# Neural Network Framework for Raspberry Pi Project Report

Carnegie Mellon University, 15-418: Parallel Computer Architecture and Programming

Ethan Gruman (egruman) and Eli Yu (yiluny)

May 8, 2018

## 1 Summary

An artificial neural network is a machine learning algorithm that draws inspiration from biological neural networks found in animal brains. It uses an interconnected layer of nodes to perform complex calculations. Conventionally, neural networks are run on heavy-duty machines with GPU's. Neural networks, however, can also be highly useful on embedded devices for robotics and computer vision tasks. In this project, we created a feed forward convolutional neural network library for the Raspberry Pi 3, a cheap, credit-card-sized computer with a quad core ARM Neon SIMD CPU. We built a pipeline that loads pre-trained weights and graph definition from Tensorflow. We exploited the CPU's SIMD and four cores using Open MP to speed up our library. In the end, we were able to achieve an overall speedup of $\approx 4.0\times$ for most layers on 4 cores.

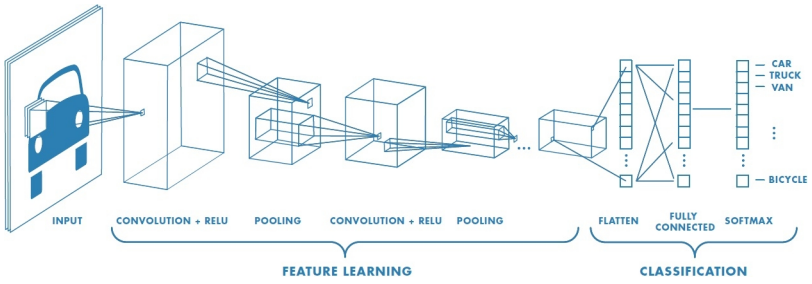


**Figure 1:** Structure of a standard convolutional neural network

## 2 Background

**Key Data Structures** The data structures used in the implementation of the neural network mainly are tensors. Tensors are generalizations of vectors in $n$ dimensions. For example, an input RGB image to a neural network would be a 3D tensor with dimension $H \times W \times C$, where $H$ and $W$ are the height and width of the image; $C$ is the number of channels.

**Operations** Operations on the data are known as "layers". Each layer performs some function based on the input tensors and spits out a new tensor. The different layers are:

**Convolutional** This layer applies a filter over the input data, which typically would be an image, working like a blurring function. An 2D input of arbitrary size convolved with a $3 \times 3$ filter can be visualized as Figure 2.

A 3D convolution is similar to a 2D convolution. Suppose the input is $H \times W \times C$, the filter must have the same number of channels as the input. In our library, we further constrain the other 2 dimensions to be the same size, as is the case in most convolution filters. So the filters are $F \times F \times C$. In the convolution step, 2D convolution is applied on each of the $C$ layers, the $C$ numbers are then summed up to get the final result. The output tensor has dimension $H \times W$. Figure 3 illustrates 3D convolution in more detail.

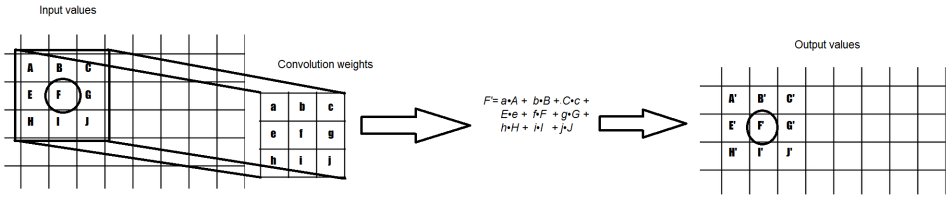


**Figure 2:** Visualization of 2D Convolution

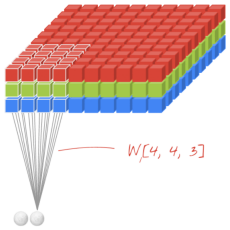


**Figure 3:** Visualization of 3D Convolution

**Pooling** This layer reduces the size of a 2D input tensor by splitting it into $2 \times 2$ boxes. Then the maximum value in each $2 \times 2$ box is extracted and placed in the output tensor. If the input tensor has dimension $H \times W$, then the output would have dimension $\frac{H}{2} \times \frac{W}{2}$. For a 3D tensor of dimension $H \times W$, we apply 2D pooling on each of the input's $C$ channels. The output would have dimension $\frac{H}{2} \times \frac{W}{2} \times C$. Figure 4 illustrates pooling for 2 dimensions.

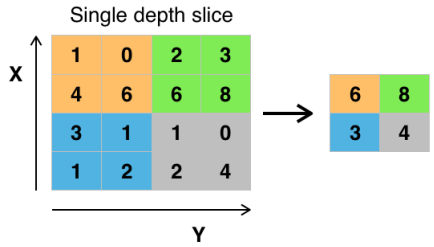


**Figure 4:** Visualization of pooling

**Fully Connected** This layer takes a 1D tensor $I$ of length $A$ and outputs another 1D tensor $O$ of length $B$. Each unit $O_i$ in the output is the dot product between a vector $W_i$ of length $A$ and the input $I$.

**Relu** This layer takes an arbitray tensor as input and outputs a tensor of the same dimension. For each element in the input, if it is less than 0 then the output will be 0. If it is greater than 0, the output will stay the same.

**Softmax** This layer is usually at the end of a neural network. It is defined as $O_j = \frac{e^{z_j}}{\sum_{k=0}^{K} e^{z_k}}$, for $j \in \{0, \ldots, K\}$, $O_j$ is the $j^{th}$ output.

**Inputs/Outputs** Our library first accepts pre-trained weights and computation graph from Tensorflow. Then it takes inputs to the neural network, which will propagate through the layers and produce an output.

**Points of Parallelization** We noticed that the structure of the network gave good opportunities for parallelism. This is because, for each layer, each node can calculate its own value independently. We also saw ways in which filtering methods can be parallelized using SIMD.

**Breakdown of Workload** The dependencies are that each layer has to be computed before later layers in the graph.

**Convolution** Suppose in the convolution layer we want to convolve with $N$ filters. Then the layer is data parallel across the filters. To get one output value we essentially calculate $C_1 \times X_1 + C_2 \times X_2 + \ldots$, where $C_i$ are the filter values, and $X_i$ are the values in a region of the input. The calculation of each output value is also data parallel. The $X_i$ used are in general not in a contiguous piece of memory, as it is a smaller cube inside a bigger 3D tensor. (see Figure 6) To get a single element in the output, we can convolve row by row. This approach can get good spatial locality because we will cache a region

larger than the cube. When we move on to calculating the next output value, we shift the our cube one unit right, which is likely cached when we calculated the previous output. The Raspberry Pi Model 3 has L1 cache of 32KB and a cache line of 64 bytes. This means the L1 cache can store 500 cache lines. The filters are usually small ($3 \times 3 \times C$ or $5 \times 5 \times C$ for $C$ channels). Since we calculate row by row, we will cache approximately $5C$ cache lines. We will also cache the filter. Unlike the cube in the input, the filter is stored as a contiguous line in memory. This means it will only take approximately $\frac{C \times 5 \times 5}{64}$ cache lines. As long as this number is smaller than the 500 cache lines, we will have good spatial locality and won't encounter thrashing. There is fairly good temporal locality because we access the filters repeatedly, which can be stored in cache. Additionally, when we shift the input region one unit to the right to calculate the next output value, we will reuse most of the input values stored in the cache. This approach is not very good for SIMD. Since the filters are usually short, SIMD can only do about 1 full load before having to move on to the next row.

**Pooling** Each max pooling operation on a $2 \times 2$ box on the input tensor is an independent operation, which means it is data parallel. Pooling has good spatial locality because each time we read two rows to calculate the max of the first $2 \times 2$ square in those two rows. When we calculate the max of the next $2 \times 2$ square to the right, these input values are already stored in cache. This operation is not suitable for SIMD because we need to load 2 adjacent elements twice.

**Fully Connected** The calculation of each output value is data parallel because it only requires the input vector and a vector of weights. The fully connected layer has good spatial locality because we will use every value in input and weight vectors, which are both contiguous pieces of memory. This layer is suitable for SIMD because we can efficiently load 4 values from both the input and weight vectors and multiply them together.

**Relu** This operation is data parallel since every output is only dependent on 1 input. It has good spatial locality because we can step through every element in a contiguous piece of memory. This layer is suitable for SIMD because we can efficiently load 4 values from the input. ARM's SIMD intrinsics allows us to execute conditional execution based on masks. The operation is simple enough that there will be no divergence.

**Softmax** This operation requires the entire input vector to calculate a single output value. It has good spatial locality because we can step through every element in a contiguous piece of memory. It is bad for SIMD because calculating $e^x$ can take considerably different amount of time for different $x$ values. So if the SIMD vector contains values that are very different, divergence can be significant.

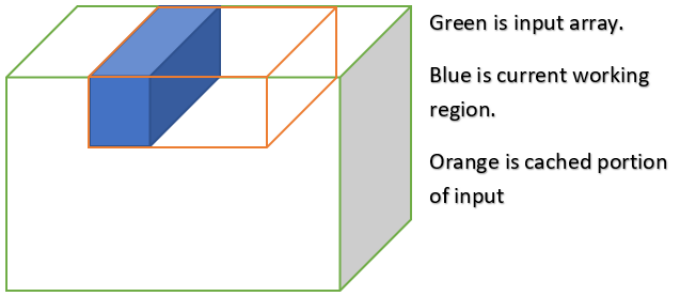


**Figure 5:** Cache of input tensor

# 3 Approach

We implemented a sequential version of our library as baseline. We generated test cases and thoroughly tested the sequential implementation. First, we generated smaller test cases manually. For larger test cases, we defined a network using Tensorflow, ported the network to our library and made sure that the output of our library was similar (absolute value $< 0.1$) to the output produced by Tensorflow. Next, we implemented a SIMD and a SIMD+OMP version. We tested the correctness of these versions by making sure its output matches the sequential version given the same input. We used TensorFlow-Examples github repository to build some models in Tensorflow: https://github.com/aymericdamien/TensorFlow-Examples. We also tried to use Ne10 for some ARM Neon functions, but we chose to use the Neon intrinsics directly instead.

### 3.1 Parallelization Technique: Fully Connected Layer

The fully connected layer is essentially a matrix-vector multiplication problem. Given a matrix $M$ of dimension $H \times W$, and vector $x$ of dimension $H$, we want to compute $Mx$. The basic idea is to take the dot product of each row of $M$ with $x$. `Algorithm_A` shows the pseudocode for naive SIMD implementation. A "sum" operation would be called every inner loop. This operation would sum up the values in the SIMD vector. It would flush the pipeline every time it's called and therefore is expensive. `Algorithm_B` shows a modified version where we kept a SIMD vector of accumulators which would only "reduce" after the inner loop is completed. We added a OpenMP around the outer loop since each row of the matrix $M$ can be calculated independently.

```
Algorithm_A() {
    for(int i=0; i<H; i++){
        for(int j=0; j<J; j+=4){
            float32x4_t in1 = load_4_floats(&x[j]);
            float32x4_t in2 = load_4_floats(&M[i*W+j]);
            output[i] += sum(in1*in2);
        }
    }
}

Algorithm_B() {
    for(int i=0; i<H; i++){
        float32x4_t accumx4 = {0.0,0.0,0.0,0.0};
        for(int j=0; j<J; j+=4){
            float32x4_t in1 = load_4_floats(&x[j]);
            float32x4_t in2 = load_4_floats(&M[i*W+j]);
            accumx4 += in1*in2;
        }
        store_4_floats(sum(accumx4),&output[i]);
    }
}
```

### 3.2 Parallelization Technique: Convolution Layer

The convolution layer is composed of a 3D input tensor of size $H \times W \times C$ convolved with a 3D filter of size $K \times K \times C$. This is essentially an extrapolation of 1D convolution. The method that we used to speed up the SIMD calculation of this step is as follows:
Original:

$$\underset{[a_1,a_2,a_3,a_4,a_5]}{\text{input}} \quad \times \quad \underset{[K_1,K_2,K_3]}{\text{filter}} \quad \Longrightarrow \quad \underset{\begin{bmatrix} K_1a_1+K_2a_2+K_3a_3, \\ K_1a_2+K_2a_3+K_3a_4, \\ K_1a_3+K_2a_4+K_3a_5 \end{bmatrix}}{\text{output}}$$

Our Method:

$$\begin{array}{ccccccc} \text{input} & & \text{temp vec} & & \text{intermediate result} & & \text{output} \\ [a_1,a_2,a_3,a_4,a_5] & \times & [K_1,K_1,K_1] & \Longrightarrow & [\langle K_1a_1, K_1a_2, K_1a_3\rangle, K_1a_4, K_1a_5] & & \\ [a_1,a_2,a_3,a_4,a_5] & \times & [K_2,K_2,K_2] & \Longrightarrow & [K_2a_1, \langle K_2a_2, K_2a_3, K_2a_4\rangle, K_2a_5] & \xRightarrow{\text{add triplets}} & \begin{bmatrix} K_1a_1+K_2a_2+K_3a_3, \\ K_1a_2+K_2a_3+K_3a_4, \\ K_1a_3+K_2a_4+K_3a_5 \end{bmatrix} \\ [a_1,a_2,a_3,a_4,a_5] & \times & [K_3,K_3,K_3] & \Longrightarrow & [K_3a_1, K_3a_2, \langle K_3a_3, K_3a_4, K_3a_5\rangle] & & \end{array}$$

To generalize this technique to a 2D case, we would still first iterate over each value in the filter, and make four copies of it to make a SIMD vector. Then for each row in the input, we would multiply the constant vector with

every element in the input row and store the resultant 4 wide vector in the output. See Figure 6. In the 3D case, suppose the input tensor is $H \times W \times C$, we just apply the 2D algorithm described above on each of the input's $C$ channels and add the values produced by different channels together. This algorithm allows us to better utilize SIMD capabilities because as we pointed out previously, the old algorithm can only do few SIMD operations because the width of the filters are short. In this case, we can apply SIMD to an entire surface of the input.

Locality: For a $H \times W \times C$ input and $3 \times 3 \times C$ filter, convolution layer will generate a $(H-2) \times (W-2)$ output, which is around $H \times W$ assuming that $H$ and $W$ are both considerably larger than 3. In the new algorithm, when we are working a specific channel, we will use a $H \times W$ rectangle from input, a $3 \times 3 \times C$ filter, and a $H \times W$ output. Overall this is about $2 \times H \times W$. We mentioned that the Raspberry Pi has 32KB of L1 cache. If $H$ and $W$ are equal, this means they need to be less than 128. This is enough for most tasks that we expect to run. MNIST for example is $28 \times 28$.

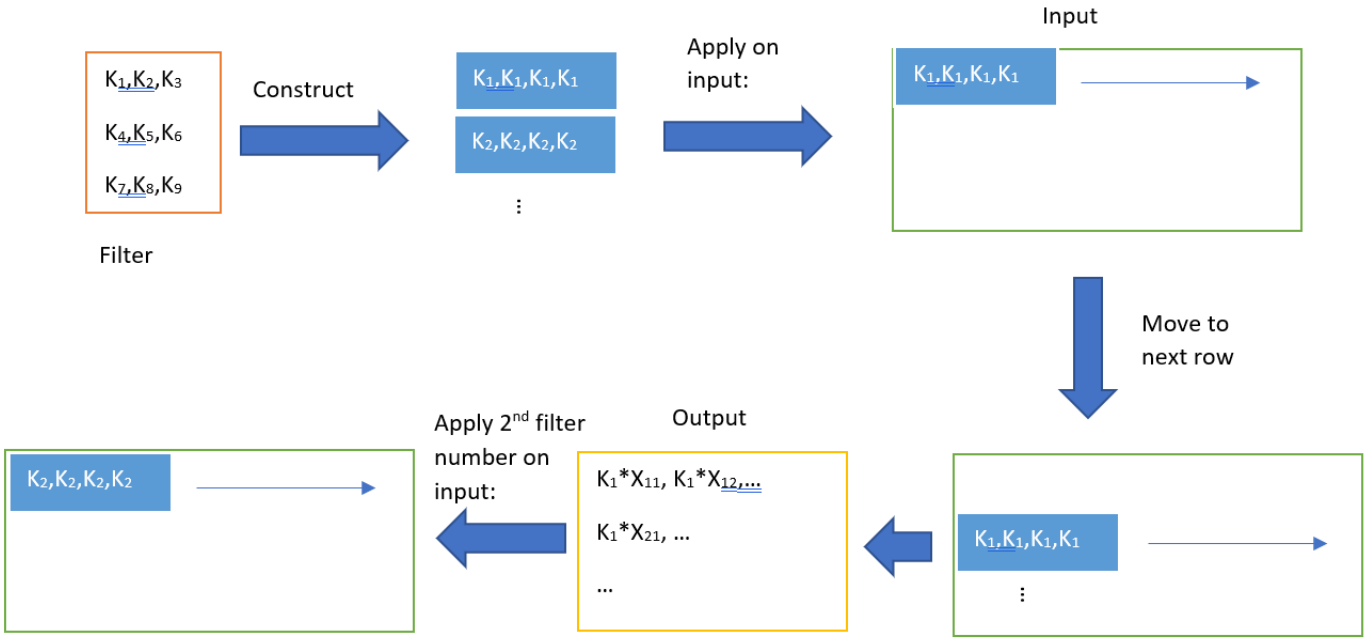


**Figure 6:** 2D convolution

We set the number of OpenMP threads to 4 since we had 4 cores. We used it to parallelize across the filters in the convolution layer since each filter's operations are independent of each other.

## 3.3 Parallelization Technique: Pooling Layer

First we tried using SIMD vector to load the 2 numbers in the top half of a $2 \times 2$ box and use another SIMD load for the bottom 2 numbers in the box. Then we used 2 SIMD compare operations to get the biggest number in the box. This approach did not achieve much speedup at all because it only loads 2 numbers at a time instead of 4.

The second approach was to calculate two adjacent $2 \times 2$ boxes together. We first loaded all 4 numbers on the top row into a vector $v_1$ and then the 4 numbers in the bottom row into $v_2$. Then we used a SIMD intrinsics function to take the element wise max of $v_1$ and $v_2$ and store it into $v_3$. Then we used another SIMD intrinsics function that can find the max between $v_3[0]$ and $v_3[1]$, and $v_3[2]$ and $v_3[3]$ in one step. The result is a vector of length 2 that is the maximum of the two boxes. Lastly, we store the numbers into memory and move on to the next group of 2 boxes.

We used OpenMP to parallelize across the channels of the input since they are independent.

### 3.4 Parallelization Technique: Relu Layer

We used OpenMP to parallelize across the channels of the input since they are independent. We loaded 4 values from input and used a vector of zeros to check if they are greater than 0. Based on this output mask, we could conditionally choose to store 0 to the output or the original value.

## 4 Results

We compared the performance of three version of our library on convolution, fully connected, pooling, softmax, relu layers and measured the speedups compared to the sequential version. We generated input data by randomly generating numbers for an input tensor as well as weights. We used the same inputs and weights for all three versions of our library. We did not run any programs on the Raspberry Pi other than our ssh session to run code and the neural network itself to ensure fairness. We also used a desktop fan to keep it cool. For convolution, the input is $100 \times 100 \times 10$ and the output is $96 \times 96 \times 10$. For fully connected, we used a flattened $100 \times 100$ image as input and the output is 10 values. For pooling we accepted a $100 \times 100 \times 4$ tensor and produced a $48 \times 48 \times 16$ tensor. For Relu, we tested on $100 \times 100 \times 3$ tensors. We repeated the same process many times to simulate forward passing new images to the network. We did not test Softmax because it is a small layer typically used on the output. We think there is not enough computation to see a significant improvement. The sequential version is a single threaded CPU based implementation.

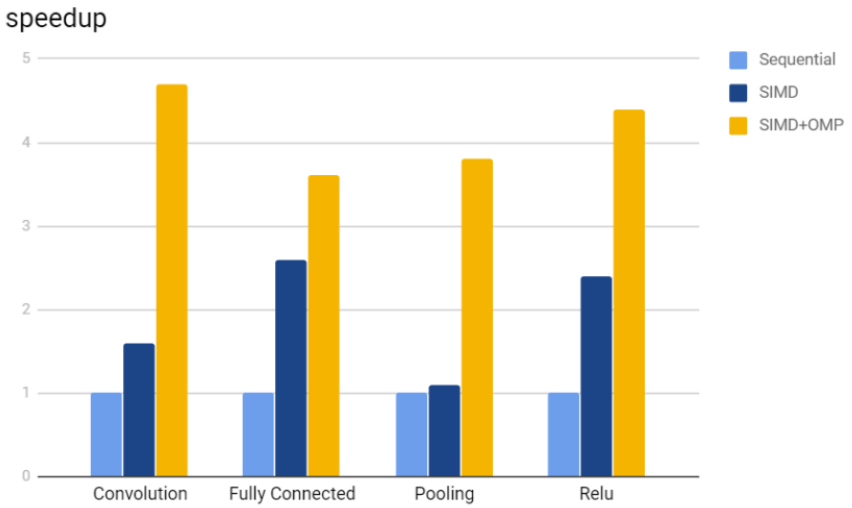


**Figure 7:** Speedup Results

Next we measured the performance of convolutional layer. We wanted to thoroughly understand this layer because it is the most computationally expensive layer and also one of the most commonly used layers in a convolutional neural network.

We measured the speedup versus number of filters. We ordered threads to execute in parallel across the filters. We expected that when we increase the number of filters. All cores would do approximately the same amount of work. Using dynamic scheduling would perhaps make it more even. Our results show that dynamic and static scheduling attain the same speedup. This means each filter takes about the same amount of time to compute. We achieve higher speedup for more filters because we can more evenly divide a large number of filters amongst the processors.

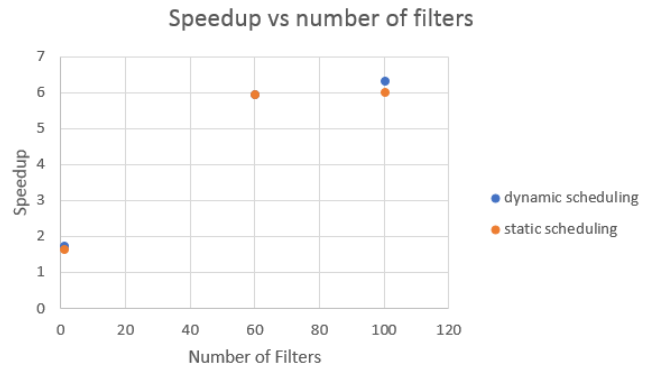


**Figure 8:** Speedup vs num filters, input images are $50 \times 50 \times 10$, filters are $5 \times 5 \times 10$, 100 iterations

When the number of filters is low, we tried to parallelize in the inner loop during the multiplication of filter value and the input rows. Dynamic scheduling in inner loop should slower because of false sharing of output array.

We expect that increasing number of input channels would increase speedup because doing this would increase the working set of sequential version but not parallel. We did not have time test this.

We also tried to see if changing the width of the input while fixing the height would affect the speedup. We fixed height of input to be 100. We expected that at width $\approx 160$ the speedup will drop sharply because the working set exceeds L1 cache. In reality this is not entirely true. Speedup dropped somewhat smoothly, and for extremely large width we do not get much speedup at all.

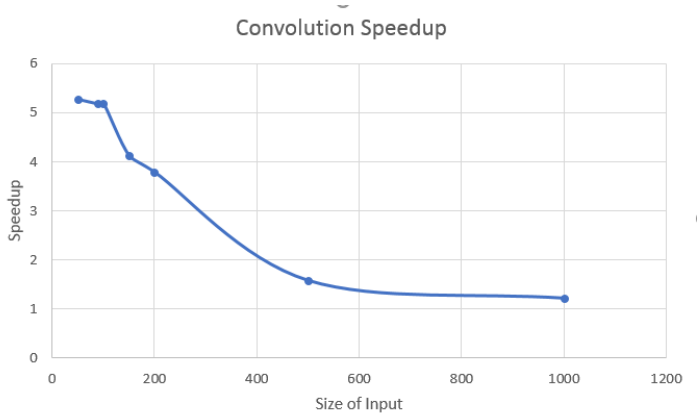


**Figure 9:** Speedup Results

In addition, we found the parallel code to run at $1.8\times$ SIMD+OMP speedup on average when we removed arithmetic operations. This means the computation was fairly heavy in arithmetic compared to memory operations.

# 5 Conclusion

In this project, we learned a lot about the architecture of the Raspberry Pi's CPU and how to exploit its SIMD library. We were surprised by the amount of processing power such a cheap computer offers. We were also happy that we achieved a good amount of speed up. In the future, we plan to investigate further if there are opportunities for more optimization. One potential improvement would be using loop unrolling. Another idea is to use assembly to execute SIMD instead of using ARM Neon intrinsics. Lastly, we discovered close to the end of our project that a project called py-videocore exposes GPGPU programming interface to the Raspberry Pi's GPU. We can use this project to implement a new version of our library that uses GPU.

# 6 References


- https://www.mathworks.com/discovery/convolutional-neural-network.html
- https://www.tensorflow.org
- https://developer.arm.com/technologies/neon/intrinsics
- https://github.com/yilunyu/Raspberry_CNN
- https://github.com/aymericdamien/TensorFlow-Examples


# 7 Division of Work

Equal work was performed by both project members.