Hardware Accelerated Convolutional Neural Networks

Kiuran Naidoo, Othniel Konan and Luke Schwartzkopff
Department of Electrical Engineering
University of Cape Town
P20 Rondebosch, Cape Town, South Africa

***Abstract*—The FANNtom Menace project is an attempt to create an FPGA accelerated neural network. The concern is not to facilitate the training of neural networks on the FPGA but rather to use traditional methods to train a model that can be verified and run on a PC. This model, once trained to an acceptable level of accuracy, should then be used to configure the FPGA using appropriate synthesisable code. Due to the highly parallel nature of a neural network a significant speedup is expected here over traditional computational hardware. Not only do we expect the FPGA to improve on latency due to each layer of the neural network being able to be computed in parallel, we also expect a great increase in throughput due to each layer of the neural network acting as a pipeline stage on the FPGA. The example used to test our implementation will be the classification of an image as either a cat or a dog (a binary classifier).**

The speedup of this solution will be compared to a golden measure implemented on a PC - specifically our C++ implementation setup and compiled for an ordinary x86 CPU. Both latency as well as throughput will be considered as well as the resource consumption on the FPGA (amount of BRAM required etc).

I. Introduction And Background Theory

Neural Networks are a biology-inspired form of machine learning that borrows the idea of a vast network of simple processing units (neurons). Neurons receive inputs from a previous layer, perform processing on these inputs and then pass on the output to neurons in the following layer. The way these neurons are connected between layers, the intermediate processing that is done and the non-linearities introduced vary from model to model and application to application[1].

Convolutional Neural Networks (*CNN*) are a very popular form of contemporary neural network used for analysing grids of data with their primary applications lying in image classification and other computer vision tasks. In this paper, we deal with the network in two distinct development phases: *Training* and *Synthesis and Implementation*. Using the *Keras Framework* we train the model before migrating and implementing the trained model architecture and weight values to an FPGA to conduct image classification on new images the FPGA is fed.

The purpose of this project is not simply to create an accelerator for the purposes of increased performance by leveraging FPGA parallelism - but also for the purposes of cost and power consumption for embedded systems. If a GPU or CPU powerful enough to reach this tier of performance were placed in an embedded system, its cost and power consumption would be huge. Thus, with an FPGA we can create practical embedded systems for purposes of image recognition and other functions which require the use of neural networks.

II. Methodology

The methodology utilised for creating the device described previously can be broken into two distinct components, the training methodology and the synthesis and implementation methodology. Those methodologies are illustrated in Figure II-B3 which shows the general workflow of the FANN design.

A. Training Methodology

In this aspect, a simple convolutional network was created with *Keras Framework* using the *Theano* for the back-end. The model created in this environment uses a modified *LeNet* model of a simple *CNN*. This architecture was chosen because it is simpler than other *CNN* architectures and as such would ease the model conversion to hardware. The training methods use employ the back-propagation and gradient-descent algorithms. Additionally, a

dropout layer was used in the model during training. This dropout layer sets 50 percent of the neurons in the layer to have a transfer function of 0 randomly every training epoch. This helps prevent over-fitting, especially when using small datasets.

B. Synthesis and Implementation Methodology

1) C Implementation: The *C* Code that is utilised to create the convolutional layers is based on macros which can be programmed using a higher-level abstraction. A sample of the convolutional layer is given in the appendix. These structures will take in values from the model generated in Keras and tranfirm this data into statically declared matrices that can be used for synthesis in *Vivado*.

2) Hardware Description Language: This stage makes use of the power of *Vivado Design Suite*. Vivado has a *High Level Synthesiser* which can convert any synthetisable *C* code into a *Hardware Description Language*. Vivado will be used to convert the Neural Network implementation made in *C* to a *Verilog* solution. Prior to that, the *C implementation* will need to be optimized on different levels.Those levels ranges from splitting the *BRAM* used to into independent blocks to unrolling or even pipelining the *for loops*. This section will be more experimental as various optimization parameters will modify to achieve the best resource utilization.

3) Testing: The *CNN* used will be trained with hundred of images. Those images will be used for testing the *FANN*. The images will be uploaded to the *FPGA's BRAM* during compilation time. The result of the classification of the image will be display on the *seven segments display*.
Ideally, the system would be able to stream images from a camera in real time and perform classification.

III. Design

In this section, the design aspects of the neural network as well as its implementation on the FPGA will be explained, discussed and justified.

A. Over-fitting and Under-fitting

Over-fitting and under-fitting are problems that arise in statistics when attempting to model a real world problem. Over-fitting means that the model is too tuned to the examples that have been used to derive it and as such will not be very accurate on other real-world examples that the model hasn't 'seen' before. Under-fitting means that the model is not complex enough to accurately predict the behaviour of the system it is attempting to model. These ideas are taken into consideration when designing the CNN architecture.

B. Elements of a Convolutional Neural Network

The following section describes the various layers and the functions that they perform in order to create an effective, fully functioning network. These layers were then judiciously selected and arranged in order to create a network most applicable for our purposes. How these layers were chosen and arranged is discussed in Design.

1) Convoutional Layer: A convolutional layer provides a way to filter certain elements of an image in order to create a mapping that has certain distinguishing characteristics that allow the rest of the neural network to be trained using these accentuated physical features. This means that the neural network obtains better results without having to utilise many fully connected layers which is useful considering that an image is a large input vector and the training process with many fully connected layers would be computationally intensive. The standard convolutional layer utilises a matrix operation that works on a block of data such that:

$$\sum_{n} x_i y_i \tag{1}$$

where x is the block of the image and y is the filter that the image is being convoluted with. The block of the image and the filter produce a dot product.

2) Max Pooling: The purpose of max-pooling is to decrease the matrix size of that is used in the operations of the neural network. It takes the maximum value of a neighbourhood of pixel values. If this is done too coarsely than the the features that are accentuated by convolution are lost and the features become irrevocably distorted[2]. This means that max pool mapping needs to occur at such in such way that reduces operational size while retaining fidelity. Generally, it is agreed by literature that a suitable neighbourhood of pixels is 2x2 [3]. Max pooling as an operation is not immensely expensive as it computes the maximum value from a series of pixels.

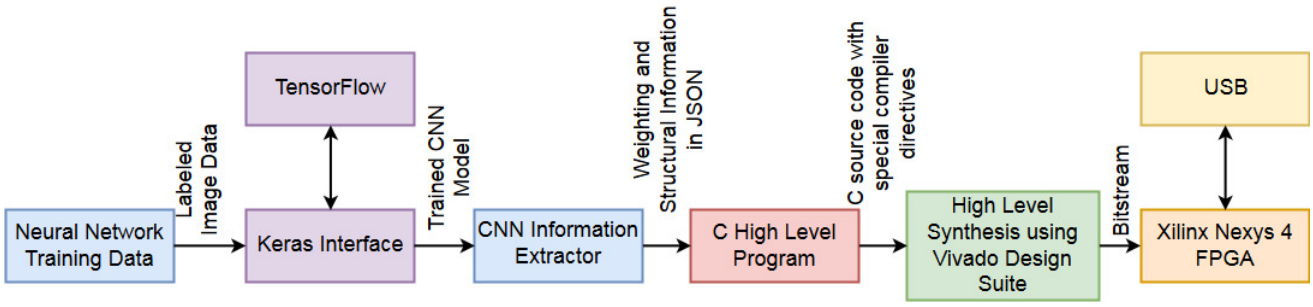

Fig. 1. General workflow

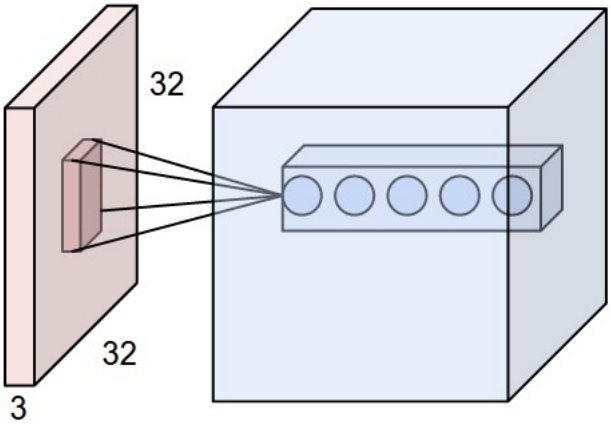

Fig. 2. An example of how convolutional layers work

3) Fully Connected Layer (Dense Layer): The fully connected layers result from the connection of convolutional neural layers to standard neurons in a dense structure. This process can be quite computationally challenging as all neurons are connected and the resulting complextiy of all the permutations can be expensive. However these permutations can be parallelised considerably by calculating all the permutations independently and this calculation should prove more effective on the FPGA compared to a normal general purpose CPU. The resulting permutations from a fully connected layer is of complexity $n!$ where n is the number of neurons in a layer.

C. Activation Layers (Non-Linear)

In general, non-linear activation functions are used to allow non-linearity in the model such that certain functions can be better approximated than if the system were fully linear. They generally also help to reduce over-fitting.

1) Sigmoid Function: The *Sigmoid* layer is often used to rectify the output of a neuron between 0 and 1. This is useful in the output layer (especially in binary classifiers) in order to quantify the probability of one of the output vectors being the correct output from the input vector. The sigmoid function is as follows: $S(x) = \frac{1}{1+e^{-x}}$

2) ReLu (Rectified Linear Unit): This non-linearity layer acts an activation for the the convoluted data neurons. The non-linearity can be expressed as a function of $max(0, p)$ where p represents an element of the output matrix. This means that an element in the matrix cannot not take on negative values. ReLu activation functions have been shown over time to allow for much faster training of networks due to their algorithmic simplicity while not offering significant accuracy deterioration with respect to sigmoid or tanh activation functions.

D. LeNet Architecture

The *LeNet* architecture is a very simple architectural structure for a *CNN*. It was first described in 1997 and forms the most basic practical CNN[3]. It has been used in the past to accurately classify handwritten digits. It was used as the basis for the custom CNN we designed for Cat/Dog classification.

E. Chosen Design

The chosen design for the CNN implemented is shown in Fig 2. It is the simplest architecture used that can give reasonably accurate results. The simplicity is important for successful FPGA implementation. The max pooling layers are used to reduce dimensionality as well as to stop overfitting. ReLu activiation functions are largely used in order to quicken training and classification time. The

final layer utilises a flattening as well as a densely connected layer in order to concatenate the convolutional spaces into a 1D space that can be used to generate a binary output. Finally, the sigmoid function is used to rectify the output between 0 and 1.

F. Hardware

The *FANN* runs on an *ARTIX-7*. It takes its input images from a *USB*. Two buttons are used to select the images from the *USB* to be sent to the *FANN's RAM*. After the image is being processed, the *FANN* display the output using the on-board *seven segment display*. Figure III-F shows the hardware connection of the *FANN*.

G. Software

Figure III-G shows the steps performed for image classification.

IV. Proposed Development Strategy

The current development plan for *FANN* consists of multiple phases. These can be listed as:

1) Create a neural network model for image classification utilising the Keras Framework
2) Implement the blocks to model the elements of a neural network in hardware utilising C and Verilog
3) Create an interface between the Keras Framework and Vivado to exchange weight values
4) Utilise Vivado to synthesise the neural network using the weighting values from Keras
5) Test new data on CPU, and FPGA and compare results.

As a final product, the FANN would have a Neural Network performing image classification based on trained data used in the Neural Network. It would require two inputs: an image coming from camera and a file containing the trained data for the network. The output of the result would still be displayed on the seven segments display. For efficiency of the process, the trained data would be imported only when it is a new data.

Furthermore it is important to note that the digital accelerator being created is the first of its kind and should it be successful, further development around this topic will take place. As such the main points of this project can be refined in order to create a better experience for the user. An envisioned final use for the *FANN* would to make the entire learning process transparent to the user and create a product that was a neural network on a chip where the user could download the weights and these would be automatically placed on the FPGA. This means that the product could be used for multiple purposes.

Finally the *FANN* could in the future do the entire neural network on an FPGA. This means that the device could complete backward propagation as well as feed-forward. This is in contrast to the current version which only accelerates the feed-forward model based on offline training.

V. Experiments and Results

The *FANN* is being implemented on a *Nexys 4 DDR*. This FPGA is an *Artix-7* device running with an internal clock of $450MHz+$ *(although we will only use 100Mhz)*, a *DDR2* of $128MiB$, a *Block RAM* of $4,860$ and $15,850$ *Logic Slices*. The performance of the *FANN* will be compared to a *CNN* using a similar implementation running on a$1.5Ghz$ 2 core x86-64 CPU (the golden measure).
The performance test will be based on the:

- Percentage of error
- Latency
- Resource utilization

A. Golden Measure

The golden measure used for this experiment is the *C code* mentioned in section II-B1. It was decided that this would be used instead of the python implementation as it is far more similar to what will be run on the FPGA - giving a more relevant comparison. The C code that was used for Vivado synthesis was retooled to work on a general purpose computer. Using the standard Clang LLVM compiler the C code was built for the x86-64 architecture under MacOS Sierra (10.12.5). The compiled program was run and the standard C time library was utilised to obtain a benchmark of runtime performance was established. The specifications for the MacBook Air used was: 1.5GHz Intel i5, 8GB RAM.

The C code implementation was created by recreating the neural network structure in C utilising *for* loop structures. In addition Python scripts were made that dumped the weight and image data into C style arrays that could be included with the C code to create a full feedforward neural network. This code required minimal changes so

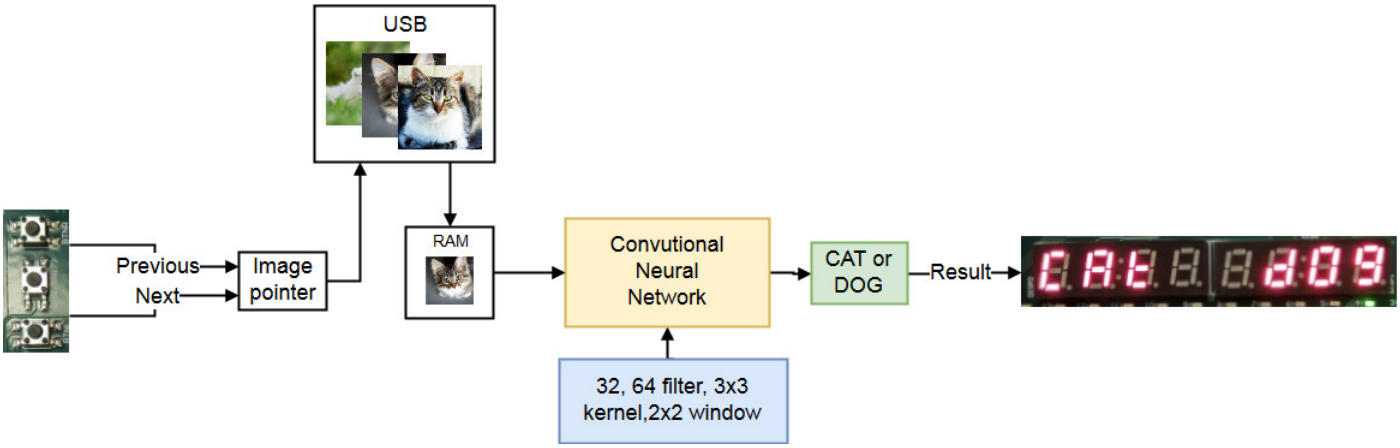

Fig. 3. Prototype design of the FANN

that it could be synthesised on the FPGA using Vivado. The code described here can be found at: *https://github.com/kiuran/FANNtomMenace/*

B. Percentage of error

The *CNN* trained achieve a certain performance. This experiment consists of evaluating the veracity of the result of the *FANN* compared to the one obtained from the *C implementation*.

C. Latency

The speedup of the *FANN* will be analysed. The latency of the *CNN* on the *FANN* will be timed using a pulse. By setting a pin high then low before and after the *CNN* process, the latency of the process would be obtainable by probing the corresponding pin.

D. Resource utilization

Resource utilization of the *FANN* will be optimised using *Vivado*. Vivado offers certain operations to maximize optimization of the FPGA. As an example, *for loops* can be flattened, pipelined, unrolled, ... These optimization are entirely done in the C implementation. Figure V-D illustrates the example described above.

E. Expected result

The Neural Network used has roughly 10 layers of significant computational demand. Each layer will be pipelined, and therefore expect a throughput increase of **x10** in the pipelining process. Three of the layers are *convolutional layers* containing **six nested for loops each**. By the use of optimization mentioned in section II-B2, each for loop could be unrolled or even pipelined to various extents. By means of unrolling, the speedup in these layers can be improved by n^6, n being the number of loops per section. The value of n will depend on the availability of resources, but in a less pessimistic approach, n could be 2 allowing a speedup of 32 for these layers. The *BRAM* block can be optimized. The BRAM can be partitioned into many blocks. Ideally, it would be fully partitioned decreasing drastically memory access latency.
A decrease in latency due to intra-layer parallelism is estimated to be roughly 10x based on other studies and the availability of CLBs on the Nexys 4. Thus, we can expect the overall throughput as the result of increased intra-layer parallelism and layer pipelining *FANN* to be at least $10 \times 10 = 100$.

The predicted performance of the *FANN* can only be broadly speculative for now as is difficult to perform exact or even very good approximation as doing so will require knowing exactly how the logic slices of the FPGA will be used - meaning specific deterministic knowledge of the workings of Vivado is needed.

VI. Results

Initially, the C implementation synthesised successfully but could not be transferred to the *FANN* due to the small amount of BRAM on the FPGA. Streaming in the weight values would be absolutely

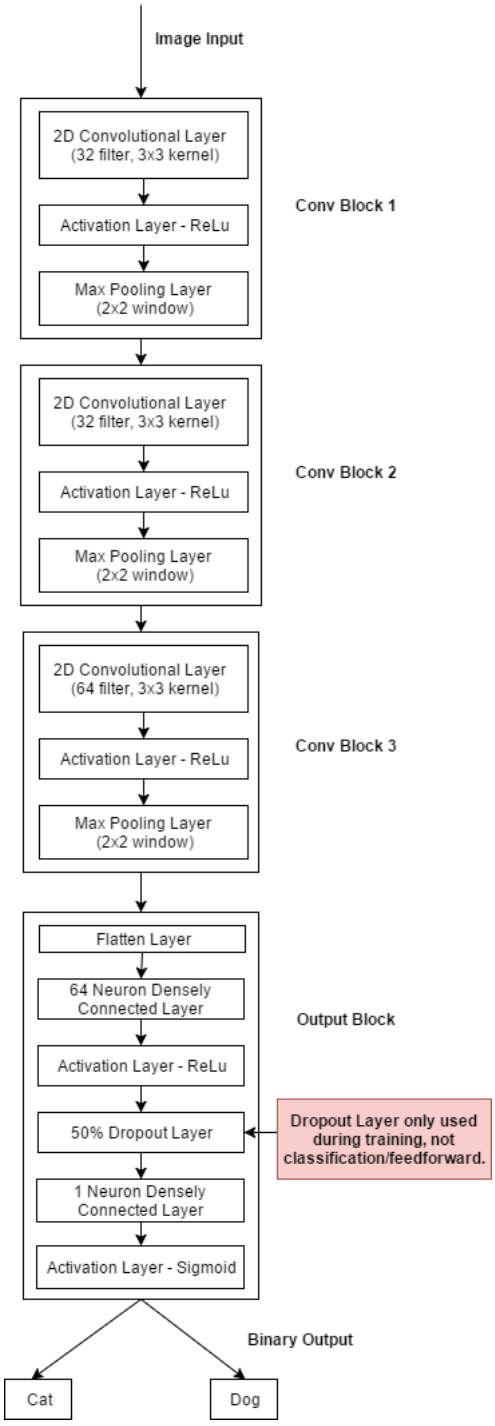

Fig. 4. Design of Neural Network using a reworked LeNet Structure

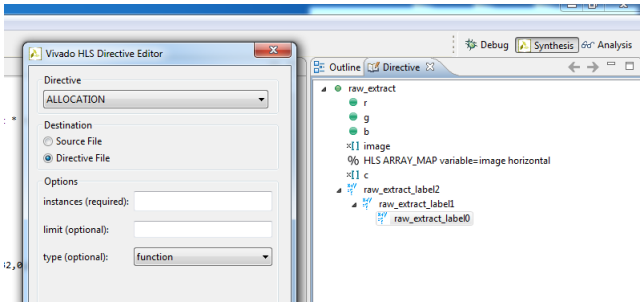

Fig. 5. Vivado HLS Directive Editor

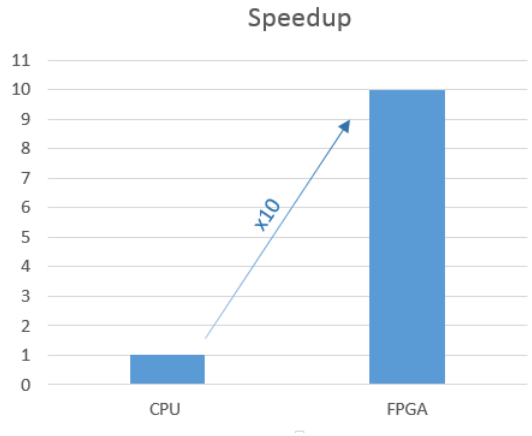

Fig. 6. Expected result

pointless and incredibly slow. A 5*mb* *HD*5 weighting file was produced - 16*mb* when uncompressed into C arrays. This resulted in around a 3000% BRAM overusage (Figure VI).

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	0	4899	3864
FIFO	-	-	-	-
Instance	-	34	6250	7629
Memory	7848	-	64	0
Multiplexer	-	-	-	2159
Register	-	-	2698	-
Total	7848	34	13911	13652
Available	270	240	126800	63400
Utilization (%)	2906	14	10	21

Fig. 7. Synthesis summary

A. Remedial actions

The solution proposed was to sacrifice the neural network complexity - thus reducing its size in a trade-off with classification accuracy.

1) Attempt 1: Convolutional layer and dense layer have their filter sizes halved (to 16,16, 32, 32). This results in a good start as the HD5 file is reduced to 1.4mb, but it is not nearly good enough. Accuracy remains acceptable (Figure VI-C).

2) Attempt 2: Convolutional layer and dense layer have their filter sizes halved again (to 8, 8, 16, 16). This results in another good reduction as the HD5 file is reduced to 340kb, but it is still not good enough. We estimate we require a weight file of under 150kb to suitably synthesise. Accuracy remains acceptable (Figure VI-C).

3) Attempt 3: Convolutional layer and dense layer have their filter sizes halved again (to 4, 4, 8, 8). This results in another good reduction as the HD5 file is reduced to 110kb. This is possibly small enough to synthesise. Accuracy is now possibly unacceptable for some purposes though - with an error rate of greater than 25%. However - Trials showed that BRAM usage was still at 214% (Figure VI-C).

4) Attempt 4: Convolutional layer and dense layer have their filter sizes halved again (to 2, 2, 4, 4). This results in another good reduction as the HD5 file is reduced to 49kb. Accuracy is again reduced to around 67%. This was barely too large to synthesise at 113% BRAM capacity (Figure VI-C).

5) Attempt 5: Convolutional layer and dense layer have their filter sizes halved again (to 1, 1, 2, 2). This doesnt reduce the size much (to 35kb) and results in catastrophic underfitting essentially resulting in a very expensive coin-toss network. The training trends show that the data is underfit and any short-term gains are example specific. **This does fit in BRAM though!** (Figure VI-C).

B. HD5 File Size vs CNN Accuracy

Figure VI-C shows the HD5 file size versus the CNN accuracy. There are clearly diminishing returns to simplifying the network. The rate at which the returns diminish is problem dependent.

C. Golden Measure vs FPGA

The results for the golden measure and the eventual FPGA implementation are given below in (Figure VI-C). The particular network implemented on the FPGA was the same simple and inaccurate network used on the FPGA in order to have a fair comparison.

It can be seen that there is a speedup of roughly 10x for a single iteration, and roughly 20x for 1000 iterations. This indicates that the primary performance improvement is because of intra-layer parallelism (offering roughly 10x) while pipelining offers a 2x overall performance increase. This indicates that the FPGA was constrained and could not implement all of the neural network's layers simultaneously. This is confirmed by the CNN implementation barely meeting synthesis memory limits.

VII. Conclusion

FPGA implementations of neural networks are certainly viable and beneficial. If a larger FPGA with more memory was used - the original network could have been implemented and a high-accuracy model could have been implemented. If additional memory beyond this was available, the FPGA would be able to more fully roll out the loops and parallelise the code - further increasing performance as well.

In this case, the network used was too small to tackle this particular problem. If a more simple recognition problem was tackled, it is more likely that an accurate neural network could have fit in the FPGA's BRAM. The inaccurate network was used as proof of concept and shown to offer good speed increases.

FANNs are practical and beneficial with respect to a CPU based implementation provided that the FPGA is powerful enough and the CNN structure of the particular problem is simple enough.

VIII. References

[1] I.Goodfellow, Y.Bengio and A.Courville, *Deep Learning*. Cambridge, Mass: The MIT Press, 2017.

[2]"Dogs vs. Cats — Kaggle", Kaggle.com, 2017. [Online]. Available: https://www.kaggle.com/c/dogs-vs-cats/data. [Accessed: 17- Jun- 2017].

[3] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems*. 2012.

[4]"CS231n Convolutional Neural Networks for Visual Recognition",

	FPGA	CPU
Num. Loops	1000	1000
Time	0.76s	16s

	FPGA	CPU
Num. Loops	1	1
Time	2.16ms	22ms

Fig. 8. FPGA vs Golden Measure Performance Comparison

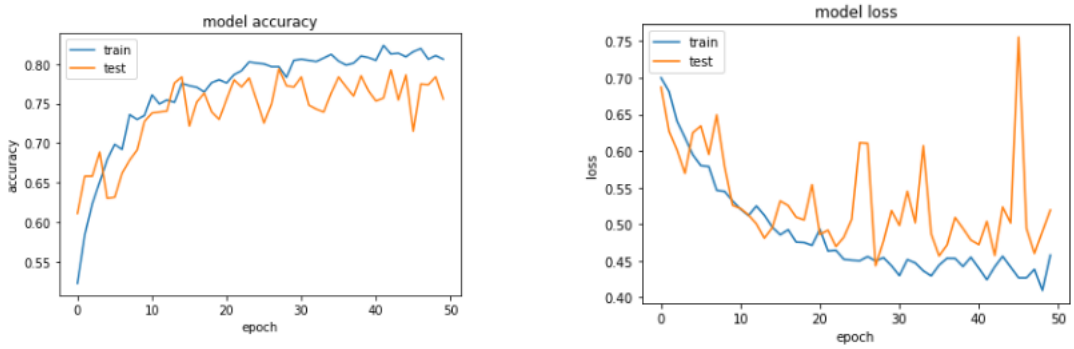

Fig. 9. Model accuracy and model loss of the CNN using filter size of 16,16, 32, 32

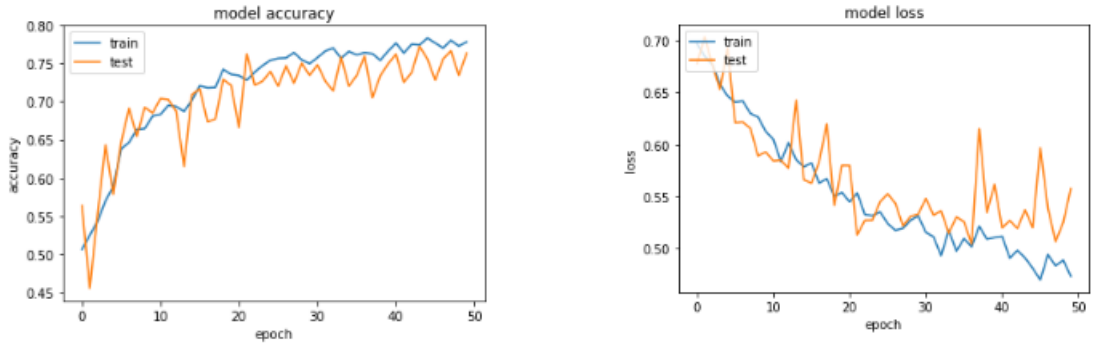

Fig. 10. Model accuracy and model loss of the CNN using filter size of 8,8, 16, 16

Cs231n.github.io, 2017. [Online]. Available: http://cs231n.github.io/convolutional-networks/. [Accessed: 17- Jun- 2017].

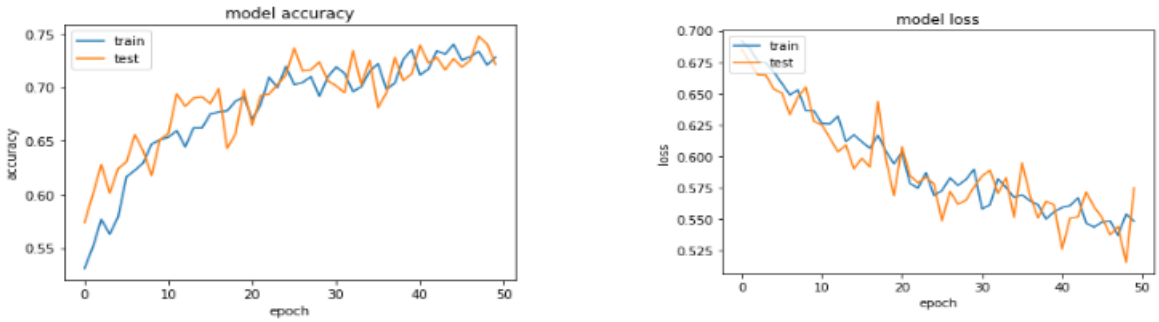

Fig. 11. Model accuracy and model loss of the CNN using filter size of 4, 4, 8, 8

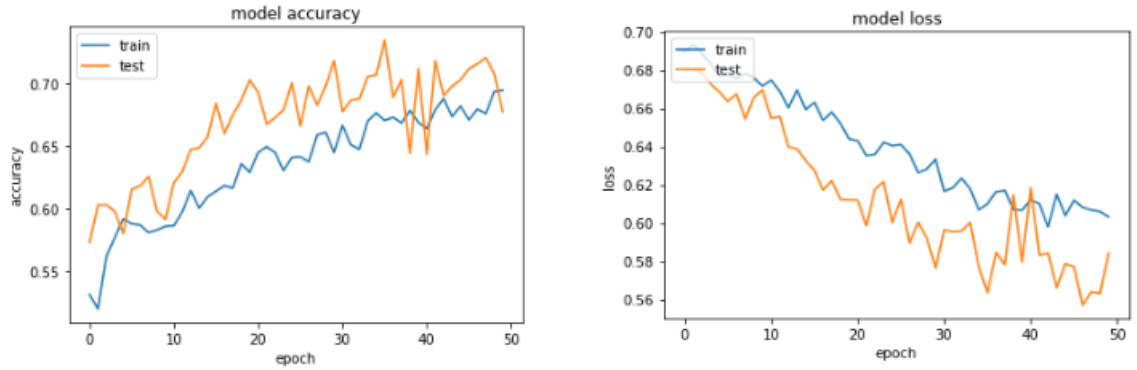

Fig. 12. Model accuracy and model loss of the CNN using filter size of 2, 2, 4, 4

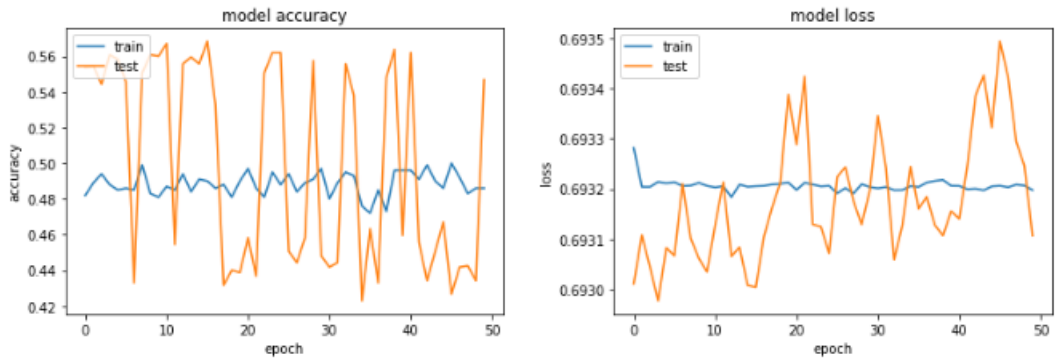

Fig. 13. Model accuracy and model loss of the CNN using filter size of 1, 1, 2, 2

Trial Num	File SIze (kB)	Accuracy
1	5000	83
2	1400	78
3	340	74
4	110	70
5	49	67
6	35	50

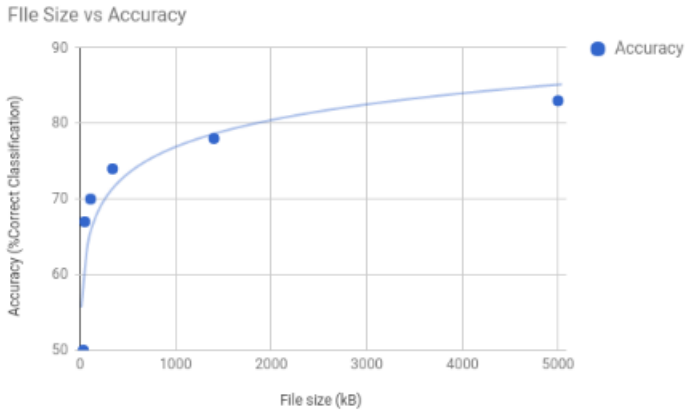

Fig. 14. HD5 File Size vs CNN Accuracy