**Implementation of a Convolutional Neural Network in C**

*Dean Conway*

**Abstract**:

Convolutional neural networks utilize convolution layers to reduce parameter count and increase accuracy in computer vision scenarios. A classic example, and the informal “Hello World”, of neural networks is the recognition of 70,000 images of handwritten digits from the MNIST database. The implementation of such networks is commonly done in Python or MATLAB. This is due to the existence of many libraries that make the process easier for programmers of all experience levels. This paper describes an original implementation of a convolutional neural network in C utilizing only standard C libraries. This implementation is expected to improve upon the run-time efficiency of implementations in other languages. The network is constructed from 2 convolutional and pooling layers of length 6 and 12 respectively. The second pooling layer is concatenated into a vector of its elements. This concatenated vector is fully connected to an output layer consisting of 10 nodes. As such, this network consists of 3,898 parameters to be trained.

The training process for the network took well over 24 hours and used unconventional means to train more quickly such as halting the training process and adjusting the learning rate at my discretion. Many modern networks automate the learning rate with algorithms designed to optimize learning. I intend to address this in future work. The accuracy of the network at this time is approximately 95%, meaning the network correctly identifies 95% of the images it is presented with. This accuracy is behind state-of-the-art implementations of this recognition problem and I hope to improve upon it in future research.

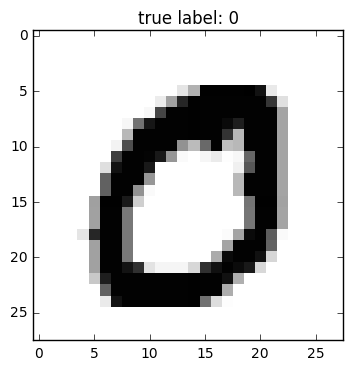


Figure : Example Image from the MNIST Database

**Introduction:**

The purpose of this paper is to outline the network I’ve implemented in C, the network architecture I used, the various structures in the network, the networks’ configuration, and the potential benefits of a C-based convolutional neural network. The architecture portion will briefly discuss the two different architectures I considered and why I chose the convolutional architecture. The implementation portion will cover the various structures defined to store the network in the program and keep the data organized. The configuration section will discuss the parameters / hyper-parameters of the network. Before the conclusions, I will discuss the results of my implementation and why they are what they are.

**Neural Network Architectures for Image Processing**:

For this implementation, I considered two architectures of neural networks. The first being the widely known multilayer perceptron. A fully connected network of nodes organized into non-vertically connected layers. This approach has been used to some success [2] however, convolutional networks have been shown to be well-suited for visual analysis (Cite 3 from the Microsoft article).

The major structural decisions such as how many convolutional layers to make and the kernel size was decided by an example in a paper by Zhifei Zhang [3] deriving backpropagation in a convolutional network. I chose to use this instead of making the structural decisions myself because the focus of this research is the implementation and not the design, with which I have no experience yet. This example is shown in figure 2.

The convolutional network approach is also going to have fewer parameters than a multilayer perceptron with enough layers to properly generalize. This implementation was not designed with optimization algorithms such as momentum, so the fewer parameters to train, the better. I intend to address this in future research.

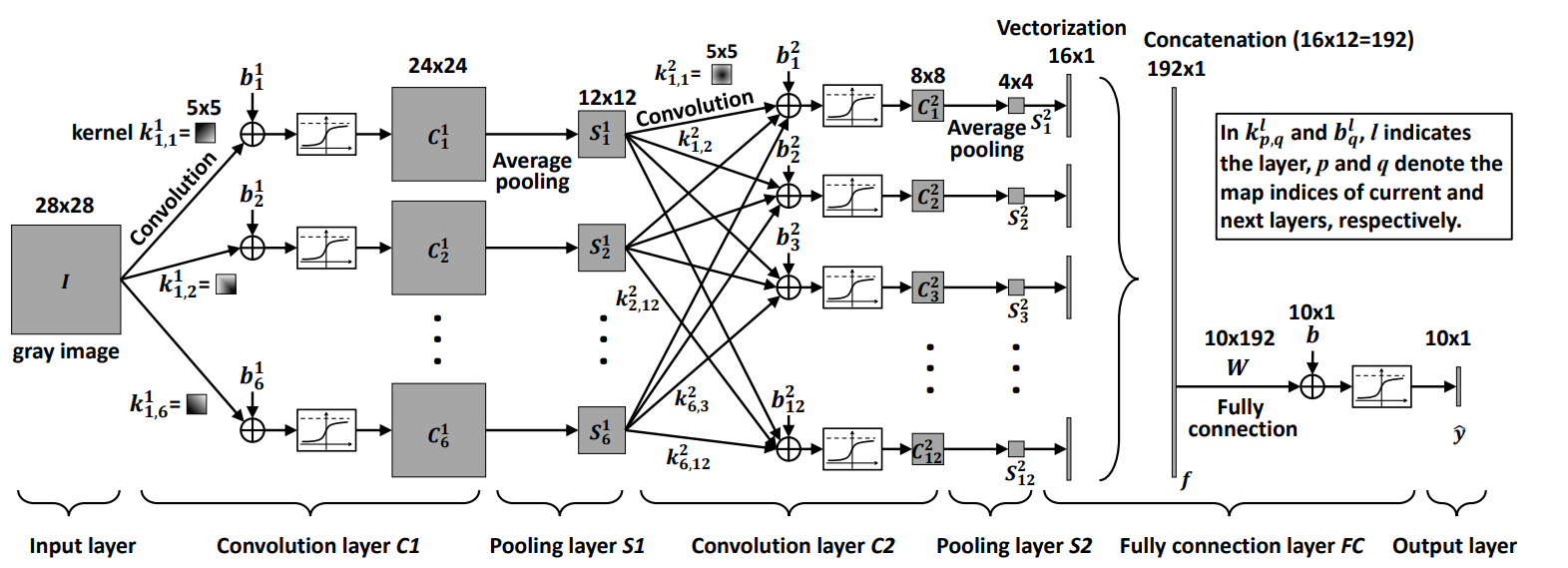


Figure : CNN Structure [3]

**Structure Implementation**:

To implement a convolutional network in C, various structures needed to be created such that each member of the network could be accessed, changed, and in some cases, iterated over. While C does have native support for vectors and matrices in the form of 1-dimensional and 2-dimensional arrays, I created structs to support these more explicitly. This helped to define more complex features such as a kernel array, a 2-dimensional array of 2-dimensional arrays, which would have been more difficult to represent otherwise.

The various structs created required functions to allocate the proper amount of memory and populate them with values. The functions for the less complex structures became useful in declaring the more complex structures. For example, when declaring a kernel array, it was useful to have the matrix function since the kernel array is an array of matrices. All such functions returned pointers to these objects rather than the objects themselves because this makes it easier to free the memory used for these objects when it becomes necessary.

The most complex structure in the program is the network structure itself. This contains pointers to all the members that contain values. To keep this structure somewhat adaptable, the values that define the size of these members are written as defines that are used throughout the program. This means that changing certain aspects of the network such as the length of individual layers is relatively easy and doesn’t require a thorough overhaul. The function for creating this structure also returns a pointer because it’s more sensible to pass a pointer from function to function than it is to pass the entire network.

The network pointer is not the only structure of its kind in the program. Duplicate networks with all the same parameters are created to store the image gradient and the batch gradient. These are the changes to a particular weight desired by an image and the batch gradient is a sum of the image gradient over a batch. It is after every batch that the batch gradient is added to the network to adjust the weights, biases, and kernels to perform gradient descent.

**Network Configuration**:

As mentioned in the network architecture section of this paper, I followed an architecture similar to that from a paper by Zifhei Zang [3]. However, I made one change to it in regards to the activation function, opting to use a leaky ReLu instead of the softmax function defined for the original network. I chose to do this because it’s computationally simpler and typical of convolutional neural networks.

Other network parameters were the batch size, the learning rate, and the bounds for the weight initialization. The batch size was 200 images. This was done to try and have enough images to be representative of the whole dataset. The learning rate ranged from 0.1 to 0.01 at my discretion. During training, if the networks loss began to stagnate, I would pause the training and adjust the learning rate. This more manual approach is hard to reproduce and inefficient so it’s certainly not recommended but for the purposes of the initial implementation, it made the project more manageable. The last major parameter was the weight initialization. This parameter was also kept simple and initialized weights between -1 and 1. This parameter can also be optimized for a given dataset but again, that was excluded for the sake of maintaining a manageable scope for the initial program. I intend to address all of these parameters more thoroughly in future research.

**Results / Training**:

The training process for this network took approximately 36 hours. Compared to other implementations, this is an unacceptably long time as many of them achieve better results in a much shorter time. However, many of those implementations use multiple processing cores where as the one described in this paper currently uses just one processing core. In testing the network, I used the OpenMP library in an effort to accurately hypothesize how much faster the process could be with the current implementation. Utilizing 4 CPU cores, the training process was reduced to approximately 8 hours. However, this implementation was not designed to perform with OpenMP so the calculations, while all performed, did not work across threads properly.

For training this network, I used all 60,000 images of the MNIST training set and for testing its performance, I used the separate testing set of 10,000 images. I also iterated over the entire training set approximately 400 times. This is another reason for the networks long training time. Many other implementations require only 10 epochs to reach similar classification rates. This is due in large part to not using an adaptive learning rate. This means that the initial epochs in my network were not making large strides towards converging and instead were moving very slowly from start to finish.

At the end of training, the network successfully classified 94.92% of the images in the test set. This is well behind the state-of-the-art networks used for this classification problem however as the first neural network I’ve ever written, it’s very encouraging.

**Conclusions**:

I have created a convolutional neural network from scratch in a programming language that is largely ignored for the task. The results of which show that there is significant potential for improving upon implementations in other languages. Using only 4 CPU cores, the training time was cut down to 8 hours, and the process involved 400 epochs, nearly 4 times that of an implementation with adaptive learning rates. If an estimate were to be made, this network has the potential to train in approximately 2 hours on just 4 CPU cores. Of course, other improvements may also be made to efficiency of the written code.

I think that with further research, this network could rival the speed and the accuracy of the more common Python implementations, possibly even surpassing them. Admittedly, much of this is only a hypothesis to be researched in the future. This paper only covers this initial implementation which is by no means a rival to other implementations.

**References**:

1. Le Cun, Y, et al. *Handwritten Digit Recognition with a Back-Propagation Network*. yann.lecun.com/exdb/publis/pdf/lecun-90c.pdf.
2. Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015
3. Zhang, Zhifei. “Derivation of Backpropagation in Convolutional Neural Network (CNN).” *Https://Pdfs.semanticscholar.org*, 18 Oct. 2016, pdfs.semanticscholar.org/5d79/11c93ddcb34cac088d99bd0cae9124e5dcd1.pdf.
4. *Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis - IEEE Conference Publication*, ieeexplore.ieee.org/document/1227801/
5. Agarwal, Mayank, and Mayank Agarwal. “Back Propagation in Convolutional Neural Networks - Intuition and Code.” *Becoming Human: Artificial Intelligence Magazine*, Becoming Human: Artificial Intelligence Magazine, 14 Dec. 2017, becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199.