PARAMETRISED NEURAL NETWORK DESIGN AND COMPILATION INTO HARDWARE

Wayne Luk, Adrian Lawrence, Vincent Lok, Ian Page and Richard Stamper

INTRODUCTION

Most artificial neural networks consist of one or more arrays of components, each of which is obtained by replicating a few simple processing elements connected together in a uniform manner. This paper illustrates the use of Ruby, a language of relations and functions, for describing such networks and for implementing them in hardware. Our objective is to enable designs to be rapidly realised and evaluated.

Ruby has a number of generic relations – such as replication and transposition – that can be used to generate interconnection patterns commonly found in neural systems. It also has a small set of constructors for building composite circuits from simpler ones. These features enable many neural architectures, for instance multi-layer perceptrons and Hopfield networks, to be captured very concisely in Ruby.

We shall also discuss how Ruby can be used to derive, from a simple expression, a complex parametrised representation for a family of architectures. For instance, a parallel design such as the perceptron network shown in Figure 5a can be systematically transformed into a serial architecture like that in Figure 7. This approach permits developing from a high-level description a range of designs with different performance trade-offs, and the features of such designs can be summarised quantitatively – see Table 1 for an example. These tables can be used to find an appropriate implementation for a particular application, given the performance required and the availability of hardware resources.

In the next section we shall provide an overview of our approach, further details of which can be found in Jones and Sheeran (1990) and Luk (1992).

DESIGN REPRESENTATION

A design will be represented by a binary relation of the form $x\ R\ y$ where x and y represent the interface signals and belong respectively to the domain and range of R. For instance, a squaring operation can be described by $x\ sqr\ y \Leftrightarrow x^2 = y$ or, more succinctly, by $x\ sqr\ x^2$.

Transformed or composite circuits are usually described by functions which map one or more relations to a relation. As an example, the converse of R is defined by $x\ R^{-1}\ y \Leftrightarrow y\ R\ x$. It can be considered as a reflected version of R.

Two components Q and R can be connected together if they share a compatible interface s which is hidden in the composite circuit (Figure 1a): $Q;R$ is given by $x\,(Q;R)\,y \Leftrightarrow \exists s.\ (x\ Q\ s)\ \&\ (s\ R\ y)$. For instance, $x\,(sqr;sqr)\,x^4$. This is, of course, just the common definition of relational composition. It is simple to show that relational composition is associative, and that $(Q;R)^{-1} = R^{-1};Q^{-1}$. A collection of such theorems constitutes a calculus for reasoning about designs, which can usually be used without the need to refer to the meaning of symbols such as Q and R.

As shown later, many useful theorems can be expressed in the form $R = P^{-1};Q;P$. The pattern $P^{-1};Q;P$ – in words 'Q conjugated by P' – will be abbreviated as $Q\backslash P$.

VLSI for Neural Networks and Artificial Intelligence
Edited by J.G. Delgado-Frias, Plenum Press, New York, 1994

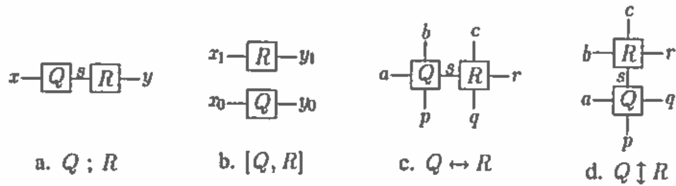


Figure 1 Binary compositions.

Parallel composition of two components Q and R, given by $[Q, R]$ (Figure 1b), represents the combination with no connection between Q and R. Given that a tuple (an ordered collection) of signals are enclosed by angle brackets, parallel composition can be defined by $\langle x_0, x_1\rangle\, [Q, R]\, \langle y_0, y_1\rangle \Leftrightarrow (x_0\ Q\ y_0)\ \&\ (x_1\ R\ y_1)$; so $\langle x, y\rangle\, [sqr, (sqr; sqr)]\, \langle x^2, y^4\rangle$. One can easily check that $[P, Q]; [R, S] = [P; R,\ Q; S]$, and that $[P, Q]^{-1} = [P^{-1}, Q^{-1}]$.

There are several operations involving pairs of signals that we will require. First of all, given that ι is the identity relation, we have the abbreviations $\mathsf{fst}\, R = [R, \iota]$, and $\mathsf{snd}\, R = [\iota, R]$. Next, the relation *fork* can be used to duplicate a signal, since x *fork* $\langle x, x\rangle$. The projection relations π_1 and π_2 extract an element from a pair: $\langle x, y\rangle\ \pi_1\ x$ and $\langle x, y\rangle\ \pi_2\ y$. Finally, we need to be able to swap the elements of a pair: $\langle x, y\rangle$ *swap* $\langle y, x\rangle$. Examples of theorems involving these operations include $\mathsf{fst}\, Q\ ;\ \mathsf{snd}\, R = \mathsf{snd}\, R\ ;\ \mathsf{fst}\, Q = [Q, R]$ and $[Q, R]\backslash swap = swap\ ;\ [Q, R]\ ;\ swap = [R, Q]$. It should also be clear that $fork; [\pi_1, \pi_2] = [\iota, \iota]$, and that $\pi_1{}^{-1}; \pi_1 = \iota \neq \pi_1; \pi_1{}^{-1}$ in general and similarly for π_2.

A rectangular component with connections on every side is modelled by a relation that relates 2-tuples, with the two components in the domain corresponding to signals for the west and north side and those in the range corresponding to signals for the south and east side. Such components can be assembled together by the beside ($\leftrightarrow$) and below ($\updownarrow$) operators (Figure 1c and Figure 1d): $\langle a, \langle b, c\rangle\rangle\ (Q \leftrightarrow R)\ \langle\langle p, q\rangle, r\rangle \Leftrightarrow \exists s.\ \langle a, b\rangle\ Q\ \langle p, s\rangle\ \&\ \langle s, c\rangle\ R\ \langle q, r\rangle$ and $Q \updownarrow R = (Q^{-1} \leftrightarrow R^{-1})^{-1}$. Theorems that have been proved for beside can readily be adapted for below, and vice versa.

It is also useful to have a conjugate operator for pairs: $Q \backslash\backslash [R, S] = [S^{-1}, R^{-1}]; Q; [R, S]$. Given that the conjugate operators have a lower precedence than all other operators except relational composition, one can show that $Q \backslash\backslash R = R^{-1}\backslash swap\ ;\ Q\ ;\ R$, and that $\mathsf{snd}\, Q^{-1}; R; \mathsf{fst}\, Q = R \backslash\backslash (\mathsf{fst}\, Q)$. We shall also use the abbreviations $\mathsf{fsth}\, R = R \leftrightarrow swap$, $\mathsf{fstv}\, R = R \updownarrow swap$, and $\mathsf{fstvh}\, R = \mathsf{fstv}(\mathsf{fsth}\, R)$.

Repeated compositions

Let us now look at the ways that we describe one- and two-dimensional arrays of components. Repeated relational composition of a given relation R cascades together copies of R (Figure 2a); it is defined inductively by the equations $R^1 = R$ and $R^{n+1} = R^n\ ;\ R$.

Repeated parallel composition, $\mathsf{map}\, R$ (Figure 2b), relates two equal-length tuples such that the corresponding elements of the tuples are related by R (note that $\#x$ denotes the number of elements in tuple x):

$$\text{if } \#x = \#y = N \quad \text{then} \quad x\ (\mathsf{map}\, R)\ y \ \Leftrightarrow\ \forall i : 0 \le i < N.\ x_i\ R\ y_i.$$

For clarity, on some occasions we shall make explicit the number of R's in a map and write it as $\mathsf{map}_N\, R$. This expression can be considered to be an abbreviation of $\mathsf{map}\, R \backslash N$ where N is the identity relation on N-tuples.

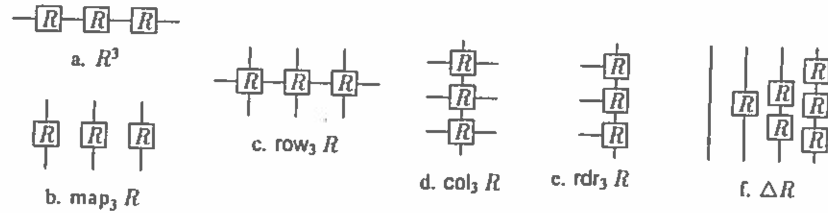


Figure 2 Repeated compositions.

A row of components (Figure 2c) is built from repeated composition of beside, and can be described by

$$\text{if } \#x = \#y = N \text{ and } ax = \langle a, x\rangle \text{ and } yb = \langle y, b\rangle \text{ then}$$
$$ax\ (\mathsf{row}\, R)\ yb \ \Leftrightarrow\ \exists s.\ (s_0 = a)\ \&\ (s_N = b)\ \&\ \forall i : 0 \le i < N.\ \langle s_i, x_i\rangle\ R\ \langle y_i, s_{i+1}\rangle.$$

A column of components (Figure 2d) can be obtained from $\mathsf{col}\, R = (\mathsf{row}\, R^{-1})^{-1}$. A degenerate form of col, called a right-reduction (rdr, Figure 2e), is also frequently used; it describes the result of applying a binary operation on a tuple in a right-associative manner, like $\langle\langle a, b, c\rangle, z\rangle\ (\mathsf{rdr}\ add)\ x \Leftrightarrow a + (b + (c + z)) = x$. Right reduction can be defined by $\mathsf{rdr}\, R = \mathsf{col}\,(R\ ;\ \pi_1{}^{-1})\ ;\ \pi_1$. The corresponding degenerate version of row, known as left-reduction, is given by $\mathsf{rdl}\, R = \mathsf{row}\,(R\ ;\ \pi_2{}^{-1})\ ;\ \pi_2$.

We shall also need the relation ΔR (Figure 2f) which relates two equal-length tuples such that their i-th elements relate to each other according to R^i. The Δ operator is useful for formulating distributive theorems for col: on the assumption that $[A, B]; R; \mathsf{snd}\, C = R; \mathsf{fst}\, B$, one can show that

$$\mathsf{col}_n\,(\mathsf{snd} B; R) = [\Delta A,\ B^n]\ ;\ \mathsf{col}_n\, R\ ;\ \mathsf{snd}\, \Delta C. \qquad (1)$$

The use of this equation in pipelining designs will be explained later.

Sometimes we shall need to interleave an array of components from two equal-length tuples. This can be achieved by *zip*, given by $\langle x, y\rangle\ zip\ z \Leftrightarrow \forall i : 0 \le i < N.\ \langle x_i, y_i\rangle = z_i$, on the assumption that $\#x = \#y = \#z = N$. For instance, $\langle\langle 1, 2, 3\rangle, \langle 4, 5, 6\rangle\rangle\ zip\ \langle\langle 1, 4\rangle, \langle 2, 5\rangle, \langle 3, 6\rangle\rangle$.

Sequential circuits and serialisation

So far we have been using relations to model a static situation – the steady state behaviour of a circuit at a particular instant of time. To deal with sequential circuits, an expression is interpreted as a relation that relates a *stream* in its domain to a stream in its range. For our purpose, a stream can be considered to be a doubly-infinite tuple containing data at successive clock 'ticks'. Notice that the clock is an abstract means for specifying data synchronisation, and it may be realised either by a global synchronous clock or by some hand-shaking mechanism.

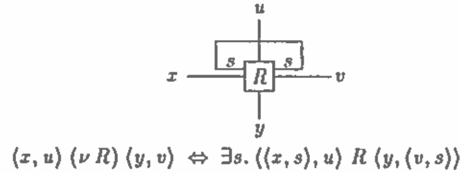
We shall use x_t to denote the t-th element from some reference point – such as the time when the circuit is initialised – in the stream x; given that x_t is a tuple, $x_{t,i}$ is its i-th element. An adder can be described in the stream model as $x\ add\ y \Leftrightarrow \forall t.\ x_{t,0} + x_{t,1} = y_t$.

There are two primitives that do not possess a static interpretation. The first is *delay*, $\mathcal{D}$, defined by $x\ \mathcal{D}\ y \Leftrightarrow \forall t.\ x_{t-1} = y_t$. An *anti-delay* $\mathcal{D}^{-1}$ is such that $\mathcal{D}; \mathcal{D}^{-1} = \mathcal{D}^{-1}; \mathcal{D} = \iota$. A

latch is modelled by a delay with data flowing from domain to range, or by an anti-delay with data flowing from range to domain.

For a circuit R which contains no primitives that possess a measure of absolute time, it is the case that $\mathcal{D}; R = R; \mathcal{D}$. With $A = B = \mathcal{D}$ and $C = \mathcal{D}^{-1}$, the pre-condition for Equation 1 becomes valid so that the transformation can be applied to distribute latches among the R's to reduce the longest combinational path. This process is usually called *retiming*, and examples of deriving pipelined circuits based on an algebraic treatment of retiming can be found elsewhere (Jones and Sheeran 1990, Luk 1992).

A serial design R with an internal feedback path can be modelled by the loop construct in Figure 3. One can show that $\nu\, R = (\mathrm{fstv}\, R) \backslash\!\backslash \mathrm{snd}\, sndfb^{-1}$ where $x\ sndfb\ \langle\langle x, s\rangle, s\rangle$.



$$\langle x, u\rangle\ (\nu\, R)\ \langle y, v\rangle \Leftrightarrow \exists s.\ \langle\langle x, s\rangle, u\rangle\ R\ \langle y, \langle v, s\rangle\rangle$$

Figure 3 A function that describes designs with feedback.

The intuitive idea behind our serialisation equations, the details of which are included in Luk (1992), is to circulate data through a processor n times to emulate the effect of n cascaded processors. A multiplexer $\underline{cmx}_n$ controls when to accept external data x and feedback data y: $\langle \ldots, \langle x_0, y_0\rangle, \langle x_1, y_1\rangle, \langle x_2, y_2\rangle, \ldots\rangle\ \underline{cmx}_3\ \langle \ldots, x_0, y_1, y_2, x_3, y_4, y_5, \ldots\rangle$, and the relation $\underline{bundle}_n$ describes converting between serial and parallel data: $\langle \ldots, x_0, x_1, x_2, x_3, x_4, x_5, \ldots\rangle\ \underline{bundle}_3\ \langle \ldots, \langle x_0, x_1, x_2\rangle, \langle x_3, x_4, x_5\rangle, \ldots\rangle$. The relations $\underline{ev}_n^{-1}$ and $\underline{ev}_n$ are used to inject and to reject dummy data when the processor is in feedback mode: $\langle \ldots, x_0, x_1, x_2, \ldots\rangle\ \underline{ev}_3\ \langle \cdots, x_0, x_3, x_6, \cdots\rangle$. The number of latches in a serialised processor, $\mathrm{slow}_n\, R$, has to be n times of that of the unserialised version R, since it contains up to n interleaved computations with each corresponding to a copy of R. As an example, the following equation can be used to serialise a row of components:

$$\mathrm{row}_n\, R \;=\; \nu\,(\mathrm{fst}\, \underline{cmx}_n\ ;\ \mathrm{slow}_n\, R\ ;\ \mathrm{snd}\,(\mathcal{D}; fork)) \backslash\!\backslash\, [\underline{bundle}_n, \underline{ev}_n]\ ;\ \mathrm{snd}\, \mathcal{D}^{-1}. \quad (2)$$

Again the corresponding theorem for a column of components can be obtained by substituting $\mathrm{row}\, R$ by $(\mathrm{col}\, R^{-1})^{-1}$.

DEVELOPING PERCEPTRONS

First, recall that if x is a tuple, then $\langle x, 0\rangle\ (\mathrm{rdr}\, add)\ \sum_i x_i$. Let $x\ !c\ y \Leftrightarrow x = y = c$, and $sndzero = \pi_1^{-1}; \mathrm{snd}\, !0$. Then $x\ (sndzero; \mathrm{rdr}\, add)\ \sum_i x_i$.

Given input x_j and weights $w_{i,j}$ where $0 \le i < m$ and $0 \le j < n$, a node in a perceptron computes the output $y_i = th\,(\sum_j w_{i,j} \times x_j)$ where th is a threshold function such as the sigmoid function; that is, $\langle x, w_i\rangle\ zmadds\ y_i$ where $zmadds = zip; sndzero; \mathrm{rdr}_n\, madd; th$, and $madd = \mathrm{fst}\, mult; add$. To pass the value of x to a neighbouring node, we use the wiring cell $wire1 = fork; \mathrm{snd}\pi_1$ to implement a broadcast circuit, so that $\langle x, w_i\rangle\ node1\ \langle y_i, x\rangle$ where $node1 = wire1; \mathrm{fst}\, zmadds$.

A layer in a perceptron consists of a row of m nodes, $layer1 = \mathrm{row}_m\, node1\ ;\ \pi_1$ (Figure 4a), and our first description of a multi-layer perceptron, $mlp1$ (Figure 4b), is assembled by arranging the layers according to left reduction:

$$mlp1 \;=\; \mathrm{rdl}\, layer1 \;=\; \mathrm{row}\,(layer1\ ;\ \pi_2^{-1})\ ;\ \pi_2.$$

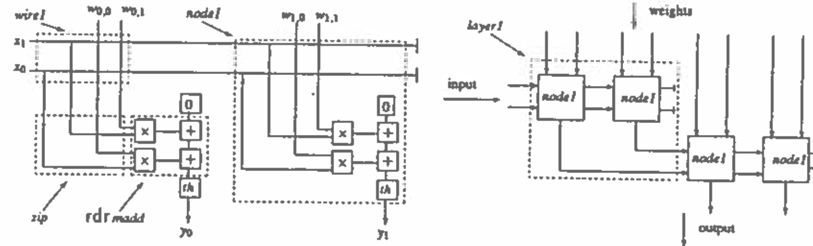


Figure 4a Design $layer1$ ($m = n = 2$).

Figure 4b Design $mlp1$.

Our next task is to distribute the multiply-adders $madd$ among the buses in the broadcast cell $wire1$; this transformation does not substantially improve performance by itself, but it enables further transformations such as pipelining and serialisation to be applied. Using equations such as $wire1 = \mathrm{fst}fork; \mathrm{fstv}\, wire1; \mathrm{snd}\pi_2$, $zip = (\mathrm{map}\, wire1) \backslash zip^{-1}$ and $fork; zip = \mathrm{map}\, fork$, we obtain $layer2$ (Figure 5a) which has a more uniform layout:

$$\begin{aligned}
layer2 &= \mathrm{snd}\,(\mathrm{map}\, sndzero)\ ;\ \mathrm{row}_m\, node2\ ;\ \pi_1, \\
node2 &= wire2 \leftrightarrow (\mathrm{col}_n\, madd2; \mathrm{fst}\, th); \mathrm{fst}\pi_2, \\
wire2 &= \mathrm{map}\,(wire1; \pi_2^{-1}) \backslash zip^{-1}, \\
madd2 &= \mathrm{fstv}\,(madd; \pi_1^{-1}); \mathrm{snd}\pi_2.
\end{aligned}$$

It can be shown that $\mathrm{snd}\, sndzero; node2 = node1$, and the size and performance of $layer1$ and $layer2$ are identical if area and delay of wires are ignored.

Pipelining and serialisation

Since $mlp2 = \mathrm{row}\,(layer2\ ;\ \pi_2^{-1})\ ;\ \pi_2$, we can use the row version of Equation 1 to pipeline it and Equation 2 to serialise it; one possibility is shown in Figure 5b. There are further opportunities in transforming the architecture of $layer2$, and we shall consider some of these next.

If all the coefficients w_i's are hardwired in $node2$, we can eliminate the $wire2$ block to give $node3$, which behaves like $\mathrm{snd}\,(\mathrm{fst}\,[w_i \mid 0 \le i < n]); node2$ while having a simpler structure. Given that $\mathrm{icol}\,\langle P, Q, R\rangle$ describes a column of heterogeneous components with P below Q below R, then

$$\begin{aligned}
node3 &= \mathrm{icol}\,\langle madd3_i \mid 0 \le i < n\rangle, \\
madd3_i &= \mathrm{fst}\,(fork; \mathrm{fst}\,(\pi_1^{-1}; \mathrm{snd}\, !w_i))\ ;\ madd2.
\end{aligned}$$

To produce a faster circuit, a theorem similar to Equation 1 can be used to pipeline $node3$; the resulting design, $node4 = \mathrm{icol}\,\langle madd3_i; \mathrm{fst}\mathcal{D} \mid 0 \le i < n\rangle$, is shown in Figure 6.

On the other hand, if we want to reduce the number of multiply-adders in $layer2$, theorems such as Equation 2 can be used to serialise it. Given that $m = ap$ such that $1 < a \le m$ and $x\ sndfb\ \langle\langle x, y\rangle, y\rangle$, we can reduce the number of columns in $layer2$ by a factor of a by

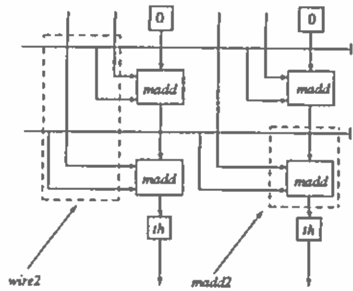


Figure 5a Design *layer*2 ($m = n = 2$).

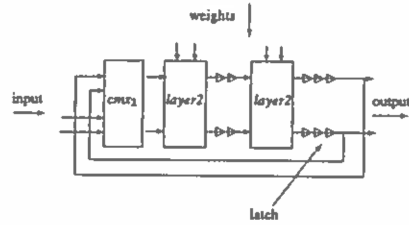


Figure 5b Pipelined and serialised *mlp*2.

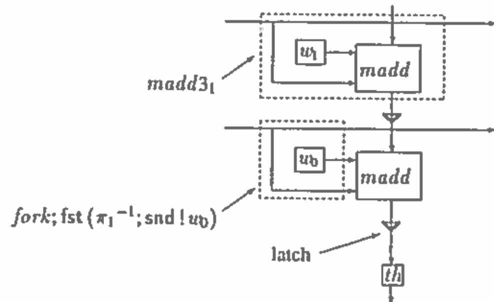


Figure 6 Design *node*4 ($n = 2$).

serialising it horizontally to obtain

$$\begin{aligned} layer5 &= pre'layer5 \;;\; \mathrm{row}_p\; node5 \;;\; \mathrm{snd}\,(\mathrm{map}\,(\mathrm{fst}\mathcal{D}; fork^{-1}))\;;\; \pi_1, \\ node5 &= wire5 \leftrightarrow (\mathrm{col}_n\,(\mathrm{fstv}\; madd2; \mathrm{fst}\, th)); \mathrm{fst}\,\pi_2, \\ pre'layer5 &= [\mathrm{map}\,(sndfb; \mathrm{fst}\; \underline{cmx}_a), \mathrm{map}\; sndzero], \\ wire5 &= \mathrm{map}\,(\mathrm{fstv}\,(wire1; \pi_2{}^{-1})) \backslash zip^{-1}. \end{aligned}$$

An example of *layer*5 will look like the one in Figure 7 without the vertical feedback wires and the associated latches. Instantiating *layer*5 with $a = m$ and $p = 1$ gives a design which is similar to that described by Baji and Inouchi (1992).

Another possibility is to reduce the number of rows of cells in *layer*2 by a factor of b (where $n = bq$ and $1 < b \leq n$) by serialising it vertically; this gives

$$\begin{aligned} layer6 &= \mathrm{snd}\,(\mathrm{map}\; sndzero)\;;\; \mathrm{row}_m\; node6\;;\; \pi_1, \\ node6 &= pre'node6\;;\; wire2 \leftrightarrow (\mathrm{col}_q\,(\mathrm{fsth}\; madd2))\;;\; post'node6, \\ pre'node6 &= \mathrm{snd}\,(\mathrm{snd}\,(sndfb; \mathrm{fst}\; \underline{cmx}_b)), \\ post'node6 &= \mathrm{fst}\,(\pi_2; \mathrm{fst}\mathcal{D}; fork^{-1}; th). \end{aligned}$$

Notice that the critical path is also reduced by a factor of b. Instantiating *layer*6 with $b = n$ and $q = 1$ gives a design which is similar to that described by Skubiszewski (1992).

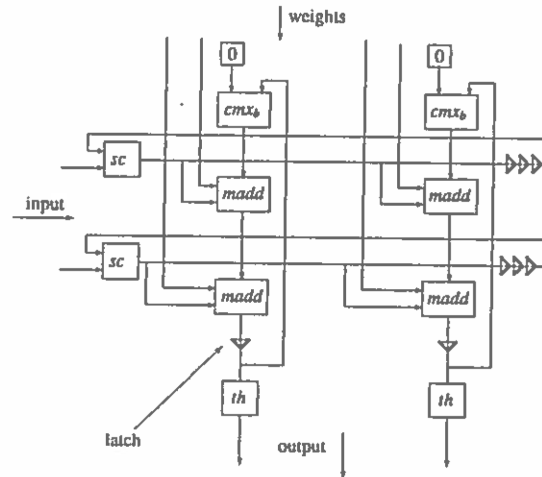


Figure 7 Design *layer*7 ($sc = \underline{scm}_{b,a}$, $m = n = 6$, $a = b = 3$, $p = q = 2$).

Finally, we describe the design *layer*7 (Figure 7), obtained by serialising *layer*2 horizontally by a factor of a and then vertically by a factor of b:

$$\begin{aligned} layer7 &= pre'layer7\;;\; \mathrm{row}_p\; node7\;;\; \mathrm{snd}\,(\mathrm{map}\,(\mathrm{fst}\mathcal{D}^b; fork^{-1}))\;;\; \pi_1, \\ node7 &= pre'node6\;;\; wire5 \leftrightarrow (\mathrm{col}_q\,(\mathrm{fstvh}\; madd2))\;;\; post'node6, \\ pre'layer7 &= [\mathrm{map}\,(sndfb; \mathrm{fst}\; \underline{scm}_{b,a}), \mathrm{map}\; sndzero], \end{aligned}$$

where $scm_{b,a} = \mathrm{slow}_b(\underline{cmx}_a)$ is a component that repeatedly extracts for b cycles the first element of a pair and for the next $(a-1)b$ cycles the second element; for instance

$$\langle \ldots, (x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots \rangle\; \underline{scm}_{2,3}\; \langle \ldots, x_0, x_1, y_2, y_3, y_4, y_5, x_6, x_7, y_8, \ldots \rangle.$$

A design similar to *layer*7 can be obtained by first serialising *layer*2 vertically and then horizontally; it will look like *layer*7 but with more latches on the vertical wires than on the horizontal wires. Note that the multiplier can itself be serialised: one such strategy can be found in Murray *et al* (1987).

The features of our designs are summarised in Table 1; note that T_{ma} and T_{th} correspond to the combinational delay of cell *madd* and *th*, and wire delays are assumed to be insignificant. Such tables, when they are reasonably complete, can be used in checking whether designs can be appropriately parametrised to meet requirements for a specific application. Promising designs can then be implemented on Field-Programmable Gate Arrays using the prototype compilers for various dialects of Ruby (Luk and Page 1991).

EXAMPLE

In this section we report some experimental results from software simulations which can be used to guide the construction of hardware accelerators for neural systems.

The benchmark problem we examined was learning the parity of some set number of binary inputs (Tesauro and Janssens 1988); we concentrated on the 4-parity problem. We

Table 1 Comparison of perceptron designs for computing $th\,(\sum_j w_{i,j} \times x_j)$, where th is a threshold function and $0 \leq i < m$ and $0 \leq j < n$.

Design	Seriali-sation factor	Minimum cycle time	Number of inputs and outputs	Number of *madd* in array	Number of *th* in array	Number of latches in array
layer2	1	$nT_{ma} + T_{th}$	$m + n + mn$	mn	m	0
layer3	1	$nT_{ma} + T_{th}$	$m + n$	mn	m	0
layer4	1	$\max(T_{ma}, T_{th})$	$m + n$	mn	m	mn
layer5	a	$nT_{ma} + T_{th}$	$(m + na + mn)/a$	mn/a	m/a	n
layer6	b	$\max(nT_{ma}/b, T_{th})$	$(n + mb + mn)/b$	mn/b	m	n/b
layer7	ab	$\max(nT_{ma}/b, T_{th})$	$(na + mb + mn)/ab$	mn/ab	m/a	$(m/a) + n$

studied three-layer feed-forward perceptrons with 12 units in the hidden layer, using the standard sigmoid threshold function. For simplicity in simulation, no momentum term was used in back-propagation training. For the same reason we used incremental learning, back-propagating error and updating connection weights after each presentation of a training case.

A network simulator was written in C, using a fixed-point representation for integers. The results of all arithmetic operations were clipped to within a range determined by the number of bits being employed. The sigmoid function was evaluated using floating point exponentiation and division, but the result was appropriately quantised. Two questions regarding this fixed-point approximation are: What is the minimum acceptable *range*? What is the minimum acceptable *precision*?

Investigations into the minimum range revealed that no clipping occurred when five bits (including the sign bit) were used for representing integers, and the amount of clipping that occurred when four bits were employed had no significant effect on training.

To investigate precision, we trained nets with different fixed-precisions, but all with 4 bits for sign and integer. Training was considered to have succeeded when all outputs were within 0.4 of their desired values. If success had not been achieved within 2000 epochs, the net was regarded as having failed to train. Table 2 gives the (hypergeometric) mean number of epochs for training, and the number of times that training failed in a series of 100 trials. The number of bits given is the *total* number in the fixed-point representation. Performance is satisfactory with 13 bits or more; with fewer, training fails too often.

Table 2 Effect of precision on training time.

Bits	17	16	15	14	13	12	11	10
Average epochs	192	189	185	183	176	191	280	557
Failures	0	1	2	5	10	27	51	76

Although at least 13 bits are required for training, fewer are needed when applying a net. This was investigated by training a network with 15-bit fixed-point numbers, then truncating those weights successively down to 6 bits (Table 3). The second line of the table gives the percentage of trials in which there was no deterioration in performance. Truncation has a steady cumulative effect down to 8 bits, after which performance collapses. A more

sympathetic approach is to train the network to a greater degree; outputs must be within 0.2 of their desired values for success during training, but only within the 0.4 threshold when testing (Table 3, third line). We now see no deterioration in performance down to 9 bits, and the success rate at 8 bits is acceptable. The effect of reducing the range for execution was also explored: a reduction to 3 integer bits always significantly reduced the success rates.

Table 3 Effect of truncation on execution.

Bits	15	14	13	12	11	10	9	8	7	6
% (threshold 0.4)	100	91	89	88	77	60	68	62	11	0
% (threshold 0.2)	100	100	100	100	100	100	100	97	46	0

Thus, for training, we found that a minimum of 13 bits are required for the numerical representation, with 4 integer bits. For execution of a pre-trained network, however, as few as 8 bits may be sufficient. These results agree with those of other studies (Holt and Hwang 1992).

We then used our compiler and timing analysis tools to estimate the speed of multipliers implemented in Field-Programmable Gate Arrays manufactured by Algotronix Limited (Algotronix 1990). Using a simple shift-and-add architecture, the maximum clock frequency was found to range from 1.1 MHz for multiplying two 13-bit numbers, to 1.8 MHz for multiplying two 8-bit numbers, to 7.5 MHz for multiplying two 2-bit numbers. A fully-pipelined multiplier, operating in a bit-serial or in a bit-parallel fashion, can run at 16 MHz; this would be attractive for applications such as video processing which demands high-throughput while tolerating large latency. A more detailed evaluation of various ways of implementing neural structures on a number of hardware platforms is currently being undertaken.

Note that the threshold function th can be implemented as a look-up table. An example of how this was achieved is given in Cox and Blanz (1992).

CONCLUDING REMARKS

We have described a method of developing parametrised descriptions of neural networks with different trade-offs in size and performance. Our framework provides a basis for theories and computer-based tools to systematise and formalise design expertise, so that a variety of architectures can be generated and evaluated rapidly. Future work will include conducting further case studies, enhancing our libraries of components and transformations, and extending them to handle optimisations such as weight sharing (Boser *et al* 1992).

ACKNOWLEDGEMENTS

The support of Rank Xerox (UK) Limited, the U.K. Science and Engineering Research Council (GR/F47077), Scottish Enterprise and Algotronix Limited is gratefully acknowledged.

REFERENCES

Algotronix Limited, *CAL 1024 Datasheet*, 1990.

Baji, T. and Inouchi, H., "Systolic Processor Elements for a Neural Network", US Patent 5,091,864, 25 February 1992.

Boser, B.E., Sackinger, E., Bromley, J., leCun, Y. and Jackel, L.D., "Hardware Requirements for Neural Network Pattern Classifiers", *IEEE Micro*, February Issue, pp. 32–40, 1992.

Cox, C.E. and Blanz, W.E., "Ganglion – A Fast Field-Programmable Gate Array Implementation of a Connectionist Classifier", *IEEE J. Solid-State Circuits*, vol. 27, pp. 288–299, 1992.

Holt, J.L. and Hwang, J.N., "Finite Precision Error Analysis of Neural Network Hardware", to appear in *IEEE Trans. Neural Networks*, 1992.

Jones, G. and Sheeran, M., "Circuit Design in Ruby", in *Formal Methods for VLSI Design*, J. Staunstrup (ed), North-Holland, pp. 13–70, 1990.

Luk, W., "Systematic Serialisation of Array-Based Architectures", to appear in *Integration, Special Issue on Algorithms and VLSI Architectures*, 1992.

Luk, W. and Page, I., "Parametrising Designs for FPGAs", in *FPGAs*, W. Moore and W. Luk (ed), Abingdon EE&CS Books, pp. 284–295, 1991.

Murray, A.F., Smith, A.V.W. and Butler, Z.F., "Bit-Serial Neural Networks", *Proc. BIPS Conf.*, pp. 573–583, 1987.

Skubiszewski, M., "A Hardware Emulator for Binary Neural Networks", *Proc. FPL 92*, Vienna, 1992.

Tesauro, G. and Janssens, B., "Scaling Relationships in Back-Propagation Learning", *Complex Systems*, vol. 2, pp. 39-84, 1988.