AN13331

Glow with PyTorch Model for Embedded Deployment Using eIQ

Rev. 0 — 06-Sep-2021 Application Note

1 Introduction

Glow is a machine learning compiler that accelerates the performance of neural network frameworks on different hardware platforms. The compiler takes in machine learning frameworks such as PyTorch and Tensorflow and produces optimized code for accelerators. Glow requires the machine learning model to be in specific formats (that is, ONNX, Caffe2, and TFLite) for conversion. Users should review Glow documentation to verify supported formats.

PyTorch is an open source machine learning framework developed by Facebook. The API used by PyTorch is seamlessly integrated into Python which allows for an easier coding experience. PyTorch also has built-in support for ONNX (an open format for ML models). ONNX allows easy transition between models from different frameworks and tools.

This application note describes the process to generate a model with PyTorch for the Cifar10 dataset. It also elaborates how to deploy it on an i.MX RT1060-EVK board using the Glow Library provided by eIQ software environment.

2 Overview

This application note demonstrates the process of creating a simple image recognition neural network written in PyTorch and trained on the CIFAR-10 dataset. The neural network is exported in ONNX format for compatibility with Glow. Glow tools created object files from the model to run on the i.MXRT1060 platform.

Machine Learning concepts are introduced and PyTorch layers are explained to familiarize the user with ideas used during model creation and training in PyTorch.

2.1 Machine Learning concepts

Some common terms used in machine learning are referred throughout the document. These are described below.

1. **Convolutional Layer:**

   In machine learning, a convolutional layer (or a convolution) is created by passing a moving filter along the input matrix or other convolutional layers. This applied filter extracts features from the data (image) and creates a feature map (matrix) as the output. When stacking convolutional layers, the filter is applied to the extracted features of the previous layer. The act of convolution also reduces the shape of the model.

2. **Channels:**

   In convolutional neural networks, channels correspond to the amount of feature maps created from the convolutional layer. When relating to images, channels correspond to the total color channels used. In CIFAR-10, the dataset is in RGB (Red Green Blue) format, which means the input image has 3 channels.

3. **Activation Function:**

   An activation function is a non-linear function that is used to introduce non-linear properties to the model. The addition of a non-linear activation function between convolutional layers allows for more complex relationships to be mapped between the input and output.

4. **Fully Connected Layer**:

   The fully connected layer allows the result of convolutional layers to be output as a linear classifier. Before the fully connected layer, the result of the convolutional layers should be flattened to a 1D array.

5. **Optimizer:**

The optimizer is an algorithm used during training that updates weights to minimize the loss. The optimizer is needed in the training process to allow neural networks to learn efficiently. How the optimizer updates the weights or learning rates in the model is dependent on the optimizer chosen.

6. **Epoch:**

   Epoch defines the number of times the user traverses through the entire training dataset. By increasing this value, the model weights have more chances to learn the important features to extract. When too many epochs are used, the training might cause your model to overfit the training dataset.

7. **Learning Rate:**

   Learning rate determines how much the weights are adjusted when the model updates. The value is a very small positive value in the range of (0,1). The learning rate also affects how the model can find a solution. When a learning rate is too small, the training process takes longer and can potentially get stuck during training. When the learning rate is too large, the training can result in a suboptimal solution.

8. **Batch Size:**

   Batch size during training refers to the number of examples from the dataset included in 1 iteration/step. The number of iterations used in 1 epoch is given by:

   - iterations = (training dataset size) / (batch size).

9. Larger batch sizes can allow for more parallelization (less training time per epoch) if CPU memory allows for it. Typically, smaller batch sizes allow for a more generalized model and larger batch sizes allow for a larger learning rate.

3 Software and hardware installation

This section describes the steps required to install the eIQ software and PyTorch on your computer system.

3.1 Glow installation

1. Install MCUXpresso IDE 11.2.0 or later.
2. Install a terminal program like TeraTerm.
3. Download and install the Glow package from eIQ™ for Glow Neural Network Compile website. This installs the Glow compiler and helper programs into **C:\NXP\Glow.**
4. Add the installation path (" **C:\NXP\Glow**") to your environment variables.

3.2 ML tools installation

1. Download and install Python 3.7.0 (64-bit) from https://www.python.org/downloads/. **Note:** The 64-bit edition is required. Using a later version of python might cause problems with the imported packages.
2. Open a command prompt and verify that the Python command corresponds to Python 3.x. You must use "`python3`" for all the commands instead of "`python`".

```
python -V
```

NOTE

If the command does not find Python, turn off Python in '**Manage app execution aliases**' in the **Settings** tab. Update the python installer tools:

3. Use the commands below:

```
python -m pip install -U pip
python -m pip install -U setuptools
```

4. Install Python packages used in training scripts:

```
python -m pip install numpy==1.18.5
python -m pip install bokeh==2.1.1
```

5. Install **`torch`** and **`torchvision`** packages to use PyTorch. To perform this step via command line, visit the PyTorch website (https://PyTorch.org/). Then, select your preferences in the "**Quick Start Locally**" section to receive an install command.

   Setup used for this exercise

   - `Build: Stable`
   - `Your OS: Windows`
   - `Package: Pip`
   - `Language: Python`
   - `Cuda: None (CPU)`

   Note: Without CUDA enabled, PyTorch packages run on the CPU. Problems might occur if a CUDA variant is used on a system without a GPU driver installed.

6. The "**`Cifar10 Pytorch`**" directory structure contains a training directory, a dataset directory, and a saved model directory. Below is a summary of the scripts and important files in each directory.

- Dataset
  - **Cifar-10-python.tar.gz**: Pickled file containing Cifar-10 training and testing dataset.
- PyTorch Models
  - **Cifar.onnx**: Pretrained model using the training described in this document.
- Training
  - **Train_Cifar.py**: Trains ML model from scratch and exports in ONNX format.
  - **Train_TransferLearning.py**: Trains ML model from scratch and exports in `.pth` format.
  - **Retrain.py**: Uses `.pth` model input to continue training and export in `.pth` format.
  - **Convert_Pth_to_ONNX.py**: Converts `.pth` model to `.ONNX` format for embedded deployment.

The **Train_Cifar.py** is the primary script used to train the neural network from scratch. The next chapter describes the behavior of this script.

4 PyTorch model with CIFAR-10 Dataset and Glow for Embedded Deployment Using eIQ

4.1 Layer and training description of the model

This section shows the different layers and training options that can be considered when making a model of your own. When the layers have multiple parameters the descriptions will show how to call this layer with PyTorch and what each parameter represents.

2D Convolution: Conv2d(input channels, output channels, kernel size, padding)

Performs 2D convolution on input image to create N output channels that will be trained to extract N features from the image. This convolution is performed by moving a matrix referred to as the kernel across the input image matrix.

When deciding on kernel size, always consider the input size of the image and the amount of details in the image. Generally, smaller kernels will extract more details whereas larger kernels allow for a loss of information. Another trend in image recognition is to use a square shaped kernel with an odd number as the kernel size. A square shaped kernel is more commonly used when there is no preference of which direction a rectangular pattern can be detected. An odd filter size is preferred due to the symmetry around

a center pixel. With an even filter size, the model would have to account for distortions across layers. The kernel size parameter is commonly set to smaller values such as 3 or 5 and assumes a square shape (3x3 or 5x5) from an int.

To ensure that the shape of the model does not reduce from convolution, padding is included, using the following equations:

$$H_{out} = \frac{(H_{in} + (2 * Padding) - (dilation) * (kernel\_size - 1) - 1)}{Stride} + 1$$

Figure 1. Equation 1

$$W_{out} = \frac{W_{in} + (2 * Padding) - (dilation) * (kernel\_size - 1) - 1}{Stride} + 1$$

Figure 2. *Equation 2*

Where

- H_{in}= input height (before convolution)
- H_{out}= output height (after convolution)
- W_{in}= input width (before convolution)
- W_{out}= output width (after convolution)
- Dilation= spacing between kernel elements (default=1)
- Kernel_size= shape of matrix to perform convolution
- Padding = zero-padding added to both sides of the input (column and row)

For more detailed information about the parameters, visit PyTorch's nn.Conv2d Documentation.https://pytorch.org/docs/master/generated/torch.nn.Conv2d.html

Activation function

There are a variety of activation functions that can be chosen at the output. However, this application note implements the most common functions, which are ReLU, LeakyReLU, Sigmoid, Tanh, and Softmax.

The below figures depict the graphical representations of the non-linear activation function.

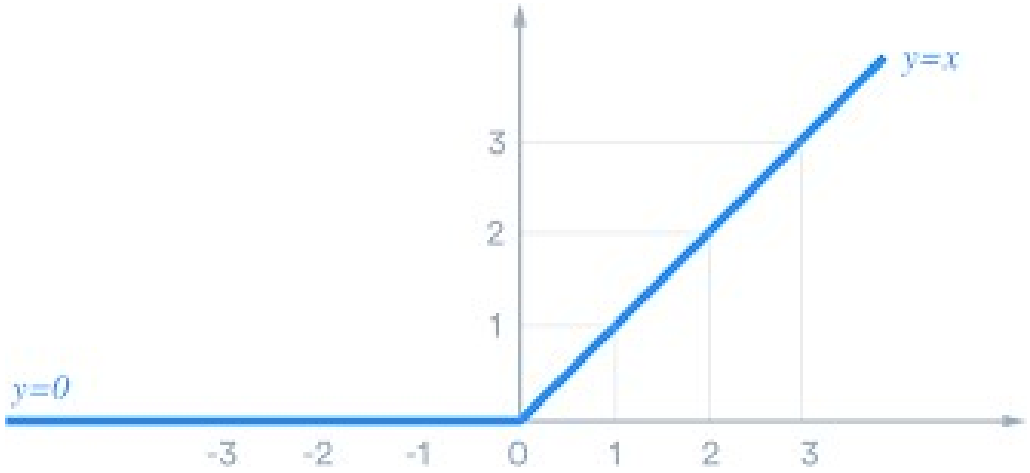

Figure 3. ReLU

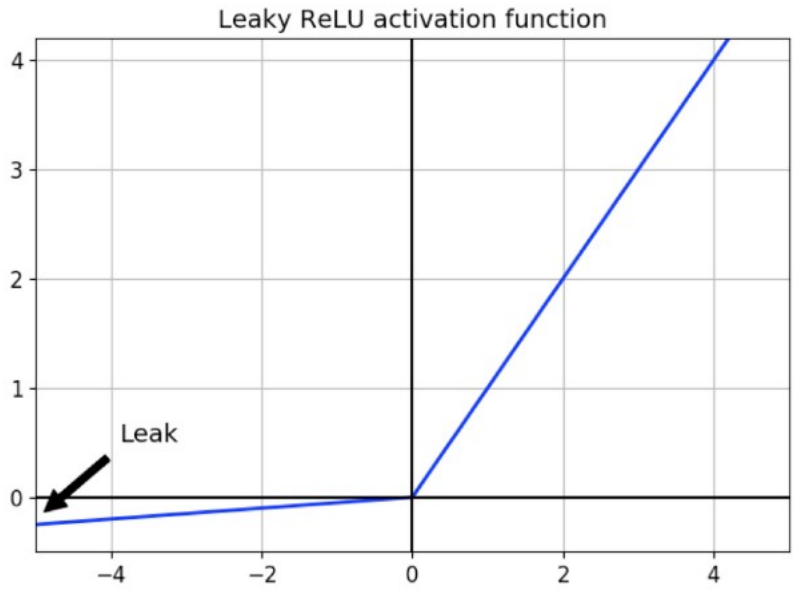

Figure 4. Leaky ReLU

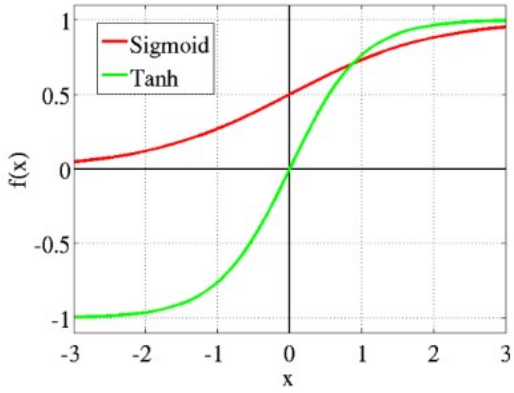

Figure 5. Sigmoid Vs. Tanh

NOTE

Softmax is calculated by using the equation below:

$$Softmax(y_i) = \frac{e^{y_i}}{\sum_1^n e^{y_k}}$$

Figure 6. Softmax calculation

This is similar to the sigmoid function in that the range of values is between 0 and 1, but the denominator is used to make the values span from [0,1] and sum to 1 (similar to probabilities). From this information, a single-variable (2d) plot is not shown because it is a multi-variable function. Softmax is mainly used at the output of a model to convert the output model values into a probability used later for confidence.

Key differences:

As seen in the activation, ReLU, and LeakyReLU plots only differ for inputs x < 0. When using ReLU, a problem might arise (Dying ReLU Problem), where certain weights can update such that they will always be 0. This mainly occurs when the learning rate is too high. LeakyReLU attempts to fix the problem by adjusting x < 0 activations to instead have a very small slope (PyTorch uses the slope .01).

Tanh is a rescaled and shifted Sigmoid with the equation: **tanh(x) = 2sigmoid(2x) – 1**.

The main differences in these functions is the steepness of the gradient of tanh compared to sigmoid and the range of output values.

- **Batch Normalization: BatchNorm2d(num_features)**

Effectively, this technique converts outputs between layers into a standardized format. This allows the future layers to more efficiently process the outputs of previous layers. The main parameter for this call is named num_features, which corresponds to the amount of channels in the previous connecting layer. The use of this function allows for higher learning rates and faster training speeds.

- **Pooling: Maxpool2d(kernel_size = 2, stride = 2)**

Pooling scales down the image by a factor of 2. In this case, we use max pooling by taking the max value in each 2x2 kernel. This effectively reduces the amount of calculations needed in the model and provides the ability to generalize inputs. With the addition of pooling, the model (conv2d) can afford more channels/calculations in deeper layers of the network.

- **Dropout: dropout(p=.5)**

Reduces the chance of overfitting by shutting down parts of the neural network at random during training. The parameter p is the probability that a specific neuron in the model will be zeroed out. By default it is set to a 50% probability but can be adjusted. While this helps in preventing overfitting it increases the amount of training time needed for convergence. For the purposes of the application note dropout will not be included in the model.

- **Linear Layer: Linear(input_channels, output_channels)**

The Linear Layer/fully connected layer is used to interpret the data to output as a classifier. Input_channels correspond to the shape of the image multiplied by the amount of channels in the previous layer (ex: 32 * 32 * channels). Output_channels correspond to the intended size of the 1-Dimensional output. In this document output_channels shapes the output to fit into 10 categories to allow for classifying images (ex: CIFAR-10 has 10 animal categories therefore output_channels=10).

- Optimizer: Commonly used optimizers include the following:
  - SGD: optim.SGD(model.parameters(), learning_rate, momentum)
  - Adam: optim.Adam(model.parameters(), learning_rate)
- SGD: optim.SGD(model.parameters(), learning_rate, momentum)

SGD utilizes the gradients of each sample image to update the model. In PyTorch when uploading the data via DataLoader(), if batch size > 1, the optimizer is considered a mini-batch SGD. This means the optimizer will update weights for each batch/iteration of the epoch. SGD faces the issue of getting stuck in a local minimal while finding the optimal solution. Although the model may occasionally be able to make progress again, training resources are inefficiently used.

Momentum combats this by using an exponentially weighted average of past gradients to update weights instead of relying solely on the current batch. This accelerates the learning in a contextually relevant and optimal direction. Momentum effectively reduces the training time needed for convergence.

- Adam: optim.Adam(model.parameters(), learning_rate)

Adam utilizes the advantages of RMSProp optimization to compute adaptive learning rates and momentum to speed up training. From RMSProp, Adam utilizes recent gradients instead of all past gradients to avoid aggressively decreasing the calculated learning rate. Keep in mind, the learning rate hyperparameter remains constant and learning rate refers to the adjusted learning rate calculated by the optimizer.

4.2 Model definition and visuals

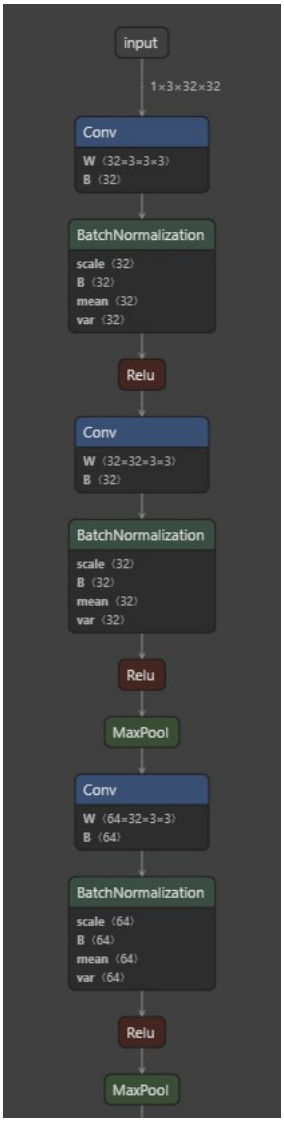

Figure 7. Model visualization in Netron

```
channel = 32
class NeuralNet(nn.Module):                                         #Create model class to inherit from nn.Module
    def __init__(self):
        super(NeuralNet, self).__init__()
        self.layer1 = nn.Sequential(                                # Container Class
            nn.Conv2d(3, channel, kernel_size=3, padding=1),        # Extracts 32 (32x32) feature maps from RGB image
            nn.BatchNorm2d(channel),                                # Normalize extracted features to each batch
            nn.ReLU(),                                              # Activate normalized data for non-linearity (3)
            nn.Conv2d(channel, channel, kernel_size=3, padding=1),  # Extracts 32 (32x32) feature maps from processed feature maps (1)
            nn.BatchNorm2d(channel),                                # Normalize extracted features to each batch
            nn.ReLU(),                                              # Activate normalized data for non-linearity (3)
            nn.MaxPool2d(kernel_size=2, stride=2))                  # Pooling to reduce training parameters and generalize learned features
        self.layer2 = nn.Sequential(
            nn.Conv2d(channel, channel*2, kernel_size=3, padding=1),    # Extracts 64 (16x16) feature maps from processed pooled features
            nn.BatchNorm2d(channel*2),                                  # Normalize extracted features to each batch
            nn.ReLU(),                                                  # Activate normalized features for non-linearity
            nn.MaxPool2d(kernel_size=2, stride=2))                      # Pooling to reduce training parameters and generalize learned features
        self.layer3 = nn.Sequential(
            nn.Conv2d(channel*2, channel*2, kernel_size=3, padding=1),  # Extracts 64 (8x8) feature maps from processed pooled features
            nn.BatchNorm2d(channel*2),                                  # Normalize extracted features to each batch
            nn.ReLU(),                                                  # Activate normalized data for non-linearity
            nn.MaxPool2d(kernel_size=2, stride=2))                      # Reduce training parameters (4x4) connected to Linear layer
        self.fcLayer = nn.Linear(4 * 4 * channel*2, 10)                 # Convert Learned training into a classifier output


    def forward(self, x):                   #Path through the model using nn.Sequential container class
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = out.reshape(out.size(0), -1)  # Reshape 2D weight matrix to 1D Array
        out = self.fcLayer(out)


        return out
```

Figure 8. PyTorch Model definition and forward pass

NOTE

nn.Sequential is a container class used by PyTorch, which is a useful way of stacking layers to have a more readable forward pass through the model. The container class is used when initializing the neural network object (as seen in the figure above) and allows for the functions to be called in the order listed.

NOTE

The initial channel size of 32 is chosen because it was increased until the increase in model quality was less noticeable. This isn't the most accurate model but is intended to optimize size and accuracy.

Structure of layers

Through experimentation, the highest accuracy was achieved by using the following layer structure:

NOTE

Due to the input image size of 32x32, pooling is used in each layer with a maximum of 3 layers to make an output shape of 4x4. Further pooling is not used to avoid oversimplifying the features that were extracted. To increase the accuracy of the model, this layer structure is added to the start of the first layer without pooling. In different dataset cases or while using larger channels, pooling to a shape of that size may potentially cause a loss of information/accuracy.

High Level Model Visual

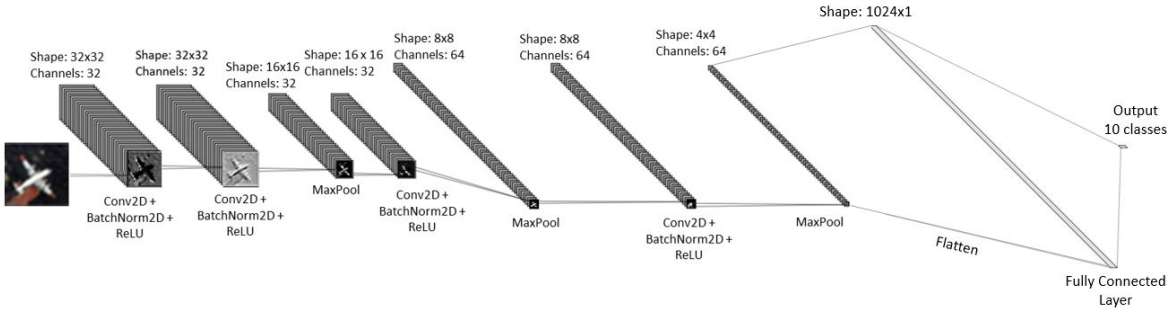

Figure 9. Visual of the model architecture and the image processing through the layers

The images in front of the layers are extracted from the intermediate layers of the model by using PyTorch. As the image of the plane traverses through the model, the images extracted become less readable. This is most apparent toward the final layers as the shape of the image gets reduced by pooling.

Although stacked together, it is the Conv2d layer that learns to filter the image in different ways. The BatchNorm2d and ReLU layers prepare the data as intermediate steps for later convolutions.

4.3 Model/training experimentation

The measurements taken during this experiment are run in PyTorch. The test accuracy percentage is measured from the performance of the model on CIFAR-10 validation set (comprising 10,000 images). The test loss is calculated by calculating the average of the cross-entropy loss over the validation set. Some measurements were taken 2 to 3 times to verify the consistency of the training parameters used. Settings that resulted in accuracy below 50% were taken only once because they were shown not viable to the solution.

The items highlighted in **bold font** yield the best results from training. The highlighted items will be used to update the setup for future tests and the final version of the model.

Current setup:

- Epoch: 5
- Batch Size: 64
- Learning Rate: 0.001
- Optimizer: Adam

Activation functions:

Table 1. Activation functions:

Activation Functions	Test Accuracy	Test Loss
ReLU	77.1% - 77.9%	0.65 - .071
ReLU + Softmax output	70% - 71.5%	0.6 - 0.65
Sigmoid	52.94% - 58.42%	1.1918 - 1.5617
LeakyReLU	76.3%-77.81%	0.66 - 0.7435
Tanh	71.8% - 72.24%	0.7856

Conclusion

Sigmoid and Softmax at the output underperformed in training. The most accurate and consistent of the tested functions are ReLU and LeakyReLU. Although LeakyReLU combats the 'Dying ReLU' problem, it does not result in a notable increase in performance. Due to more consistent accuracies and loss, the ReLU activation function is chosen for this model.

NOTE

Keep in mind the utility of Softmax as the last layer is to allow for the output to be in terms of probability. When running the model on the board, this is interpreted as the confidence of the prediction. Without this function, the confidence values have a larger range and must be converted at runtime to get an accurate reading of confidence.

Optimizer

When deciding to build your model, it is important to consider the optimizer that directly impacts the training of your model.

Table 2. Optimizer functions:

Optimizer	Test Accuracy	Test Loss
Adam	77.1% -77.9%	0.6 - 0.65
SGD	16.43%	2.29
SGD + Momentum (.9)	42.48%	1.6277
SGD + Momentum(.99)	70% - 71.32%	0.8435 - 0.87

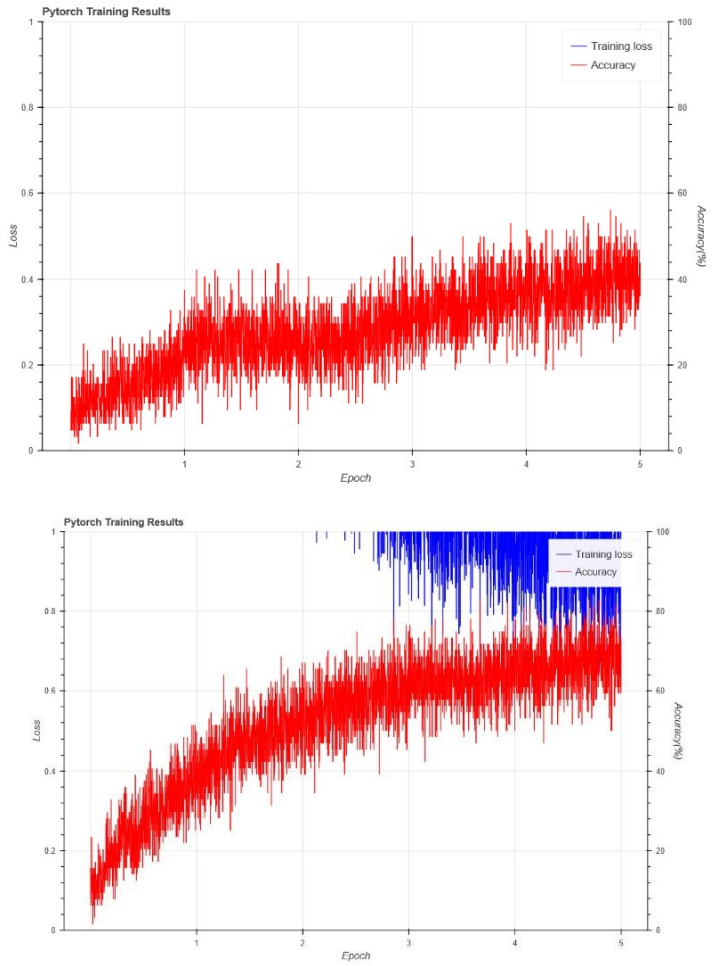

Figure 10. Training progression using SGD with Momentum 0.9 and 0.99.

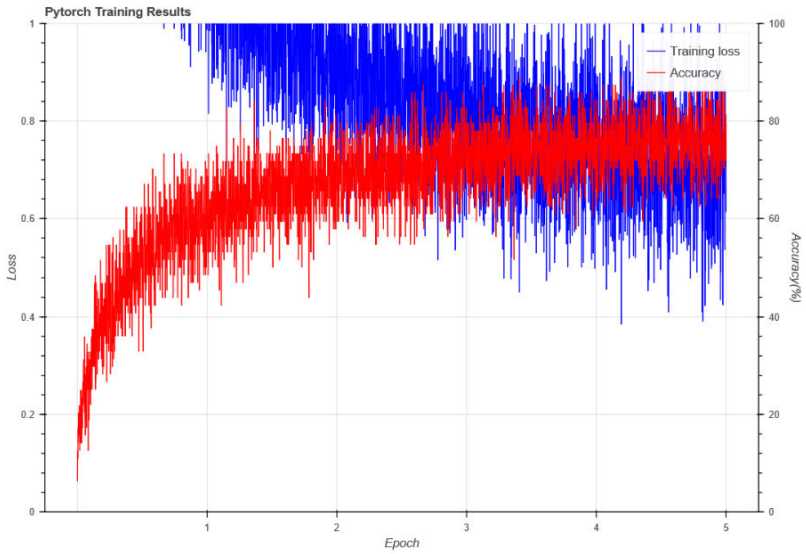

Figure 11. Training progression using Adam Optimizer

Conclusion:

The Figure 10 visualizes how SGD can get stuck during training. As momentum is increased, the use of SGD with momentum is more viable. Therefore, the main consideration for this application note is between Adam and SGD with a large (.99) momentum value. As shown in the training results,

- Adam optimizer allows for a higher accuracy and lower loss. In Figure 11, the Adam optimizer can be seen adapting to the solution at a quicker rate than SGD in this figure.
- Adam also has the advantage of requiring less tuning because the default optimizer is effective in many instances. On the other hand, SGD requires more experimentation to find the optimal use of the optimizer.

Due to the performance of Adam over SGD, Adam is used as the optimizer of choice for this model.

Batch Size and Learning Rate

Table 3. Batch Size with Learning Rate 0.01

Batch size	Test Accuracy	Test Loss
16	77.2% - 77.6%	.6435
32	76.21%-77.12%	.6496 - .6644
64	76.39%-77.1%	.6591 - .6674
128	76.19% - 76.8%	.667 - .696
256	75.2% - 75.8%	.71 - .7386

Batch Size with Learning Rate 0.001

Table 4. Batch Size with Learning Rate 0.001

Batch size	Test accuracy	Test loss
16	77.3% - 78.5%	.59-.64
32	76.51%-77.8%	.6121 - .6694
64	76.6-77.1%	.6332 - .6721
128	75.2% - 75.7%	.6863 - .6919
256	74.1% - 74.8%	.7216 - .7421

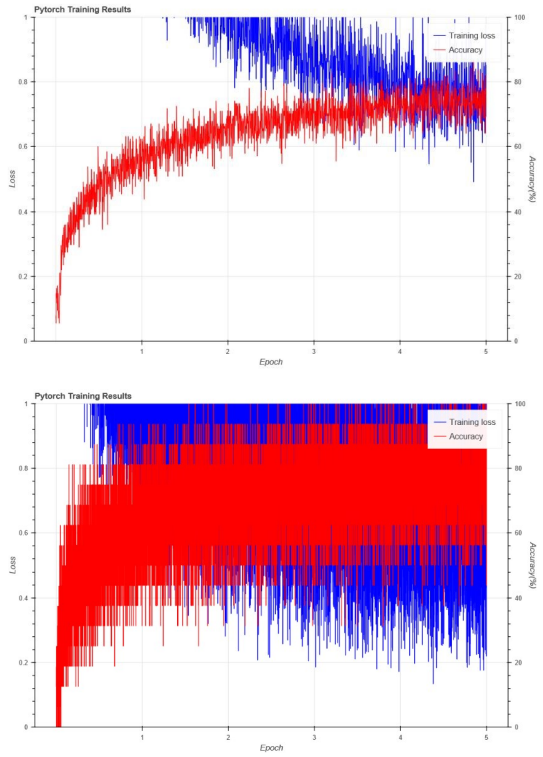

Figure 12. Training progression for batch sizes 16 and 128 respectively using a learning rate of .01.

NOTE

The smaller batch size yields the highest accuracy but has the noisiest training progress. This is due to how sensitive the batch accuracies are for small batches (training accuracy = correct/16 vs training accuracy = correct/128).

Conclusion:

Increasing the batch size value also decreases the total amount of iterations needed for 1 epoch.

Consider the example with a CIFAR10 training dataset of 50,000 images. The test used a batch size of 64 results in 782 steps (50,000/64 = 781.25) and a batch size of 256 results in 196 steps.

Since the model updates after each batch, smaller batches provide more generalized (but noisy) loss calculations and update the model more frequently. It can be inferred from the data that a batch size of 16 with a learning rate of 0.001 yields the most optimal results.

NOTE

Using a batch size of 16 results in 240 seconds per epoch. Depending on CPU memory, training time per epoch can decrease by increasing the batch size.

Table 5. Epoch experimentation results

Epoch	Test Accuracy	Test Loss
5	77.3%-78.5%	0.59 - 0.65
10	80.5%-81.95%	0.55 - 0.62
15	82.51% - 82.9%	0.5112 - 0.5176

Conclusion:

During epoch experimentation, the largest epoch yields the highest accuracy. The increase from 5 to 15 epochs increases the accuracy by ~4% but lengthens the training time from 20 to 60 minutes. Larger epoch values can be used but can face the risk of overfitting by learning the noise in images as important features.

Final Model:

- Optimizer: Adam
- Activation Function: ReLU
- Epoch: 15
- Batch Size: 16
- Learning Rate: 0.001
- Test Accuracy: 82.51% - 82.9%
- Test Loss: 0.5112 - 0.5176

Now we can train the model and export it in ONNX format so that Glow can compile it. Follow the steps below.

1. The training script **Train_Cifar.py** can be found at **\Cifar10_Pytorch\Training**. Running this script trains the model from scratch and exports it in ONNX format.
2. Users can modify the training parameters.
3. Code Snippet of `Train_Cifar.py` where training parameters can be set is shown below:

```
import time
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transform
import torchvision.datasets as dataset
from torch.utils.data.dataloader import DataLoader as Load
from bokeh.plotting import figure
from bokeh.io import show
from bokeh.models import ColumnDataSource, Range1d, LinearAxis
import numpy as np

model_save_path = '..\\Pytorch Models\\'
data_path = '..\\dataset\\'

# Training Parameters
epoch_Tot = 5
batch_size = 16
learning_rate = .001
```

4. Run the PyTorch training script in your Python editor/IDE.
5. You should see the output in the console begin by downloading the CIFAR10 dataset then displaying training characteristics after model updates like as shown below:

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ..\dataset
\cifar-10-python.tar.gz
HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))
Extracting ..\dataset\cifar-10-python.tar.gz to ..\dataset\
Epoch [1/5], Step [100/3125],  Loss: 1.8159, Accuracy: 31.25%
Epoch [1/5], Step [200/3125],  Loss: 1.6179, Accuracy: 37.50%
Epoch [1/5], Step [300/3125],  Loss: 1.6160, Accuracy: 43.75%
Epoch [1/5], Step [400/3125],  Loss: 1.5218, Accuracy: 43.75%
Epoch [1/5], Step [500/3125],  Loss: 1.6116, Accuracy: 31.25%
Epoch [1/5], Step [600/3125],  Loss: 1.4814, Accuracy: 50.00%
Epoch [1/5], Step [700/3125],  Loss: 1.4362, Accuracy: 50.00%
Epoch [1/5], Step [800/3125],  Loss: 2.0941, Accuracy: 37.50%
Epoch [1/5], Step [900/3125],  Loss: 1.3630, Accuracy: 50.00%
Epoch [1/5], Step [1000/3125],  Loss: 0.9267, Accuracy: 62.50%
Epoch [1/5], Step [1100/3125],  Loss: 1.3452, Accuracy: 50.00%
Epoch [1/5], Step [1200/3125],  Loss: 1.1693, Accuracy: 56.25%
Epoch [1/5], Step [1300/3125],  Loss: 1.5581, Accuracy: 50.00%
Epoch [1/5], Step [1400/3125],  Loss: 0.8525, Accuracy: 75.00%
Epoch [1/5], Step [1500/3125],  Loss: 1.9241, Accuracy: 56.25%
Epoch [1/5], Step [1600/3125],  Loss: 1.0069, Accuracy: 50.00%
```

Figure 13. Display of training characteristics on console

6. Once the ONNX model is in your **PyTorch Models** directory, the model is ready to be prepared for inferencing.

5 Exporting PyTorch model in ONNX format

PyTorch has built-in support for exporting models in ONNX format that can be used while building your model from scratch. It is recommended to specify input/output node names if you plan to reference these nodes later.

The following code snippet is used at the end of scripts to export the model in ONNX format. The code is expanded to clarify the variables being used.

Exporting PyTorch model in ONNX format

```
 # Save the model with input shape of the image for inferencing

Batch Size = 1   #Corresponds to the amount of images to inference on
in_channels = 3  #The input channel size based on image (RGB = 3, GrayScale = 1)
imgHeight = 32   # Input image dimensions
imgWidth = 32    # Input image dimensions
input_shape = torch.randn(Batch size, in_channels, imgHeight, imgWidth)
torch.onnx.export(model, input_shape, model_save_path + 'Model.onnx', input_names= ['input'],
output_names= ['output'])
```

NOTE

Be careful not to include certain special characters in node names to avoid problems when exporting the model to other programming languages.

Importing ONNX format models into PyTorch

PyTorch currently does not support importing models in ONNX format. So optionally the model can be exported in a `.pth` format to allow for model reuse in further training or analysis.

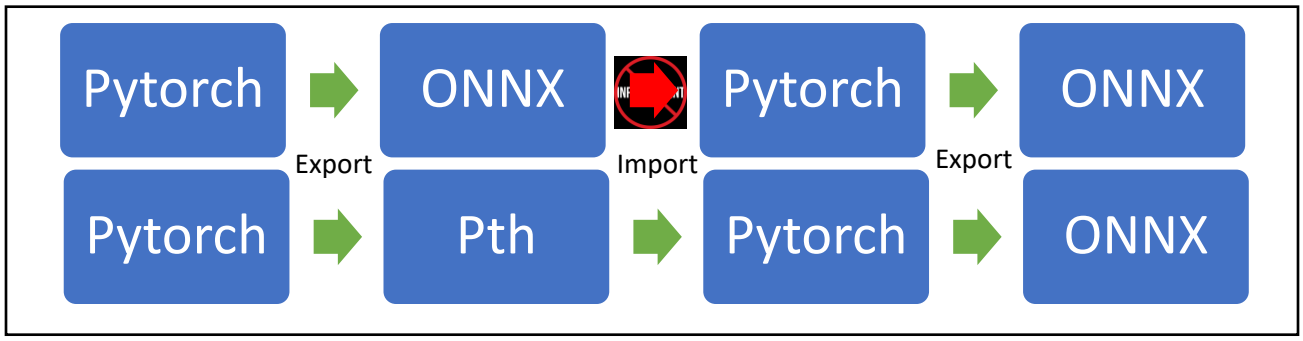

Figure 14. Visualizing PyTorch model conversions

If the model is exported as a `.pth` file, the model can be imported with the following code:

```
model = NeuralNet()
state_dict = torch.load(MODEL_LOAD + "Cifar_model.pth")
model.load_state_dict(state_dict)

#Training or analysis
#Export in ONNX format
```

The above code shows importing models in `.pth` format allows for model reuse in PyTorch.

6 Creating Glow files

To setup the environment for the below commands please install the “Glow Cifar10” directory (Here). These commands are run in the **GlowCifar10** directory in the command prompt. The figure below shows the folder structure of **GlowCifar10** directory:

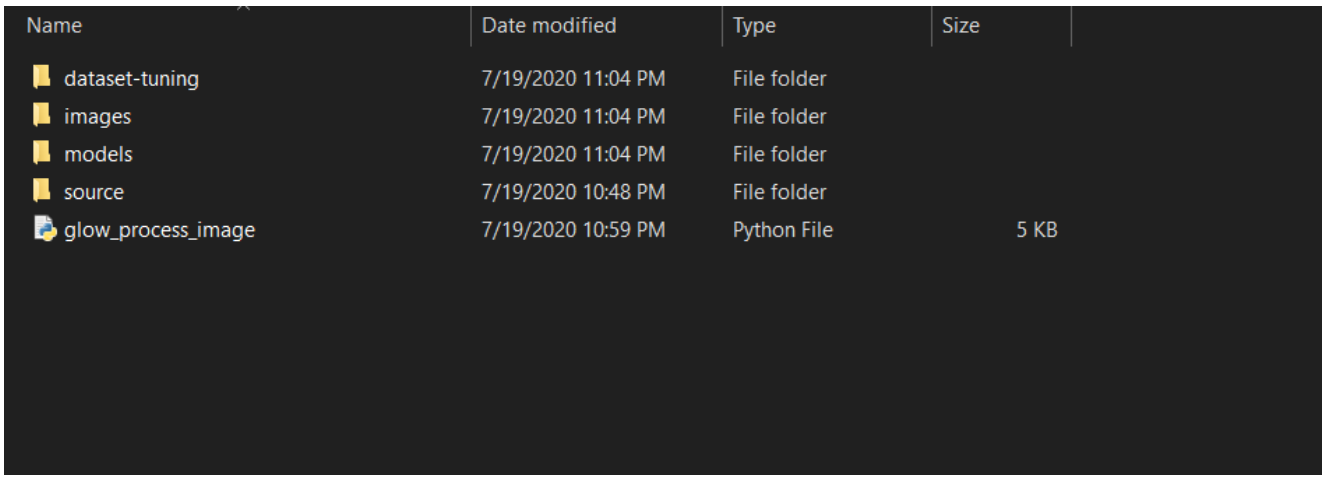

Figure 15. GlowCifar10 directory structure

- **`Dataset-tuning`**: Contains 30 images per class named “ClassName_XXXX” and a .csv containing the labels
- **`Images:`** Contains 10 folders (1 per classification) each with 100 32x32 images of the respective class
- **`Models`**: Contains PyTorch model in ONNX format
- **`Source`**: Output directory for Glow files
- **`Glow_process_image.py`**: Converts image to C array for inference

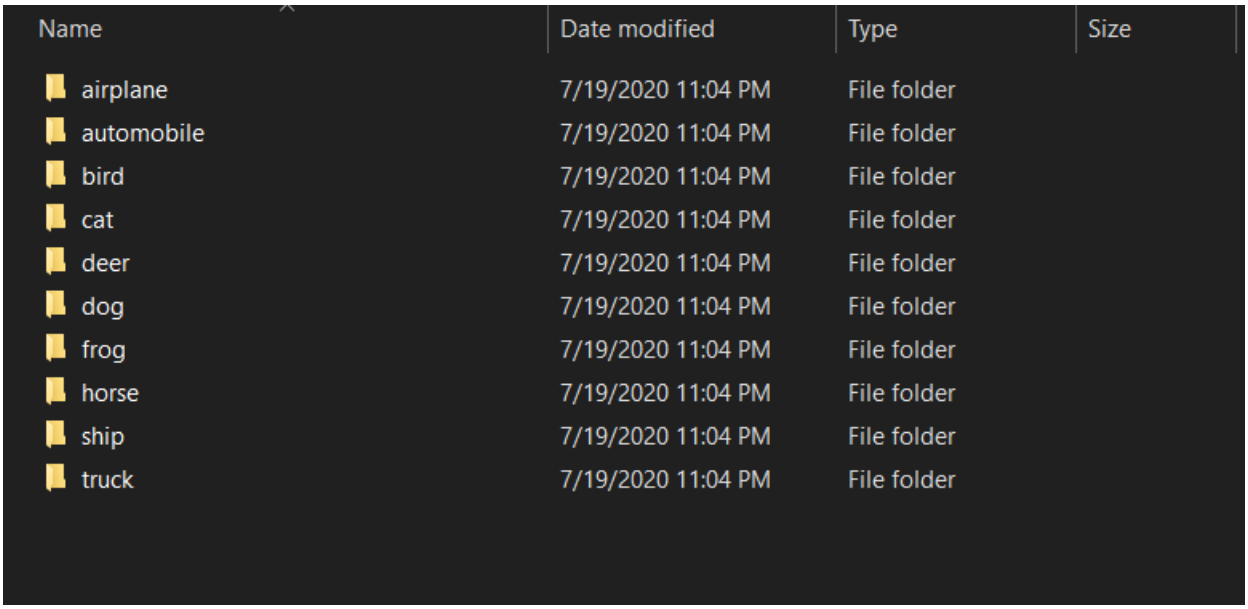

Figure 16. Folder structure of the Images directory

The model specific arguments are elaborated below. For more information on the hardware/quantization arguments and the functionality of the tools visit the “eIQ: Glow for RT1060 Lab”.

6.1 Creating a quantization profile

Quantization profile is generated with example images using **image-classifier**. This tool generates a **profile.yml** file that can be used to optimize quantization when compiling the model.

This should be one long continuous line:

```
image-classifier
images\airplane\0001.png
images\automobile\0001.png
images\bird\0001.png
images\cat\0001.png
images\deer\0001.png
images\dog\0001.png
images\frog\0001.png
images\horse\0001.png
images\ship\0001.png
images\truck\0001.png
-image-mode=neg1to1
-image-layout=NCHW
-image-channel-order=RGB
-model=models\Cifar.onnx
-model-input-name=input
-dump-profile=profile.yml
```

where the parameters are described below:

```
images\class\XXXX.png
```

- Images to perform the profiling on (1 image from each class).

```
image-mode=neg1to1
```

- The value for all PyTorch tensors are in range [0,1], this model normalizes images using .5 for each color channel which changes the range to [-1,1] (neg1to1).

```
-image-layout=NCHW
```

- Corresponds to Num, Channels, Height, Width. This is the image shape syntax used when exporting to ONNX format

```
-image-channel-order=RGB
```

- Specifies the channel order of images (Red-Green-Blue)

```
-model=models\Cifar.onnx
```

- Onnx model filename.

```
-model-input-name=input
```

- Name of input layer of the model. Specified when exporting PyTorch model to ONNX format.

```
-dump-profile=profile.yml
```

- Loads the quantization profile yielded in previous step.

```
-dump-tuned-profile=profile_tuned.yml
```

- File to upload tuned profiling to.

```
-target-accuracy=.9
```

- Stops tuning when accuracy has reached 90% from provided images.

6.2 Tuning quantization profile

The quantization profile can be further tuned to provide better accuracy. The model-tuner tool will take images from **\dataset-tuning** and the labeled dataset file in **\dataset-tuning\Labels.csv.** The optimized quantization profile is output as **profile_tuned.yml** to differentiate it from the previous quantization profile.

```
model-tuner
-dataset-file=dataset-tuning\Labels.csv
-dataset-path=dataset-tuning
-image-mode=neg1to1
-image-layout=NCHW
-image-channel-order=RGB
-model=models\Cifar.onnx
-model-input=input,float,[1,3,32,32]
-load-profile=profile.yml
-dump-tuned-profile=profile_tuned.yml
-backend=CPU
-quantization-precision=Int8
-quantization-schema=symmetric_with_power2_scale
-target-accuracy=.9
```

This should be one long continuous line:

Where:

- -dataset-file=dataset-tuning\Labels.csv

  is the CSV containing image file names and labels

- airplane_0001.png, 0,
- automobile_0001.png, 1,
- -dataset-path=dataset-tuning

  is the Path to directory with images specified in CSV file. 30 images per class used to tune profiling (300 total).
- -load-profile=profile.yml

  Loads the quantization profile yielded in previous step.
- -dump-tuned-profile=profile_tuned.yml: It is the file to upload tuned profiling to.
- -target-accuracy=.9: Stops tuning when accuracy has reached 90% from provided images

6.3 Creating source files

Using the profile created in the **model-tuner** tool, the **model-compiler** tool generates the compiled Glow executable.

This should be one long continuous line:

```
model-compiler
-model=models\Cifar.onnx
-model-input=input,float,[1,3,32,32]
-emit-bundle=source
-backend=CPU
-target=arm
-mcpu=cortex-m7
-float-abi=hard
-load-profile=profile_tuned.yml
-quantization-schema=symmetric_with_power2_scale
-quantization-precision-bias=Int8
-use-cmsis
-network-name=cifar
```

- -load-profile=profile_tuned.yml

Using the tuned profile this argument tells model-compiler to quantize the model

- -networkname=cifar

Name for generated source files. “cifar” will be referred to later in MCUXpresso steps.

6.4 Image processing for interference

To test the accuracy of the model on an embedded system the input image must be converted to a C array to be used for inference on the board. The **glow_process_image.py** script outputs the converted image to the same directory as the Glow executable files: **\GlowCifar10\source**

```
python glow_process_image.py
-image-path=images\horse\0001.png
-output-path=source\input_image_test.inc
-image-mode=neg1to1
-image-layout=NCHW
-image-channel-order=RGB
```

NOTE

The same image arguments as in previous steps are used.

7 Running Glow on RT1060

After the Glow files have been generated, the next step is to use them in the MCUXpresso IDE project using Glow and run it on the i.MXRT1060 board.

1. Open up MCUXpresso IDE and select a new workspace.
2. Download the current public SDK release that includes the glow packages needed. Visit the MCUXpresso SDK Builder, select the RT1060 board to build the SDK. Ensure to include **eIQ middleware** for the Glow packages.
3. Install the Glow RT1060 SDK into the “Installed SDKs” tab by dragging-and-dropping the **SDK_2.8.0_EVK-MIMXRT1060.zip** into the installed SDK window. The dialog box shown below is displayed. Click **OK** to continue the import.

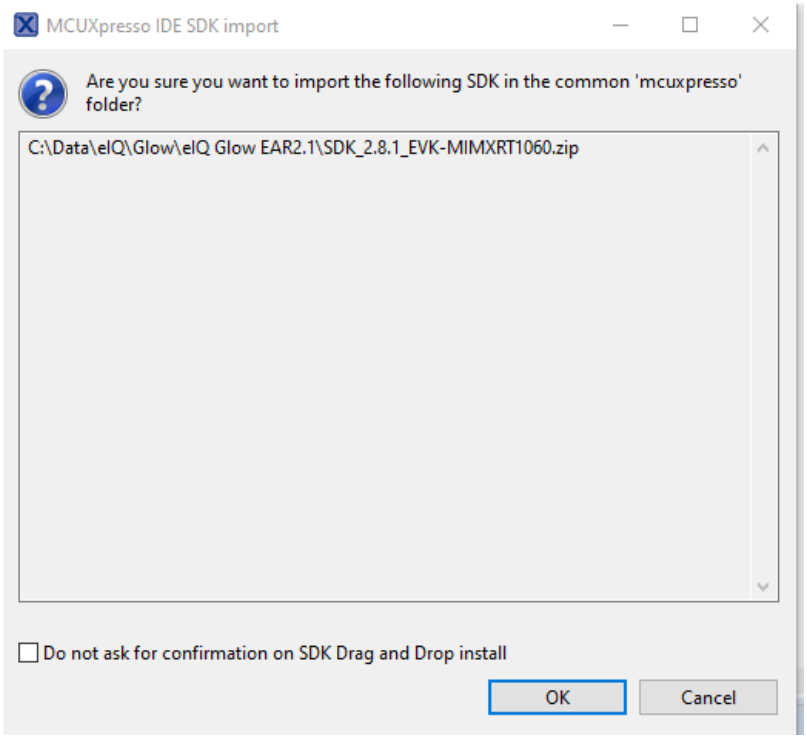

Figure 17. Importing the SDK

4. It will look like the following when complete:

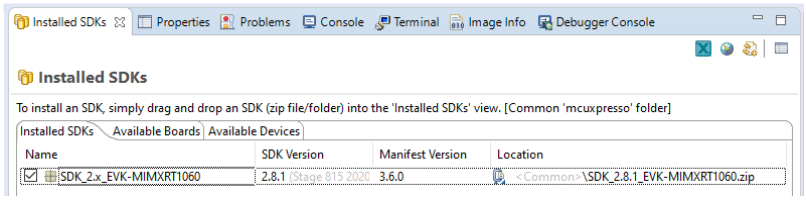

Figure 18. Installed SDKs

5. In the Quickstart Panel on the lower left corner, click **Import SDK example(s)...**.

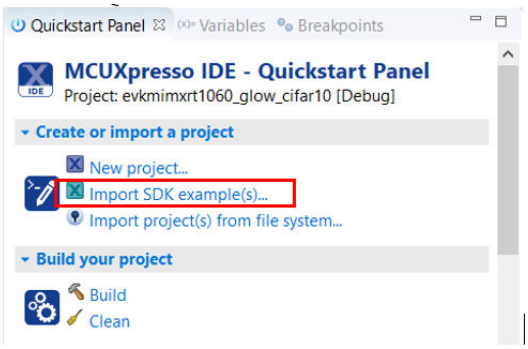

Figure 19. Importimg SDK example(s)

6. Select the **RT1060 board** and click **Next**.
7. Expand the **eiq_examples** category and select the **glow_cifar10** example. Click **Finish**.

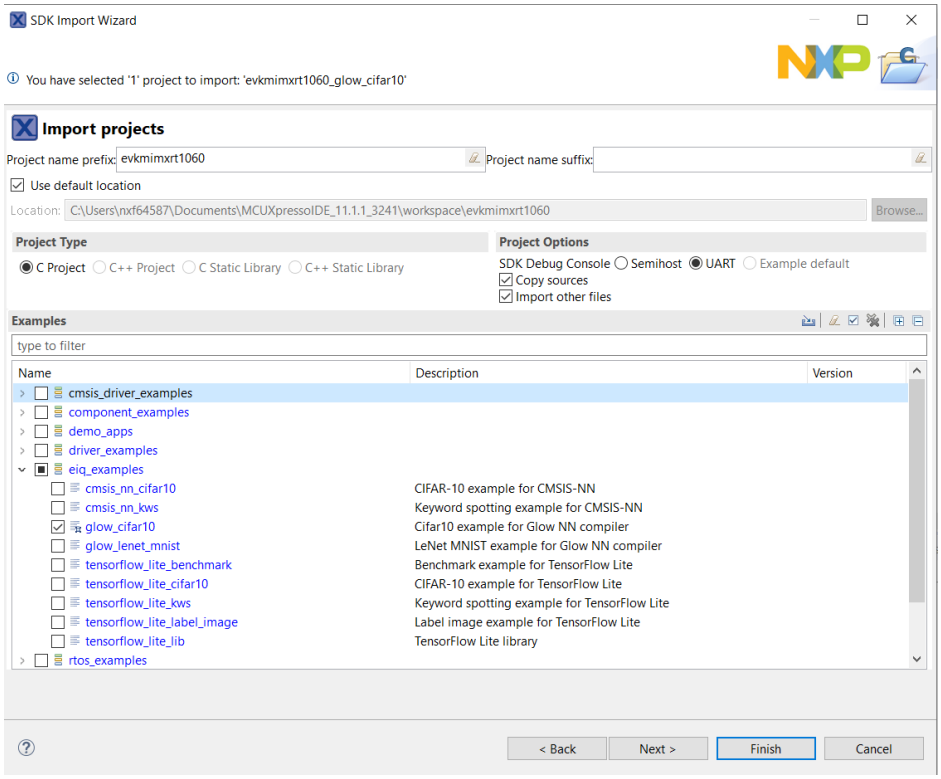

Figure 20. Selecting the example

8. Click the **evkmimxrt1060_glow_cifar10** project name and then in the menu bar, click on **Project->Properties**.

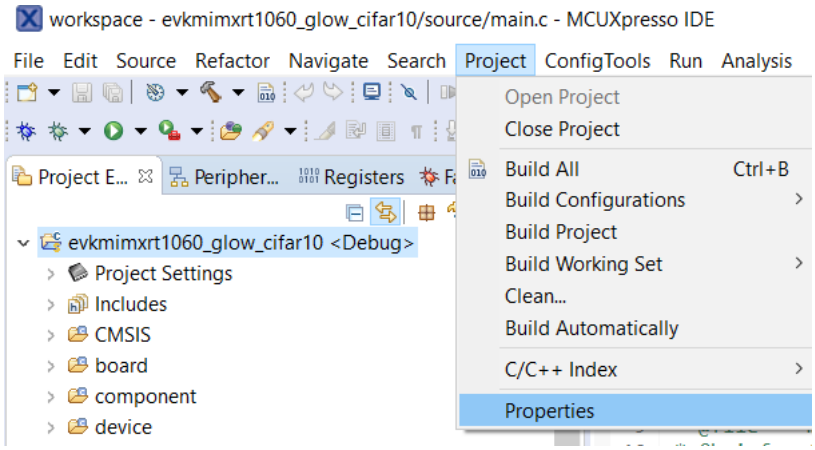

Figure 21. Editing Project properties

9. Then in the **Resources** category, in the **Location** field, click the **Open** icon. This displays a Windows Explorer view of that directory location.

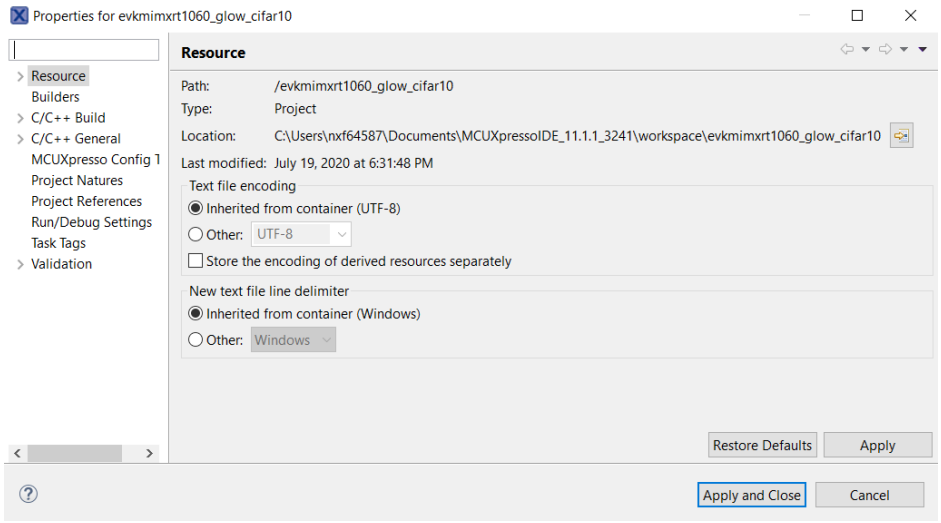

Figure 22. Viewing the directory

10. Find the **cifar.h**, **cifar.o**, **cifar.weights.txt and input_image_test.inc** files from the Glow **source** directory made in the **Creating Glow Files** section.Copy these files into the **evkmimxrt1060_glow_cifar10/source** directory. Remember that the "**cifar**" name comes from the "`-network-name`" argument, which you gave while running **model-compiler**.

    Input_image_test.inc comes from the python script to generate the data to do the inferencing on.

    It looks like the following when done:

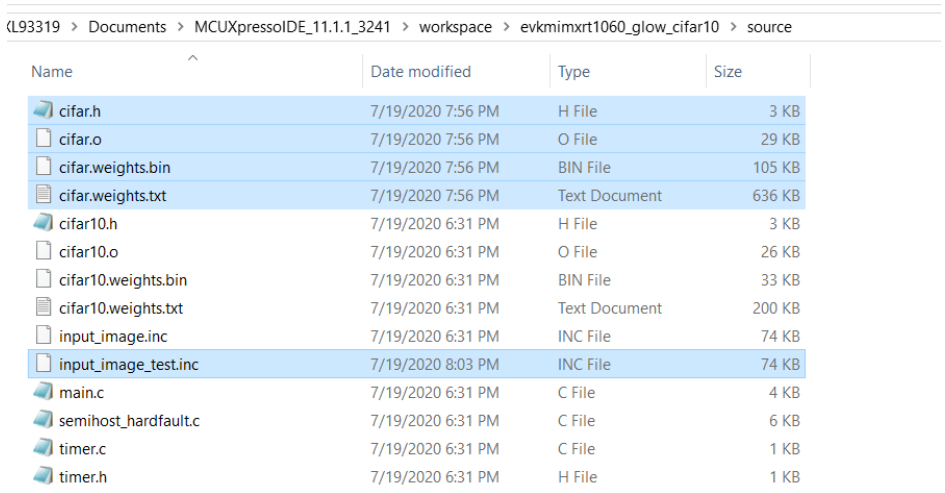

Figure 23. Viewing the selected source files

11. Now go back to MCUXpresso IDE.
12. Change the project to the Release settings that have highest compile optimization, by going to **Project->Build Configurations->Set Active->Release (Release build)**.

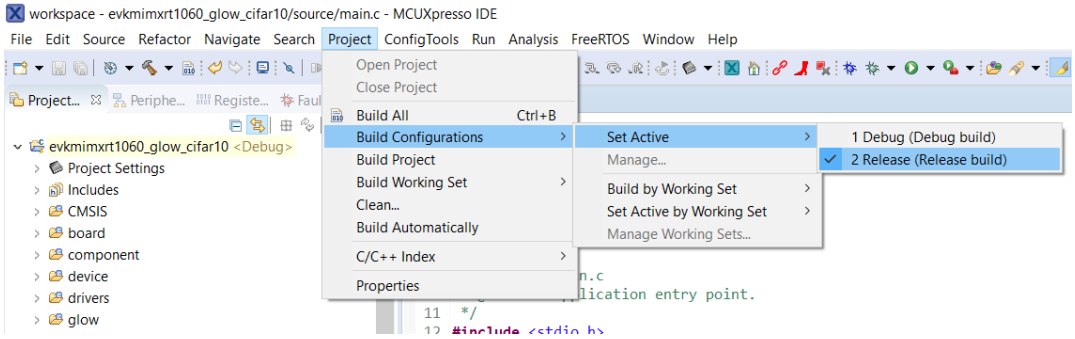

Figure 24. Changing the Release settings

13. Next, open up the Project Properties by right clicking on the project name again and select **Properties**.
14. In the **Properties** window, select the **C/C++ Build->Settings->MCU Linker->Miscellaneous** screen and double click the item in "**Other Objects**" to change the object file to the one that was just added:

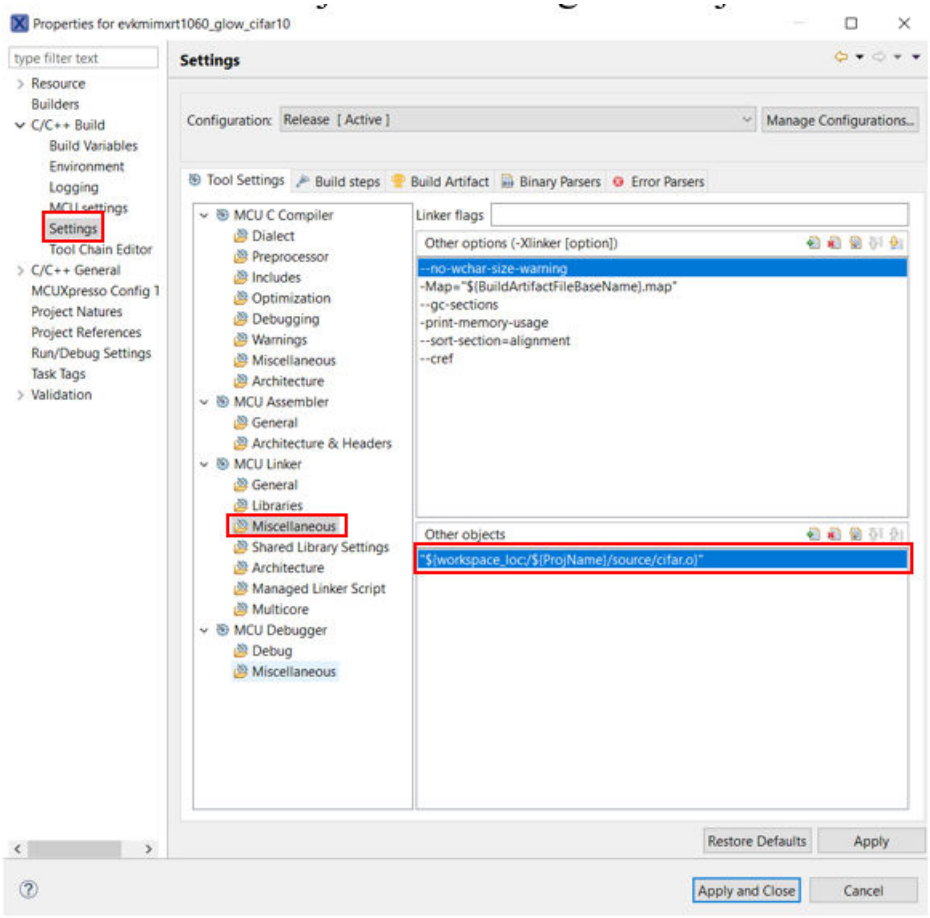

Figure 25. Updating Build settings

15. Click **Workspace**. Then, navigate to the source folder and select "**cifar.o**". Click **OK**.

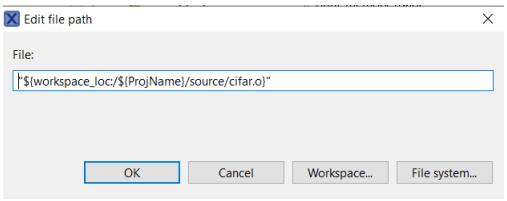

Figure 26. Selecting the source file

16. Then, click "**Apply and Close**" to close the **Properties** dialog box.
17. Next, you need to modify **main.c** to use the new file names. If you had chosen to use "`-network-name=cifar10`" then these changes are not necessary since the original name and the old name would match up in the source code. On the other hand, using the new network name "cifar" has the benefit that a user can walk through the structure of the code.
18. Everywhere in the **main.c** file that cifar10 is used, it should be changed to just "cifar" and use the new variable names created by the generated files. This includes:
    - Line 23 to include the generated header file "`cifar.h`".
    - Lines 26-27 for the new Glow variable names (Replace `CIFACIFARR10` with).
    - Line 28 to include the generated weights file: "`cifar.weights.txt`".
    - Lines 32-37 to use the new Glow variable names.

- Line 40 should set the inputAddr pointer to the network name plus the name of the model's input layer (**CIFAR_input** in this example). This name was specified during exporting the model in ONNX format.
- Line 43 should set the outputAddr pointer to the network name plus the name of the model's output layer (**CIFAR_output** in this example). This name was specified during exporting the model in ONNX format.
- Line 47 should be set to the input size of the model (Height*Width*Channels).
- Line 50 should be set to the number of classes of the model.
- Line 55 to include the generated test image "`input_image_test.inc`".
- Line 94, which starts the inference by calling "`cifar(constantWeight, mutableWeight, activations)`".

NOTE

The image shown below is for reference purpose only. The line numbers may vary slightly depending on the SDK version used.

```
21 // Bundle includes.
22 #include "glow_bundle_utils.h"
23 #include "cifar.h"
24
25 // Statically allocate memory for constant weights (model weights) and initialize.
26 GLOW_MEM_ALIGN(CIFAR_MEM_ALIGN)
27 uint8_t constantWeight[CIFAR_CONSTANT_MEM_SIZE] = {
28 #include "cifar.weights.txt"
29 };
30
31 // Statically allocate memory for mutable weights (model input/output data).
32 GLOW_MEM_ALIGN(CIFAR_MEM_ALIGN)
33 uint8_t mutableWeight[CIFAR_MUTABLE_MEM_SIZE];
34
35 // Statically allocate memory for activations (model intermediate results).
36 GLOW_MEM_ALIGN(CIFAR_MEM_ALIGN)
37 uint8_t activations[CIFAR_ACTIVATIONS_MEM_SIZE];
38
39 // Bundle input data absolute address.
40 uint8_t *inputAddr = GLOW_GET_ADDR(mutableWeight, CIFAR_input);
41
42 // Bundle output data absolute address.
43 uint8_t *outputAddr = GLOW_GET_ADDR(mutableWeight, CIFAR_output);
44
45 // ---------------------------- Application -----------------------------
46 // Cifar10 model input data size (bytes).
47 #define CIFAR_INPUT_SIZE    32*32*3*sizeof(float)
48
49 // Cifar10 model number of output classes.
50 #define CIFAR_OUTPUT_CLASS  10
51
52 // Allocate buffer for input data. This buffer contains the input image
53 // pre-processed and serialized as text to include here.
54 uint8_t imageData[CIFAR_INPUT_SIZE] = {
55 #include "input_image_test.inc"
56 };

92   // Perform inference and compute inference time.
93   start_time = get_time_in_us();
94   cifar(constantWeight, mutableWeight, activations);
95   stop_time = get_time_in_us();
96   duration_ms = (stop_time - start_time) / 1000;
```

Figure 27. Updating `main.c` file

19. To have the confidence output as a probability, we will implement a softmax function in the project to the output of the model. To implement the Softmax function, be sure to add the following to **main.c**:

- In the `#include` section near line 12, add **#include <math.h>**
- Above the main function, add the following code to implement the softmax function:

```
 static void softmax(float *input, size_t input_len) {
 float m = input[0];
 for (size_t i = 1; i < input_len; i++)
{
 if (input[i] > m) {
 m = input[i];
                 }
 }
 float sum = 0.0;
 for (size_t i = 0; i < input_len; i++)
{
 sum += expf(input[i] - m);
 }
float offset = m + logf(sum);
for (size_t i = 0; i < input_len; i++) {
input[i] = expf(input[i] - offset);
}
}
```

- Add the following function call before the model output is traversed for the correct prediction around line 122:

```
softmax(out_data, 10);
```

NOTE

The images used below clarify where to include the above code.

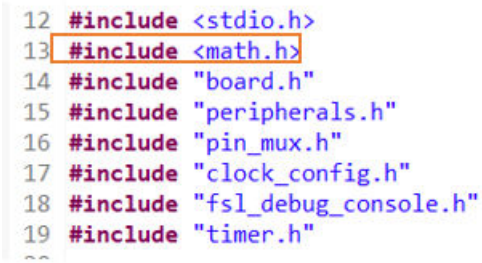

Figure 28. Adding the function call to the code

```
73 static void softmax(float *input, size_t input_len) {
74   assert(input);
75   // assert(input_len >= 0);  Not needed
76
77   float m = input[0];
78   for (size_t i = 1; i < input_len; i++) {
79     if (input[i] > m) {
80       m = input[i];
81     }
82   }
83
84   float sum = 0.0;
85   for (size_t i = 0; i < input_len; i++) {
86     sum += expf(input[i] - m);
87   }
88
89   float offset = m + logf(sum);
90   for (size_t i = 0; i < input_len; i++) {
91     input[i] = expf(input[i] - offset);
92   }
93 }
94
95 /*
96  * @brief   Application entry point.
97  */
98 int main(void) {
```

Figure 29. Softmax function

```
115   // Perform inference and compute inference time.
116   start_time = get_time_in_us();
117   cifar(constantWeight, mutableWeight, activations);
118   stop_time = get_time_in_us();
119   duration_ms = (stop_time - start_time) / 1000;
120
121   // Get classification top1 result and confidence.
122   float *out_data = (float*)(outputAddr);
123   float max_val = 0.0;
124   uint32_t max_idx = 0;
125   softmax(out_data, 10);
```

20. Now you can build the project. However, because new Glow files were copied into the project, you must do a clean first. Failing to do a clean could cause the newly imported weight data to become misaligned in memory, and cause accuracy errors during the inferencing. This is only required when new Glow files are copied into the project. Click on "**Clean**" in the Quickstart Panel first.

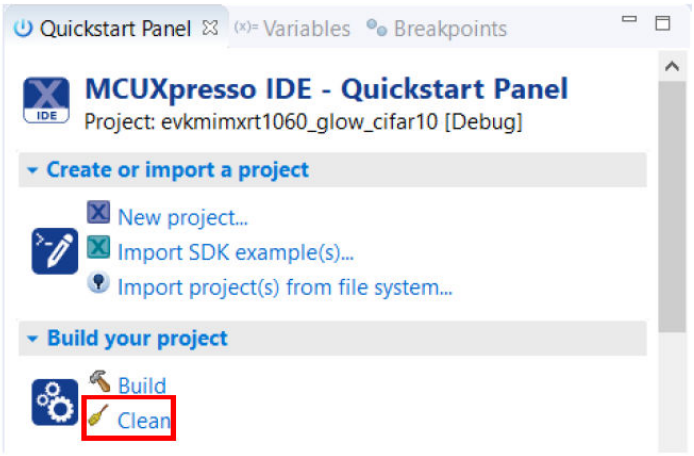

Figure 30. Cleaning the project

21. Build the project by clicking "**Build**" in the Quickstart Panel.

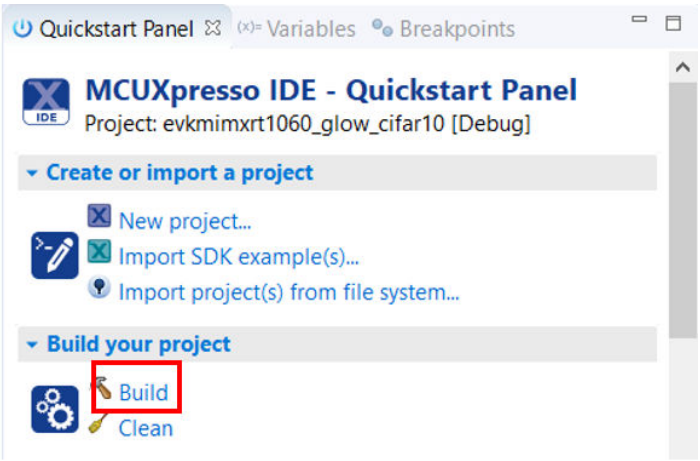

Figure 31. Building the project

22. Plug the micro-B USB cable into the board at J28.
23. Open TeraTerm or other terminal program, and connect to the COM port that the board enumerated as. Use 115200 baud, 1 stop bit, no parity.
24. Debug the project by clicking on "**Debug**" in the Quickstart Panel.

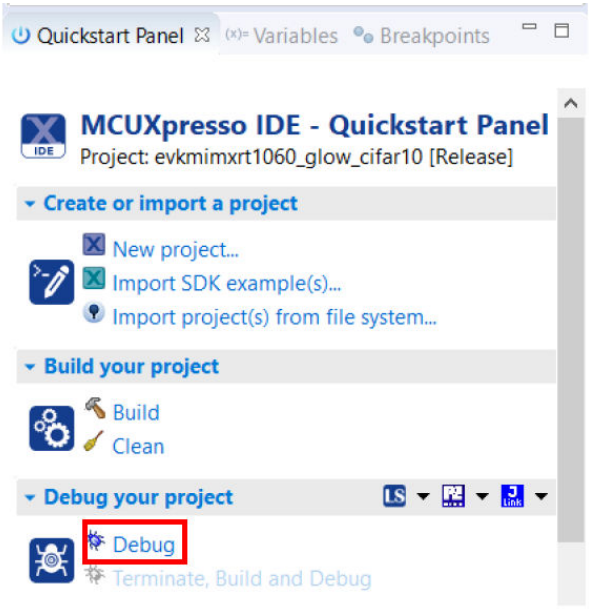

Figure 32. Selecting 'Debug'

25. Select the debug interface of your board (that is, CMSIS-DAP) and click **OK**.

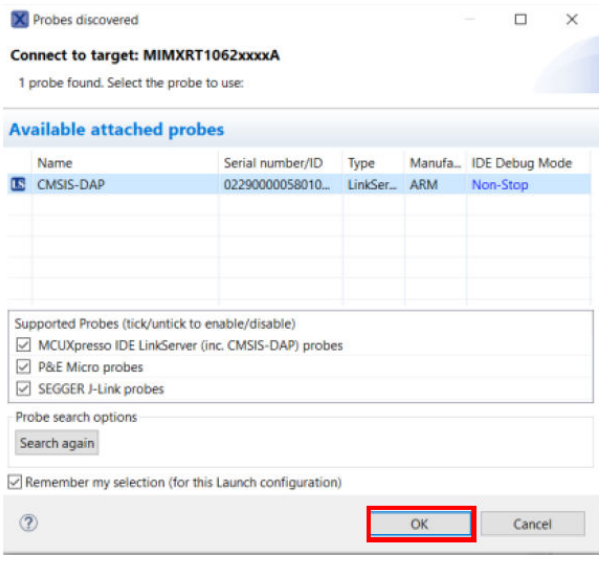

Figure 33. Selecting the debug interface

26. The debugger will download the firmware and open up the debug view. Click on the **Resume** button to start running the application.

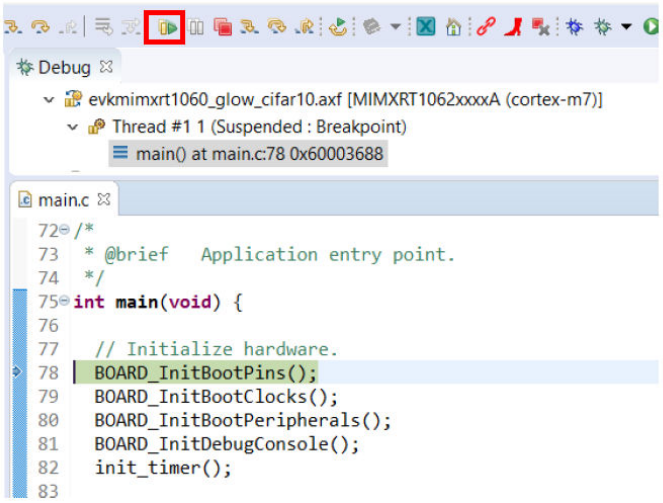

Figure 34. Running the application

27. You should see the output on the console:

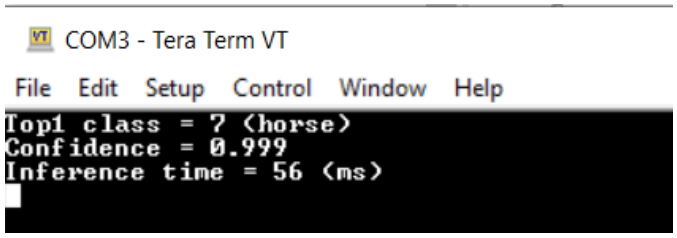

Figure 35. Console output

28. This is the memory footprint of the model on the RT1060 and the RT685 using the M33 chip and using the Hifi4 DSP. These values are taken from the MCUXpresso console after building the project.

Table 6. Memory footprint of the model

	Flash	SRAM	Confidence	Inference Time (ms)
RT1060	157460	211532	.999	56
RT685 (M33)	157336	225616	.999	205
RT685 (DSP)	831248	226008	.999	15

As seen in the preceding table, RT1060 and RT685 have similar memory footprints when using the M33 chip. When using the Hifi4 DSP, the inference time significantly decreases. However, the size of the DSP NN library is quite large as shown in the QSPI_Flash. There is also an extra 667 KB of RAM that is not reported by the compiler. This extra RAM consists of copying the DSP library to the board memory.

8 Retraining the model (optional)

As an optional step, users can train the model in **Train_TransferLearning.py** to train and export the model in .pth format for further reuse. The only difference between this training script and **Train_Cifar.py** is the model is exported in .pth format instead of ONNX format. Running **Train_TransferLearning.py** does the following:

- Train the model from scratch.

- Export the final model with .pth extension (allows user to re-upload the model file for further training or model visualization).

Running the **retrain.py** training script, the model can continue training where it left off. With the path of the saved model the user can run retrain.py to:

1. Extract model weights from the `.pth` file.
2. Continue training process. **Or** Retrain the model on a different dataset (To preserve model and convolutional dimensions the input image should have 3 channels (for example, RGB), 32x32 pixels and the dataset needs 10 separate classes).
3. Export the model in `.pth` format.

```
#Create model instance
model = NeuralNet()
#Load model weights from .pth file
state_dict = torch.load(model_load + "Cifar.pth")
# Load weights to model instance
model.load_state_dict(state_dict)

# Save the model and plot
input_shape = torch.randn(1,3,32,32)
torch.save(model.state_dict(), model_save_path + 'Cifar_retrain.pth')
```

Figure 36. Loading and saving model in retrain.py

NOTE

Be sure to specify the correct `.pth` model when loading. All the models are stored in the same directory. Hence, to ensure each model is saved and not overwritten, specify or differentiate the naming of the models between these scripts.

The last step is to run **Convert_Pth_to_Onnx.py** to convert your `.pth` model in a format readable by Glow.

```
# Paths for data and models
model_save_path = '..\\Pytorch Models\\'
data_path = '..\\dataset\\'
model_load = '..\\Pytorch Models\\'

#Create model instance
model = NeuralNet()
#Load model weights from .pth file
state_dict = torch.load(model_load + "Cifar_retrain.pth")
# Load weights to model instance
model.load_state_dict(state_dict)

# Save the model with input shape (Batch size, channels (RGB), img width, img height)
input_shape = torch.randn(1,3,32,32)
#Specify Model's Input and Output node names
torch.onnx.export(model, input_shape, model_save_path + 'Cifar_convert.onnx', input_names= ['input'], output_names= ['output'])
```

Figure 37. Conversion of model in Convert_Pth_to_Onnx.py

Once the model is in ONNX format, the steps described in Running Glow on RT1060 can be applied for deploying the model.

9 Conclusion (points to note)

In the initial stages, while porting the ideas and learning from other models, dropout is a common way to prevent overfitting by randomly choosing nodes to disconnect from the network. While this is true, keep in mind that it also increases the amount of training time needed for the model to find its most optimal solution. This was useful in simpler datasets such as the MNIST dataset (Handwritten Digit Recognition). The reason is that the model very quickly finds its solution and dropout makes sure further training will not cause overfitting.

Dropout is less effective on the CIFAR10 dataset due to the fixed amount of training time used for this application note. Theoretically, if you are training the model to convergence (much larger than 5 epochs), dropout would be more useful. The removal of dropout increased the model accuracy from ~74% to ~78% on CIFAR-10 validation set.

Using the optional Retraining Model section, the `.pth` extension was used to visualize the layers of the neural network. Identifying and isolating certain layers allowed to see the input image in different stages of the model. This was used to create the model visual in Model definition and visuals. If an input image is not supplied, the visualization of the convolutional layers show what filter is being applied to the image in different channels.

10 Revision history

The table below summarizes revisions to this document.

Table 7. Document revision history

Revision	Date	Topic cross-reference	Substantive change
0	06-Sep-2021	-	Initial release

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Limited warranty and liability — Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer’s applications and products. Customer’s responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer’s applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX,EMBRACE, GREENCHIP, HITAG, ICODE, JCOP, LIFE, VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, CodeWarrior, ColdFire, ColdFire+, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, Tower, TurboLink, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org. M, M Mobileye and other Mobileye trademarks or logos appearing herein are trademarks of Mobileye Vision Technologies Ltd. in the United States, the EU and/or other jurisdictions.

© NXP B.V. 2021. **All rights reserved.**

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 06-Sep-2021
Document identifier: AN13331