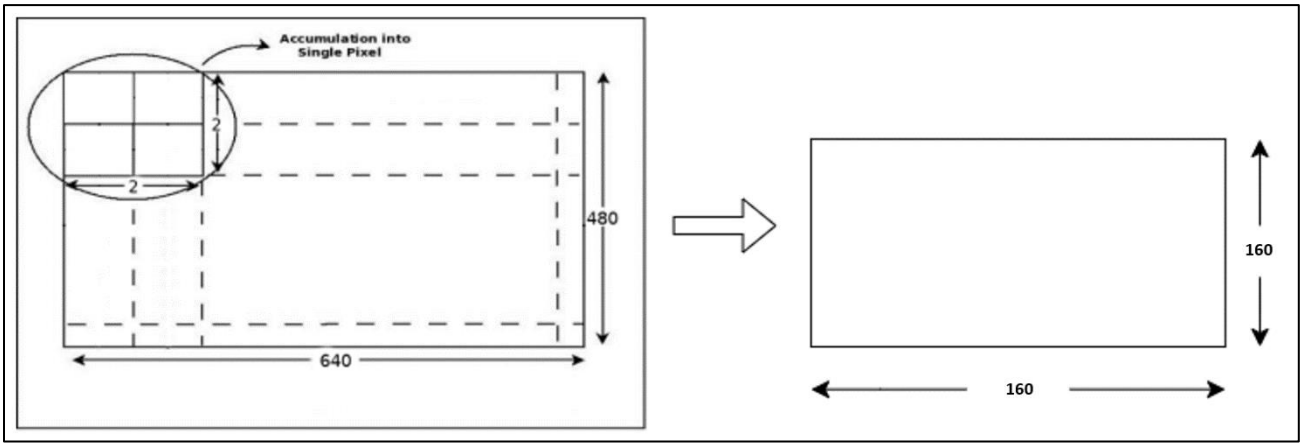

Figure G.3. Yolov5: Downscaling Image

- The accumulated pixel values are written into accumulation buffer. While reading the data from the buffer is sent to the ML engine for inference via the focus layer.

Post-processing operation

The post-processing operation is explained as follows:

- After the image is down scaled to 160 x 160, it is passed to the ML engine via the focus layer module.
- When the ML engine completes processing the input image, it asserts the enable signal and starts transmitting processed data at its output port which is stored in FIFO. This data is the used in calculating bounding boxes in *det_out_filter.v.*
- When bounding box co-ordinates are calculated, it is passed to *osd_back_qvga_human_count.v.*
- The *osd_back_qvga_human_count.v* module receives the 160 x 160 downscaled image from the pre-processing module and the ML engine data from the post processing block.
- Using the pixel information of barcode detection received from post-processing block, the OSD module performs text and graphics addition over the 160 x 160 downscaled image.
- In the final segmentation output display, the barcode present can be observed in grayscale with bounding box.
- Upon completion of data processing in the OSD, it is sent to FX3 via rgb2ycbcr.

The pre-processing and post-processing cycle continues until the board is powered on and the real time image is captured by the camera.

References

- CertusPro-NX FPGA web page
- Lattice CertusPro-NX Reference Designs
- Lattice Radiant Software FPGA web page
- Lattice senseAI web page
- Lattice Insights for Lattice Semiconductor training courses and learning plans

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

For frequently asked questions, refer to the Lattice Answer Database at www.latticesemi.com/en/Support/AnswerDatabase.

Revision History

Revision 1.0, December 2023

Section	Change Summary
All	Initial release.

www.latticesemi.com