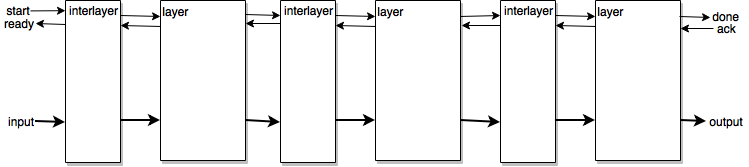
**Architecture**

In order to facilitate modularity and extensibility, a simple high-level architecture consisting of a sequence of linked “layer” and “interlayer” blocks is proposed (figure 1).

**Figure 1.** Diagram showing the chosen high-level architecture. A thin arrow means a one bit signal and a bold arrow means a bus of several bits.

In this architecture, the layer block is charged with accomplishing the actual calculations of a neural network layer, while the interlayer serves as an adapter to ensure proper communication and data propagation between two consecutive layers. Note that there is no global controller; every layer block carries its own controller and coordinates with the adjacent layers using control signals (arrows from *start* to *done* and from *ack* to *ready* on figure 1, some types of layers may use additional control signals). This serves to enable pipelining of multiple inputs inside the circuit; indeed, the general idea is that data can only progress from a layer to the next when the previous layer is *done* (has finished its previous calculation) AND the next layer is *ready* (has already passed forward the data it was processing). This simple protocol, generally mediated by the interlayer, enables calculations to be performed simultaneously in different layers while preventing any collision.

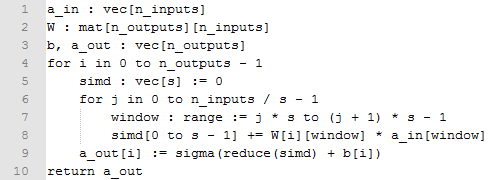
Since the system is so modular, it can in theory accommodate any number of different types of layer. Two types, fully-connected layers and convolutional layers, are presented below.

Fully-connected layer

Fully-connected layer blocks (abbreviated to *fc layer*), like their software equivalent, perform the vector-matrix operation

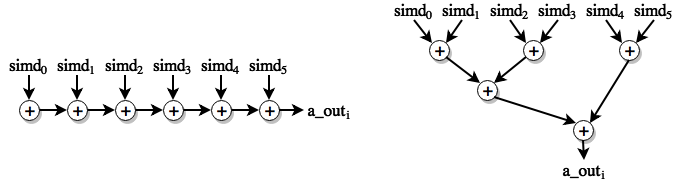


where **a**i is the vector of inputs or activations from the previous layer, **W**i is the matrix of weights, **b**i is the vector of biases, σi is the layer’s activation function applied element-wise to each vector component and **a**i+1 is a vector of the layer’s activations. Concretely, the vector-matrix multiplication between **a**i and **W**i can be implemented in many different ways. A sequential implementation where each element is multiplied and added one after the other is possible, but suboptimal given the FPGA’s high ability for parallelism. On the opposite end of the spectrum, a fully parallel implementation which computes all outputs at once, while theoretically possible, is prohibitively expensive in terms of FPGA resource utilization. Between these two extremes, alternatives with moderate levels of parallelism exist. For example, it would be possible to, at each iteration, multiply an input with several weights and accumulate the result, thereby producing several outputs at the same time. An inconvenient of this approach is that either an activation function block must be instantiated for each computed output (which could be expensive depending on the function) or the result data must be serialized before being passed through the activation function (which means reduced parallelism). Another possibility is to multiply *s* inputs with weights at each iteration to get *s* values, each of which then accumulates with the product of another input until all inputs have been multiplied. The *s* values are then summed together and the activation function is applied to the result to give one activation. This process is repeated until all the output activations have been produced. This algorithm can be illustrated by the pseudocode given in listing 1.



**Listing 1.** Computation for a fully connected layer with parallelization on the inputs.

This is the method which was chosen to implement the fc layer block, due to the fact that it allows the level of parallelization to grow without duplicating the activation function or introducing a serialization bottleneck. However, a few compromises are worth mentioning. First, in order for the computation to cover the inputs exactly, the number of inputs has to be a multiple of *s*. If needed, this can be solved by padding the inputs and weights with zeros as necessary. Second, an implementation of function *reduce*, which adds together the values of *simd*, has to be provided. A naïve version of it would add each value one after the other, yielding an *O*(s) latency (path from first value to result) and an *O*(s) resource utilization (number of adders). However, it is also possible to leverage the parallel nature of the FPGA and arrange the adders in a binary tree to get an *O*(log s) latency and an *O*(s) resource utilization. In order to maximize performance, this last solution was chosen.



**Figure 2.** Illustration of both possible *reduce* implementations for *s* = 6. The naïve version (left) has 6 adders on its longest path, while the chosen solution (right) has 3.

Regardless of the architecture, an implementation of an activation function has to be provided. Rectified linear units (ReLU) are rather trivial to implement on FPGA, but other common functions, such as the sigmoid, aren’t. The sigmoid is characterized by the formula



yielding a value between 0 and 1. Immediately, we notice two problems for an efficient FPGA implementation: an exponential function and a division. While methods exist to perform those computations (e.g. Goldschmidt’s method for division), a faster, more straightforward method would be preferred. This work solves this problem by linearly interpolating the sigmoid between points sampled at intervals of a power of 2. More precisely, an array is pre-populated with the values and slopes of the right half of the sigmoid at intervals of Δz = 2k (where k is an integer) up to z = 6 (any higher z gives a result of 1). Then, given a fixed point representation of numbers, the most significant bits of the input (down to the bit with value 2k) can be extracted to get a value *z’* which can be reinterpreted as an index *i* into the array when the fixed point is elided. Then, the interpolation can be performed such that



. If *z* is negative, then it is first made positive before the operation, and, thanks to the sigmoid’s symmetry, the result can mirrored relative to *σ*(*z*) = 0.5. It bears mentioning that other fast methods exist for calculating such functions; for instance, the related hyperbolic tangent can be computed using the discrete cosine transform [1].

Using all the strategies expounded previously, a hardware architecture for the fc layer can be proposed (figure 3).

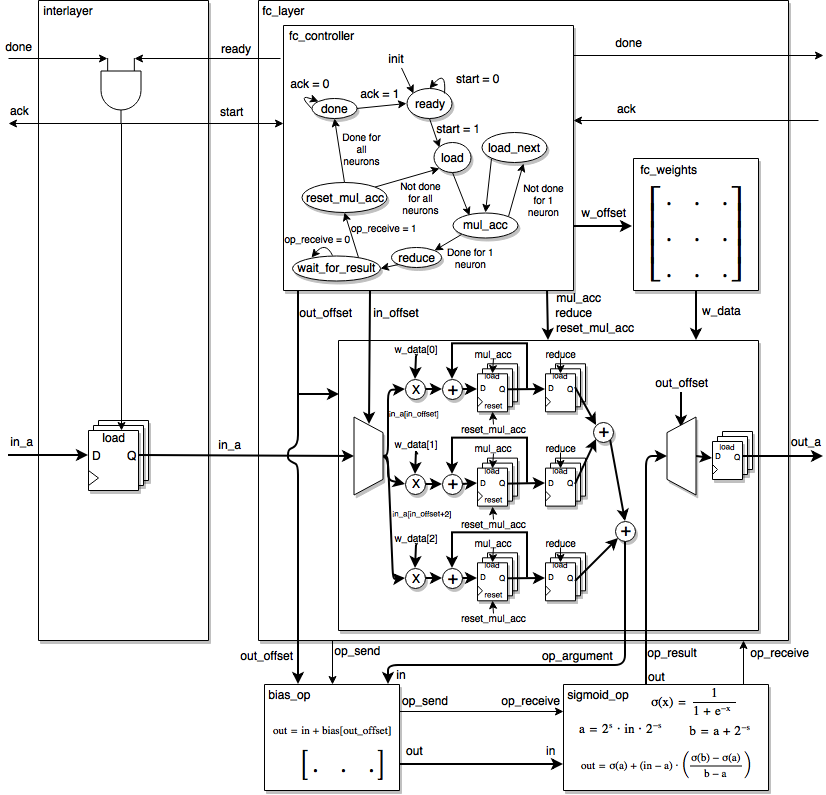


Figure 3. Hardware architecture for the fc layer and the interlayer between two fc layers.

First of all, both the *ready*-*done* protocol for synchronization between layers and the process described in listing 1 are directly recreated in the interlayer’s top part and in the controller’s state machine. Furthermore, the “simd window” used to perform the computation is directly represented as the array of 3 multiplier-accumulators (assuming *s* = 3) and the *reduce* function is represented by the two adders to the right of said array.

In order to allow high design flexibility, all functions that act on the result of the vector-matrix multiplication (including the addition of the bias and the activation function) are implemented as blocks connected to (but distinct from) the main layer block. As a consequence of this flexibility and since each block can take any number of cycles to complete its computation, a simple mechanism to follow the flow of information was added; a block starts its computation when it receives a ‘1’ on its *op\_receive* port and emits a ‘1’ on its *op\_send* port when the result of the computation is available to the next block. The fc layer’s controller can therefore safely wait in the *wait\_for\_result* state until the *op\_receive* signal has been received.

An important detail which has been implied during the discussion of the sigmoid implementation is that data is represented as fixed point numbers. Floating point numbers are the most common data format for neural networks in CPUs and GPUs, but most FPGAs do not support them natively. While it would be possible to provide an implementation of floating point arithmetic on FPGA, it was deemed that the core advantage of floating point numbers, dynamic range, was probably not worth the loss of performance in the context of a feedforward network. Beyond that, another advantage of fixed point numbers is that the sizes of their integral and fractional parts can be trivially parametrized. Such parametrization provides an additional way to fine-tune resource utilization and result precision which is available in a more limited capacity in CPUs and GPUs. Finally, while it could in theory be possible to normalize fixed point numbers based on statistics (for example, bring values between 0 (minimum) and 1 (maximum)), this implementation instead opts to preserve the unscaled values, to resize the number as necessary during an operation and to provide points where the designer can specify a new bit size (e.g. at the output of the layer). An inconvenient of this approach is that, when using an activation function like ReLU which does not restrict values between 0 and 1, and since it is not realistic to keep an ever increasing number of bits between layers, the designer has to carefully choose the number of bits in the integral part to preserve. However, this also avoids a lot of implementation complexity and potential performance bottlenecks (statistics on inputs and rescaling operations).

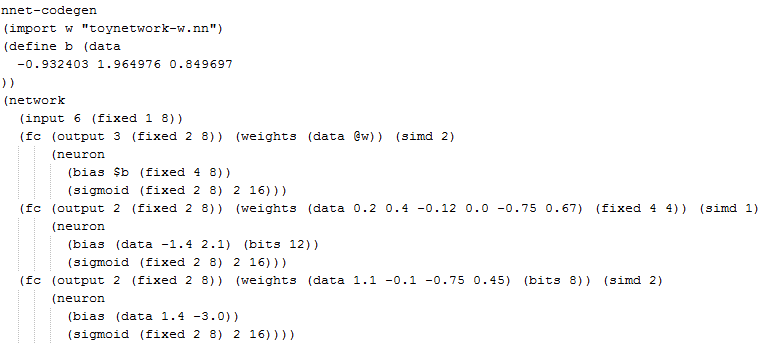
Convolutional layer

……………………………………stuff……………..

**Methodology**

In order to have full control over the design, hardware description languages VHDL and Verilog were used to implement the various blocks necessary. Furthermore, parametrization (with generics in VHDL and parameters in Verilog) was heavily employed to enable extra flexibility while allowing connected blocks to retain a compatible port interface. While this is appropriate for blocks which seldom change in a radical manner (like the conv and fc layers), it is less so for the network as a whole. Indeed, in order to avoid having to manually change parameters and reconnect every wire and port every time a slight architectural change is made, it would be highly desirable to have a high-level description of the network which could be converted into low-level HDL code. In theory, such a high-level description could be found in code using existing neural networks libraries (e.g. Keras). However, many important parameters for our architecture (fixed point size, simd width, etc.) are not generally (or obviously) specifiable in such libraries and it has therefore been decided to create a custom format.

This format, hereafter referenced as “.nn files”, uses a syntax based on s-expression lists, which makes parsing quite simple while allowing arbitrarily complex expressions. Any number of networks can be listed, and each network specification contains an input specification followed by an arbitrary number of layer specifications. Those input and layer specifications expose all the necessary parameters in a straightforward manner. .nn files are also endowed with a very simple macro system which inserts s-expressions (“define” directive) and the contents of other files interpreted as s-expressions (“import” directive) into the remainder of the file. A concrete example of such a file is given in listing 2. More details about the format can be found in the associated git repository [2]. A command line/library tool made in C++ for converting the .nn format to VHDL is also available in the git repository (source and makefile available at https://github.com/fireyoshiqc/stage2017/tree/master/Gabriel/codegen).



**Listing 2.** Example of a .nn specification of a network of fc layers with structure 6-3-3-2.

[FOR RESULTS: USE FC ARCHITECTURE ON ZEDBOARD? DEPENDS ON WHETHER CONV NETWORK WILL WORK]

**References**

1. Abdelsalam, A.M. et al. Accurate and Efficient Hyperbolic Tangent Activation Function on FPGA using the DCT Interpolation Filter. arXiv:1609.07750, 2016.
2. Demers, G. and Boulet, F. The Neural Network (NN) file format. https://github.com/fireyoshiqc/stage2017/blob/master/Gabriel/codegen/README.md