Object Classification: Video Stream Analysis Reference Design

Reference Design

FPGA-RD-02265-1.2

November 2024

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
CNN	Convolutional Neural Network
FPGA	Field-Programmable Gate Array
LED	Light-emitting diode
ML	Machine Learning
NN	Neural Network
NNC	Neural Network Compiler
SD	Secure Digital
USB	Universal Serial Bus

1. Introduction

This document describes how to set up and run the Object Classification: Video Stream Analysis Reference Design, which uses the Lattice Avant™-AT-E Evaluation Board Rev. D.

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 Neural Network: Creating the filter and firmware binary files using Lattice sensAI 6.0
3. FPGA Design: Creating the FPGA bitstream file.
4. FPGA bitstream and quantized weights and instructions: Flashing the binary and bitstream files to the Lattice Avant-AT-E Evaluation Board.

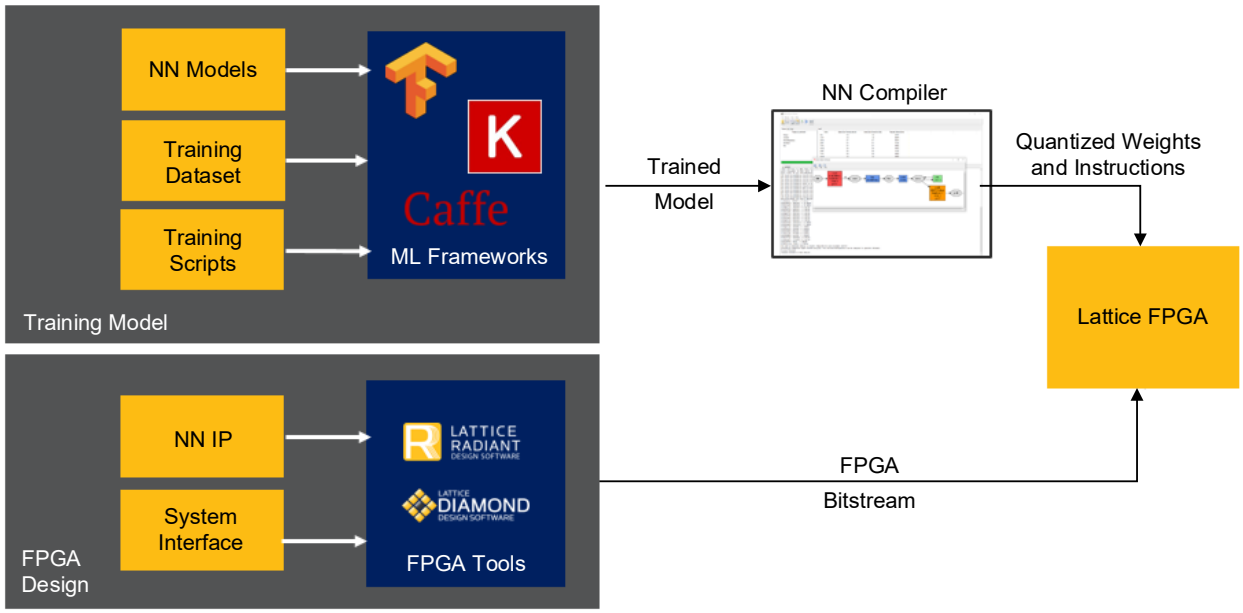

Figure 1.1. Lattice Machine Learning Design Flow

2. Hardware and Software Requirements

2.1. Hardware Requirements

Figure 2.1 shows the hardware requirement for the Lattice Avant-AT-E Evaluation Board Rev. D.

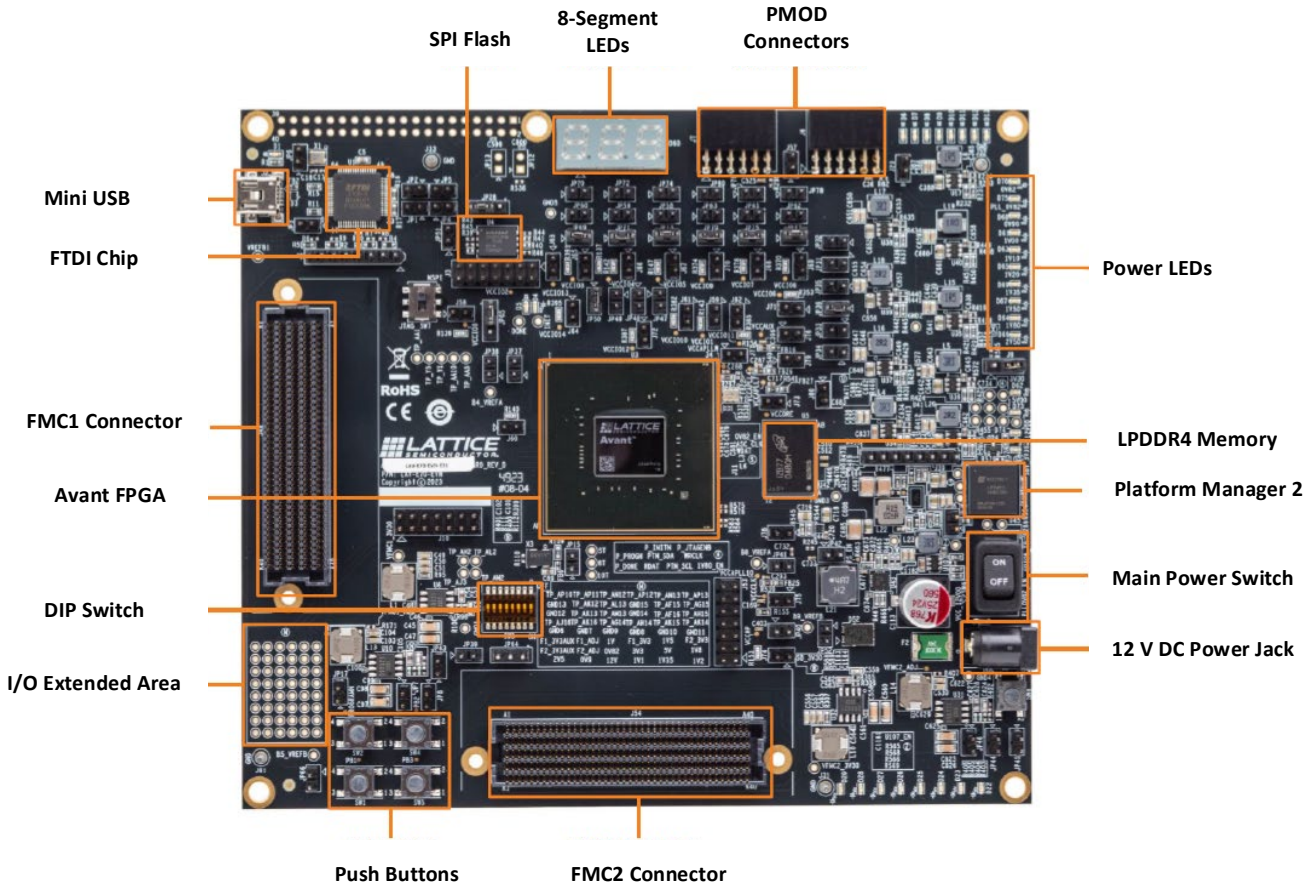

Figure 2.1. Top View of Lattice Avant-AT-E Evaluation Board

2.2. Software Requirements

- Lattice Radiant™ software version 2024.1. Refer to http://www.latticesemi.com/latticeradiant.
- Lattice Radiant Programmer version 2024.1. Refer to http://www.latticesemi.com/latticeradiant.
- Lattice sensAI Compiler v 7.0 – Refer to https://www.latticesemi.com/Products/DesignSoftwareAndIP/AIML/NeuralNetworkCompiler.

3. Setting Up the Basic Environment

3.1. Setting Up the Linux Environment for Machine Training

This section describes the steps for NVIDIA GPU drivers and/or libraries for the 64-bit Ubuntu 16.04 operating system. NVIDIA library and TensorFlow version are dependent on the PC and Ubuntu/Windows version.

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

3.1.1.1. Installing the CUDA Toolkit

To install the CUDA toolkit, run the following commands in the order specified below:

```
$ curl -O
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/cuda-
repo-ubuntu1604_10.1.105-1_amd64.deb
```

```
$ curl -O https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/cuda-repo-ubuntu1604_10.1.105-1_amd64.deb
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  2832  100  2832    0     0   2204      0  0:00:01  0:00:01 --:--:--  2205
```

Figure 3.1. Download CUDA Repo

```
$ sudo dpkg –I ./cuda-repo-ubuntu1604_10.1.105-1_amd64.deb
```

```
$ sudo dpkg -i ./cuda-repo-ubuntu1604_10.1.105-1_amd64.deb
Selecting previously unselected package cuda-repo-ubuntu1604.
(Reading database ... 5287 files and directories currently installed.)
Preparing to unpack .../cuda-repo-ubuntu1604_10.1.105-1_amd64.deb ...
Unpacking cuda-repo-ubuntu1604 (10.1.105-1) ...
Setting up cuda-repo-ubuntu1604 (10.1.105-1) ...

The public CUDA GPG key does not appear to be installed.
To install the key, run this command:
sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub
```

Figure 3.2. Install CUDA Repo

```
$ sudo apt-key adv --fetch-keys
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.
pub
```

```
$ sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub
Executing: /tmp/tmp.a2QZZnTMUX/gpg.1.sh --fetch-keys
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub
gpg: key 7FA2AF80: public key "cudatools <cudatools@nvidia.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

Figure 3.3. Fetch Keys

```
$sudo apt-get update
```

```
$ sudo apt-get update
Ign:1 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  InRelease
Hit:2 http://archive.ubuntu.com/ubuntu xenial InRelease
Get:3 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Release [697 B]
Get:4 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Release.gpg [836 B]
Hit:5 http://archive.ubuntu.com/ubuntu xenial-updates InRelease
Hit:6 http://security.ubuntu.com/ubuntu xenial-security InRelease
Hit:7 http://archive.ubuntu.com/ubuntu xenial-backports InRelease
Ign:8 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Packages
Get:8 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Packages [428 kB]
Fetched 429 kB in 1s (386 kB/s)
Reading package lists... Done
```

Figure 3.4. Update Ubuntu Packages Repositories

```
$ sudo apt-get install cuda-9-0
```

```
$ sudo apt-get install cuda-9-0
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

Figure 3.5. CUDA Installation

3.1.1.2. Installing the cuDNN

To install the cuDNN:

1. Create 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 below 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
```

4. $ 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 3.6. cuDNN Library Installation

3.1.2. 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 16.04. Anaconda provides one of the easiest ways to perform machine learning development and training on Linux.

3.1.2.1. Installing the Anaconda Python

To install the Anaconda and Python 3:

1. Go to https://www.anaconda.com/products/individual#download-section
2. Download Python3 version of Anaconda for Linux.
3. Run the command below to install the Anaconda environment:

```
$  sh  Anaconda3-2019.03-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 3.7. Anaconda Installation

4. Accept the license.

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

Figure 3.8. Accept License Terms

5. Confirm the installation path. Follow the instructions on the screen 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 3.9. Confirm/Edit Installation Location

6. After installation, enter No as shown in Figure 3.10.

```
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 3.10. Launch/Initialize Anaconda Environment on Installation Completion

3.1.3. Creating a New Environment with Python 3.6

1. Activate the base conda environment using the below command.
```
$ source <conda directory>/bin/activate
```
2. To create a new environment run the below command.
```
$ conda create -n <Name of New environment> python=3.6
```
3. Activate the newly created environment.
```
$ conda activate <Name of New environment>
```

3.1.4. Installing the TensorFlow v1.15

1. Install the TensorFlow by running the command below:
```
$ conda install tensorflow-gpu==1.15
```

```
$ conda install tensorflow-gpu=1.15
Collecting package metadata (current_repodata.json): done
Solving environment: failed with initial frozen solve. Retrying with flexible solve.
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/anaconda3.1/envs/tf1.15-enet

  added / updated specs:
    - tensorflow-gpu=1.15
```

Figure 3.11. TensorFlow Installation

2. After installation, enter **Y** as shown in Figure 3.12.

```
Proceed ([y]/n)? y


Downloading and Extracting Packages
tensorflow-1.15.0    | 4 KB      | ################################### | 100%
cudatoolkit-10.0.130 | 261.2 MB  | ################################### | 100%
tensorboard-1.15.0   | 3.2 MB    | ################################### | 100%
gast-0.2.2           | 155 KB    | ################################### | 100%
cudnn-7.6.5          | 165.0 MB  | ################################### | 100%
numpy-1.21.5         | 10 KB     | ################################### | 100%
numpy-base-1.21.5    | 4.8 MB    | ################################### | 100%
tensorflow-base-1.15 | 156.5 MB  | ################################### | 100%
tensorflow-estimator | 271 KB    | ################################### | 100%
typing_extensions-4. | 28 KB     | ################################### | 100%
six-1.16.0           | 18 KB     | ################################### | 100%
cupti-10.0.130       | 1.5 MB    | ################################### | 100%
webencodings-0.5.1   | 19 KB     | ################################### | 100%
werkzeug-0.16.1      | 258 KB    | ################################### | 100%
importlib-metadata-4 | 40 KB     | ################################### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

Figure 3.12. TensorFlow Installation Confirmation

3.1.5. Installing the Python Package

To install the Python package:

1. Install Easydict by running the command below:

```
$ conda install -c conda-forge easydict
```

```
(base) $ conda install -c conda-forge easydict
Solving environment: done
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - easydict
```

Figure 3.14. Easydict Installation

2. Install Joblib by running the command below:

```
$ conda install joblib
```

```
(base) $ conda install joblib
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - joblib
```

Figure 3.15. Joblib Installation

3. Install Keras by running the command below:

```
$ conda install keras
```

```
(base) $ conda install keras
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - keras
```

Figure 3.16. Keras Installation

4. Install OpenCV by running the command below:

```
$ conda install opencv
```

```
(base) $ conda install opencv
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - opencv
```

Figure 3.17. OpenCV Installation

5. Install Pillow by running the command below:

```
$ conda install pillow
```

```
(base) $ conda install pillow
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - pillow
```

Figure 3.18. Pillow Installation

4. Code Directory Structure

Download the Object Classification: Video Stream Analysis training code. Figure 4.1 shows the directory structure.

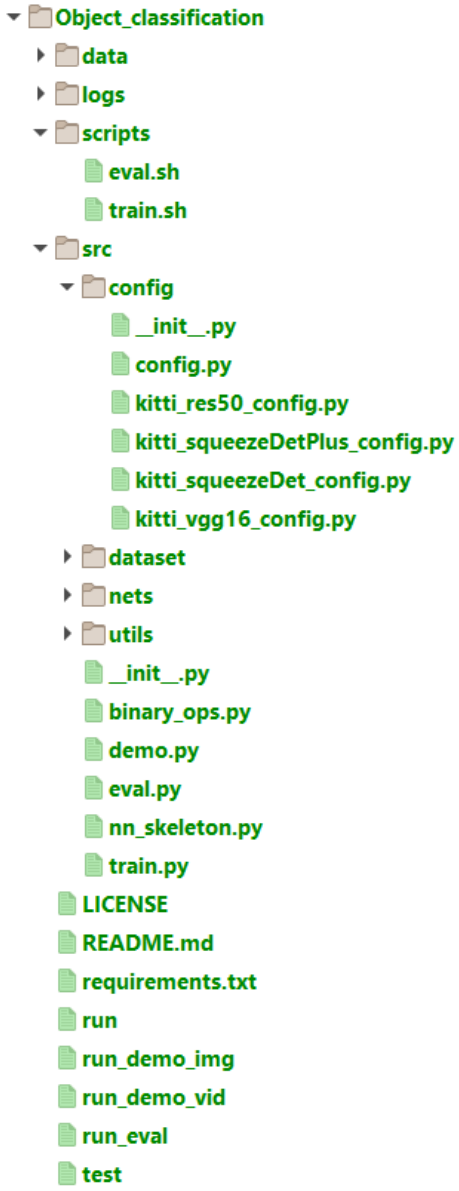

Figure 4.1. Training Code Directory Structure

5. Preparing the Training Code

5.1. Neural Network Architecture

This section provides information on the Convolution Neural Network (CNN) configuration of this reference design. Figure 5.1 shows the basic building block of the model architecture. Table 5.1 shows the entire detection model using the basic building block.

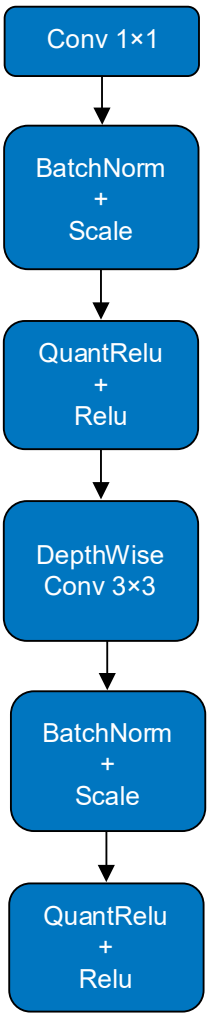

Figure 5.1. Basic Building Block

Table 5.1. Detection Model

Layer Name	Layer Content	Layer Output Dimensions
Input	Input layer	228 × 480 × 1
Fire1	Convolution3×3 + BatchNormalization + QuantRelu + Relu + MaxPool(k=2,s=2)	144 × 240 × 8
Fire2	Basic Building Block	144 × 240 × 8
Fire3	Basic Building Block	144 × 240 × 8
Fire4	Basic Building Block + MaxPool(k=2,s=2)	72 × 120 × 8
Fire5	Basic Building Block	72 × 120 × 32
Fire6	Basic Building Block	72 × 120 × 32
Fire7	Basic Building Block	72 × 120 × 32
Fire8	Basic Building Block + MaxPool(k=2,s=2)	36 × 60 × 32
Fire9	Basic Building Block	36 × 60 × 64
Fire10	Basic Building Block	36 × 60 × 64
Fire11	Basic Building Block	36 × 60 × 64
Fire12	Basic Building Block + MaxPool(k=2,s=2)	18 × 30 × 64
Fire13	Basic Building Block	18 × 30 × 128
Fire14	Basic Building Block	18 × 30 × 128
Fire15	Basic Building Block	18 × 30 × 128
Fire16	Basic Building Block + MaxPool(k=2,s=2)	9 × 15 × 128
Fire17	Basic Building Block	9 × 15 × 192
Fire18	Basic Building Block	9 × 15 × 192
Fire19	Basic Building Block	9 × 15 × 256
Fire_o	Convolution3×3	9 × 15 × 168

The diagram model contains Convolution (Conv), BatchNormalization (bn), Relu, and MaxPool layers.

5.1.1. Convolutional Layer

In general, the first layer in a CNN is always a convolution layer**.** Each layer consists of a number of filters (sometimes referred to as kernels) which convolve with the input layer/image and generates activation map (or a 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, 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 it is looking for (such as curve for example) is in the input volume.

5.1.2. Activation Layer (Relu)

Immediately after each convolution layer, it is convention to apply a nonlinear layer (or activation layer). This layer introduces nonlinearity to a system that has just been computing linear operations during the convolution layers. These operations may just be element-wise multiplications and summations. In the past, nonlinear functions such as tanh and sigmoid are used. Researchers, however, found out that Relu layers work far better because the network can train a lot faster, because of the computational efficiency, without making a significant difference to the accuracy. The Relu layer applies the function f(x) = max (0, x) to all of the values in the input volume. In basic terms, this layer just changes all the negative activations to 0. It increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the convolution layer.

5.1.3. 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 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 subregion that the filter convolves around.

The intuitive reasoning behind this layer is that once it is known that a specific feature is in the original input volume (there is 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:

- Lessens the computation cost since the parameters or weights are reduced by 75%
- Controls overfitting

The term overfitting is used 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 overfitting is having a model that gets 100% or 99% on the training set, but only 50% on the test data.

5.1.4. Batch Normalization Layer

The batch normalization or BatchNorm layer reduces the internal covariance shift. To train a neural network, some preprocessing is applied to the input data. For example, normalizing all data so that it resembles a normal distribution (that means, zero mean and a unitary variance). This prevents the early saturation of non-linear activation functions such as the sigmoid function, assuring that all input data is in the same range of values, and so on.

The problem, however, 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 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 steps below during training time:

1. Calculate the mean and variance of the layer inputs.
2. Normalize the layer inputs using the previously calculated batch statistics.
3. Arrange 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 the user to be carefree about weight initialization. It works as regularization in place of dropout and other regularization techniques.

5.1.5. Dropout Layer

Dropout layers have a specific function in neural networks. After training, the weights of the network are so tuned to the training examples they are given that the network does not perform well when given new examples. The idea of dropout is simplistic. This layer *drops out* a random set of activations in that layer by setting them to zero. It forces the network to be redundant. That 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 overfitting problem. An important note is that this layer is only used during training, and not during testing.

5.1.6. Fully-connected layer

The fully-connected layer takes an input volume (whatever is the output of the convolution, 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.

5.1.7. Quantization

Quantization is a method that brings the neural network to a reasonable size, while also achieving high performance accuracy. This is especially important for on-device applications wherein 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 above architecture provides nonlinearities and preservation of dimension that help to improve the robustness of the network and control overfitting.

5.2. Object Classification: Video Stream Analysis Network Output

From the input image model, the feature maps are first extracted and overlaid with a W × H grid. Each cell then 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)*IOU)
- C° conditional class probability

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 detection

The model has a total of 22680 output values, which are derived from the following:

- 9 × 15 grid
- Twenty-four anchor boxes per grid
- Seven values per anchor box. It consists of:
  - Four bounding box coordinates (x, y, w, h)
  - Two class probability (The model is trained for Person and Car classes)
  - One confidence score

As a result, there is a total of 9 × 15 × 24 × 7 = 22680 output values.

5.2.1. Training Code Overview

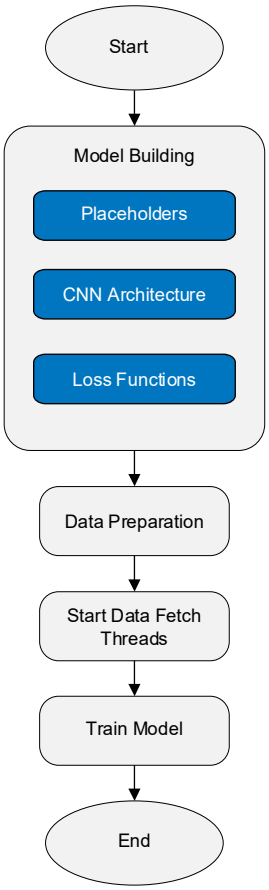

Figure 5.2. Training Code Flow Diagram

The training code can be divided into the following parts:

- Model Configuration
- Model Building
- Model Freezing
- Data Preparation
- Training for Overall Execution Flow

5.2.2. Model Configuration

This model is trained for two classes: *person* and *car*. The design uses KITTI dataset format and SqueezeDet model architecture. To run this Reference Design, the user is expected to bring a dataset in KITTI format.

kitti_squeezeDet_config() maintains all the configurable parameters for the model. Below is the summary of configurable parameters.

5.2.2.1. 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          = 480
mc.IMAGE_HEIGHT         = 288
```

Figure 5.3. Code Snippet – Input Image Size Configuration

Since there are four pooling layers, grid dimension is H = 9 and W = 15. *anchor_shapes* variable of *set_anchors()* in *src/config/kitti_squeezeDet_config.py* indicates anchors width and heights. Update it based on anchors per gird size changes.

```
def set_anchors(mc):
  H, W, B = 9, 15, 24
```

Figure 5.4. Code Snippet – Anchors per Grid Config #1 (Grid Sizes)

5.2.2.2. Batch size

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

5.2.2.3. Anchors per Grid

Change mc.ANCHOR_PER_GRID in src/config/kitti_squeezeDet_config.py to configure anchors per grid.

```
mc.ANCHOR_BOX             = set_anchors(mc)
mc.ANCHORS                = len(mc.ANCHOR_BOX)
mc.ANCHOR_PER_GRID        = 24
```

Figure 5.5. Code Snippet – Anchors per Grid Config #2

Change hard coded anchors per grid in *set_anchors()* in *src/config/kitti_squeezeDet_config.py*. Here, B (value 24) indicates anchors per grid.

```
anchor_shapes = np.reshape(
    [np.array(
        [
          [320, 256],
          [320, 192],
          [320, 128],
          [256, 256],
          [256, 192],
          [256, 128],
          [256,  64],
          [192, 256],
          [192, 192],
          [192, 128],
          [192,  64],
          [128, 256],
          [128, 192],
          [128, 128],
          [128,  64],
          [128,  32],
          [ 64, 256],
          [ 64, 192],
          [ 64, 128],
          [ 64,  64],
          [ 64,  32],
          [ 32, 192],
          [ 32, 128],
          [ 32,  64]])] * H * W,
    (H, W, B, 2)
)
```

Figure 4.6. Code Snippet – Anchors per Grid Config #3

5.2.2.4. Training Parameters

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

```
mc.WEIGHT_DECAY          = 0.0001
mc.LEARNING_RATE         = 0.01
mc.DECAY_STEPS           = 30000
mc.MAX_GRAD_NORM         = 1.0
mc.MOMENTUM              = 0.9
mc.LR_DECAY_FACTOR       = 0.9

mc.LOSS_COEF_BBOX        = 5.0
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 #0.
mc.NMS_THRESH            = 0.4
mc.PROB_THRESH           = 0.005
mc.TOP_N_DETECTION       = 16 #64

mc.DATA_AUGMENTATION     = True
mc.DRIFT_X               = 90 #60
mc.DRIFT_Y               = 60 #42
mc.EXCLUDE_HARD_EXAMPLES = False
```

Figure 5.6. Code Snippet – Training Parameters

5.2.3. Model Building

SqueezeDet class can be configured from *src/nets/squeezeDet.py*. SqueezeDet class constructor builds the model, which is divided into the following sections:

- Forward Graph
- Interpretation Graph
- Loss Graph
- Train Graph
- Visualization Graph

5.2.3.1. Forward Graph

- The CNN architecture consists of Convolution, Batch Normalization, Relu, and Maxpool.
- The forward graph consists of 19 fire layers as indicated in Table 5.1.

```
depth = [8, 8, 8, 8, 32, 32, 32, 32, 64, 64, 64, 64, 128, 128, 128, 128, 192, 192, 256, 320, 320]
```

Figure 5.7. Code Snippet – Filter Values

- Filter sizes of each convolutional block are shown in Table 5.1, which can be configured by changing the values of depth from SqueezeDet class in *src/nets/squeezeDet.py*, as shown in Figure 5.8.

```
fire1 = self._fire_layer('fire1', self.image_input, oc=depth[0], freeze=False, w_bin=fl_w_bin, a_bin=fl_a_bin, pool_en=True, min_rng=min_rng, max_rng=max_rng, bias_on=bia
fire2 = self._bsconv_layer('fire2', fire1, oc=depth[1], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on, mu
fire3 = self._bsconv_layer('fire3', fire2, oc=depth[2], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on, mu
fire4 = self._bsconv_layer('fire4', fire3, oc=depth[3], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=True, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on, mul
fire5 = self._bsconv_layer('fire5', fire4, oc=depth[4], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on, mu
fire6 = self._bsconv_layer('fire6', fire5, oc=depth[5], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on, mu
fire7 = self._bsconv_layer('fire7', fire6, oc=depth[6], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on, mu
fire8 = self._bsconv_layer('fire8', fire7, oc=depth[7], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=True, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on, mu
fire9 = self._bsconv_layer('fire9', fire8, oc=depth[8], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on, mu
fire10 = self._bsconv_layer('fire10', fire9, oc=depth[9], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on,
fire11 = self._bsconv_layer('fire11', fire10, oc=depth[10], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on
fire12 = self._bsconv_layer('fire12', fire11, oc=depth[11], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=True, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on,
fire13 = self._bsconv_layer('fire13', fire12, oc=depth[12], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on
fire14 = self._bsconv_layer('fire14', fire13, oc=depth[13], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on
fire15 = self._bsconv_layer('fire15', fire14, oc=depth[14], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on
fire16 = self._bsconv_layer('fire16', fire15, oc=depth[15], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=True, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on,
fire17 = self._bsconv_layer('fire17', fire16, oc=depth[16], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on
fire18 = self._bsconv_layer('fire18', fire17, oc=depth[17], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on
fire19 = self._bsconv_layer('fire19', fire18, oc=depth[18], freeze=False, w_bin=ml_w_bin, a_bin=ml_a_bin, pool_en=False, min_rng=min_rng, max_rng=max_rng, bias_on=bias_on
fire_o = fire19

#################################################################

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 5.8. Code Snippet – Forward Graph of model

5.2.3.2. Interpretation Graph

The Interpretation Graph consists of the following sub-blocks:

- interpret_output
  This block interprets output from the network and extracts predicted class probability, predicated confidence scores, and bounding box values.
  The output of the convnet is a 9 × 15 × 168 tensor – there are 168 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. This means the 168 channels are not stored consecutively but are scattered all over and need to be sorted. Figure 4.10 show the details. For each grid, cell values are aligned as shown in Figure 4.10.

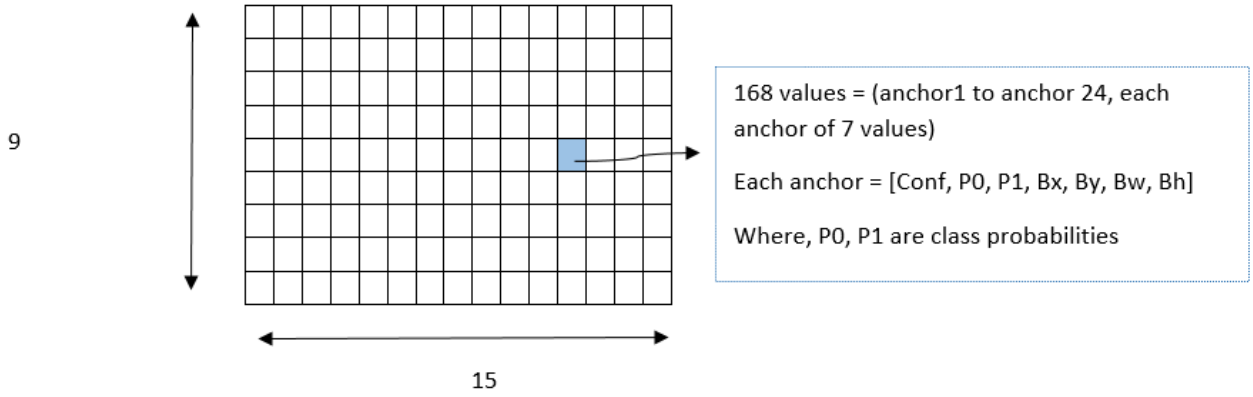

Figure 5.9. Grid Output Visualization

```
# 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')
# probability
num_class_probs = mc.ANCHOR_PER_GRID*mc.CLASSES+num_confidence_scores
print ('ANCHOR_PER_GRID:', mc.ANCHOR_PER_GRID)
print ('CLASSES:', mc.CLASSES)
print ('preds2:', preds[:, :, :, num_confidence_scores:num_class_probs])
print ('ANCHORS:', mc.ANCHORS)

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')
```

Figure 5.10. Code Snippet – Interpret Output Graph

Output from the fire_o layer is a 4D array of batch size × 9 × 15 × 168 that needs to be sliced with the proper index to get all values of probability, confidence, and coordinates.
For the confidence score, this must be a number between 0 and 1, as such, sigmoid is used.
For predicting the class probabilities, there is a vector of NUM_CLASS values at each bounding box. Apply a softmax to make it probability distribution.

- bbox
  This block calculates bounding boxes based on the anchor box and the 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.

5.2.3.3. Loss Graph

This block calculates different types of losses, which needs to be minimized. To learn detection, localization, and classification, 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 5.11. Code Snippet – Bbox Loss

- Confidence Score
  - To obtain meaningful confidence score, the predicted value of each box is regressed against the real and predicted box. During training, compare the ground truth bounding boxes with all anchors and assign them to the anchors with the largest overlap (IOU).
  - 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, the user only includes the loss generated by the *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 5.12. Code Snippet – Confidence Loss

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

```
with tf.variable_scope('class_regression') as scope:
  # cross-entropy: q * -log(p) + (1-q) * -log(1-p)
  # add a small value into log to prevent blowing up
  #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'
  #)
  self.labels_rev = tf.reverse(self.labels, [-1])
  '''
  self.class_term2 = tf.reduce_sum((1-self.labels)*(-tf.log(1-self.pred_class_probs+1e-8)))
  self.class_term2_1 = self.labels
  self.class_term2_2 = -tf.log(self.pred_class_probs+1e-8)
  self.class_term2_3 = self.pred_class_probs
  self.class_term2_4 = (1-self.labels)*(-tf.log(1-self.pred_class_probs+1e-8))
  self.class_term2_5 = self.labels_rev
  '''
  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+1e-7)))
          * self.input_mask * mc.LOSS_COEF_CLASS),
      self.num_objects,
      name='class_loss'
  )
  tf.add_to_collection('losses', self.class_loss)
```

Figure 5.13. Code Snippet – Class Loss

In one model architecture, the user obtains the bounding box prediction, the classification, as well as the confidence score.

5.2.3.4. Train Graph

This block is responsible for training the model with momentum optimizer to reduce all losses.

5.2.3.5. Visualization Graph

This block provides visitations of detected results.

5.2.4. Training

```
if mc.NUM_THREAD > 0:
    _, loss_value, conf_loss, bbox_loss, class_loss = sess.run(
        [model.train_op, model.loss, model.conf_loss, model.bbox_loss,
        model.class_loss], options=run_options)
else:
    feed_dict, _, _, _, _ = _load_data(load_to_placeholder=False)
    _, loss_value, conf_loss, bbox_loss, class_loss = sess.run(
        [model.train_op, model.loss, model.conf_loss, model.bbox_loss,
        model.class_loss], feed_dict=feed_dict)
```

Figure 5.14. Code Snippet – Training

sess.run feeds the data, labels batches to network, and optimizes the weights and biases. The code above handles the input data method in case of multiple threads preparing batches, or data preparation in the main thread.

5.3. Training from Scratch and/or Transfer Learning

To train the machine:

1. Go to the top/root directory of the Lattice training code from the command prompt.

   The model works on 288×480 images.

   Current training code uses mean = 0 and scale = 1/128 (0.0078125) in pre-processing step. Scale can be changed in training code *@src/dataset/imdb.py* as shown in highlight below.

```
# scale image
im = cv2.resize(im, (mc.IMAGE_WIDTH, mc.IMAGE_HEIGHT), interpolation=cv2.INTER_AREA)
im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY) # gray color instead of RGB
org_img = im.copy()
im /= 128.0
org_img = cv2.cvtColor(org_img, cv2.COLOR_GRAY2BGR)
```

Figure 5.15. Training Code Snippet for Mean and Scale

   The dataset path can be set in the training code *@src/dataset/kitti.py* and can be used in combination with the *--data_path* option while triggering training using *train.py* to get the desired path. For example, the user can have *<data_path>/training/images* and *<data_path>/training/labels*.

```
def __init__(self, image_set, data_path, mc):
  imdb.__init__(self, 'kitti_'+image_set, mc)
  self._image_set = image_set
  self._data_root_path = data_path
  self._image_path = os.path.join(self._data_root_path, 'training', 'images')
  self._label_path = os.path.join(self._data_root_path, 'training', 'labels')
  self._classes = self.mc.CLASS_NAMES
```

Figure 5.16. Training Code Snippet for Dataset Path

2. Modify the training script. *@scripts/train.sh* is used to trigger training.
   Figure 4.18 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="$TRAIN_DIR/train" \
  --net=$NET \
  --summary_step=100 \
  --checkpoint_step=500 \
  --max_steps=1500000 \
  --gpu=$GPUID
```

Figure 5.17. Training Input Parameter

- $TRAIN_DATA_DIR – dataset directory path.
- $TRAIN_DIR – log directory where checkpoint files are generated while model is training.
- $GPUID – gpu id. If the system has more than one gpu, it indicates the one to use.
- --summary_step – indicates at which interval loss summary should be dumped.
- --checkpoint_step – indicates at which interval checkpoints are created.
- --max_steps – indicates the maximum number of steps for which the model is trained.

3. Execute the run command script (by modifying if required) which in turn triggers train.sh script and starts the training.
```
$ ./run
```
4. Start TensorBoard.
```
$ tensorboard –logdir=<log directory of training>
```
   For example: tensorboard –logdir='./logs/'

5. Open the local host port on the web browser by giving path to the log directory.
6. Check the training status on TensorBoard.

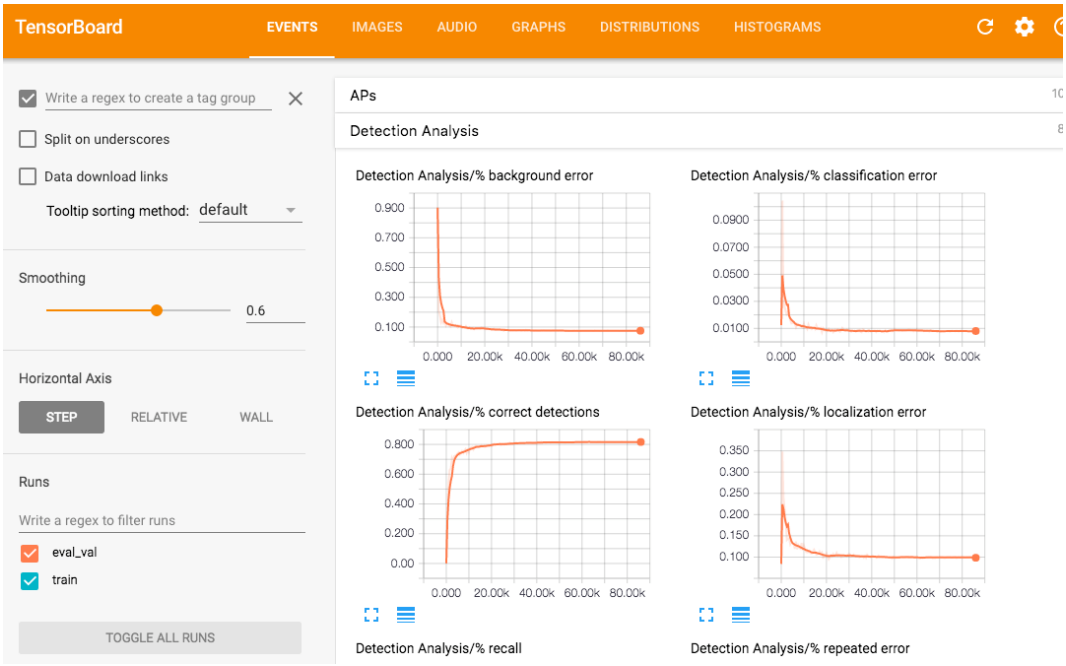

Figure 5.18. TensorBoard

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

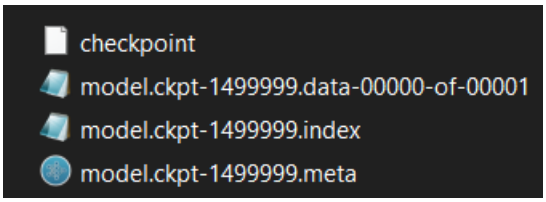

Figure 5.19. Example of Checkpoint Data Files in Log Folder

6. Creating Frozen File

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

6.1. Generating the Frozen .pb File

Generate .pb file from latest checkpoint using the command below from the root directory of the training code.

```
$ python src/genpb.py –ckpt_dir <log directory> --freeze
```

For example, *python src/genpb.py –ckpt_dir logs/demo/train –freeze*.

This command generates model.pbtxt in inference mode which is then internally used by genpb.py to create frozen pb file.

Figure 5.1 shows the generated .pb file in the log directory

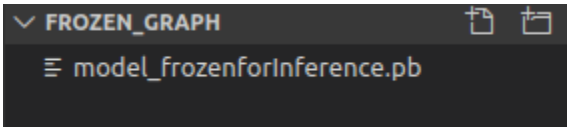

Figure 6.1. Frozen .pb File

7. Creating Binary File with sensAI

This section describes how to generate binary files using the Lattice sensAI version 7.0 program.

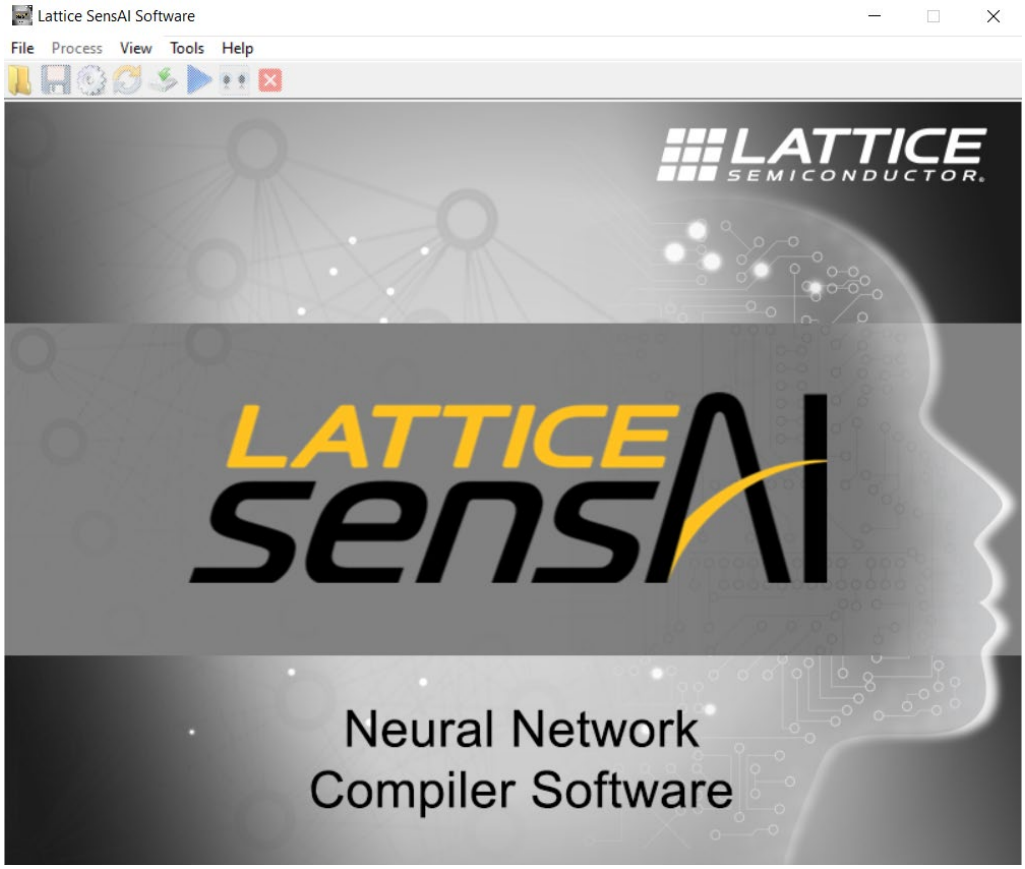

Figure 7.1. sensAI – Home Screen

To create the project in the sensAI tool:

1. Click **File > New**.
2. Enter the following settings:
   - **Project name**
   - **Framework – TensorFlow**
   - **Class – CNN**
   - **Device – AVANT**
   - **IP – Advanced_CNN**

3. Click **Network File** and select the network (.pb) file.

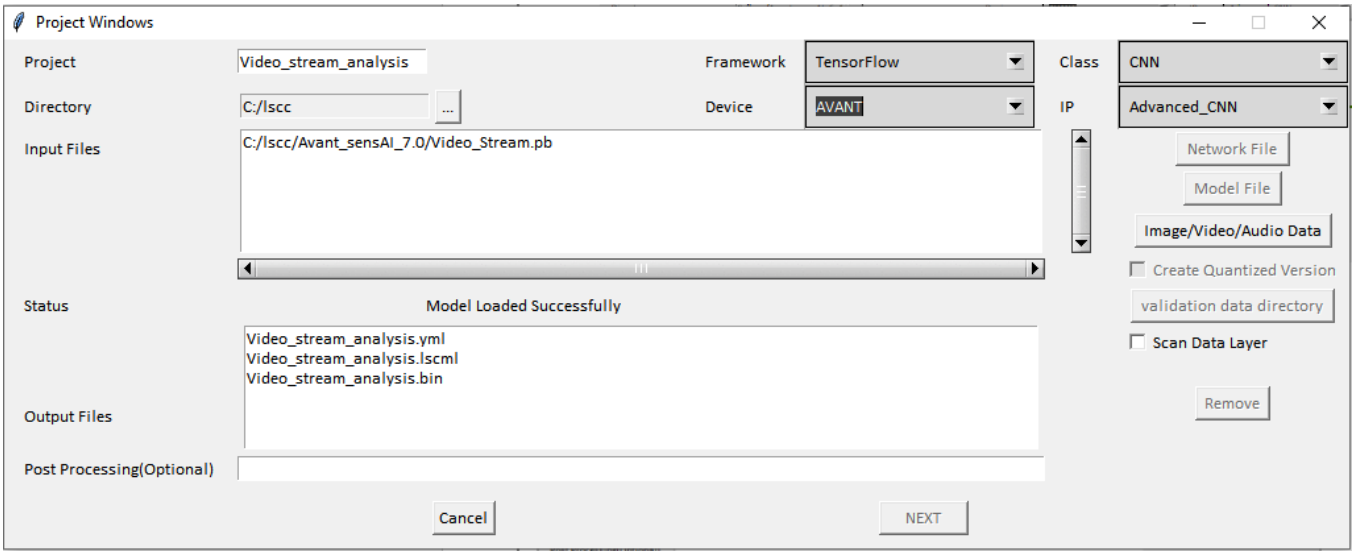

Figure 7.2. sensAI – Select Framework, Device, and Network File

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

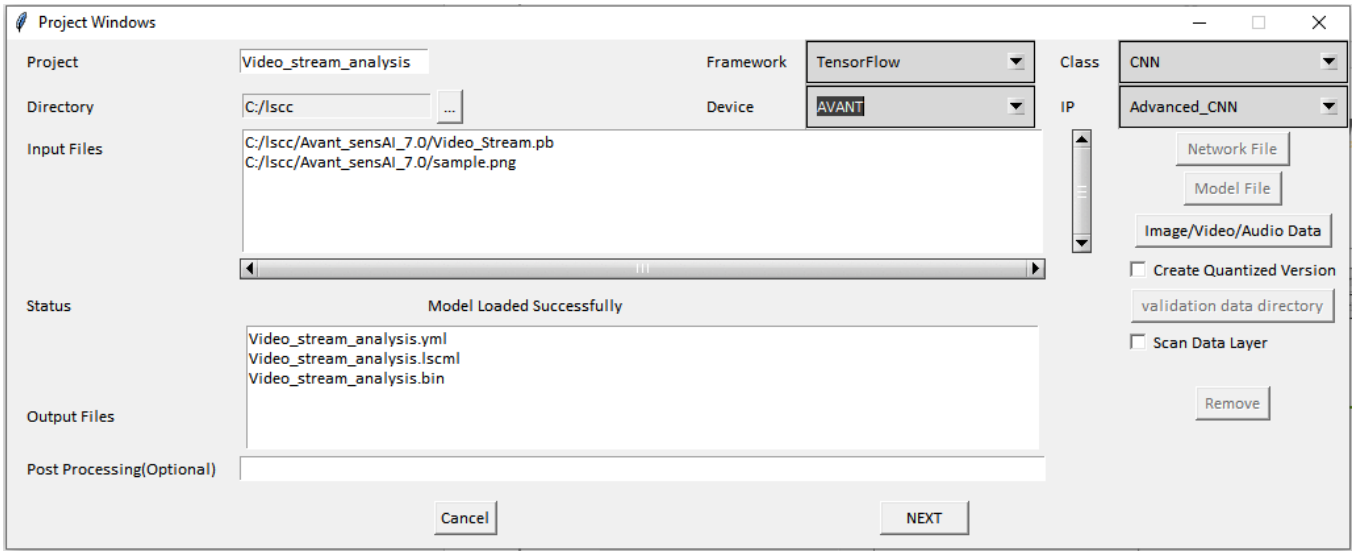

Figure 7.3. sensAI – Select Image Data File

5. Click **Next**.
6. Uncheck the **Store Input, Store Output** and **Result** Readout box, and click Next.

7. Change default values of following attributes to:
   a. **Number of Segments – 8**
   b. **Number of Convolution Engines - 1**

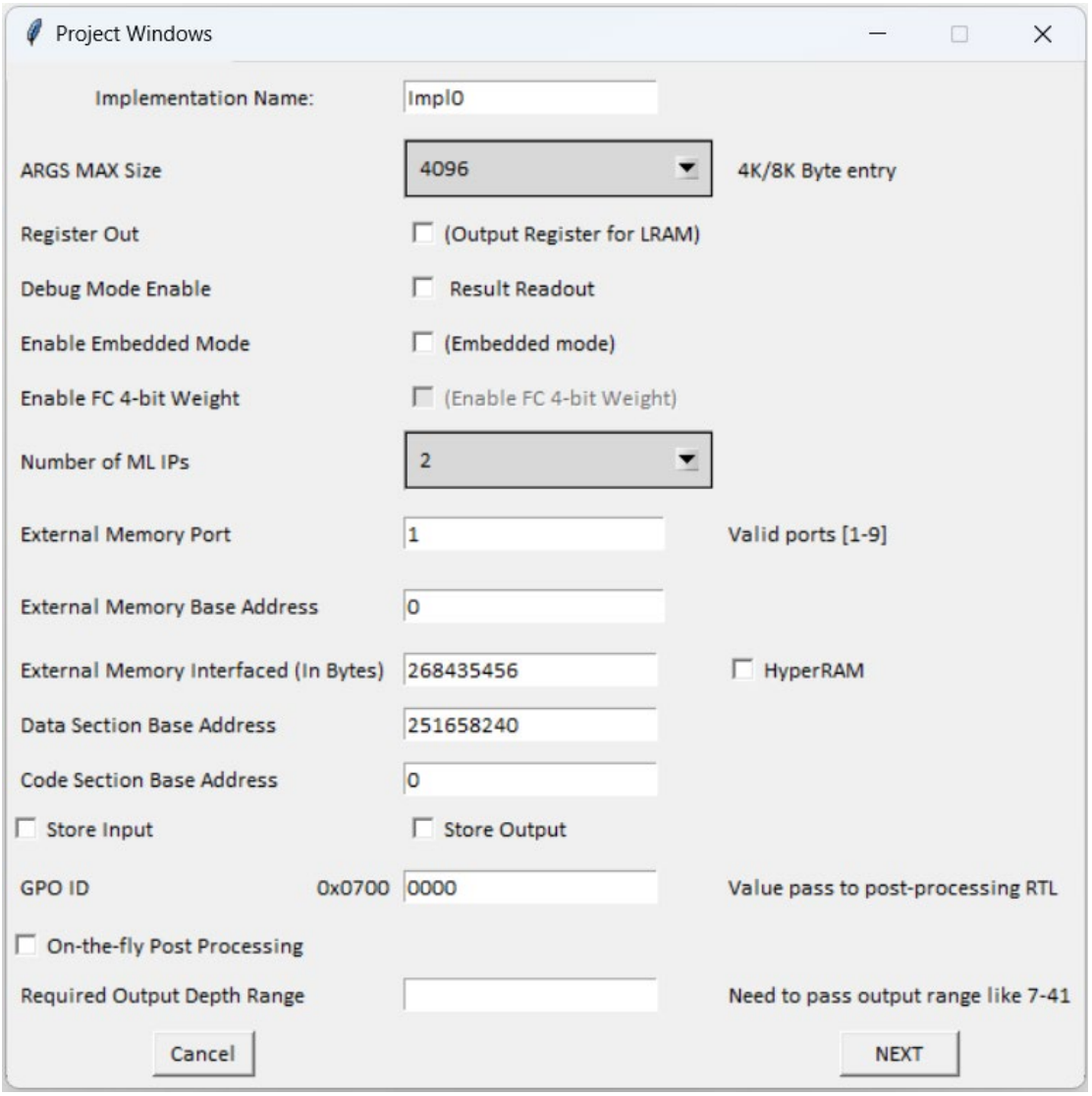

Figure 7.4. sensAI – Update Project Settings (1)

Note: Please ensure that ‘Store Input’, ‘Store Output’ & ‘Result Readout’ check boxes are un-checked.

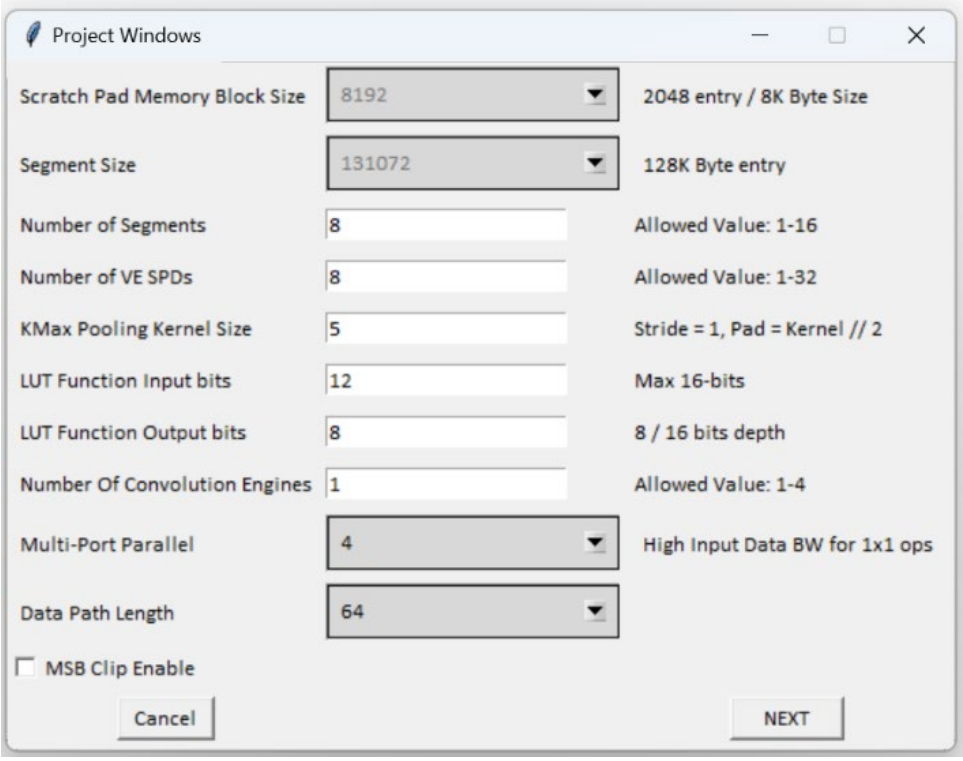

Figure 7.5. sensAI – Update Project Settings (2)

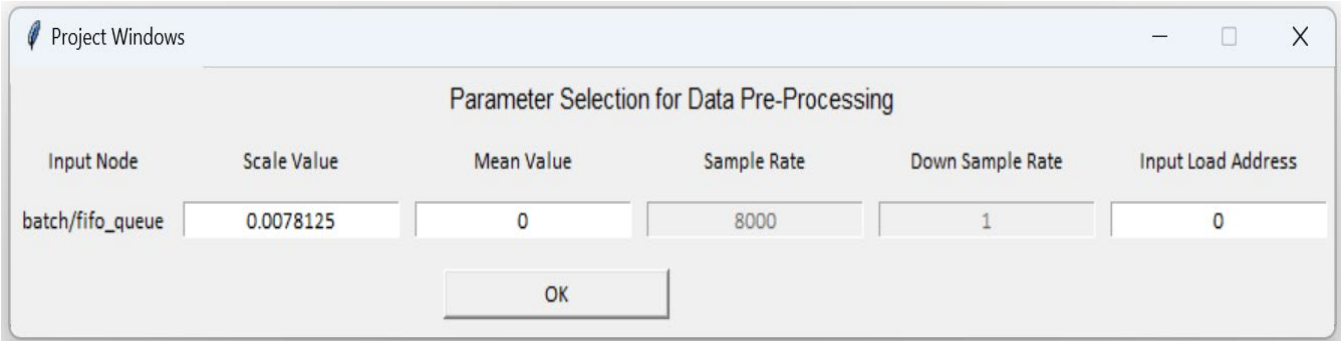

Figure 7.6. sensAI – Update Project Settings (3)

8. Click **OK** to create the project.
9. Double-click **Analyze**.

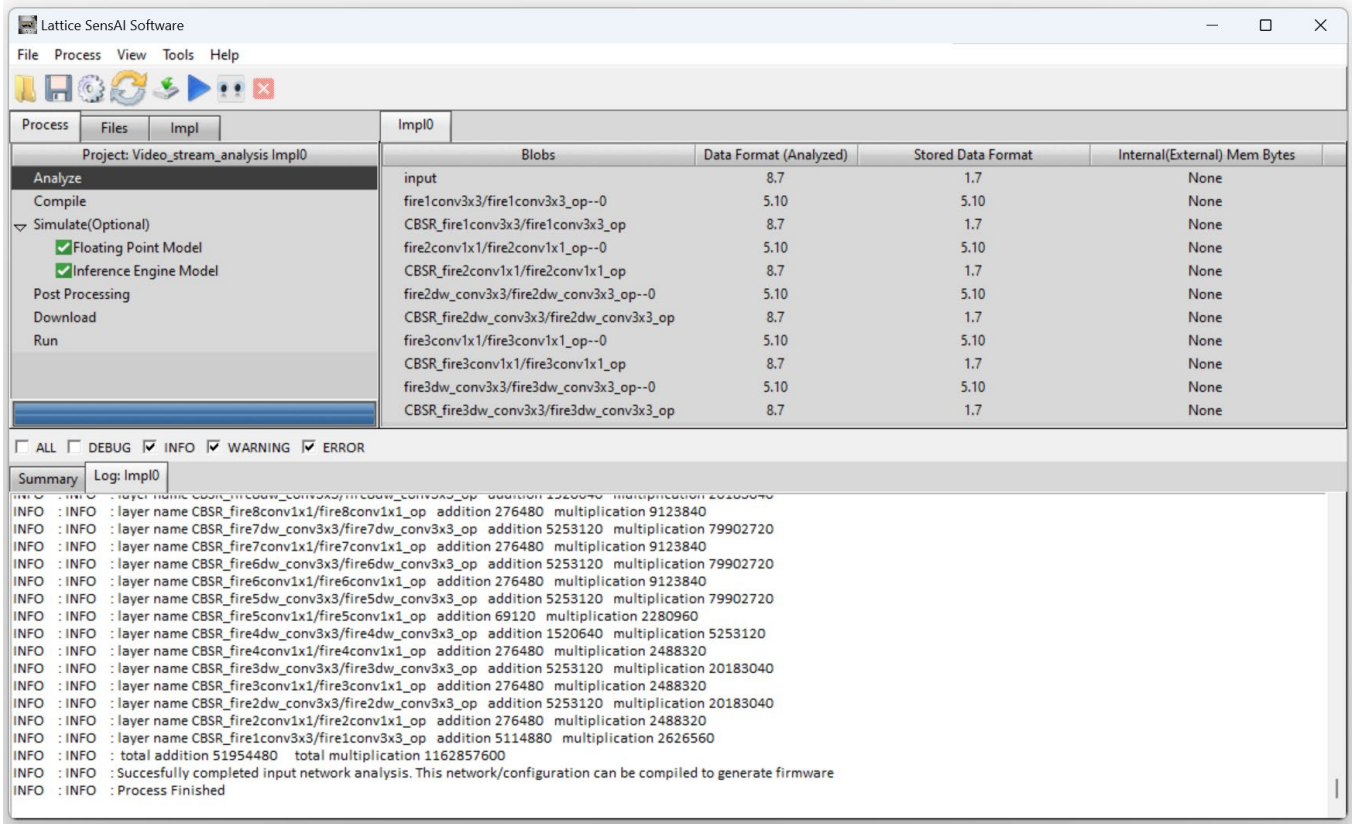

Figure 7.7. Analyze Project

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

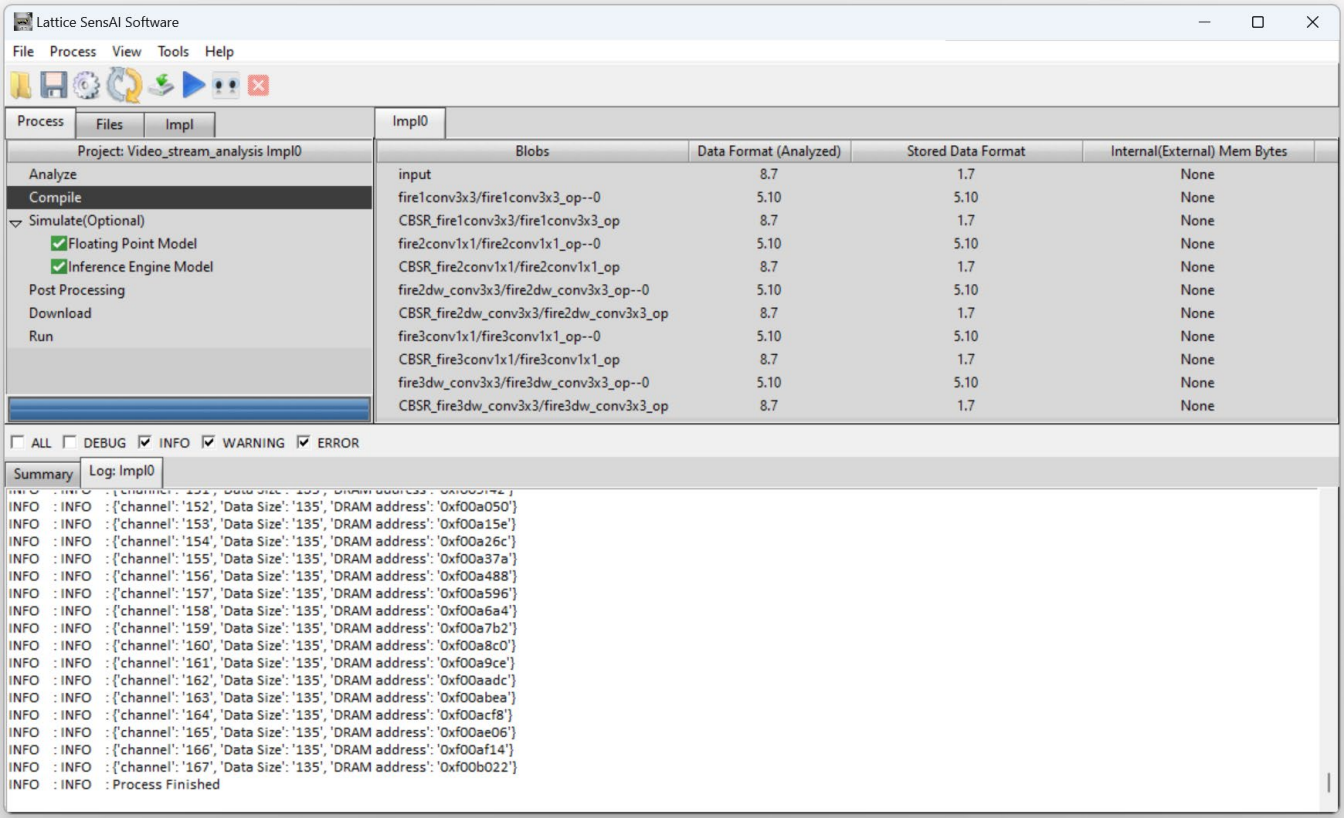

Figure 7.8. Compile Project

11. Two firmware .bin files are displayed in the compilation log. Use the generated firmware .bin files on hardware for the respective ML engine testing.

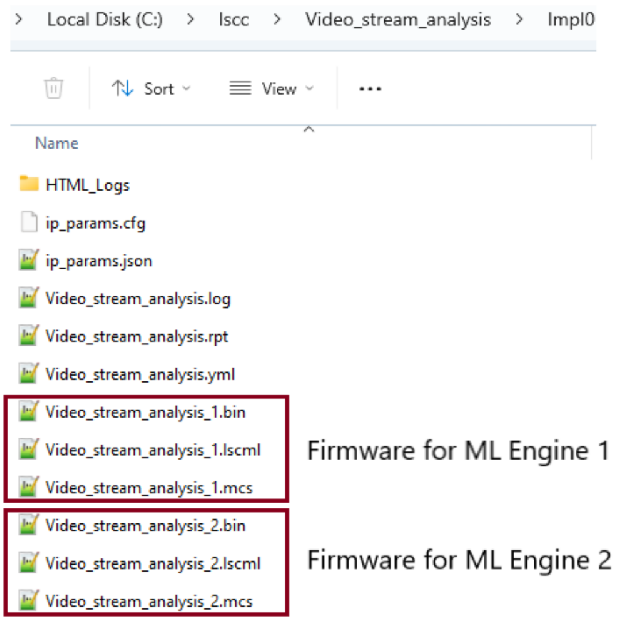

Figure 7.9. Firmware files for ML Engine 1 and 2

8. Hardware (RTL) Implementation

8.1. Top Level Information

8.1.1. Block Diagram

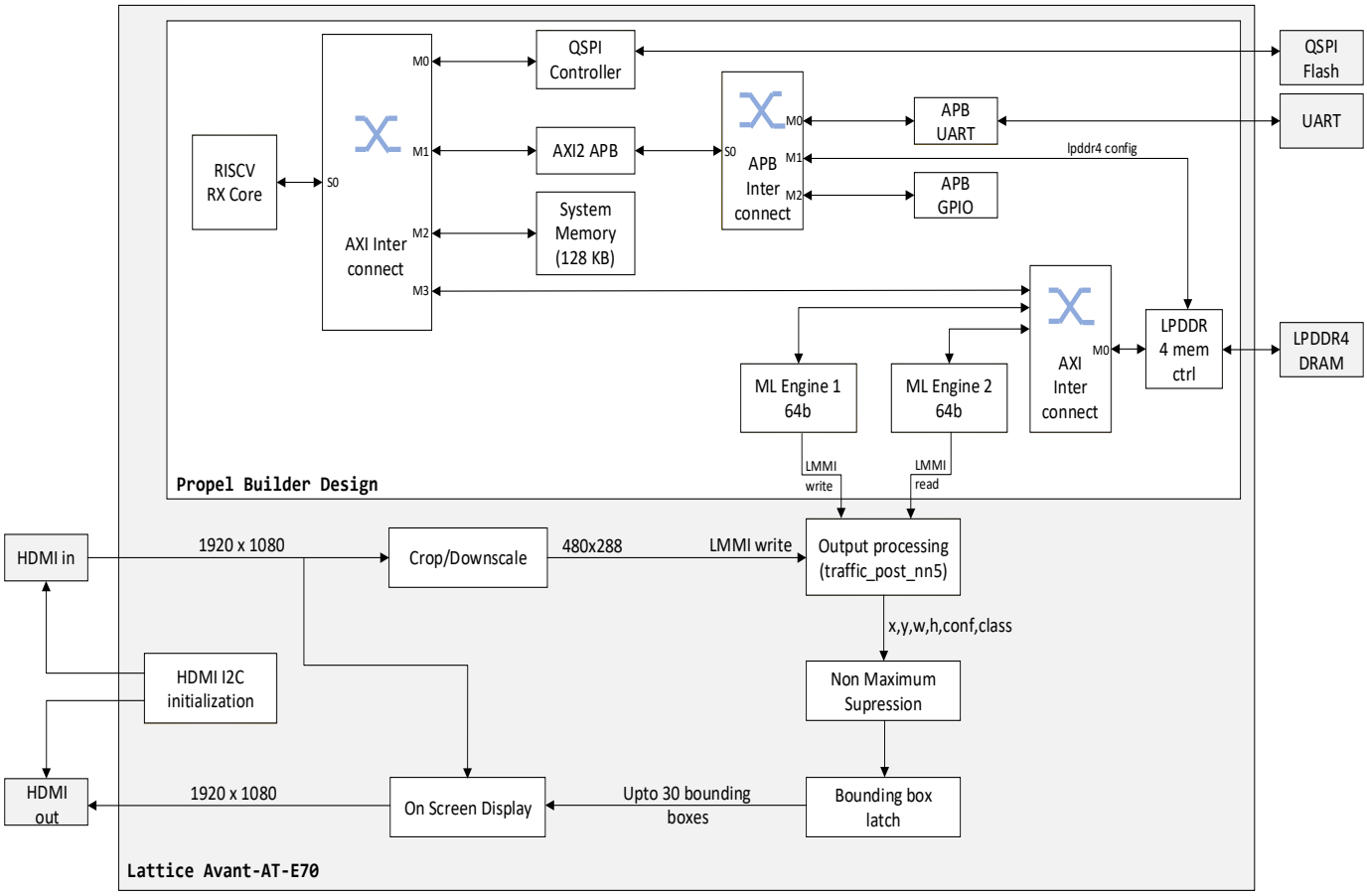

Figure 8.1. Top Block Diagram of Avant Traffic Demo with Lattice Avant-AT-E Evaluation Board (Rev D)

8.1.2. Architecture

- The FPGA design interfaces with Alinx HDMI I/O module (LPC FMC card) to capture frames from a 1080p HDMI video input source, perform object classification ML processing on it, and finally overlays the resultant bounding boxes on the input video, which is output through HDMI out of Alinx card.
- There are two Advanced CNN Accelerator v2.0.1 ML Engines. . They operate in 64b data path mode for higher performance. They have one AXI manager interface to fetch firmware instructions and data and an LMMI subordinate interface for external modules to read data from or write data into the ML engine's internal memory.
- The design uses the LPDDR4 DRAM on the Lattice Avant-AT-E Eval board as the external memory for storing the firmware instructions necessary for the ML Engine to execute the ML model, as well as to store and load intermediate layer outputs during ML processing.
- The LPDDR4 memory is exposed as an AXI subordinate for the rest of the design through the Avant memory controller and AXI interconnect. The AXI interconnect manages this subordinate interface. The interconnect itself has the ML engines and the RISC-V RX v2.4.0 core as the three managers working with external memory.

- The ML firmware compiled from the neural network model using the sensAI compiler is programmed in QSPI Flash at a known address, to preserve firmware across power cycles. During system initialization after power on and bitstream configuration, this firmware from QSPI flash is read by the RISC-V Core using QSPI Controller and written to LPDDR4 memory at a predefined address. ML Engine 1 firmware is loaded at address 0x80000000 in the system address map, which is address 0x0 of LPDDR4 memory. ML engine 2' firmware is loaded at address 0x81000000 in the system address map, which is address 0x01000000 of the LPDDR4 memory.
- The RISC-V core is initializing the LPDDR4 and SPI memory modules during boot up. After successfully loading the firmware of both the ML IPs, The RISC-V core indicates the successful loading to rest of the logic using memory mapped GPIO pin [0]. The RISC-V core firmware resides in 128 KB system memory implemented inside the FPGA using EBR resources.
- During system initialization, the HDMI encoder and decoder chips on the Alinx FMC card, are programmed over I2C interface to enable the 1080p streaming video pipeline.
- The AXI bus interfaces are clocked at 100 MHz, while the ML Engines perform processing at 170 MHz. The RISC-V core runs at 25 MHz
- The on-screen display module combines the input video stream with the results of ML processing (bounding boxes) to provide a single output video stream for HDMI out.

8.1.3. System Address Map

Table 8.1. ML Engine 1 and ML Engine 2 Address Map

Target	Start Address	End Address	Size
LPDDR4 Memory	0x8000_0000	0xBFFF_FFFF	1 GB

Table 8.2. RISC-V Core Address Map

Target	Start Address	End Address	Size
System Memory	0x0000_0000	0x0001_FFFF	128 KB
APB GPIO	0x4000_0000	0x4000_0FFF	4 KB
APB UART	0x4009_0000	0x4090_0FFF	4 KB
LPPDDR4 ctrl APB	0x4009_2000	0x4009_2FFF	4 KB
QSPI Controller	0x4030_0000	0x4030_0FFF	4 KB
LPDDR4 Memory	0x8000_0000	0xBFFF_FFFF	1 GB

8.1.4. Operational Flow

- The real-time input image data is streamed in from Alinx card as parallel 24-bit RGB per clock, at 1080p resolution. This is then sent to the crop/downscale module which will downscale the 1080p(1920x1080) image data received from camera to 480x288(with bottom padding). The downscaler uses a frame buffer to improve the performance.
- This downscaled image is provided as input for the NN model. The downscaled image is written directly into the internal memories of the ML Engine 1 over the LMMI interface. The ML Engine has a single LMMI subordinate interface, which interfaces with the manager interface in traffic_post_nn5.v module. The crop downscale data is written to ML engine through this traffic_post_nn5 module.
- Once input is fully written, the ML Engine is ready to start execution of the ML Model. It does so by fetching firmware instruction codes from the external memory and executing layers according to it.
- Once the ML engine processing is completed, the output processing module reads the neural network output over LMMI interface of ML engine 2 and formats it into x,y,w,h, confidence, and class scores that characterize the bounding boxes.
- As part of the post-processing, non-maximum suppression is performed to filter the bounding boxes based on threshold.
- The final bounding boxes from this module are latched (up to 30) and sent to the on-screen display module for drawing these bounding boxes on the input video stream.

9. Creating FPGA Bitstream file

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

To create the FPGA bitstream file:

1. Open the Lattice Radiant software tool, as shown in Figure 9.1.

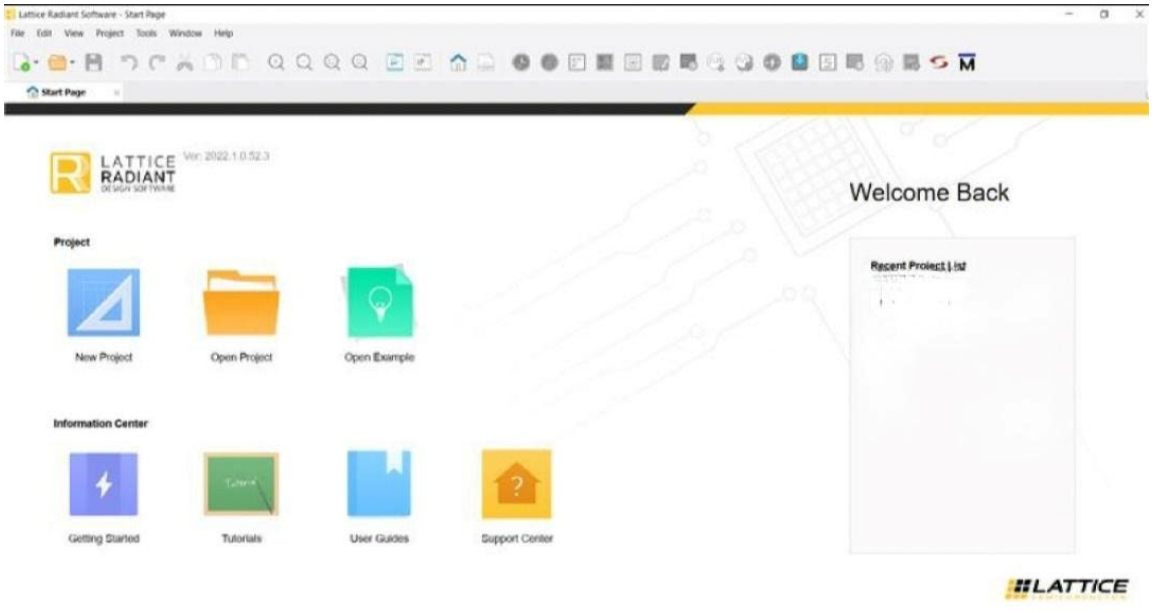

Figure 9.1. Lattice Radiant Software

2. Click **File > Open Project**.
3. From the project database, open the Lattice Radiant project file (.rdf), as shown in Figure 7.2.

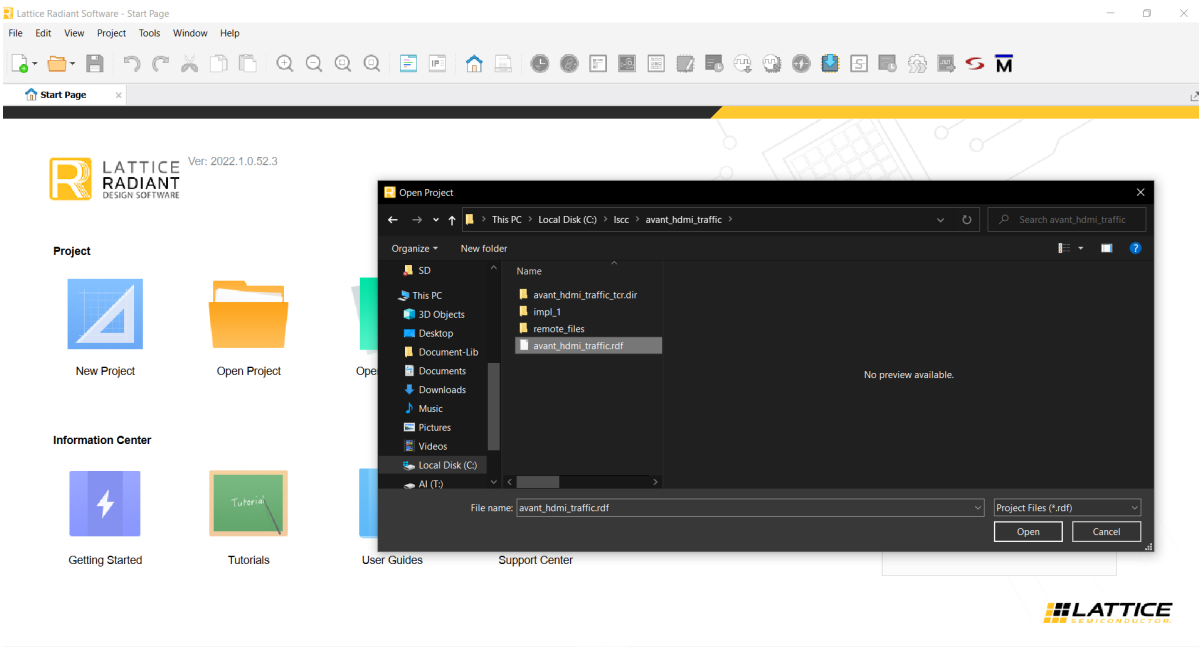

Figure 9.2. Lattice Radiant Software – Open Project

4. Click **Export Files** to generate the .bit file.
5. Click **Export Files** to generate the .bit file. The user can see that this process is completed in the output window, as shown in Figure 9.3. Find the generated .bit file at location */project_folder/impl_1*.

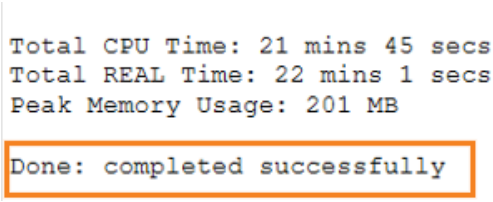

Figure 9.3. Lattice Radiant Software – Bitstream Generation Export Report

References

For complete information on the Lattice Radiant Project-Based Environment, Design Flow, Implementation Flow, and Tasks, as well as on the Simulation Flow, refer to the Lattice Radiant software user guide.

- Avant-E web page
- Lattice sensAI Stack web page
- Lattice Radiant Software web page
- Lattice Insights for Lattice Semiconductor training courses and learning plan.

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/Support/AnswerDatabase.

Revision History

Revision 1.2, November 2024

Section	Change Summary
All	• Made editorial fixes. • Updated *Lattice Avant-AT-E Evaluation Board Rev B* to *Lattice Avant-AT-E Evaluation Board Rev D*.
Disclaimers	Updated the boilerplate.
Hardware and Software Requirements	• Updated Figure 2.1. Top View of Lattice Avant-AT-E Evaluation Board. • Updated section 2.2 Software Requirements.
Creating Binary File with sensAI	• Updated Figure 7.2. sensAI – Select Framework, Device, and Network File, Figure 7.3. sensAI – Select Image Data File, Figure 7.4. sensAI – Update Project Settings (1), Figure 7.7. Analyze Project and Figure 7.8. Compile Project. • Added Figure 7.5. sensAI – Update Project Settings (2), Figure 7.6. sensAI – Update Project Settings (3) and Figure 7.9. Firmware files for ML Engine 1 and 2. • Updated the procedure sections of creating binary file with sensAI.
Hardware (RTL) Implementation	• Updated Figure 8.1. Top Block Diagram of Avant Traffic Demo with Lattice Avant-AT-E Evaluation Board (Rev D). • Added section 8.1.3 System Address Map. • Updated sections Architecture and Operational Flow.

Revision 1.1, December 2023

Section	Change Summary
Disclaimers	Updated with the latest disclaimers.
Hardware and Software Requirements	• Removed Lattice Avant evaluation FMC HyperRAM card, Alinx FMC board, WD TV live media player, microSD card, and AiTrip MicroSD card module from the Hardware Requirements section. • Updated the Lattice Radiant software version from 2022.1 to 2023.1 in the Software Requirements section. • Removed the Win32 Disk Imager tool from the Software Requirements section.
Creating Binary File with sensAI	• Updated Lattice sensAI version from 6.0 to 6.1. • Updated procedure for creating a project in the sensAI tool.
Hardware (RTL) Implementation	Newly added section.
References	Newly added section.

Revision 1.0, March 2023

Section	Change Summary
All	Initial release.

www.latticesemi.com