ECE 196 Project Report

Geffen Cooper

Winter Quarter - March 9th, 2021

Table of contents

Project Overview

This section explains the purpose and goals of this project that were outlined at the start of the quarter.

Project Topic: Application of edge machine learning

This project seeks to apply machine learning at the edge using Maxim Integrated's MAX78000 microcontroller which utilizes a CNN accelerator for inference. The task explored is real-time vision-based detection of facial features for applications such as driver drowsiness detection.

Overarching Question:

What level of vision-based classification and detection tasks can be achieved in an embedded system such as the MAX78000 considering: (For exact specs see Technical Details section)

1. Reduced memory for network storage and intermediate data processing
2. Quantization of network parameters to 8-bit values
3. Constraints on network architecture (e.g., limited kernel size)

Goals:

1. Determine if an image contains a face assuming that the majority of the pixels in the current frame contain the face (binary classification)

2. Given an image of a face, localize the face with a bounding box and specify the location of facial landmarks or identify the regions of prominent facial features (eyes, mouth, nose)

3. Track prominent facial features across frames for higher-level tasks such as facial pose estimation and driver monitoring

Conditions:

Due to time constraints, the project will currently focus on detecting facial features in the simple case when one face is present in the majority of the frame and there is good lighting.

Tasks Completed: (For additional details, see the Project Log section)

	Task	**Date**
1.	Classify handwritten digits from real-time camera data	01/27/2021
2.	Determine if a single face is present in the image (binary classification)	2/07/2021
3.	Localize a detected face in an image using a bounding box (this was done on a pc using a network architecture that fits the constraints of the MAX78000) Synthesize a working model to the MAX78000	2/14/2021 3/1/2021
4.	Localize Eyes from a detected face (runs on the MAX78000)	3/2/2021

Report Structure

The next three sections of the report explain the stages of development for this project. These sections provide high level context as well as in depth details. Following these sections are a conclusion and extra information sections including technical details about the MAX78000, a project log, and a credits section.

The MAX78000 and MNIST

This section discusses the first stage of the project which involved setting up the Maxim Integrated tools and implementing a handwritten digit classifier based on MNIST using real-time camera data.

Development with the MAX78000

One challenge of programming on a bare-metal microcontroller such as the MAX78000 is that there is very limited memory and no operating system. Unlike more powerful embedded systems such as the Raspberry Pi it cannot utilize libraries such as OpenCV or any Linux-based tools. Fortunately, Maxim Integrated does provide drivers for interacting with peripherals such as the LCD.

Accordingly, all the development and training of deep learning models is done off the board on a more powerful computer with Maxim Integrated's custom set of software tools. Development of a new model is divided into two steps: (specific details can be found on their GitHub Training Repo)

1. **Training**
    a. Maxim Integrated provides a set of customized PyTorch classes that are subclasses of the core `torch.nn.Module`. These classes add metadata needed to create a model that can run on the MAX78000 CNN accelerator. The API and workflow is very similar to standard PyTorch.

2. **Quantization and Synthesis**
    a. Once a model is trained it needs to be quantized from 32-bit floating point to 8-bit fixed point (Q7 format) using the provided quantization tool.
    b. The quantized model can then be synthesized into C code using the provided synthesis tool which also auto-generates helper functions to load the network input and output data.

Implementing a Handwritten Digit Classifier

In order to learn the development workflow of the Maxim Integrated tools, I replicated the provided MNIST example for classifying handwritten digits. This involved training a 5 layer model (4 convolution layers and 1 fully connected layer) on the MNIST dataset, quantizing the model, and synthesizing it.

However, the provided MNIST example does not use camera data. It uses sample images from the test dataset which are defined in a header file. Accordingly, I expanded the example to use live data from the onboard camera module. While the Maxim SDK (software development kit) provides drivers for interacting with the camera, multiple preprocessing steps are required to transform the camera data into a format that the CNN model expects as input. These steps are described in detail on the next page.

1. The first step is to initialize the camera. This involves setting the image dimensions to 56x56 pixels and to operate in the YUV422 pixel format rather than RGB565. The CNN model is trained on 28x28 grayscale images so the input data must match the format during inference (*note: the image is captured as 56x56 then decimated to 28x28 because 28x28 is hard to see on the LCD*). Two pixels from each format are shown in **Figure 1**; each pixel is 16 bits.

2. The second step is to request a frame from the camera and extract the luminance value (grayscale, Y byte shown in **Figure 1**) using a bitwise AND operation.

3. Once the luminance byte is extracted from a pixel, **binary thresholding** and **negation** operations are applied. As stated earlier, the input into the network must be the same form as the data in the training dataset. **Figure 2** shows the camera output before and after these operations. As shown, the network incorrectly interprets a '0' as a '5' when the raw grayscale values are fed to the network.

4. The final step is to convert the luminance byte into a signed 8 bit value (this is a requirement of the MAX78000 CNN accelerator) and store it into an output buffer that will get loaded into the network as input. The overall control flow is shown in **Figure 3**. Additional details of the input format can be found in the GitHub [Training Repo]().

The code for this example can be found here:
[MAX78000 MNIST Example]()

A video demo of this example can be found here:
[MAX78000 MNIST Demo]()

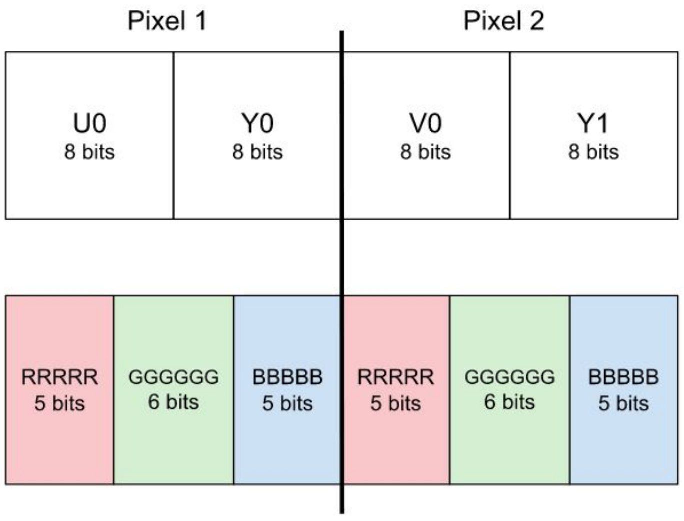


Figure 1: Pixel Formats (top is **YUV422**, bottom is **RGB565**)

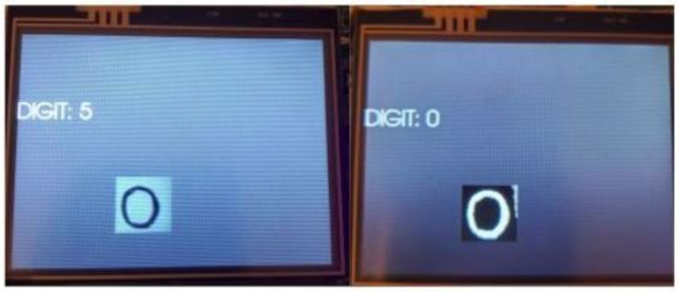


Figure 2: Binary Thresholding and Negation

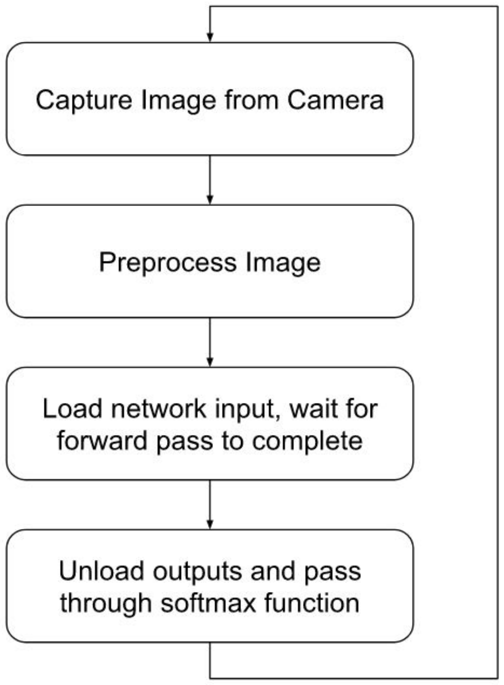

Capture Image from Camera
Preprocess Image
Load network input, wait for forward pass to complete
Unload outputs and pass through softmax function

Figure 3: Program Control Flow

Building the Binary Classifier Base Network

This section discusses the second stage of the project which was building the binary classifier network for determining whether a face is in the image. This 'base' network is later used as a feature extractor.

Training Constraints

A common practice when developing a CNN is to use pretrained networks as the backbone for your model and only train the last layers (or continue training all the layers) for a specific task (reference for transfer learning). The reasoning for this is that training a state-of-the-art model from scratch is very time consuming and requires large compute resources. In addition, assembling a large high quality dataset may not be feasible.

With the Maxim Integrated tools it is currently not possible to use pretrained networks for a few reasons.

- While the Maxim Integrated tools use a form of PyTorch, the network must eventually be synthesized into C code that can run on the MAX78000. This synthesis step requires extra metadata that is not part of standard PyTorch.
- Even if this metadata could be added to a pretrained model, most models are far too large to fit within the weight memory of the MAX78000 which is ~430 KB.
- The Maxim Integrated synthesis tool requires a very precise configuration file that specifies which processors to use for each layer in the CNN and which memory location to output the results after a forward pass through a given layer (specifics can be found in the Technical Details section). For a complex network architecture this is very difficult.

The remainder of this section describes the steps taken to train a CNN from scratch.

Assembling a Dataset

The first step of building the binary classifier was to assemble a dataset. Since there are only two classes, 'face' and 'no face', the training set only needs images that contain faces and images of anything else. As specified in the project overview, this network is focused on the specific case where an image contains a single prominent face with good lighting. The datasets used for training the binary classifier were downloaded from Kaggle and are summarized in **Table 1** below.

Dataset	Description	Size	Sample Image
FFHQ	Faces	70K	
Caltech 256	Random Objects (Classes with people were manually removed)	30K	

MIT Indoor Scenes	Indoor Scenes	15K	
Stanford STL-10	Objects	25K taken out of 100K	

Table 1: Dataset Summary

Network Architecture

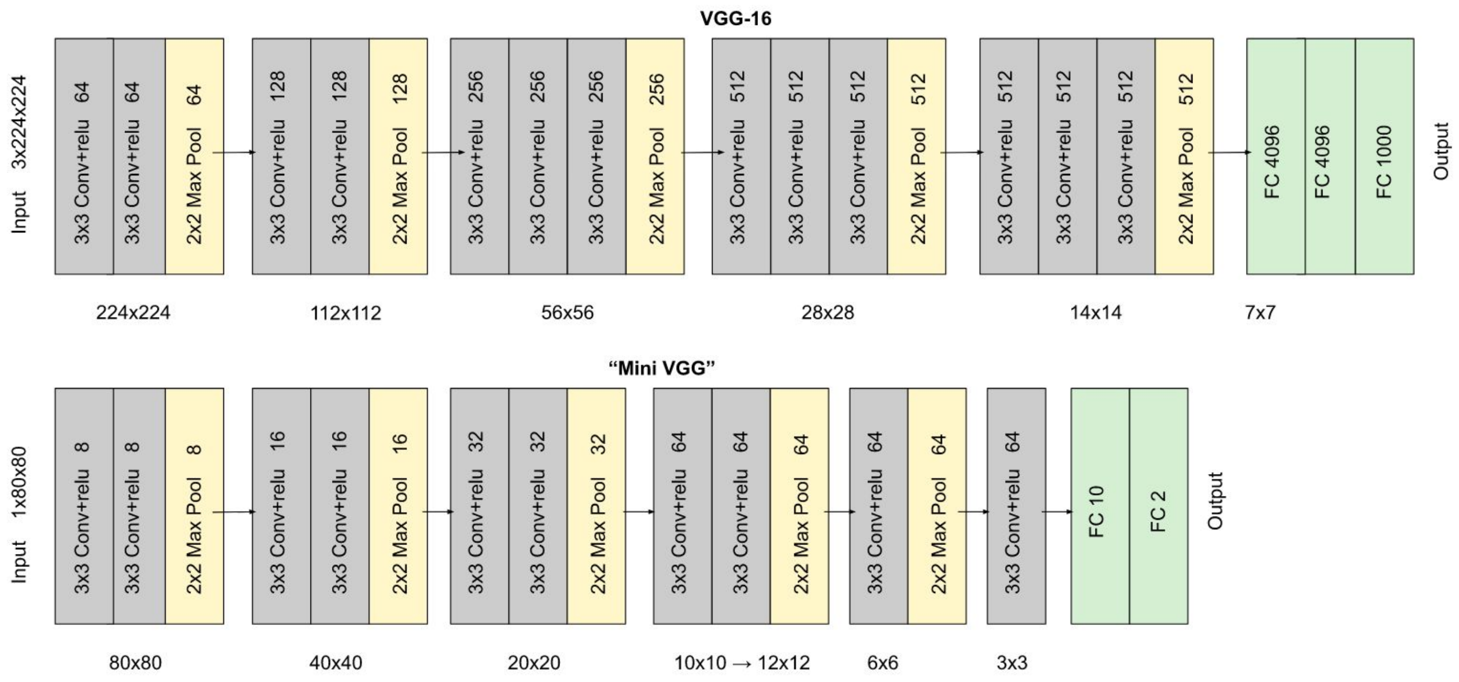


Figure 4: Network Architectures

The architecture I used for the binary classifier is based off of **VGG-16**, a well known network architecture for image classification. The two architectures are shown in **Figure 4**.
The characteristics I tried to emulate with "Mini VGG" are

- Doubling the channel dimension in each 'block'
- Putting multiple convolution layers after each pooling layer
- Using multiple fully connected layers at the output

While the "Mini VGG" architecture shown does not exactly follow "VGG-16", I heuristically observed that this architecture works well for classifying faces with a greater than 98% accuracy on the test set. One thing to note is that the MAX78000 CNN accelerator cannot flatten a 2D image with more than 1,024 parameters into a fully connected layer. For this reason I kept the channel dimension at 64x64; a

128x3x3 layer contains 1,152 parameters which is larger than the 1,024 limit. While 113x3x3 fits the limit I did not observe any substantial improvement in the accuracy.

Data Augmentation

One challenge that arose with the FFHQ dataset is that most of the images are centered with similar sized faces. This resulted in poor results when testing the network on live camera data because a face would only be detected if I placed my head in the center close to the camera.

As a result I introduced scaling and translation data augmentations. Some samples are shown in **Figure 5**. Note that the input into the network is 80x80 grayscale. The MAX78000 CNN accelerator cannot process input images larger than 90x91 without the use of *streaming*. Streaming is a more complicated procedure in which the CNN accelerator only needs to load the first *n* rows of the prior layer's output before beginning to process it (more details are in the GitHub Training Repo).

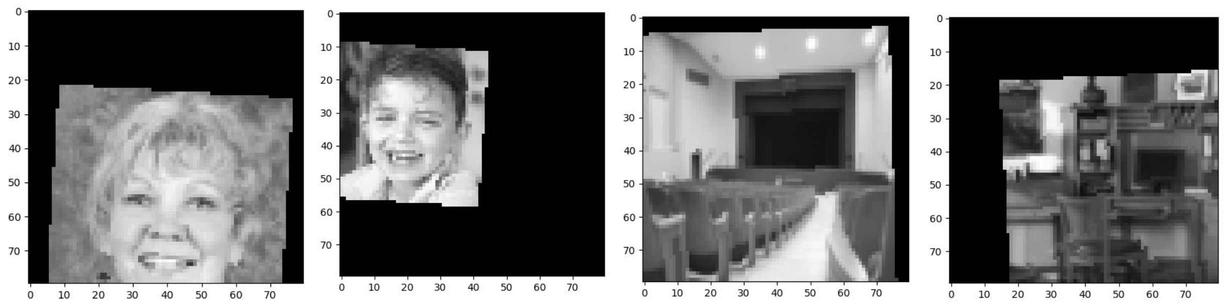


Figure 5: Data Augmentations

The code for the binary classifier can be found here: Binary Classifier Code
A video demo of the binary classifier can be found here: Binary Classifier Demo

Binary Classifier as a Feature Extractor

This section discusses the final stage of the project which uses the binary classifier as a feature extractor in order to localize a face in an image using a bounding box or locate the eyes using points.

Interpreting the Binary Classifier's Output

As mentioned in the last section, a common practice when building a CNN is to use a pretrained model as a backbone. Rather than train a new CNN from scratch to detect bounding boxes or eyes, I used the convolution layers of the binary classifier as a feature extractor. There are a couple reasons for this approach:

1. Sparse availability of images with a **single** face and a corresponding bounding box
2. The binary classifier network weights are already trained on a large dataset of faces

This methodology assumes that the binary classifier is looking for identifiable facial features which may or may not be true. This type of approach is generally referred to as transfer learning. To get a better sense of what inputs the network responds to, the Maxim Integrated training tools use an external tool called **SHAP** (SHapley Additive exPlanations) which is "a game theoretic approach to explain the output of any machine learning model." In general, the tool can help identify how much each portion of an input image is contributing to the output class. The output of the SHAP tool is shown in **Figure 6** where red pixels increase the model's output and blue pixels decrease the model's output.

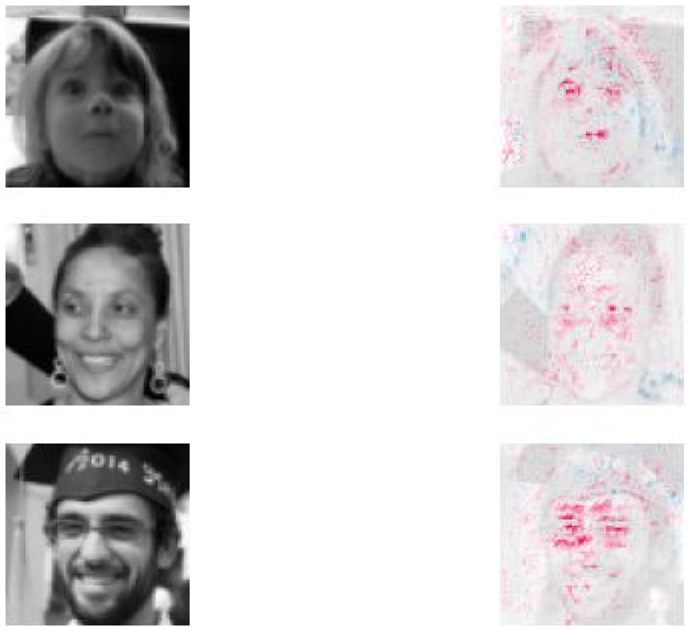


Figure 6: SHAP output

In the examples shown in **Figure 6** it seems that the binary classifier is in fact 'activated' in some sense by prominent facial features such as the eyes. Similar results are observed in several other examples.

Extending the Binary Classifier to Detect Bounding Boxes

Based on the heuristic results above, I used the convolution layers of the binary classifier as a feature extractor. Accordingly, I added an additional fully connected layer for detecting bounding boxes in parallel to the binary classifier output. In retrospect this approach created some difficulties when trying to synthesize the network (an alternative approach is discussed in the Conclusion). The extended architecture is shown in **Figure 7** where the bounding box outputs are the x-y coordinate of the top left corner and the width and height of the bounding box.

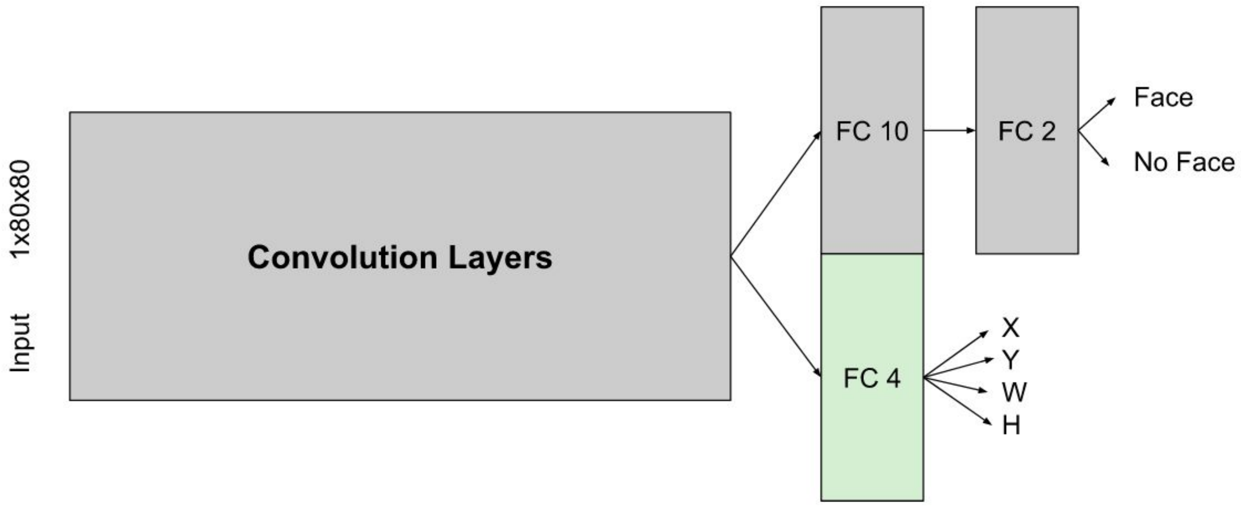


Figure 7: Bounding Box Extension

In this network the gray layers are *frozen* meaning that the weights for those layers are not being updated during backpropagation. Only the weights of the green fully connected layer for the bounding box coordinate outputs are changing. In addition it is important to mention that bounding box detection is a **regression** task rather than a **classification** task. The main difference between these tasks is the loss function. In a classification task there are a finite number of discrete output classes whereas with the regression task the output represents a set of continuous values.

A common choice of loss function for bounding box regression tasks is the L1 or L2 loss.

The L1 loss for a given sample is defined as: $L1 = \left|(Y_i - \widehat{Y}_i)\right|$

The L2 loss for a given sample is defined as: $L2 = \sqrt{\sum_{i=1}^{n} (Y_i - \widehat{Y}_i)^2}$

A very popular evaluation metric for bounding box predication is IoU (intersection over union) as shown in **Figure 8**. However, a paper from a couple years ago (GIoU Paper) shows that minimizing the L2 or L1 loss does not necessarily minimize the IoU. An image from the paper shown in **Figure 9** shows how a set of bounding boxes defined by their top left and bottom right corners can have the same L2 loss value but different IoU values. In the image one corner is fixed while the other corner can only move by a fixed amount around a circle. This results in the same L2 loss but different IoU values.

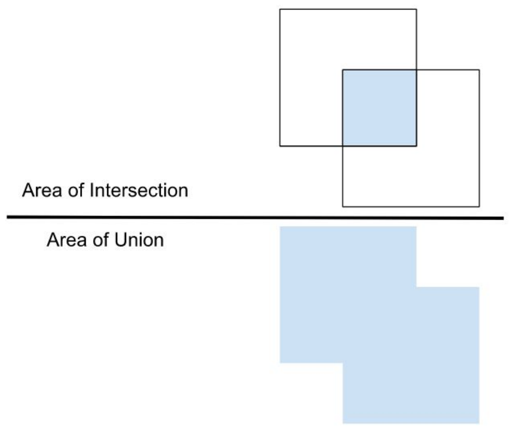


Figure 8: IoU

The proposed loss function in this paper is *1-GIoU*. GIoU (generalized IoU) is defined at the top of **Figure 9**. In this expression A and B are the bounding boxes and C is the smallest box that encloses A and B. This term accounts for the case when the two boxes have no overlap.

$$GIoU = IoU - \frac{|C\backslash(A \cup B)|}{|C|}$$

The loss function I used for training directly uses the IoU value and is defined as *L = - ln(IoU)*. I added a small term to the area in case the two boxes do not overlap. Overall this loss function resulted in decent results which can be seen at the end of this section.

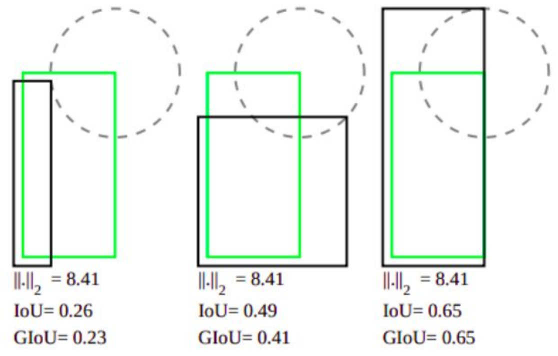


Figure 9: L2 loss vs IoU loss vs GIoU loss

Dataset and Augmentation

The dataset I used for training the bounding box output layer was a subset of the WIDER FACE dataset. This dataset contains many images of faces with labeled bounding boxes (x,y,w,h). However, the majority of these images contain multiple faces so I had to write a script that would only collect the images which were labeled to have a single face. In addition to this dataset I manually labeled a couple thousand images from the FFHQ dataset with bounding boxes.

However, as mentioned in the last section, a major problem was the low variance of these images. Most of the faces were of similar size and location causing poor results outside a certain region of the image frame. Accordingly I created a function to perform scaling and translation augmentations as shown in **Figure 10**. One concern is the black area created by the augmentation may cause the CNN model to learn unwanted features such as the image borders. These augmentations had positive and negative effects on the result. Before applying these augmentations, the model struggled to adjust the size of the bounding box if the present face became smaller. After applying these augmentations the model was able to resize the bounding box accordingly. However, the overall result was not very robust. In some regions of the image the bounding box was predicted accurately whereas in other parts of the image it was completely wrong. This may be due to lighting or objects in the background.

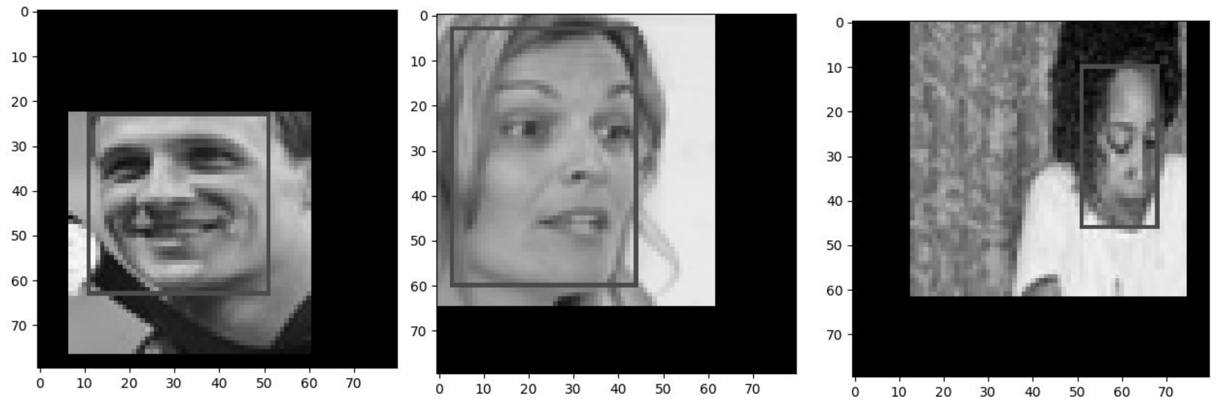


Figure 10: Bounding box augmentations

The code for the bounding box detector can be found here: Bounding Box Code
A video demo of the bounding box detector can be found here: Bounding Box Demo

Localizing Eyes

The final attempted task was to localize eyes from the image. Originally the approach was to utilize the bounding box information in some way to help localize the eyes but this caused the network architecture to become too complicated as additional branching or cascading would be necessary. Instead, I retrained the fully connected layer for the eye coordinates rather than the bounding box coordinates using the exact same architecture. One of the main challenges with eye detection is that I only had 1K images of manually labeled data. For this model, the L2 loss was used rather than the IoU-based loss function.

The code for the eye locator can be found here: Eye Locator Code
A video demo of the eye locator can be found here: Eye locator Demo

Conclusion

This section explains the results of the project and discusses some of the areas which need improvement and should be pursued further.

Overall Results

From the original project outline, goals 1 and 2 (classifying a face and localizing a face and facial features in an image under ideal conditions) were achieved to a certain extent. Given that this project was mainly an experimental opportunity to explore the capabilities of the MAX78000, there are not any quantifiable results to report. However, the observed heuristic results show that it is possible to accomplish reasonably complex vision based tasks such as basic object detection using this board.

Improvements and Further Work

While the current model serves as a valid proof of concept it is still flawed in a few ways. The main improvements that could be made and further areas to pursue are listed below

- When training the bounding box layers, the binary classifier output was not taken into account. For example, if a face was not detected (even though all training samples contain a face) the sample was still used to calculate the error and backpropagate through the fully connected layer. Ideally the binary classifier output should be utilized when training the bounding box layers.
- The structure of the network could be simplified rather than have two output layers in parallel. It is necessary to train for classification and regression separately but all the outputs could be in a single layer. During each training phase, the error for the unused outputs should be set to zero. For example, when training the classifier, the error for the bounding box should be zero.
- The loss function for bounding box regression should be experimented with further. In addition to the paper discussed in the last section, there are other variants of IoU loss that add additional terms such as distance from the center of the two bounding boxes (Paper on Distance IoU loss).
- The amount of training data was relatively small. In order to create a robust face detector, far more training data is required under various lighting conditions.
- In the future, a different network architecture may need to be explored. While the current architecture did classify images well, the strategy of ‘transfer learning’ may not be the best option given the constraints of the system. In addition, only ~33% of the total weight memory was utilized so there is room to expand the network or cascade networks if necessary
- In order to detect small facial features, higher resolution input images may need to be used. This will need to implement streaming as mentioned in the last section.

Technical Details

This section explains extra technical details including a brief explanation of the how the MAX78000 works and links to the code repo

Links to the relevant Maxim Integrated documentation are below:
Main GitHub Repo
Maxim SDK (software development kit)
Synthesis Repo (technical details can be found here in the README)
Training Repo (technical details can be found here in the README)

Technical Specifications

Some of the main technical specifications are listed below and are taken directly from Maxim Integrated documentation. For the full list, explore the above repos.

Conv2d layers:

- Kernel sizes must be 1×1 or 3×3.
- Padding can be 0, 1, or 2. Padding always uses zeros.

- Stride is fixed to [1, 1].

Activations

The supported activation functions are ReLU and Abs, and a limited subset of Linear.

Pooling:

- Both max pooling and average pooling are available, with or without convolution.
- Pooling does not support padding.
- Pooling strides can be 1 through 16. For 2D pooling, the stride is the same for both dimensions.

Dimensions

- The number of input channels must not exceed 1024 per layer.
- The number of output channels must not exceed 1024 per layer.
- The number of layers must not exceed 32 (where pooling and element-wise operations do not add to the count when preceding a convolution).
- The maximum dimension (number of rows or columns) for input or output data is 1023.
- When using data greater than 90×91, streaming mode must be used.
- The weight memory supports up to 768 * 64 3×3 Q7 kernels

How Data Flows through the MAX78000 CNN Accelerator

The MAX78000 CNN accelerator contains 64 processors that can work in parallel across the channel dimension. An example of the first 8 processors is shown in **Figure 11**.

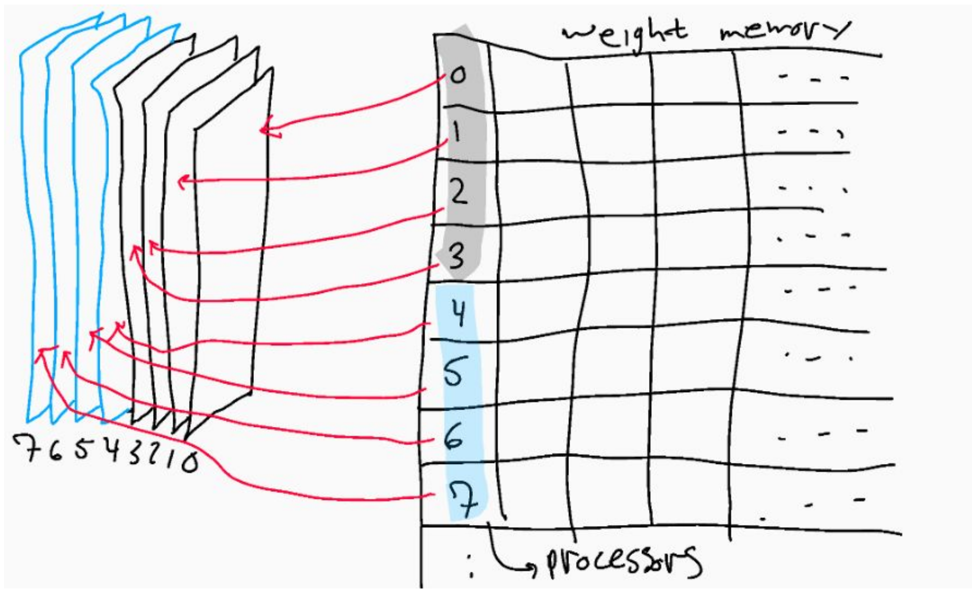


Figure 11: Processors in parallel

In **Figure 11** a given layer has 8 input channels (shown on the left 0-7) and an arbitrary number of output channels (not shown). Each processor will work on each channel in parallel as shown in the diagram. The number of rows in the weight memory reflect the number of input channels whereas the number of columns in the weight memory reflect the number of output channels. The blue and black color denote how the processors share memory. A set of four sequential processors (0-3, 4-7) share an instance of intermediate data memory. This data memory is used to store the input/output results for each layer. A subset of a diagram from the Maxim Integrated documentation shows the data instances for the first 16 processors in **Figure 12**.

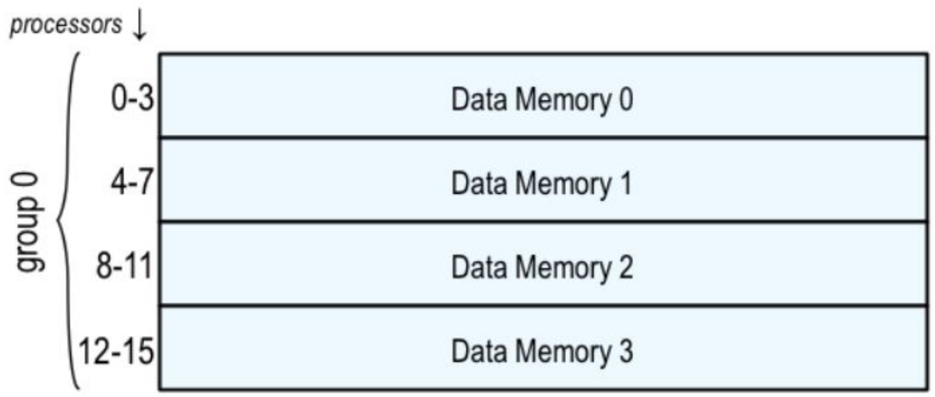
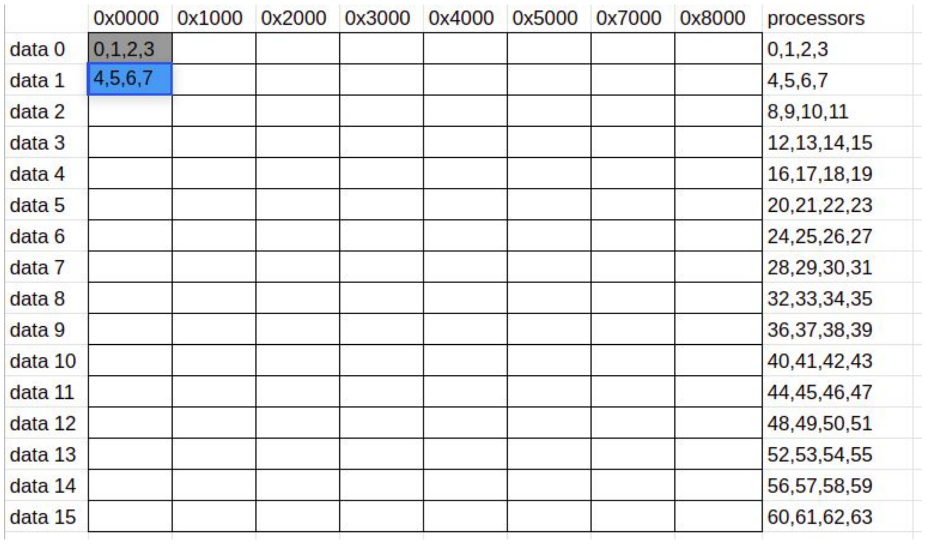


Figure 12: Data memory

Each data instance has a size of 32KB which is what limits the dimensions of an image if streaming is not used. The example in **Figure 13** shows how data flows through these memory instances.



	0x0000	0x1000	0x2000	0x3000	0x4000	0x5000	0x7000	0x8000	processors
data 0	0,1,2,3								0,1,2,3
data 1	4,5,6,7								4,5,6,7
data 2									8,9,10,11
data 3									12,13,14,15
data 4									16,17,18,19
data 5									20,21,22,23
data 6									24,25,26,27
data 7									28,29,30,31
data 8									32,33,34,35
data 9									36,37,38,39
data 10									40,41,42,43
data 11									44,45,46,47
data 12									48,49,50,51
data 13									52,53,54,55
data 14									56,57,58,59
data 15									60,61,62,63

Figure 13: Data flow through the memory

The above figure is based on the example shown in **Figure 11** and assumes the dimensions of the input data are 8x32x32. *32*32*4 = 4096* which is the number of input bytes for each set of four layers shown in **Figure 11**. This takes up the first 8th of the data instances for the first two sets of processors as shown in **Figure 13**. These details of which processors to use and which memory location to store the output data must be specified for each layer which makes the synthesis process difficult for complicated architectures.

A final example from the Maxim Integrated documentation is shown in **Figure 14**. This shows how weights are stored in memory.

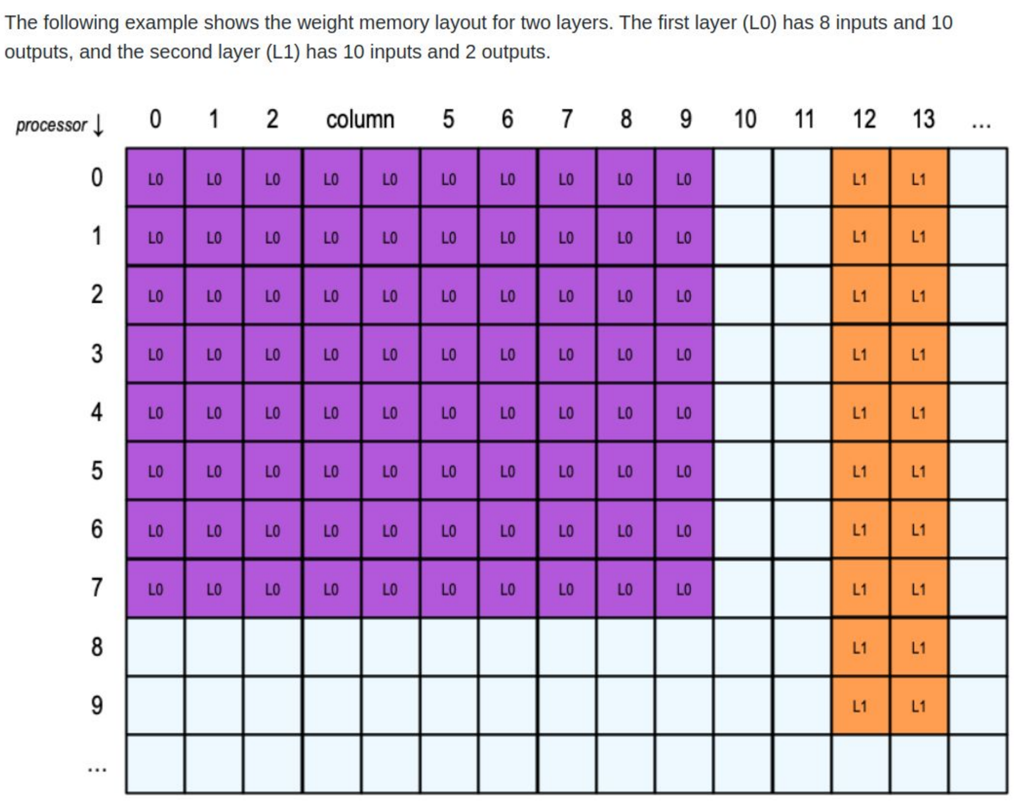

The following example shows the weight memory layout for two layers. The first layer (L0) has 8 inputs and 10 outputs, and the second layer (L1) has 10 inputs and 2 outputs.

Figure 14: Weight memory

Project Log

This section shows a log of progress made throughout the quarter

These tasks were based off of a project schedule shown here: [Project Schedule]

Task	Notes and Conclusions	Date
Set up the board and tools	• Created GitHub repos and ran Maxim's example code	01/21/2021
Learn Camera Interface: show video on TFT display	• Created a set of helper functions for capturing images and displaying them to the screen utilizing maxim libraries • Slight delay for images greater than 200x150	01/24/2021
Create live MNIST Classifier using pretrained network	• Read raw pixels from camera in YUV422 format and extracted the 8-bit luminance value needed for MNIST (bytes come from camera in little endian) • Preprocessed grayscale image through binary thresholding, negation, decimation to 28x28, convert from signed to unsigned • Displayed image to TFT display, loaded pixels to CNN buffer for inference, output results to the terminal	01/27/2021

Retrain MNIST from scratch using new architecture (go through training, quantization, and synthesis)	● Went through entire training, quantization, and synthesis process of building a new network for MNIST classification using Maxim's provided tools ● Training is doable without GPU for smaller networks but can take several hours for deeper networks	01/30/2021
Compare Quantized MNIST to FP32	● The accuracy of the quantized MNIST network on the test dataset was within 1% of the FP32 network	1/31/2021
Try to quantize and synthesize a pretrained PyTorch network that does not use Maxim's custom classes	● Attempted to quantize and synthesize a network trained using normal PyTorch with no success ● Maxim's modified version of PyTorch interests extra metadata into the network that is used for quantization and synthesis ● Reach out to Maxim engineer to see if this is feasible	2/1/2021
Explore how to utilize multiple processor groups for running two CNN	● Had a meeting with Maxim Integrated and they said at the moment this would be extremely difficult	2/16/2021
Experiment with Maxim face detection	● It seems that they don't do any face detection, they display a grid for the user to put their face in and then generate an embedding for face recognition	2/7/2021
Read papers on face detection	● Some notable papers for face detection such as MTCNN cascade multiple networks or steps together for high level feature extraction followed by bounding box/landmark coordinate prediction ● This cascading may be difficult to implement on the maxim board. An alternative is some sort of SSD that may have a simpler architecture	ongoing
Get a working face detector up and running (binary classification)	● Created a binary classifier on 64x64 images for faces detection. ● The network was trained on ~12000 images of single faces that take up >65% of the frame ● The negative images were taken from a large dataset of indoor locations ● Initially there were many false positives, increasing the size of the dataset helped ● May want to add more images and data augmentation for more robust classification ● Using one of the visualization tools in the maxim training repo, it can be seen that the network in some sense distinguishes certain features such as eyes and the nose	2/7/2021
add layers to the pretrained binary classifier and train on bounding box/landmark coordinates	● Implemented a bounding box detector that has reasonable accuracy on PC ● The network architecture capable of being synthesized but quantization has major effects on regression tasks	2/14/2021

	● Can visualize facial features with SHAP tool	
Get working on feather board	● Everything is compatible with the featherboard ● Only need to change a variable in the makefile to compile the drivers for the featherboard	2/17/2021
Localize a face in the image	● I changed the architecture of the network to be similar to VGG and added more training data and augmentation ● This made the underlying classifier much more robust ● However I am still having issues with localizing a face ● The current input is 80x80 grayscale	2/21/2021
Synthesize the face detector	● With the help Erman from Maxim Integrated I was able to synthesize the face detector onto the MAX78000 ● It works pretty well but needs to be more robust and needs to be trained on more diverse data	3/1/2021
Detect facial features from the localized face	● Using the same architecture as the face detector I trained a network for detecting eyes on a very small dataset that I labeled manually (<1k images) ● The performance is not very good but definitely not random, it at least tracks the relative position of the eyes	3/2/2021
Final changes and Project Report	● Augmented the bounding box data to get improved performance and manually added more training data	3/5/2021

Credits and References

This section lists links to the resources referenced for this project. The numbers don't have any meaning

1. https://cs231n.github.io/transfer-learning/ (Transfer learning)
2. https://arxiv.org/pdf/1902.09630.pdf (Generalized IoU loss function)
3. https://arxiv.org/pdf/1911.08287.pdf (Distance IoU loss function)
4. https://github.com/MaximIntegratedAI (Maxim Integrated Resources)
5. https://arxiv.org/abs/1806.08342 (Quantization paper)
6. https://arxiv.org/abs/1604.02878 (MTCNN, state of the art face detector architecture)
7. https://arxiv.org/abs/2002.03728 (Driver drowsiness detection implementation for Android)
8. https://arxiv.org/abs/1409.1556 (VGG paper)
9. https://github.com/slundberg/shap (SHAP tool)