Convolutional Nerual Network with Intel® Threading Building Blocks

ECE4900

List of Authors: Matthew Horvath, Ryan Marten
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: mhorvath@oakland.edu, rmarten@oakland.edu

I. Introduction

Convolutional neural networks (CNN) are currently used worldwide in a varying spectrum of application areas. Each of these applications share a common objective of being able to learn features from their massive data bases and generalize outputs based upon occurrences not learned within the training phase. Utilizing data from the MNIST handwritten database, this project will assimilate this database in order to fully implement, through training and testing, a convolutional neural network for addressing these handwritten digits. This application will be able to complete two separate tasks of training and testing, which will sufficiently output results that will be able to demonstrate the capabilities of a CNN. Upon the completion of such a design, the final output will be able to tell the user if the program guessed the correct digit or not. With this output, it will be able to address various error issues, as well as let the user know other important statistics about the network overall (sample size, training time, etc.) These statistics will be able to help in the comparison between sequential implementation and TBB. Motivation for this project stemmed from the fact that both students have previous experience with neural networks and wanted to implement a convolutional layer, along with parallelization, to further their knowledge on neural networks overall. Convolutions within a neural network remain the approach of choice for addressing complex image recognition tasks, and in combination with TBB, this approach theoretically will be able to sufficiently provide clear results with time reduction and increased accuracy. This approach also aided in further expanding research into CNN's by each student to better understand the fundamentals of such a project. Furthermore, this provides applicable details and knowledge into how convolutional neural networks, whether parallelized or sequential, are appropriate for real world applications.

II. Methodology

A. Architecture

This implementation will consider two separate sections of training and testing. This will utilize the MNIST database with an image size of 28x28. Both of these sections are used in tandem in order to achieve a successful convolutional neural network. The network consists of two sides, the first being the convolutional and pooling side, with the second being the fully connected side. The convolutional side will create a feature map of our image which will better help the fully connected side classify the image.

The convolutional side of the network features two layers. The first layer acts as the input of the entire system, while the second layer will act as the input into the fully connected layers. This first layer of the convolution network will perform 6 separate "narrow convolutions" across the original input image using a 5x5 kernel. This operation of a "narrow convolution" means that only the places on the original image where the entire 5x5 kernel can fit will be where the convolutions will occur. This results in each of the output images from this operation to be only 24x24 (down from 28x28). These six convoluted images are called feature maps, as the convolution process in each image ideally will have extracted key "features" from the original image, with each of the six having a different
feature.

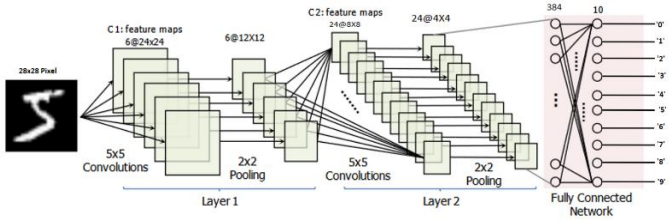

Figure 1. *CNN Architecture*

Next, these six feature maps are now fed into a pooling layer. For this implementation, we use 2x2 pooling checking for the max in each 2x2 area. Performing this process will cut these feature maps down to 12x12 size. The idea behind performing these pooling operations is to cut down on the unnecessary information that will eventually be fed into the fully connected network. By looking at each 2x2 "block" on the 24x24 feature map, we can see which of those four values is the greatest and map it to the pooled image. Doing this over the entire feature

map effectively shrinks the original feature map by a factor of two in each direction.

The output of the first layer (convolution and pooling) is now fed into the second layer. Here, similar operations are performed as were in the previous layer. We will take each of the now six pooled 12x12 images, and we will convolve them with another 5x5 kernel. This time however, we will add them up and output that sum to a single feature map. The final value that is sent to the feature map is sent through a sigmoid function before being stored. Like before, "narrow convolutions" are performed, which will shrink the 12x12 pooled image down to a 8x8 image. When adding each of these convolutions up, we perform a simple sigmoid operation on each value, effectively limiting the maximum value of each to be between 0 -1. This process of convolving each of the pooled 12x12 images, adding them up, and storing them in a 8x8 image is performed 24 times, resulting in our second set of feature maps. These 24 feature maps, each 8x8 in size, are now ready to be pooled. The same process as the first pooling is performed here, as we will use 2x2 pooling looking for the maximum value in each of the 2x2 blocks on the 8x8 feature maps. This will result in 24 4x4 pooled images which will be the final output of the convolution side of the network.

Now that all the convolution and pooling operations have been completed, we now need to feed the 24 pooled images into the fully connected layer. With how this code is set up, we need to flatten the pooled images. Each of these pooled images are stored as 2-D arrays, while our fully connected layers operate on 1-D arrays. A simple formula to map a 2-D array to a 1-D array is used in order to compute the final input for the fully connected layer.

The fully connected side of the network features three layers: the input layer, a hidden layer, and an output layer. The input layer consists of 384 separate neurons that will each take in one of the values in the now flattened final output from the convolution side. Each of these neurons will perform a mathematical function on the input, that will consist of multiplying the input with a given weight, before passing it on to the next layer.

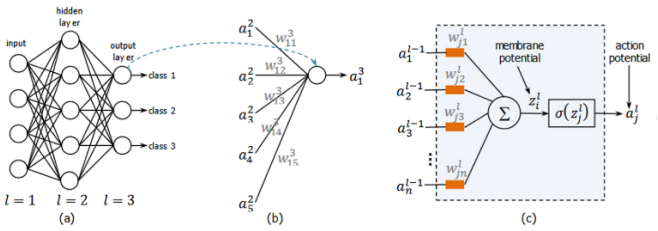

Figure 2.
(a). *Fully Connected Layer Example.*
(b). *Connection from previous layer to single neuron.*
(c). *Accumulation and Activation of Neuron Output*

The hidden layer, or the middle layer, of the network, consist of only 128 neurons. Each of these neurons is connected to the previous layers neurons, and each of these connections are represented by a weighted value. These hidden layer neurons will take in the input from each and every previous layer output, multiplied by that given weight, and accumulated. Once these values are calculated, the output of each of these neurons will be sent through an activation function, which will help keep the values within a given range. For the purpose of this network, we use the sigmoid (same as the one used in the convolutions).

For the final output layer, the same process that was performed from the first layer to the second layer is done, this time for the middle layer to the output layer. This final output layer has only 10 outputs, each corresponding to the digit 0-9. As mentioned before, the sigmoid operation is performed on the output of each layer. The benefit of this is that the value will always be between 0-1, meaning that when looking at the final outputs, each of the 10 values will be in this range. The highest value in these outputs should correspond to what the handwritten digit is.

B. Training

The general architecture as explained above makes up the core of the training model used in the application. The one difference for when we train the system is the addition of the back-propagation process. This training process is shown below in Figure 3. The general idea is that when an image is inputted into the network, we can compare what the calculated output is compared to the actual input. By doing this, we can go back though the network and change the weight values accordingly to make sure it correctly predicts an image. Each iteration, these weights will change more and more, and after a subsequent number of training runs, the final set of weights can be used in the testing application.

For the backpropagation, very simple gradient descent is implemented. With this particular system, no biases are used, so the accuracy will only ever reach so high. Because of the lack of biases however, it makes the backpropagation much simpler to implement. The first step is to calculate the error of the 10 output neurons. Once these are found, we can then find the error of the neurons in the hidden layer, with respect to the output layer. These errors that are found act as a link in a chain; any error from the first layer to the second layer, affects the third layer. This is why the training will always start with checking the error of the output layer, then working backwards. With these errors now calculated, we can now perform a multiplication operation on the outputs of both the second layer neurons and the first layer neurons. The goal of the operation is to find new weights that will better estimate the correct value of the image. With the addition of some global variables that are set at the compile time, the final equation for updating each weight value is shown below.

*Delta = (Learning Rate *Hidden Layer Error * Hidden-Layer Neuron) + (Momentum * Delta)*

This "Delta" value is the change in the weight value from epoch to epoch. Ideally, with each iteration the delta will get smaller and smaller, based off of the error values getting smaller and smaller. The momentum and learning rate values are chosen by us and can be changed to try and produce a more accurate outcome, or possibly train the system quicker.

This process of training is what takes the most amount of time when it comes to CNN. Each input training image is sent through the system anywhere from 500-1000 times, each time updating all of the weights, before we train the next image. In total, we need to train roughly 40,000-60,000 images in order to calculate a good set of weights to be used in the testing application. Because of these amounts of training runs needed, the benefit of the TBB libraries will be most apparent when using it on the training application.

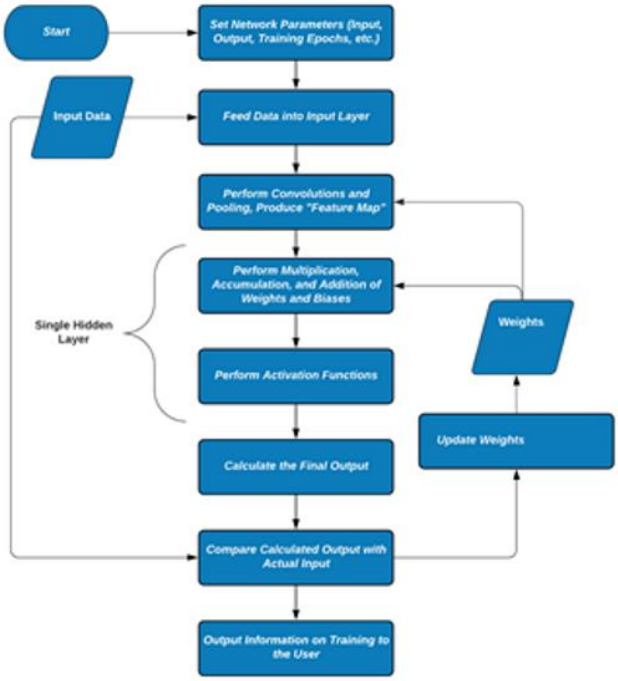

Figure 3. *Training Flow Diagram*

C. Testing

After completion of the training portion of the design, the testing of the CNN can now commence. This testing section considers the outputted weight model file from the training portion for use within this section. This will also consider testing images from the MNIST dataset for the CNN testing phase to approximate its answer for what image is outputting. Finally, this also takes into account the same kernel as the previous training section utilized. All of these files will set the parameters for the testing section and feed the data into the input layer. This layer will then go into the hidden layers that will consider the various functions that will need to be able to calculate the final output. This calculated output will then compare to the actual digit that it read and be able to tell the user what it thinks that digit is. This output to the user will then give a yes or a no, as well as and both the result and the digit value it compared to. By the chance it is incorrect, this will go to an error counter that will give a percentage error to the user upon completion of the execution. All of these statistics will again compare to the TBB implementation against the sequential. The overall timing of each subsequent section of the design should be much more rapid than the training counterpart, with the theoretical timing of each being roughly the same. Figure 4 devolves the flow diagram for this section of the implementation.

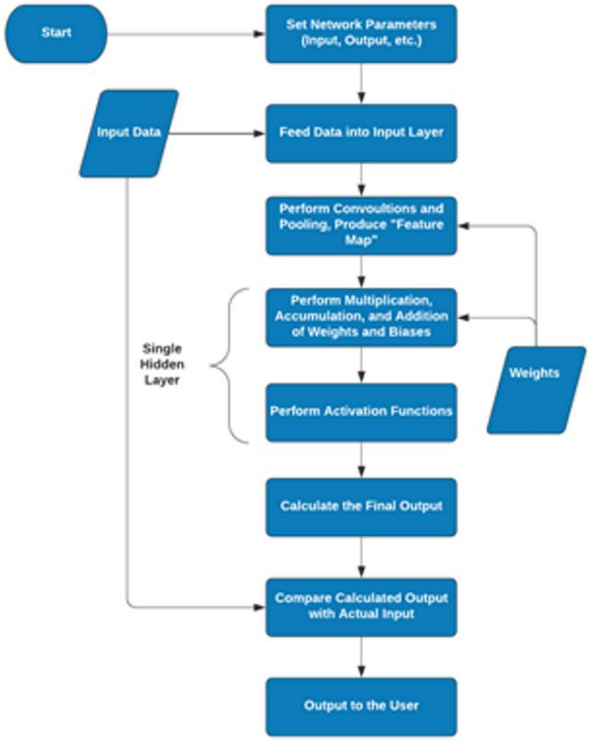

Figure 4. *Testing Flow Diagram*

III. Experimental Setup

In order to develop the results needed for the overall project, a variety of testing methods were utilized through the use of a virtual machine with the Linux Ubuntu kernel installed and the KDE advanced text editor software, Kate. This allowed for each student to have efficient testing,

without the use of transferring data between personal computers and the board. This also increased efficiency when debugging the program as a whole. Finalizing the design, the provided Terasic DE2i-150-FPGA Development Kit was employed in order to test the performance of the project upon an Intel Atom® N2600. The use of the board provided the project to be correctly compiled, sequentially and through TBB, allowing for various tests to demonstrate the project.

Specifically, the sequential tests revolved around the timing of the training segment (10 images, 1,000 images, 15,000 images, etc.), and then taking the outputted model weight file into the testing phase. This testing phase would then appropriately take in this model file for use when running through the various images for guessing. This will then output a time taken, as a guess for each of the images it will run through.

The testing of the TBB parallel portion of the CNN goes about this training and testing in the exact same manner, in order to achieve a clear comparison between its sequential counterparts. This TBB will train its separate model weights file, which will then go into the testing file, both of which will also output a time taken to train and test.

Theoretically, the training section for the sequential and TBB should take a substantially long time, with the TBB being theoretically much faster; this faster gain being the goal of the project as a whole. The testing portion of both networks should follow in a different manner, as the testing should run rapidly for each network, with the benefits of TBB being possibly non-existent. However, the theoretical benefits of parallelizing the training portion would still be able to demonstrate the advantages of TBB. Following the outputs of each system, the full comparison between the two systems is able to be completed.

IV. Results

Starting with the training of the sequential, along with the TBB, the results demonstrated the drastic time improvements that come from the benefit of having the design parallelized through TBB. Figure 5 and 6 show these time comparisons between the two systems.

```
Sequential Training Timings
Average CNN Time: 28668 us
Average FCN Time: 1824 us
Average Back Propagation Time: 3739 us
Average Time for each computation (Per Epoch): 34250 us
Average Time for each computation (Per Training Image): 17536053 us
```

Figure 5. *Sequential Training Timing*

```
TBB Training Timings
Average CNN Time: 9892 us
Average FCN Time: 2411 us
Average Back Propagation Time: 5882 us
Average Time for each computation (Per Epoch): 18049 us
Average Time for each computation (Per Training Image): 9241524 us
```

Figure 6. *TBB Training Timing*

Based on these results, it is clear that there is an approximate 45% decrease in the amount of time it takes for the program to run through the TBB training process. This gain was most apparent through the CNN, where the parallel_for loops were located, as the FCN and back propagation times were closely related to each other. These results were estimated theoretically to be around this significant and proves the design of our project did successfully implement the TBB functions. All of these results stem from the use of 512 epochs, which mean that each image is trained separately 512 times, whereas if we did a single epoch, the CNN improvement falls off to almost zero.

Moving forward, taking the model weights file generated from the training, each separately trained model file was implemented into the testing for TBB and sequential, respectively. Theoretically, the benefit should not be that significant, as the time to test each image is so small that the benefits of parallelization begins to fall off. However, based on Figures 7 and 8, the testing did allow for a slight improvement on the testing time.

```
Sequential Testing Timings
Average CNN Time: 31202 us(C1: 6753, C2:24433)
Average FCN Time: 1860 us
Average Time for each computation: 33067 us
Number of correct samples: 99 / 1000
Accuracy: 9.90
```

F

Figure 7. *Sequential Testing Timing*

```
TBB Testing Timings
Average CNN Time: 28010 us(C1: 8003, C2:19992)
Average FCN Time: 1771 us
Average Time for each computation: 29785 us
Number of correct samples: 99 / 1000
Accuracy: 9.90
```

Figure 8. *TBB Testing Timing*

From Figure 7 and 8, the major time improvements gain result from the convolutional 2, as this gain results in the biggest time decrease between the two systems. Taking into consideration the low accuracy, these do show the same accuracy between the two systems. This is important to note as both systems are then functioning properly and allows for the timing comparisons to be justified.

V. Conclusions

Following the results from the project, various points in regard to the project design and overall process is able to be discerned. Working with the convolutional side of a neural network and the parallelization through TBB was both a learning process and a good experience for both students, as this was a new concept they had not worked

with previously. This included various challenges, and research into how convolutional neural networks worked in order to try to correctly implement one, as well as continue working with the TBB knowledge learned within the class. Following the results, it is no question that the accuracy left much to be desired, however noticing that both the sequential and TBB ended up with the same accuracy proves that both systems worked correctly to each other. Furthermore, it is plainly seen that training had a great decrease in time through the use of parallel_for loops within the convolutional side of the TBB training. This decrease in time is largely beneficial, as it can take up to ~32 hours of training sequentially, and with the TBB, this time decreases to ~16 hours. More implementations of parallel_pipeline, as well as parallel_reduce would have seen large time decreases as well, however issues implementing these processes left them out of the final design. These improvements could be very noticeable if implemented, and a further design would surely add these into the final code. Finally, the issue of accuracy is a major impediment with the current design, and further improving this accuracy would be very beneficial to the final product. Conclusively, the knowledge gained on this project overall from the TBB multithreading to working with the convolutions within a neural network were very beneficial and is something both students hope to continue working on in the future.

VI. Bibliography

[1] C. Cdeotte, "MNIST - CNN coded in C - [0.995]," Kaggle, 21-Aug-2018. [Online]. Available: https://www.kaggle.com/cdeotte/mnist-cnn-coded-in-c-0-995.

[2] *D. C. Ciresan, U. Meier, L. M. Gambardella and J. Schmidhuber, "Convolutional Neural Network Committees for Handwritten Character Classification," 2011 International Conference on Document Analysis and Recognition, Beijing, 2011, pp. 1135-1139, doi: 10.1109/ICDAR.2011.229.I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.*

[3] *M. Y. W. Teow, "Understanding convolutional neural networks using a minimal model for handwritten digit recognition," 2017 IEEE 2nd International Conference on Automatic Control and Intelligent.*

[4] *Systems (I2CACIS), Kota Kinabalu, 2017, pp. 167-172, doi: 10.1109/I2CACIS.2017.8239052.*

[5] A. Escontrela, "Convolutional Neural Networks from the ground up," Medium, 17-Jun-2018. [Online]. Avaliable: https://towardsdatascience.com/convolutional-nerual-networks-from-the-ground-up-c67bb4145el.

[6] *J. Brownlee, "How Do Convolutional Layers Work in Deep Learning Neural Networks?," Machine Learning Mastery, 16-Apr-2020. [Online]. Available: https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/.*

[7] *HyTruongSon. "HyTruongSon/Neural-Network-MNIST-CPP." GitHub, 11 Oct. 2015, github.com/HyTruongSon/Neural-Network-MNIST-CPP.*

[8] *Mazur, Matt. "A Step by Step Backpropagation Example." Matt Mazur, 21 Aug. 2020, mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.*