(12) UK Patent (19) GB (11) 2568230 (13) B

(45) Date of B Publication 03.06.2020

(54) Title of the Invention: Processing in neural networks

(51) INT CL: ***G06F 9/30*** (2018.01) ***G06N 3/04*** (2006.01) ***G06N 3/063*** (2006.01) ***G06N 3/08*** (2006.01)

(21) Application No: **1717306.3**

(22) Date of Filing: **20.10.2017**

(43) Date of A Publication **15.05.2019**

(56) Documents Cited:
US 20160307098 A1

(58) Field of Search:
As for published application 2568230 A viz:
INT CL **G06F, G06N**
Other: **EPODOC, WPI, INSPEC**
updated as appropriate

Additional Fields
Other: **None**

(72) Inventor(s):
Stephen Felix
Simon Christian Knowles
Godfrey Da Costa

(73) Proprietor(s):
Graphcore Limited
6th Floor, 11-19 Wine Street, BRISTOL, BS1 2PH, United Kingdom

(74) Agent and/or Address for Service:
Page White & Farrer
Bedford House, John Street, London, WC1N 2BF, United Kingdom

GB 2568230 B

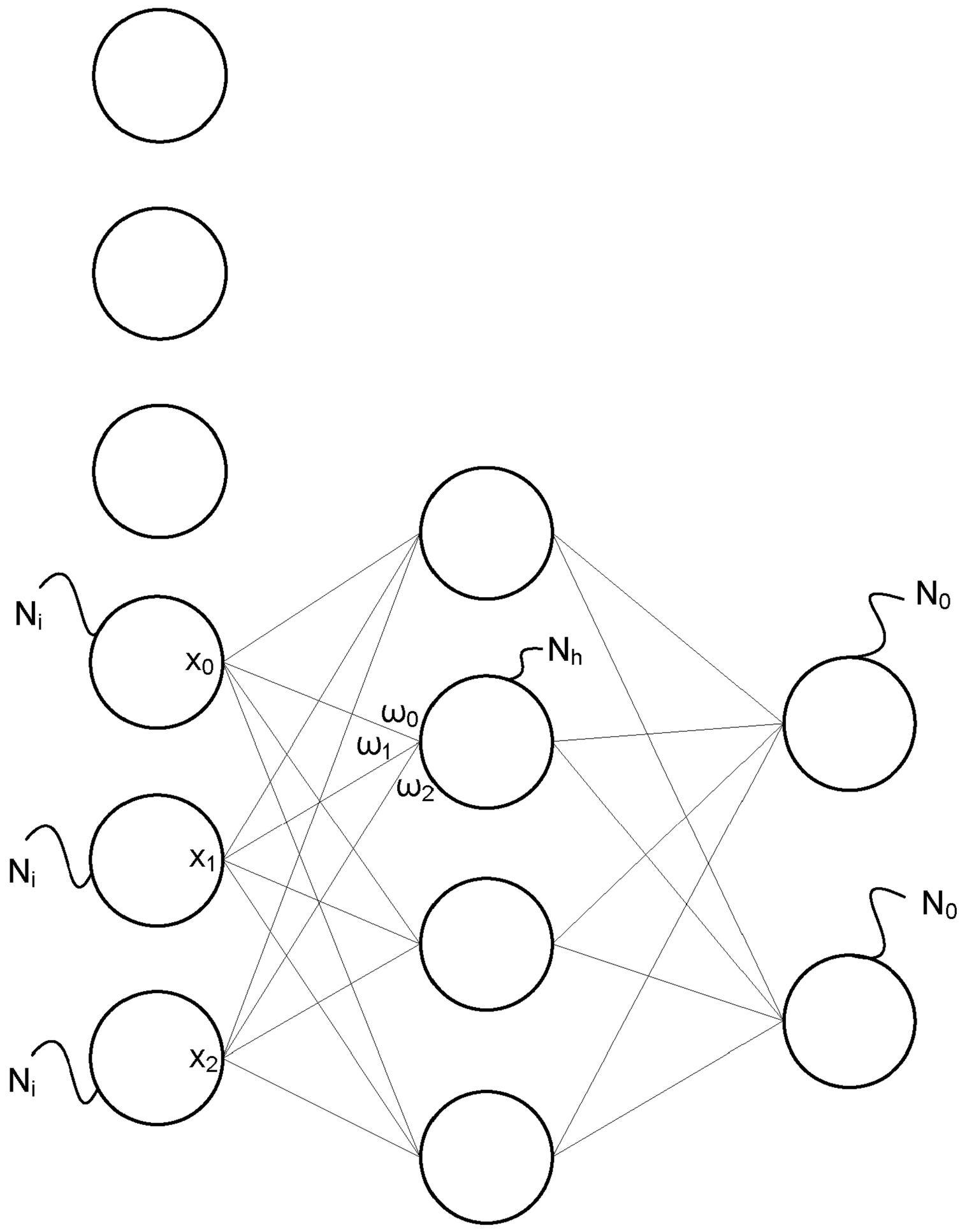
N_i
x_0
N_i
x_1
N_i
x_2
ω_0
ω_1
ω_2
N_h
N_0
N_0
Input
Hidden
Output

Figure 1

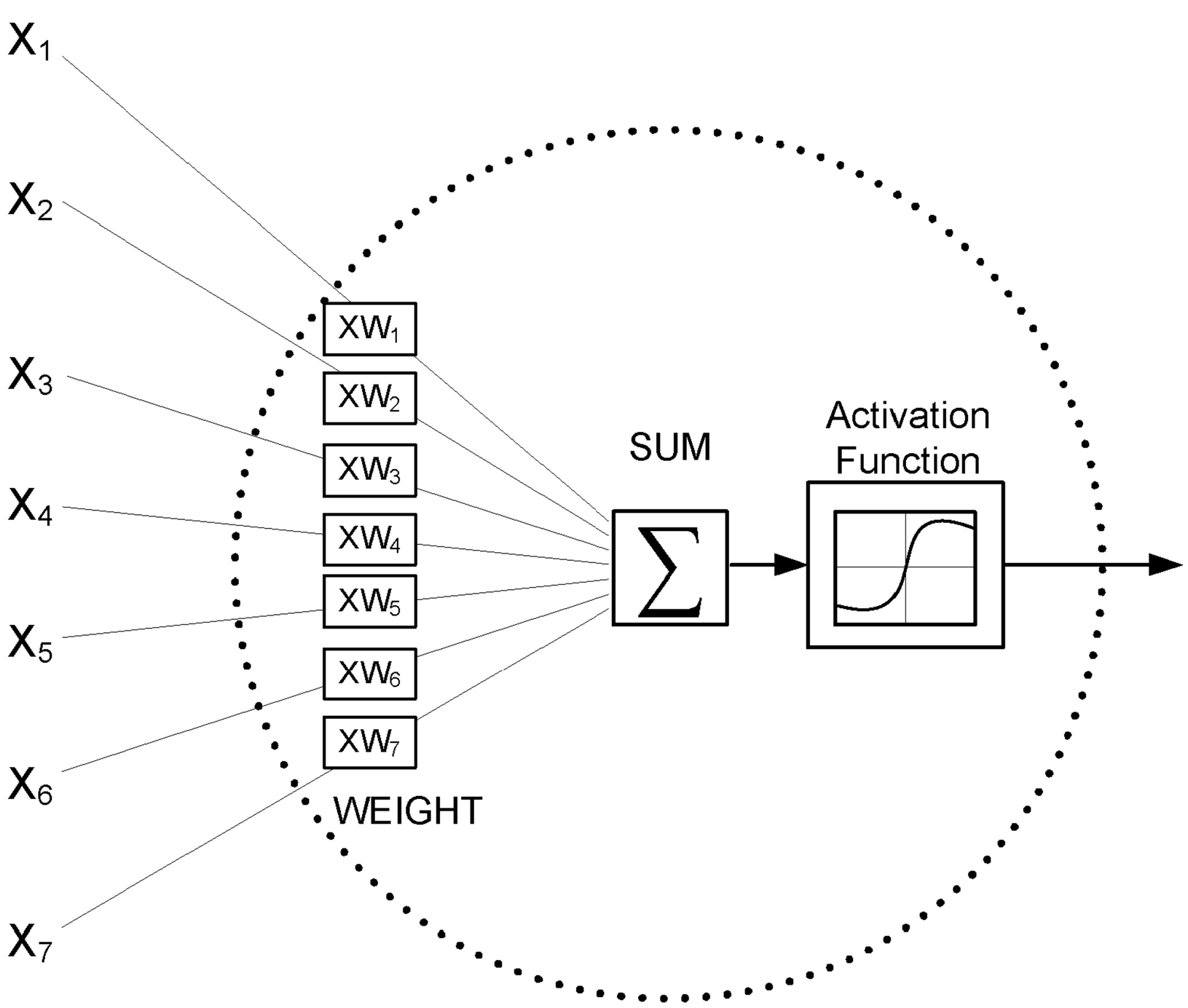
X1
X2
X3
X4
X5
X6
X7
XW1
XW2
XW3
XW4
XW5
XW6
XW7
WEIGHT
SUM
Activation Function

Figure 1A

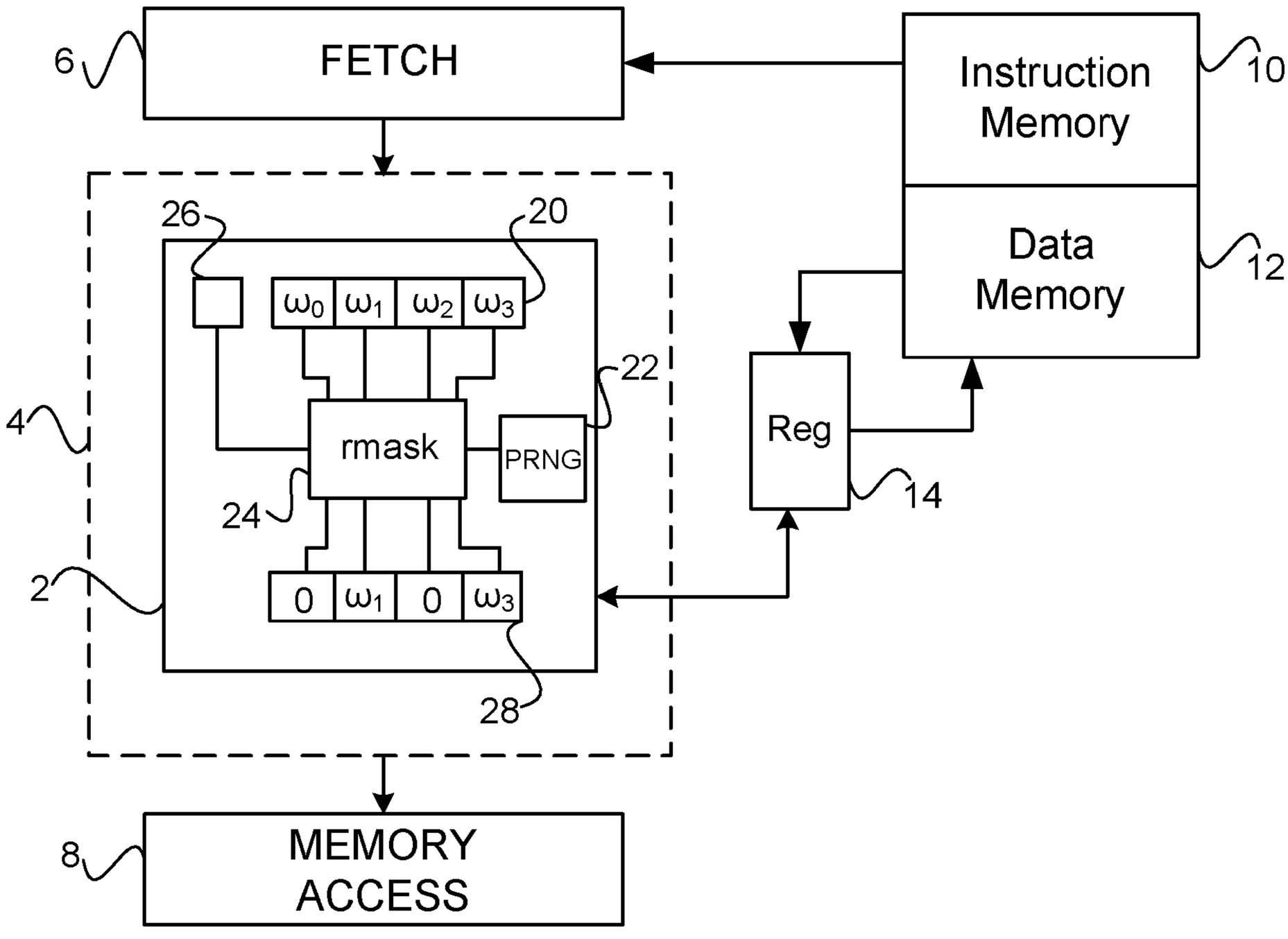

Figure 2

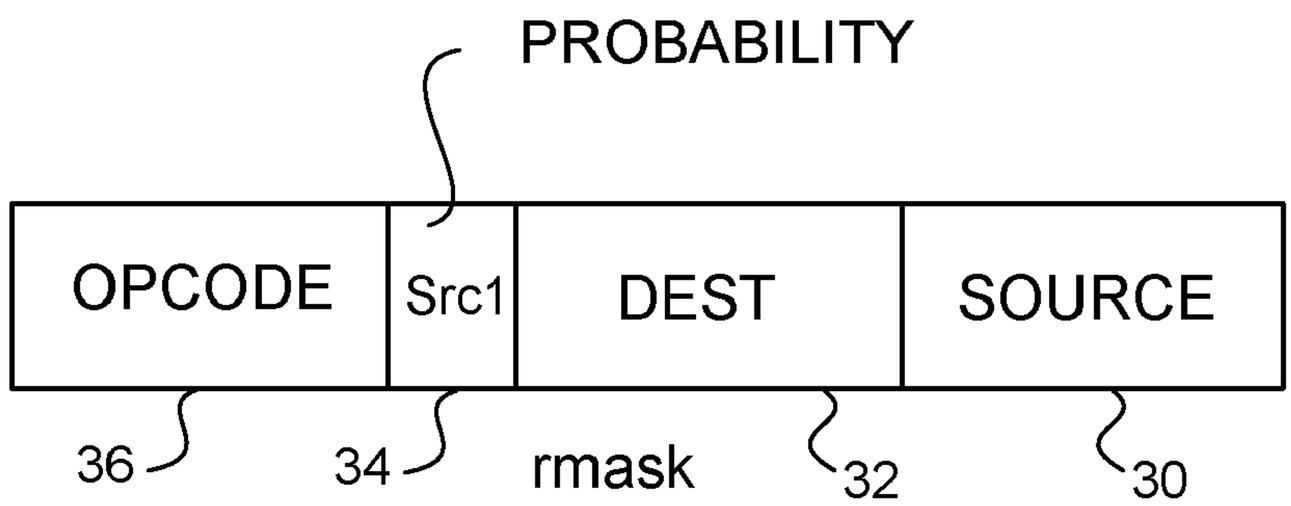

Figure 3

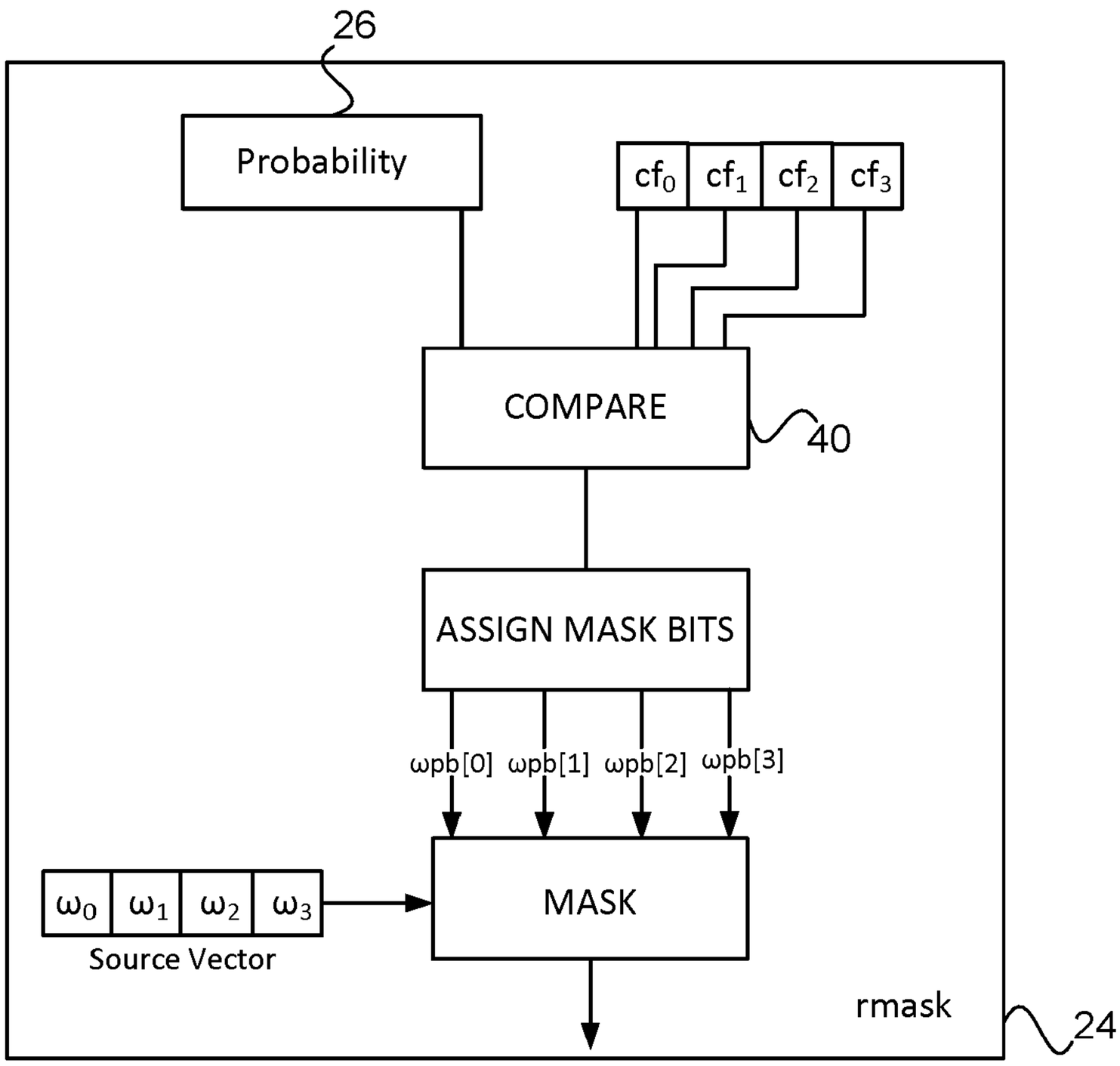
26
Probability
cf0
cf1
cf2
cf3
COMPARE
40
ASSIGN MASK BITS
ωpb[0]
ωpb[1]
ωpb[2]
ωpb[3]
ω0
ω1
ω2
ω3
Source Vector
MASK
rmask
24

Figure 4

Processing in neural networks

Technical Field

The present disclosure relates to processing data in neural networks.

Background

Neural networks are used in the field of machine learning and artificial intelligence. Neural networks comprise arrangements of sets of nodes which are interconnected by links and which interact with each other. The principles of neural networks in computing are based on information about how electrical stimuli convey information in the human brain. For this reason the nodes are often referred to as neurons. They may also be referred to as vertices. The links are sometimes referred to as edges. The network can take input data and certain nodes perform operations on the data. The result of these operations is passed to other nodes. The output of each node is referred to as its activation or node value. Each link is associated with a weight. A weight defines the connectivity between nodes of the neural network. Many different techniques are known by which neural networks are capable of learning, which takes place by altering values of the weights.

Figure 1 shows an extremely simplified version of one arrangement of nodes in a neural network. This type of arrangement is often used in learning or training and comprises an input layer of nodes, a hidden layer of nodes and an output layer of nodes. In reality, there will be many nodes in each layer, and nowadays there may be more than one layer per section. Each node of the input layer Ni is capable of producing at its output an activation or node value which is generated by carrying out a function on data provided to that node. A vector of node values from the input layer is scaled by a vector of respective weights at the input of each node in the hidden layer , each weight defining the connectivity of that particular node with its connected node in the hidden layer. In practice, networks may have millions of nodes and be connected multi-dimensionally, so the vector is more often a tensor. The weights applied at the inputs of the node Nh are labelled w0….w2. Each node in the input layer is connected at least initially to each node in the hidden layer. Each node in the hidden layer can perform an activation function on the data which is provided to them and can generate similarly an output vector which is supplied to each of the nodes N_0

in the output layer $_0$. Each node weights its incoming data, for example by carrying out the dot product of the input activations of the node and its unique weights for the respective incoming links . It then performs an activation function on the weighted data. The activation function can be for example a sigmoid. See Figure 1A. The network learns by operating on data input at the input layer, assigning weights to the activations from each node and acting on the data input to each node in the hidden layer (by weighting it and performing the activation function). Thus, the nodes in the hidden layer operate on the data and supply outputs to the nodes in the output layer. Nodes of the output layer may also assign weights to their incoming data. Each weight is characterised by a respective error value. Moreover, each node may be associated with an error condition. The error condition at each node gives a measure of whether the error in the weight of the node falls below a certain level or degree of acceptability. There are different learning approaches, but in each case there is a forward propagation through the network from left to right in Figure 1, a calculation of overall error, and a backward propagation from right to left in Figure 1 through the network of the error. In the next cycle, each node takes into account the back propagated error and produces a revised set of weights. In this way, the network can be trained to perform its desired operation.

One problem which can arise with a neural network is “overfitting”. Large networks with millions or billions of parameters (weights) can easily overfit. Overfitting causes a network to remember each training sample that has been provided to it (a training sample providing data to the input nodes), rather than be trained to extract relevant features so that the neural net is appropriate, after it has been trained, for application to more generally extract features from samples. A wide range of techniques has been developed to solve this problem by regularising neural networks to avoid overfitting/memorising.

A technique called “Drop out” is discussed in a paper by JE Hinton et al entitled “Improving Neural Networks by Preventing Co-Adaption of Feature Detectors” CoRR abs/1207.0580 (2012). According to this technique, for each training example, forward propagation involves randomly deleting half the activations in each layer. The error is then back propagated only through the remaining activations. It has been shown that this significantly reduces overfitting and improves net performance.

Another technique known as “Drop connect” is discussed in the paper entitled “Regularisation of Neural Network using Drop connect” authored by Li Wan et al published by the Department of Computer Science NYU. According to this technique, a subset of weights are randomly zeroed for each training example, during forward propagation. This technique has similarly been shown to improve the performance of neural networks. Implementing neural networks using known computer technology has various challenges. Implementing specific techniques like Drop out and Drop connect, for example using a CPU or GPU is non-trivial and may impact the full benefits that could be achieved with efficient implementation.

Summary

The present inventors have developed an execution unit for a processor which can efficiently implement Drop out or Drop connect based on a single instruction in an instruction sequence of the processing unit.

According to an aspect of the invention there is provided an execution unit according to claim 1.

The execution unit may comprise a hardware pseudo random number generator (HPRNG) configured to generate a randomised bit string from which random bit sequences are derived for randomly selecting the values to be masked.

In one embodiment each randomised bit string output from the HPRNG comprises m bits and the execution unit is configured to divide the m bits into p sequences and to compare each sequence with a probability value to generate a weighted probability indicator for selectively masking the values.

The masking instruction may comprise a source field identifying the source operand, an indication of a destination register for holding the result and a probability field defining the probability value.

The execution unit may comprise an input buffer for holding the source operand and an output buffer for holding the result.

The values of the source operand may represent weights of links in a neural network (e.g. to implement Drop connect).

The values in the source operand may represent activations defining output values of nodes in a neural network (e.g. to implement Dropout).

The sequence of instructions can comprise instructions for implementing a dot product of the result with a further set of values, and/or an instruction for writing the result to a memory location after the masking instruction has been executed.

The source operand may comprise any convenient number of values of any appropriate length e.g. four 16-bit values, two 32-bit values, or eight 8-bit values (by way of non-limiting example). In general, n values could be one or more values.

A correspond method and computer program are provided.

When used herein, the terms 'random' and 'randomly' are taken to mean random or pseudorandom.

Brief Description of the Drawings

Figure 1 is a highly simplified schematic view of a neural net;

Figure 1A is a highly simplified schematic view of a neuron;

Figure 2 is a schematic diagram of a processing unit in accordance with an embodiment of the invention;

Figure 3 shows the format of a masking instruction; and

Figure 4 is a block diagram of an execution unit for implementing the masking instruction.

Detailed Description

Figure 2 shows a schematic block diagram of an execution unit arranged to execute a single instruction for masking randomly selected values in a vector. The instruction is referred to herein as the rmask instruction. The execution unit 2 forms part of a pipeline 4 in a processing unit. The processing unit comprises an instruction fetch unit 6 which fetches instruction from an instruction memory 10. The processing unit also comprises a memory access stage 8 which is responsible for accessing a data memory 12 for loading data from the memory or storing data into the memory. A set of registers 14 is provided for holding source and destination operands for the instructions being executed at any instance by the pipeline 4. It will readily be understood that the pipeline 4 may contain many different types of execution unit for executing a variety of different instructions, for example for performing mathematical operations. One type of processing unit which may be useful with the present invention is a processing unit using barrel-threaded time slots, in which a supervisor thread may allocate different worker threads to different time slots for their execution. The rmask instruction described herein may be used with any suitable processing unit architecture. Figure 3 shows the rmask instruction format. The instruction has a field 30 which defines a source memory location from which a source operand is to be retrieved, and a field 32 for a destination memory location into which the output vector is to be placed. The memory location could be an implicitly or explicitly defined register in the registers 14, or an address in the data memory. The rmask instruction further defines a probability value Src1 34 with which each individual value within the source operand will be kept, as described in more detail later. The instruction has opcode 36 which identifies it as the rmask instruction. The rmask instruction comes in two different formats. In one format of the instruction, the source operand may represent a vector comprising four 16-bit values (a half precision vector), and in another format the source operand might define a vector comprising two 32-bit values (a single precision vector).

In each case, the rmask instruction has the effect of masking (putting to 0) randomly selected values in the source vector. Such a function has a number of different applications, including in particular functions which have been described above known in the art of neural nets as ‘Drop connect' and 'Drop out'. As already explained, in a neural net a vector representing weights may be provided, these

weights representing links between neurons in the net. Another vector may represent activations, these activations being values output to the individual neurons which are connected by the links.

A very common function in neural nets is to carry out a dot product between these two vectors. For example, the dot product could be the pre-activation value, which is the value from which a neuron output is calculated using an activation function. Applying the rmask instruction to the vector of weights prior to calculating this dot product can be used to implement the Drop-connect function. Application of the rmask instruction to the vector of weights randomly zeroes individual weights within the vector.

Applying the rmask instruction to the vector of activations prior to calculating the dot product can be used to implement the Dropout function. Applying the rmask instruction to the vector of activations randomly zeroes individual activation values within the vector.

The vectors may represent floating point numbers in both single or half precision. One type of vector may comprise four 16-bit values (a half precision vector), and another type of vector may comprise two 32 bit values (a single precision vector). In Figure 2, an execution unit is shown which operates on four 16-bit values or two 32-bit values. The execution unit 2 in Figure 2 has an input buffer 20 which can hold four 16-bit values or two 32-bit values. It is shown holding four 16-bit values, each representing a weight $w_0 \ldots w_3$. The unit has a pseudo random number generator 22 implemented in hardware for providing random numbers to an rmask module 24. An input buffer location 26 is provided to hold a probability with which each individual value within the vector will be kept. This probability value may be provided in the instruction (as shown in Figure 3), or could be set by an earlier instruction and accessed from a register or memory address with an rmask instruction. It could be a 17-bit immediate value in the information for example. The rmask module uses the random number generated by the PRNG 22 and the probability value 26 to randomly mask (put to 0) values of the input vector. The output vector is placed into an output buffer 28. In Figure 2, the first and third values are shown as having been masked to 0. Note that the probability is the probability with which each individual value will be kept (i.e. not

masked). For example, if the probability is set to .5, then each individual value has a probability of .5 that it will be masked. The probability of all four values being masked to 0 is therefore $(.5)^4$, i.e. 1/16. The random number generator introduces the required randomness into the masking by providing an indication of which of the values should be masked, based on the probability.

The rmask module operates as follows, with reference to Figure 4. The pseudo random number generator 22 produces a 64-bit output. For each output, four 16-bit fields cf_0.. cf_3 are produced and tested against the probability value in the source operand. Note that for the single precision mode, two 32-bit fields are produced and similarly tested against the probability value in the source operand.

When the mask instruction is instigated, a 64-bit output of the PRNG is provided. For each output, four 16-bit fields are formed. Res0 [15:0] refers to the least significant 16 bits of the PRNG output, and this syntax applies to the other three fields. Each of these fields is assigned to a cf field (cf standing for comparison field) and each of the comparison fields cf 0..cf 3 is 16 bits long.

```
assign cf0[15:0] = res0[15:0];
assign cf1[15:0] = res0[31:16];
assign cf2[15:0] = res0[47:32];
assign cf3[15:0] = res0[63:48];
```

Then, four weighted random bits wpb[0]…wpb[3] are derived by comparing each comparison filed cf0…3 with the probability value from the source operand src1[15:0], using compare logic 40.

```
assign wpb[0] = (src1[16]==1 | cf0 < src1[15:0])? 1'b1 : 1'b0;
assign wpb[1] = (src1[16]==1 | cf1 < src1[15:0])? 1'b1 : 1'b0;
assign wpb[2] = (src1[16]==1 | cf2 < src1[15:0])? 1'b1 : 1'b0;
assign wpb[3] = (src1[16]==1 | cf3 < src1[15:0])? 1'b1 : 1'b0;
```

The above syntax means that bit wpb[0] is assigned the value '1' if cf0<src1. Here, cf0 is a 16-bit unsigned random value, which can have values in the range {0 … 65535}; src1 is a 32-bit wide operand but only the least significant 17-bits are used for

rmask; src1 is allowed values in the range {0 to 65536}, such that the probability of being 'unmasked' is src1/65536.

Note that for the allowed src1 range of {0 … 65536}, the 17th bit of src1 (src1[16]) is set only for the value 65536. Since the maximum value of cf0 is 65535 and 65535<65536, wpb[0] is automatically '1' when src1[16]==1.>

Subsequently the four bits wpb[0..3] are used respectively to unmask the 4, 16-bit values in src0[63:0] respectively, thus:

```
assign aDst[15:0]  = (wpb[0]==1)? aSrc0[15:0]  : 16'b0;
assign aDst[31:16] = (wpb[1]==1)? aSrc0[31:16] : 16'b0;
assign aDst[47:32] = (wpb[2]==1)? aSrc0[47:32] : 16'b0;
assign aDst[63:48] = (wpb[3]==1)? aSrc0[63:48] : 16'b0;
```

Where 16'b0 denotes a 16-bit wide sequence with value "0", 1'b0 means a bit of value "0" and 1'b1 means a bit of value "1".

For example, aDst[31:16] = aSrc0[31:16] if wpb[1]==1; otherwise aDst[31:16]=0.

In one embodiment the probability value aSrc1 is used to select the probability with which values are randomly 'unmasked' (ie: not masked).

If operand aSrc1 = 65536 then all values are always unmasked

If operand aSrc1 = 49152 then each value is masked independently with a probability of 0.25 and unmasked with a probability of 0.75

If operand aSrc1 = 32768 then each value is masked independently with a probability of 0.50 and unmasked with a probability of 0.50

If operand aSrc1 = 16384 then each value is masked independently with a probability of 0.75 and unmasked with a probability of 0.25

If operand aSrc1 = 0, then all values are always masked

'unmasking' an input means reproducing that value represented by the input at the output. The output format and precision is not necessarily the same as the input format.

'masking' an input means producing at the output a symbol representing the value of zero instead of the value represented at by input.

Not that an unmasking probability == 1 – masking probability

In the implementation of rmask described herein the probability value aSrc1 == 65536 * unmasking probability. The corresponding masking probability can be derived from the masking probability: 65536*(1 – masking probability).

Thus it will be appreciated that either a masking probability or an unmasking probability be defined (the two sum to '1').

The rmask instruction has a number of different applications. For example, when implementing Drop out it could be used just prior to writing neural outputs to memory. All neurons in the next stage of the neural net would then pick up the masked activations from the previous layer.

When implementing Drop connect, the rmask instruction could be used just after reading activations from memory, just prior to calculating the dot product.

The instruction is agnostic as to what the source vector represents and therefore there are other applications beyond Drop out and Drop connect. The instruction is also agnostic to the format of the 16- or 32-bit scalar values within the source vector. In the output, each scalar value is either left unaltered or all bits are masked to zeroes. Thus, the mask instruction could be used to mask a vector of four 16-bit signed or unsigned integers or a variety of 16-bit floating point number formats. The only requirement of the scalar value format is that the value of the number represented by all bits to zero is "0".

In some number formats, there may be more than one possible symbol representing the value of zero.

For example, in the IEEE half precision floating point number format, both of the following symbols can be used to represent zero:

16'b0000000000000000 (positive zero)

16'b1000000000000000 (negative zero)

The term 'random' used herein can mean 'truly random' or 'pseudorandom'. The mask instruction could use either a pseudorandom or a true random bit sequence generator.

Pseudorandom numbers are generated by a 'pseudorandom number generator' or 'PRNG'. PRNG's can be implemented in software or hardware. True random numbers are generated by a 'true random number generator' or 'TRNG'. An example of a TRNG is a "Transition Effect Ring Oscillator". An advantage of PRNGs over TRNGs is determinism (running the same program twice with the same starting conditions always has the same result).

An advantage of TRNGs over PRNGs is that the output is truly random (while the output of a PRNG satisfies a finite set of arbitrarily chosen mathematical properties; the state and output of a PRNG is always predictable from the current state and therefore not truly random).

Claims

1. An execution unit for executing a computer program comprising a sequence of instructions, the sequence including a masking instruction, wherein the execution unit is configured to execute the masking instruction which when executed by the execution unit masks randomly selected values from a source operand of n values and retains other original values from the source operand to generate a result which includes original values from the source operand and the masked values in their respective original locations,

wherein the execution unit comprises a hardware pseudo random number generator (HPRNG) configured to generate a randomised bit string from which random bit sequences are derived for randomly selecting the values to be masked, wherein each randomised bit string output from the HPRNG comprises m bits, the execution unit configured to divide the m bits into p sequences and to compare each sequence with a probability value to generate a weighted probability indicator for selectively masking the values.

2. An execution unit according to claim 1, wherein the masking instruction comprises a source field identifying the source operand, an indication of a destination register for holding the result and a probability field defining the probability value.

3. An execution unit according to any preceding claim, comprising an input buffer for holding the source operand and an output buffer for holding the result.

4. An execution unit according any preceding claim, wherein the values of the source operand represent weights of links in a neural network.

5. An execution unit according to any of claims 1 to 4, wherein the values in the source operand represent activations defining output values of nodes in a neural network.

6. An execution unit according to any of claims 1 to 5, wherein the sequence of instructions comprises an instruction for writing the result to a memory location after the masking instruction has been executed.

7. An execution unit according to any preceding claim, wherein the source operand comprises four 16-bit values.

8. An execution unit according to any of claims 1 to 6, wherein the source operand comprises two 32-bit values.

9. A method of executing a computer program comprising a sequence of instructions, the sequence including a masking instruction, the method comprising:

responsive to execution of the masking instruction randomly selecting values from a source operand of n values and retaining other original values from the source operand to generate a result which includes original values from the source operand and the masked values in their respective original locations;

generating a randomised bit string from which random bit sequences are derived for randomly selecting the values to be masked, wherein the randomised bit string comprises m bits; and

dividing the m bits into p sequences and comparing each sequence with a probability value to generate a weighted probability indicator for selectively masking the values.

220

10. A method according to claim 9, wherein the masking instruction comprises a source field identifying the source operand, an indication of a destination register for holding the result and a probability field defining the probability value.

190

11. A method according to either of claims 9 or 10, wherein the sequence of instructions comprises instructions for implementing a dot product of the result with a further set of values.

12. A method according to any of claims 9 to 11, wherein the sequence of instructions comprises an instruction for writing the result to a memory location after the masking instruction has been executed.

13. A computer program product comprising a computer program which when executed carries out the method of any of claims 9 to 12.