2021年度　博士論文

切り捨てビットを用いた
乱数生成器不要の
制限付きボルツマンマシンの実装法と
評価ハードウェア基盤の構築

九州工業大学大学院　生命体工学研究科
生命体工学専攻
田向研究室

学籍番号　16899018
堀　三晟

概要

深層学習が人工知能，artificial intelligence (AI) の代名詞のように扱われて久しい今日，深層学習を実現する多層のニューラルネットワーク，いわゆる深層ニューラルネットワークはさまざまなアプリケーションに応用されている．これらは物体検出や画像分類，自然言語処理，画像生成など分野は多岐にわたり，日々我々が使っているスマートフォンやパソコン，Web サービスなど，今日の生活にはなくてはならないものとなっている．また，深層学習の研究では，ネットワークの構造や学習手法などさまざまな研究成果が発表されている．

一方で，深層学習は大量のデータ，大量のパラメタを扱う大規模なアプリケーションでもあり，これを実行する計算資源の需要も増加の一途を辿っている．加えて，スマートフォンなど，小型機器に深層学習を組込み，オフラインでアプリケーションを実行する用途も存在する．これらの背景から，近年では IT 企業各社が，サーバ向け，組込み向け問わず，AI 向けハードウェアを多数発表している．

本研究では，まず，AI ハードウェアで必要となる，乱数生成器に焦点を当てる．本論文では，ディジタル回路で演算を実行した際に生成される切り捨てビットを乱数の代替として用いることで，乱数生成器を削減する手法を提案する．また，本手法を検証するために，制限付きボルツマンマシン（restricted Boltzmann machine: RBM）に提案した切り捨てビットを用いる手法を実装し，FPGA 実装を見据え，高位合成環境にて固定小数点演算をエミュレートすることで，提案手法を実行した．この時に生成される切り捨てビットが一様分布に従うか，統計的検定であるカイ二乗適合度検定を実施し一様性を検証した．また，提案手法のハードウェア実装時の有効性を示すため，field-programmable gate array（FPGA）などのディジタル回路で用いられる擬似乱数生成器である xorshift や線形帰還シフトレジスタ（linear feedback shift register: LFSR）と提案手法を Verilog HDL で記述し，論理合成を実施し比較を行なった．

次に本研究では，FPGA を PC に接続し，PC 上のソフトウェアと協調動作するアプリケーションを簡単に検証できるようにする，検証用プラットフォームを構築した．本プラットフォームの動作を検証するために，簡単な画像処理回路と従来手法による RBM を実装し，FPGA 上で動作させた．

本論文では，提案した手法を用いて MNIST データセットや Fashion-MNIST データセットの学習をソフトウェア上の RBM で実施し，入出力データの交差エントロピー誤差を観測することで，学習が進行することを確認した．また，生成された切り捨てビットの一様性については，日本工業規格（Japanese industrial standards: JIS）の付属書などに記載されているカイ二乗適合度検定を用いて，一様性の検定に合格する結果を得た．さらに，ハードウェア実装時の回路資源を，xorshift，LFSR と比較し，提案手法が最も少ないことを示した．これと同時に，擬似乱数もしくは切り捨てビットを取得する際の消費電力量の見積もりも行い，提案手法の優位性を確認した．一方，検証用プラットフォームは FPGA 上で動作させることに成功し，実装したアプリケーションの動作を確認した．加えて，提案手法を用いた積和演算とサンプリングを行う回路をこのプラットフォームに実装し，その動作を確認した．

以上より，本研究の成果は，固定小数点演算で切り捨てられていた数値を用いることで，乱数生成器を実装せずに RBM をはじめとする確率的ニューラルネットワークをディジタルハードウェアに実装できる可能性を示したこと，ハードウェア開発者が提案した回路を簡単に FPGA に実装するプラットフォームを実現したこと，以上の 2 点である．

目次

図目次

表目次

第 1 章

序論

1.1 深層学習の隆盛

深層学習，ディープラーニング [1] がいわゆる人工知能（artificial intelligence: AI）の代名詞のように扱われるようになり久しい．深層学習は 2006 年の Hinton らによる多層構造のニューラルネットワーク，つまり深層ニューラルネットワーク（deep neural network: DNN）[2] の学習手法の発表に始まり，2012 年の一般物体認識コンペティション ImageNet Large Scale Visual Recognition Challenge（ILSVRC）での成功 [3] で一躍注目された．これらの成果を始め，今日までの深層学習技術の発展にはめざましいものがある．深層学習は研究対象としてはもちろん極めてホットなトピックであり，日々さまざまな学習手法や新たな深層ニューラルネットワークの構造が提案されている．一方で，深層学習の技術は単なる研究対象にとどまらず，我々の生活を豊かに，便利にするアプリケーションとしても実装され，さまざまな面で実用化がなされている．

実際に深層学習が用いられているアプリケーションについていくつか紹介する．深層学習が用いられる分野は極めて多岐にわたり，画像認識や画像処理，自然言語処理，ビッグデータ解析など枚挙にいとまがない．画像認識では畳み込みニューラルネットワーク（convolutional neural network: CNN）[4] が主に用いられ，LeNet[5] をはじめとする，さまざまなネットワークアーキテクチャが提案されている．また，ボルツマンマシン（Boltzmann machine: BM）[6] をはじめとする各種生成モデルや自己符号化器（autoencoder: AE）[7] などの研究も進められ，データの特徴抽出など様々な用途に用いられている，さらに，深層学習の技術は，Google 翻訳 [8] や DeepL[9]

といった機械翻訳の精度向上，ビッグデータ解析 [10] など様々なアプリケーションで利用されている．このような深層学習を用いるアプリケーションは大規模なデータセンタで膨大なデータを用いて学習を行い Web サービスとして提供されるものに限らず，日々我々が手に取っているスマートフォンやパソコンに搭載されるようなものまでプラットフォームを問わず実装されている．さらに，近年はロボットや自動車，モノのインターネット（internet of things: IoT）といったより応用的な分野への実装も行われている [11]．このように，深層学習による技術をはじめとした人工知能の研究成果を活用し，実社会に実装しようとする動きは日々活発に行われている．

深層学習が今日のように隆盛を極めるに至ったのは，その性能の高さやこれまで難しいと考えられていた問題への適用など，先にも述べたブレイクスルーにより，人々の注目を集めることになったことが大きい．しかし，これを実現させるに至った各種コンピュータの性能向上，インターネットの普及による膨大なデータの収集と蓄積が可能になったことも決して欠かすことのできない要因である．コンピュータの性能は，従来より唱えられていたムーアの法則に従った半導体の集積度の向上による中央処理装置（central processing unit: CPU）の性能向上や主記憶装置（random access memory: RAM）の容量向上が挙げられる．また，これらを取り巻く各種インタフェースの動作速度や通信速度，ネットワークに接続するための Ethernet などの高速化があって，これらの技術の進歩が統合され，コンピュータシステムとしての性能向上が実現している．さらに，従来はゲームなどの高画質，高フレームレートを必要とする画像の出力を主たる目的として用いられていた graphics processing unit（GPU）が一般的な計算に応用できるようになり，圧倒的な計算資源を容易に入手，運用することができるようになったことは特筆すべきである．GPU の利用により，企業，個人とわず深層学習の研究，開発，利用が大きく進んだ．

GPU は元来の用途が高度な画像処理を高速に行うことであるため，CPU と比較して単純な演算を高並列に実行できる構造をしている．近年の CPU の演算ユニット（コア）が数個〜数十個であるのに対して，GPU は数千個単位で搭載している．さらに，GPU には CPU に接続された RAM とは独立したメモリが配置され，大容量のデータをプロセッサの近くに配置することが可能である．一方で，深層学習は行列の積和演算が大量に求められるアプリケーションであり，これをいかに効率よく，高速に処理するかが DNN の学習や推論に要する時間などの実行時のパフォーマンスに大きく影響する．そこで，GPU の並列演算性能の高さが注目され，深層ニューラルネットワークの演算の高速化に多用されるようになった．このような GPU の一般的

な応用は general-purpose computing on graphics processing unit (GPGPU) と呼ばれ，GPU を一種のベクトル計算機のように用いるものである．特に，NVIDIA 社が公開している GPU 向けの汎用並列計算プラットフォームである compute unified device architecture (CUDA) [12, 13] は専用の C/C++ コンパイラや API が存在し，プログラムを同社の GPU に最適化された形でコンパイルすることが可能である．さらに，深層学習プログラミング用のフレームワークである，Pytorch[14] や TensorFlow[15] なども CUDA に対応することで，開発者は簡単に GPU を用いて深層学習を用いたアプリケーションを高速化し，実用的な速度で動作するアプリケーションを開発することが可能となった．

ここまでで述べたように，近年の深層学習の隆盛は，その性能の高さによる注目を皮切りに，さまざまな研究成果の登場，近年のコンピュータの目覚ましい性能向上，インターネットによる膨大なデータの収集，GPU による計算の高速化，深層学習プログラミング用の各種フレームワークの登場によるアプリケーション開発の平易化といった，さまざまな技術革新が複合的に組み合わさってもたらされた結果であると言える．今後もコンピュータの性能の向上や 5G に代表されるようなインターネット環境の進歩は続くため，さまざまなアプリケーションの登場や技術の応用が予想される．

1.2 深層学習の問題と専用ハードウェア

近年の CPU 開発は動作周波数の向上からコア数の増加へと舵を切り，さらにアプリケーションによるマルチスレッド処理の対応など様々な最適化 [16] により，従来型の CPU による演算能力の向上も日々進んでいる．一方で，前節で述べたとおり，深層学習のアプリケーションは GPU による処理の高速化が多用されているが，次に述べるような消費電力の問題を孕んでいる．

近年の深層学習アプリケーションは要求される演算能力が高く，これまでのような高性能な CPU や GPU の活用，これらを複数搭載したクラスタの活用ではその消費電力の増大が無視できなくなってきている．消費電力の問題については，環境負荷と組込み用途の二つの側面から捉えられる．

一つは大規模演算における膨大な電力消費に伴う環境負荷である．DNN を大規模に学習する場合，しばしば巨大なデータセンタなどで大量のコンピュータを用いて行われる．近年は環境問題に対する意識の高まりなどから，データセンタの消費

電力増加も無視できなくなりつつある．国立研究開発法人科学技術振興機構，低炭素社会戦略センターの提案書 [17, 18] によると，国内のデータセンタの消費電力は 2018 年の推定値が 14TWh/年であるのに対し，2030 年には 90TWh/年，2050 年には 12,000TWh/年と見積もられている．このうち，深層学習などに用いられるものを AI 業務として，これが消費する電力が，2018 年は 0.7TWh/年（5%），2030 年は 16TWh/年（18%），2050 年は 3,000TWh/年（25%）と推定されている．括弧内の割合はデータセンタの消費電力に対する AI 業務の消費電力の割合である．なお，AI 業務については，深層学習の手法や専用ハードウェアなどの発展が著しい分野であるので，この提案書では現在のサーバ数や直近の伸び率などから推定している．この報告書からも，データセンタにおける AI 業務向け機器の消費電力が占める割合は増加すると考えられている．また，自然言語処理を行う DNN の環境負荷について検討した研究 [19] も存在する．このような状況から，DNN の消費するエネルギーの削減は重要な課題の一つである．そこで，演算能力を犠牲とせず，高性能かつ低消費電力な演算システムが求められる．

二つ目の見方は，組込み機器に深層学習を実装する場合である．自動車やロボット，IoT 機器などの分野でも画像認識や画像処理，音声認識などさまざまなアプリケーションが求められている．しかし，これらの機器に深層学習を行うシステムを組込む場合，さまざまな制約が存在する．電力の供給源はバッテリーなどであり，貧弱であることが予想される．さらに，大規模な計算システムを組み込む空間的な余裕がない場合もある．物理的な制約のため，演算時に発生する熱を処理する能力にも乏しい可能性が高い．このような制約を加味すると，組込み機器向けには，小型かつ低消費電力でありながら，必要とされる計算能力を発揮できる演算システムが求められる．

さらに，ノイマン型コンピュータに存在する，プロセッサと記憶装置間のデータ転送におけるボトルネック（フォン・ノイマン・ボトルネック）により，大量のデータに対して，大量の演算を実施する DNN は極めて処理効率の悪いものとなる．そこで，高性能かつ低消費電力なシステムを構築するため，近年では DNN 専用のハードウェア開発が行われている．これらは，内部に実装されたディジタル回路を書き換え可能な field-programmable gate array (FPGA) による非ノイマン型アーキテクチャのシステムや，application specific integrated circuit (ASIC) による専用のチップの実装が試みられている．さらに，スマートフォンやパソコン向けに従来の汎用的なプロセッサから，メーカーが独自に開発した DNN 専用モジュールを含んだプロセッサ

へと移行する例も存在する．近年の，独自プロセッサを開発し，組込む事例は，消費電力を低減しつつも高速に動作可能なシステムを構築する目的によるものである．第２章では背景として各種 AI ハードウェア開発について紹介する．

1.3 研究目的

近年の DNN の発展は目覚ましいものがあると同時に，より高速かつ低消費電力な DNN 実行プラットフォームが求められている．そこで，本研究では，これらの制約を満たしつつ，DNN を実装するプラットフォームとして FPGA による非ノイマン型のシステムに注目した．FPGA は内部回路を自由に設計することが可能であるため，データバスやメモリ配置をアプリケーションに適した形で設計することが可能となる．これは，プロセッサに対するメモリ階層を柔軟に設計できることを意味する．つまり，ノイマン型計算機におけるプロセッサとメモリ間の通信がボトルネックとなる，フォン・ノイマン・ボトルネックを軽減することができる．さらに，書き換え可能であるという柔軟性から，ソフトウェアのように「回路をアップデートする」ことも可能となる．

そこで，本研究では，低消費電力かつ高速な FPGA の特性に焦点を当て，RBM をはじめとする確率的ニューラルネットワークの効率的な FPGA 実装法の提案と，アプリケーションを実装するための FPGA プラットフォームの実現を目的とする．特に DNN アプリケーションの FPGA 実装においては，様々な処理を実現する回路の実装が求められるが，その中でも乱数生成器は確率的なニューラルネットワークの動作を実現するために重要な回路となる．しかし，乱数生成器は比較的多くの回路資源を必要とする問題が存在する．また，開発者が FPGA にアプリケーションを実装する際，コンピュータとの通信用インタフェースの構築が大きな手間となる．そこで，本研究では，目的を達成するための目標として，乱数生成器を使用しない RBM などの確率的ニューラルネットワークのハードウェア実装手法の提案と，FPGA アプリケーションを容易に実行，検証可能にするための実験用ハードウェア基盤の開発を設定する．本研究ではまず，ディジタル回路における切り捨てビットを用いた確率的ニューラルネットワークの実装手法を提案し，制限付きボルツマンマシン（restricted Boltzmann machine: RBM）[2] に適用し動作検証を行なう．また，提案した切り捨てビットのハードウェア実装時の優位性を検証するため，切り捨てビット取得回路と他の擬似乱数生成回路をそれぞれ単体で Verilog HDL にて記述し，論理合成を行い

比較検討する．次に，x86 プロセッサを搭載したコンピュータと一般的な FPGA 評価ボードを接続し，協調動作させるための，評価用ハードウェア基盤を提案し，これを FPGA に実装し，評価する．

1.4　本論文の構成

本論文は Fig. 1.1 に示すように 7 つの章から構成されている．第 1 章は序論である．第 2 章では，ニューラルネットワークの基本理論から，生成モデルである RBM について述べる．その後，AI ハードウェアの実例を紹介し，ハードウェア乱数生成器について述べる．第 3 章と第 4 章は提案手法について述べている．まず，第 3 章では提案手法の切り捨てビットを使用した乱数生成器を用いない RBM のハードウェア実装手法について述べる．次に，第 4 章では本研究で開発した FPGA による検証用ハードウェアプラットフォームについて述べる．第 5 章では提案手法の RBM による性能評価と取得された切り捨てビットの一様性の検定，切り捨てビットを用いた回路と一般的な擬似乱数生成器のハードウェア実装時の比較，さらに，構築したハードウェアプラットフォームの動作検証実験とその結果について述べる．第 6 章はまとめとして本研究で得られた結果の考察と今後の課題について述べ，7 章で結論を述べる．

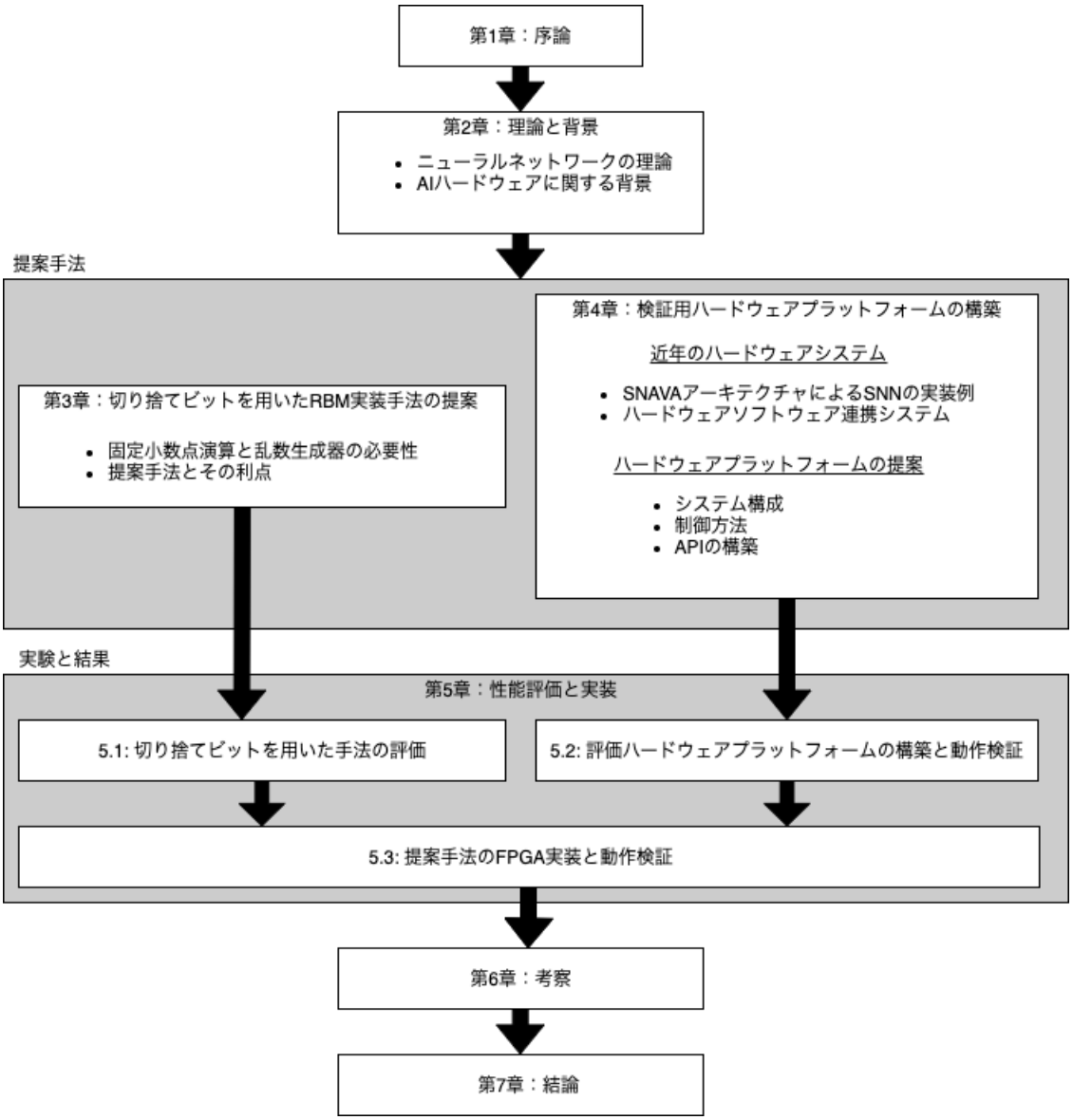

Fig. 1.1　本論文の構成

第 2 章

理論と背景

本章では，理論と背景として，まずニューラルネットワークの基礎的な理論から，制限付きボルツマンマシンの理論までを述べる．続いて，背景として近年の AI ハードウェアやプロセッサ開発の具体例を提示し，現状を俯瞰する．その後，AI ハードウェアで重要な地位を占める乱数生成器について概要を解説し，その課題について指摘する．

2.1 深層学習

深層学習（deep learning: DL）は複数の層を重ねたニューラルネットワークを学習させる手法のことであり，これを実現するのが深層ニューラルネットワークである．本節では深層学習を構成するニューラルネットワークの基本的な概念を述べ，その後，深層ニューラルネットワークや確率的モデルである制限付きボルツマンマシン（restricted Boltzmann machine: RBM）について述べる．

2.1.1 ニューロンモデル

ニューラルネットワークは脳の神経細胞の挙動を模倣し，それらを接続することで分類問題や回帰問題への適用を試みたモデルである．現在では深層学習としてより大規模なネットワークが構築され，複雑な問題へも利用されている．ここではまずニューラルネットワークの基本要素となるニューロンモデルについて解説する．

ニューラルネットワークでは，ニューロンと呼ばれる脳の神経細胞を数理モデル化し，これをコンピュータ上でシミュレーションする．このニューロンの数理モデルを

ニューロンモデルと呼び，1943 年に McCulloch と Pitts が提唱した [20] のが始まりである．

ニューロンの概形を Fig. 2.1 に示す．図に示したように，ニューロンは細胞体と呼ばれる本体が存在し，これに樹状突起がいくつも伸びている．樹状突起は周囲のニューロンから信号を受け取るための部位であり，いわば入力端子である．これは細胞体から複数伸びている．また，軸索と呼ばれる部位が存在し，これが周囲のニューロンに対して信号を伝達する役割を持つ．つまり，軸索はニューロンの出力端子である．実際の情報伝達は，この軸索の先端に存在するシナプスと呼ばれる部位で行われる．ただし，このシナプスの信号の伝達効率は全て同じではなく，シナプスごとに異なる．

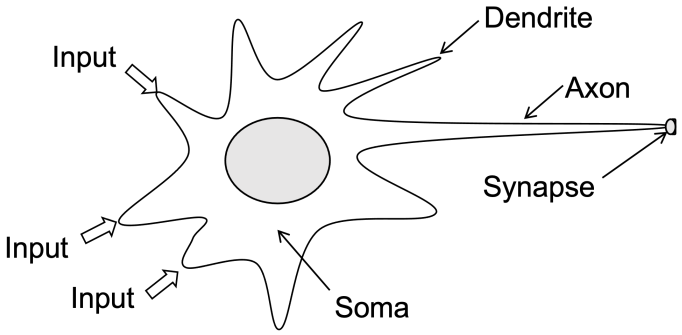

Fig. 2.1　ニューロンの概形

続いて，ニューロンの動作について解説する．ニューロンには膜電位と呼ばれる内部状態が存在する．他のニューロンからスパイクと呼ばれる入力信号が与えられたときに，この膜電位の値が変化する．膜電位がある閾値を超えたときに，ニューロンがスパイクを派生させ，軸索，シナプスを通して周囲の接続されたニューロンに信号が伝達される．この膜電位が閾値を超えてスパイクを発生させる状態となることを，ニューロンが発火したという．このニューロンを多数接続させ，シナプスの伝達効率を調整されることによって，様々な情報処理を行うことが可能となる．

このニューロンの動作を簡略化し，モデル化したものを Fig. 2.2 に示し，ここではユニットと呼ぶ．このユニットには複数の入力，$x_1, x_2, x_3, \cdots, x_n$ が存在し，二値のユニットの状態を出力として扱う．また，それぞれの入力にはシナプスに該当する重み $w_1, w_2, w_3, \cdots, w_n$ が存在する．このようにして，ニューロンの動作を簡略化している．

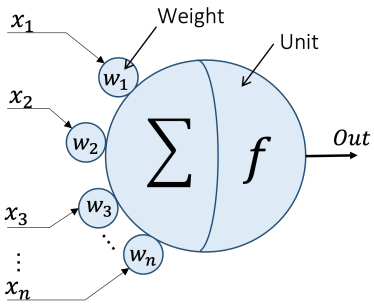

Fig. 2.2 ニューロンモデル

このユニットに入力が与えられた際の動作について述べる．このユニットは入力と重みの積和演算とその結果に対する活性化関数の適用を行なっている．具体的には以下の数式でユニットの動作は与えられる．まず，入力信号を，$x_1, x_2, x_3, \cdots, x_n$ とし，各々の入力信号に対する重みを $w_1, w_2, w_3, \cdots, w_n$ とする．また，n は入力信号の数である．また，b はユニットのバイアスである．このとき，ユニットの積和演算結果 u は以下のようになる．

$$u = \sum_{i=1}^{n} x_i w_i + b \tag{2.1}$$

この積和演算結果に対して，活性化関数 $f(u)$ を適用し，ユニットの出力とする．この活性化関数は積和演算結果がある閾値を超えると出力を 1 とし，下回っている場合は 0 とするものである．この時の，閾値を θ とし，出力と O とすると，次のように表せる．

$$O = f(u - \theta) \tag{2.2}$$

この時の活性化関数 $f(u)$ は以下のように表すことができる．

$$f(u) = \begin{cases} 1 & (u > 0) \\ 0 & (otherwise) \end{cases} \tag{2.3}$$

この活性化関数はニューロンの挙動をモデル化したものであり，出力が 1 もしくは 0 の二値となっているが，一般的にニューラルネットワークでは，シグモイド関数や双曲線正接関数（$tanh(x)$）など様々な形のものが用いられる．

2.1.2　深層ニューラルネットワーク

前節のニューロンモデルを複数接続し，層を形成し，これを多層に積み重ねたものが深層ニューラルネットワークである．深層ニューラルネットワークには多層パーセプトロンや，畳み込みニューラルネットワーク，深層ボルツマンマシン（deep Boltzmann machine: DBM），深層信念ネットワーク（deep belief netowork: DBN）[21] といった様々なアーキテクチャが存在する．深層学習はこの深層ニューラルネットワークをいかに学習するかが重要となってくる．これらに共通して言えることは多数の積和演算を内包していることである．これは以下の Fig. 2.3 に示すような 2 層に並べた順伝播型のネットワークからもわかる．この図では入力値を入力層として一つの層で表現している．x_1, x_2, x_3, x_4 はそれぞれ入力，u_1, u_2, u_3 が積和演算結果で，z_1, z_2, z_3 がそれぞれのユニットの出力である．

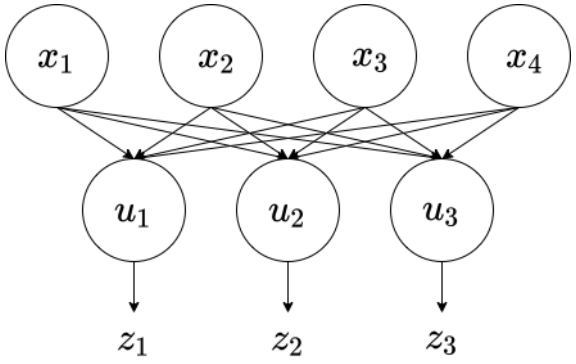

Fig. 2.3　2 層のネットワーク構造

この時の出力 z_1, z_2, z_3 は次のように計算される．ここで，i は入力ユニットの番号を表し，j は 2 層目のユニットの番号を表す．

$$u_j = \sum_{4}^{i=1} w_{ij} x_i + b_j \tag{2.4}$$

$$z_j = f(u_j) \tag{2.5}$$

この式からもわかるように，ニューラルネットワークにおける各ユニットの出力にはそのユニットに与えられた全ての入力値とそれに対応する結合荷重の積和演算が含まれる．この積和演算は入力 x のベクトル $\boldsymbol{x}$ と結合荷重行列 W との積とバイアスの加算と考えられる．これはユニット数が増大し，層数を増やすとこの積和演算の規模も増大し，大量の演算が必要となることを表している．深層ニューラルネットワークは多数の層を積み重ねた構造であり，学習時にはそれぞれの層を取り出して，一つずつ事前学習を行い，最後に全ての層を結合し，一つのネットワークとして全体の学習を行うことで，パラメタの学習を行う．

2.2 生成モデルとボルツマンマシン

本論文では提案手法や提案ハードウェア基盤に関する各種検証を RBM に対して行う．そこで，本節では RBM に関する基本的な事柄について述べる．

2.2.1 生成モデル

ニューラルネットワークは確定的な動作をするものと確率的な動作をするものの２種類に大きく分類することができる．確定的に動作をするものには，多層パーセプトロン，畳み込みニューラルネットワークや自己符号化器などが分類される．一方，確率的に動作をするものには，本論文で扱う RBM や RBM から派生した DBM や DBN が分類される．

生成モデルとは，入力として与えられたデータセットを生成する確率分布そのものを学習するように動作するものである．この枠組みでは，与えられた入力データ $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$ はある確率分布 $p(\boldsymbol{x})$ から生成されている，と捉える．そして，RBM などの生成モデルに分類されるニューラルネットワークは学習を行うことで，この確率分布をできるだけ再現するようにパラメタを調整する．この時，学習を行なったニューラルネットワークが表現する確率分布をそのパラメタ $\boldsymbol{\theta}$ を用いて，$p(\boldsymbol{x}|\boldsymbol{\theta})$ と表す．生成モデルはデータの生成される確率分布を学習しているので，学習後のニューラルネットワークからデータを抽出（サンプリング）すると，学習時に与えたようなデータを取得することができる．

ここで，生成モデルの場合はパラメタ $\boldsymbol{\theta}$ を学習によって獲得することになる．この学習は最尤推定法によって行われる．最尤推定法では，与えられた N 個のデータ

$\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N$ が $p(\boldsymbol{x}|\boldsymbol{\theta})$ から生成されたと考え，最も尤もらしい $\boldsymbol{\theta}$ を推定する手法である．この際に最適化する対象となる関数は以下に示す尤度関数であり，学習とはこれを最大化するパラメタ $\boldsymbol{\theta}$ を求める問題となる．

$$L(\boldsymbol{\theta}) = \prod_{n=1}^{N} p(\boldsymbol{x}_n|\boldsymbol{\theta}) \tag{2.6}$$

2.2.2　ボルツマンマシン

BM は生成モデルの一種であり，Fig. 2.4 に示すような全結合方のネットワーク構造を有している．それぞれのユニット同士が向きを持たない結合を持ち，各ユニットは 1 または 0 の 2 値をユニットの状態として持つ．

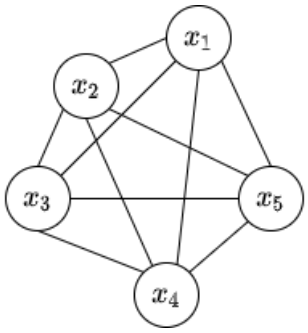

Fig. 2.4　BM の構造

BM では，M 個の各ユニットそれぞれの状態を x_i $(i = 1, 2, \cdots, M)$ とし，これらを確率変数としてみなしたときに，BM 全体が表現する状態はそれぞれのユニットの状態の組み合わせである $\mathbf{x} = [x_1, x_2, \cdots, x_M]$ となる．そこで，BM ではこの $\mathbf{x}$ が式 (2.7) で与えられる確率分布から生成されているものと考える．また，式 (2.8) をエネルギー関数と呼び，ε は BM におけるユニット間の結合を表す．また，$\boldsymbol{\theta}$ は BM のパラメタを表す．パラメタは，$\{b_i|1, \cdots, M\}$ と $\{w_{ij}|(i,j) \in \varepsilon\}$ である．b_i はバイアス，w_{ij} はユニット i と j 間の結合荷重を表す．BM はユニット同士の結合に向きのない無向グラフであるから，$w_{ij} = w_{ji}$ である．さらに，式（2.9）は BM の表現するモデル分布 $p(\mathbf{x}|\boldsymbol{\theta})$ が全ての x_i の組み合わせの確率を合算した際に 1 となるようにするための規格化定数である．BM は確率分布であるから，これの表す分布は確

率分布の条件を満たす必要があるためである．

$$p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} exp\{-\Phi(\mathbf{x}, \boldsymbol{\theta})\} \tag{2.7}$$

$$\Phi(\mathbf{x}, \boldsymbol{\theta}) = \sum_{i=1}^{M} b_i x_i - \sum_{(i,j)\in\varepsilon} w_{i,j} x_i x_j \tag{2.8}$$

$$Z(\boldsymbol{\Phi}) = \sum_{\mathbf{x}} exp\{-\Phi(\mathbf{x}, \boldsymbol{\theta})\} \tag{2.9}$$

なお，$Z(\boldsymbol{\Phi})$ に含まれる，$\sum_{\mathbf{x}}$ は，$\mathbf{x} = \{x_1, x_2, \cdots, x_M\}$ の全ての組み合わせの和を表す．この組み合わせは BM のユニットが 2 値を表すので，M 個のユニットが存在する場合は，その組み合わせ数は 2^M となるため，ユニット数の増大とともに組み合わせ数は指数関数的に増大する．

BM の学習では，モデル分布 $p(\mathbf{x}|\boldsymbol{\theta})$ がデータを生成している真の分布 $p_g(\mathbf{x})$ に最も近づくようにパラメタ $\boldsymbol{\theta}$ をデータから学習する．この学習には最尤推定法が用いられ，式（2.6）に示した尤度関数の対数をとった対数尤度関数 $logL(\boldsymbol{\theta})$ を最大化する．この時，バイアス b_i と結合荷重 w_{ij} の勾配を求め，反復的にパラメタを更新する勾配降下法を適用することができる．しかし，この勾配を求める際に，BM ではモデルの期待値を計算する必要があり，それには BM を構成する全ユニットのとりうる状態の組み合わせに関する和を求める必要がある．規模の小さい BM であれば計算可能であるが，先にも述べた通り，BM の全ユニットがとりうる状態の組み合わせは 2^M 通り存在し，指数関数的に増大するため，ユニット数の増大と共に，組合せ爆発が発生し，現実的な時間で計算を終えることができなくなってしまう．このため，通常の手法では BM の学習は難しい．

また，BM に隠れ変数を導入することで，モデル分布の表現力を高めることができる．隠れ変数を導入した BM の形を Fig. 2.5 に示す．この時，BM を構成するユニットは，実際にデータが入力される可視変数 v_i とデータの入出力には関わらず，BM の表現力向上に寄与する内部的な変数である隠れ変数 h_j に分けられる．しかし，ユニット間の結合は BM と同様に互いに自由に結合できる．学習方法は隠れ変数を持たない BM と同じ最尤推定法による尤度関数を最大化するパラメタの探索である．しかし，隠れ変数を持たない BM と同じく，ユニット数の増大と共に計算量が増大する問題を抱えている．

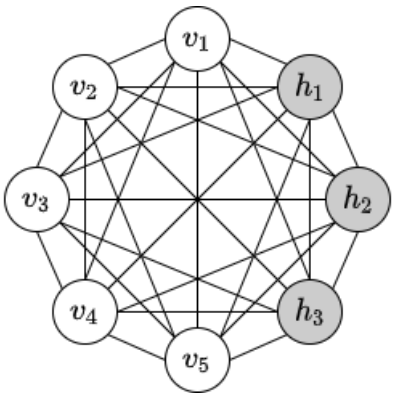

Fig. 2.5　隠れ変数を導入した BM の構造

2.2.3　制限付きボルツマンマシン

RBM は隠れ変数を導入した BM のユニット間接続に制約を持たせたアーキテクチャのニューラルネットワークであり，Fig. 2.6 に示すような構造を持っている．RBM は可視ユニットと隠れユニットでそれぞれ層を形成し，同じ層内に存在するユニット同士の結合を持たせない制限を設けた隠れユニットを持つ BM である．また，それぞれの層を可視層，隠れ層と呼ぶ．

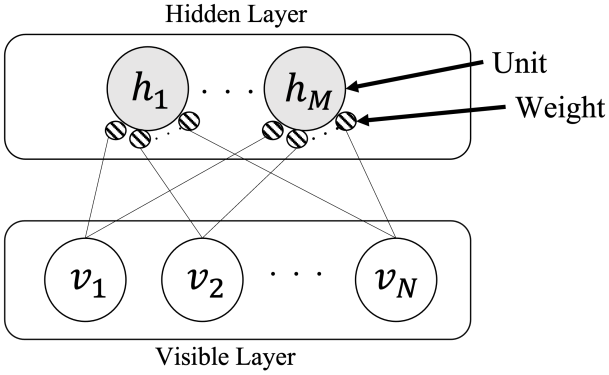

Fig. 2.6　RBM の構造

RBM の表すモデル分布は式（2.10）で与えられる通り，パラメタ $\boldsymbol{\theta}$ を条件とする，可視ユニットと隠れユニットの状態 $\boldsymbol{v}$ と $\boldsymbol{h}$ の取りうる確率分布となる．BM と同様

に，$\Phi(\boldsymbol{v}, \boldsymbol{h}, \boldsymbol{\theta})$ はエネルギー関数，$Z(\boldsymbol{\theta})$ は正規化定数である．正規化定数では可視層と隠れ層を形成する全ユニットの全ての組み合わせについて，$exp\{-\Phi(\boldsymbol{v}, \boldsymbol{h}, \boldsymbol{\theta})\}$ を計算ししているので，式（2.10）は確率分布として扱うことが可能となる．

$$p(\boldsymbol{v}, \boldsymbol{h}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} exp\left\{-\Phi(\boldsymbol{v}, \boldsymbol{h}, \boldsymbol{\theta})\right\} \tag{2.10}$$

$$\Phi(\boldsymbol{v}, \boldsymbol{h}, \boldsymbol{\theta}) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_i \sum_j w_{ij} v_i h_j \tag{2.11}$$

$$Z(\boldsymbol{\theta}) = \sum_{\boldsymbol{v}} \sum_{\boldsymbol{h}} exp\left\{-\Phi(\boldsymbol{v}, \boldsymbol{h}, \boldsymbol{\theta})\right\} \tag{2.12}$$

また，RBM において，可視層と隠れ層の取りうる確率は，それぞれ相対する層の状態を条件とした条件付き確率となる．可視層の確率分布は式（2.14），隠れ層の確率分布は式（2.14）と表現できる．

$$p(\boldsymbol{v}|\boldsymbol{h}) = \prod_i p(v_i|\boldsymbol{h}) \tag{2.13}$$

$$p(\boldsymbol{h}|\boldsymbol{v}) = \prod_j p(h_j|\boldsymbol{v}) \tag{2.14}$$

この時, 可視ユニット, 隠れユニットそれぞれが個々に持つ確率, $p(v_i|\boldsymbol{h}, \boldsymbol{\theta})$, $p(h_j|\boldsymbol{v}, \boldsymbol{\theta})$ は発火確率と呼び，そのユニットの状態が 1 となる確率を表す．つまり，$p(v_i = 1|\boldsymbol{h}, \boldsymbol{\theta})$，$p(h_j = 1|\boldsymbol{v}, \boldsymbol{\theta})$ とも表現でき，その値はそれぞれ，式（2.16），式（2.16）のように表される．この式で，$\sigma(x)$ はシグモイド関数である．

$$p(v_i = 1|\boldsymbol{h}, \boldsymbol{\theta}) = \sigma\left(a_i + \sum_j w_{ij} h_j\right) \tag{2.15}$$

$$p(h_j = 1|\boldsymbol{v}, \boldsymbol{\theta}) = \sigma\left(b_j + \sum_i w_{ij} v_i\right) \tag{2.16}$$

RBM における学習方程式も BM 同様，最尤推定法によって求められるが，モデルの期待値を求める際に全ユニットの組み合わせを計算する必要があり，ユニット数の増大とともに組合せ爆発が発生し，現実的な事件で計算を完了することができなくなる．そこで，パラメタの更新量を近似的に求める，コントラスティブダイバージェンス（CD）法 [22] と呼ばれる手法が用いられる．この手法ではまず，可視ユニットに学習データをセットし，隠れユニットの発火確率を計算し，その確率から隠れユニットの状態をサンプリングする．続いて，サンプリングされた隠れユニットの状態か

ら，可視ユニットの発火確率を計算し，可視ユニットの状態を決定する．n 回目のサンプリングで得られた可視層と隠れ層の状態を，$\boldsymbol{v}^{(n)}$．$\boldsymbol{h}^{(n)}$ と表すと，式（2.17）のような順序で計算することになる．この時，$\boldsymbol{v}^{(0)}$ は入力されたデータである．繰り返し数 k は 1 でも十分であると言われている．

$$\boldsymbol{v}^{(0)} \to \boldsymbol{h}^{(0)} \to \boldsymbol{v}^{(1)} \to \boldsymbol{h}^{(1)} \to \cdots \to \boldsymbol{v}^{(k)} \to \boldsymbol{h}^{(k)} \tag{2.17}$$

以上の手順で可視ユニットと隠れユニットの状態を決定したのち，パラメタ w，b，a の更新量を決める．なお，ε は学習率である．

$$dw_{ij} = \varepsilon \left(P\left(h^{(0)} = 1 | v^{(0)}\right) v^{(0)} - P\left(h^{(k)} = 1 | v^{(k)}\right) v^{(k)} \right) \tag{2.18}$$

$$da_i = \varepsilon \left(v^{(0)} - v^{(k)} \right) \tag{2.19}$$

$$db_j = \varepsilon \left(P\left(h^{(0)} = 1 | v^{(0)}\right) - P\left(h^{(k)} = 1 | v^{(k)}\right) \right) \tag{2.20}$$

2.3　AI ハードウェアの現状

本節では近年の AI ハードウェア開発の現状を俯瞰するため，発表されている各社の AI ハードウェア等についてまとめる．

2.3.1　IBM TrueNorth

TrueNorth[23] は従来の DNN 向けハードウェアとは違い，脳の神経細胞の動作により近い挙動を示す spiking neural network (SNN) 向けに設計されたチップである．これは，将来 1,000 億以上のニューロンと数百兆のシナプスを持つ人間の脳と同程度の規模の脳型コンピュータを実現しようとする挑戦である．大規模な脳型コンピュータを構築する場合，これまでのノイマン型コンピュータでは膨大な電力を必要とし現実的ではないため，専用のチップである TrueNorth が開発された．

2.3.2　Intel Loihi

Loihi[24] も SNN 向けのチップであり，全てディジタル回路で構築されている．Loihi には 1,024 個のニューロンを含むニューロモルフィックコアが 128 個搭載されている．それぞれのニューロモルフィックコアはメッシュ上に配置，接続され，ネッ

トワークを形成している．また，ニューロモルフィックコアには，leaky-integrate and fire モデルと呼ばれるニューロンモデルが実装されている．また，2021 年に第 2 世代である Loihi 2[25] が発表された．

2.3.3 Google TPU

これは Google によって開発された tensor processing unit (TPU) [26] と呼ばれる AI 向けプロセッサである．DNN の演算で大量に必要となる積和演算を効率よく処理するためのシストリックアレイと呼ばれるアーキテクチャを有している．初代の TPU は 8bit 固定小数点が演算に用いられ，推論向けとされていたが，第 2 世代 TPU からは浮動小数点を取り扱い，学習も可能となった．

TPU はさらにアップデートが行われ，2021 年 5 月に行われた同社の開発者向け会議 Google I/O 2021[27] の基調講演で，第 4 世代となる TPU v4 が発表された．TPU v4 では 4,096 個の TPU チップを一つにまとめたポッドと呼ばれる集合一つあたりの演算性能が 1 EFlops に達すると同基調講演で述べられている．さらに，TPU の一部は一般に公開されており，クラウドサービスとして一般のユーザでも利用可能である．

2.3.4 NVIDIA GPU

GPU は元来，画像処理を行うためのプロセッサであったが，その高い並列性からより一般的な用途への応用が行われている．しかし，高並列なプログラムを記述することは一般的に難しい．そこで，CUDA と呼ばれる専用言語が提供されている．CUDA に対応する深層学習フレームワークなどを用いることで開発者は容易に GPU で並列動作するプログラムを構築できる．

一方，最新の GPU アーキテクチャは Ampere[28] と呼ばれるもので，既に製品としてリリースされている．Ampere アーキテクチャには Tensor コアと呼ばれる AI 向けの行列の積和演算に特化したコアが搭載されている．Tensor コアは Ampere より 2 世代前の Volta アーキテクチャで初めて搭載され，改良が加えられながら現在に至る．

2.3.5　Microsoft による FPGA の活用

この事例は Microsoft が自社のデータセンタに GPU の代替として FPGA を導入し，電力効率の向上を図った取り組み [29] である．当時，一般的には演算の高速化を実現するためのアクセラレータとして GPU が主流であった．しかし，本取り組みでは，データセンタにアクセラレータとして FPGA を導入した．その結果，従来の GPU を用いた場合と比較して二倍以上電力効率が向上したとの結果が報告されている．

2.3.6　Xilinx ACAP

Xilinx は多数の FPGA シリーズを市場に投入している．この中でも，adaptive compute acceleration platform（ACAP）[30] と呼ばれるプラットフォームは一つのチップに ARM アーキテクチャの CPU，FPGA，DSP 等の専用回路を搭載したヘテロジニアスな構造を持つ．Xilinx は CPU，FPGA，専用回路をそれぞれ，Scalar Engines，Adaptable Engines，Intelligent Engines と呼んでいる．これらのエンジンはチップ内で 1Tb/s 以上の帯域の Network-on-Chip（NoC）で接続され，それぞれのエンジンが協調して処理を実行することが可能である．ACAP は AI や信号処理など用途ごとに搭載するエンジンが異なる複数の種類が提供されている．

特に，AI 向けの ACAP である AI コアシリーズには，AI エンジン [31] と呼ばれるベクトルベースのアルゴリズムに最適化された Single Instruction Multiple Data（SIMD）プロセッサが搭載されている．これを用いることで，FPGA 部分（Programmable Logic: PL と呼ぶ）に同じアプリケーションを実装した場合よりも最大で消費電力を 50% 削減できるとしている．ACAP は，AI エンジンのほかにビデオデコーダユニットなどを搭載し，各種処理を適切なコアに割り当てることで高性能で高効率なシステムの実現を目指している．

このように FPGA や専用プロセッサ，CPU を含めたヘテロジニアスな構造を持ったチップが近年登場してきている．

2.3.7 PC 向け CPU および SoC

近年の PC 向け CPU では自社開発の System-on-a-Chip（SoC）の採用や，ヘテロジニアスな構成の CPU の使用などが行われている．自社開発の SoC を使用した事例として，記憶に新しいのは，Apple による AppleSilicon[32] の採用である．Apple は自社のパソコンで従来採用していた Intel 製の x86 アーキテクチャの CPU から自社で開発した Apple Silicon と呼ばれるチップへの置き換えを進めている．このチップにもニューラルネットワーク専用の演算回路（同社は Neural Engine と命名している）が組み込まれている．また，高い処理能力を有する演算コアと電力効率の高い演算コアを組み合わせたヘテロジニアスな構成となっている．

さらに，PC 向け CPU などを製造する Intel は 2021 年 10 月に第 12 世代インテル Core プロセッサを発表した [33]．発表によると，第 12 世代 CPU は，高いパフォーマンスを有する Performance-cores（P-Cores）と高い電力効率を有する Efficient-core（E-Cores）と呼ばれる 2 種類の演算コアを搭載したヘテロジニアスな構造となっている．これにより，各種ソフトウェアの処理を適切なコアに割り当てることで低消費電力でありながら高いパフォーマンスを出すことができるとしている．

このように，データセンタやサーバ用途の専用チップのみならず，一般的なユーザが使用する PC へも独自開発の SoC 導入や，ヘテロジニアスな構造を有する CPU が発表され，搭載されつつある．

2.4 ハードウェア乱数生成器

乱数生成器は今日の様々なアプリケーションを支える上でなくてはならない重要な存在である．乱数を必要とするアプリケーションは多岐に渡り，例えば，通信などの暗号化，モンテカルロ法などの数値シミュレーションなどはその代表的なものである．さらに，深層学習をはじめとする，ニューラルネットワークにおいても乱数生成器は必要な存在となっている．ニューラルネットワークは大別して，確定的な動作を行うものと確率的動作を行うものが存在する．次章で詳しく述べるが，確率的に動作するニューラルネットワークには，生成モデルと呼ばれるものが含まれる．確率的に動作するモデルでは，与えられた確率をもとに，何らかの事象が発生するか否かを決定する必要がある．この決定するプロセスに乱数が用いられるため，確率的なニュー

ラルネットワークでは乱数生成器は非常に重要な存在となる．本節では，主にハードウェアにおける乱数生成器について概観し，いくつか具体例を取り上げる．

まず，乱数生成器は真性乱数生成器（true random number generator: TRNG）と擬似乱数生成器（pseudorandom number generator: PRNG）の 2 種類に大別できる．TRNG は完全に予測不可能かつ再現不可能な乱数列を生成する乱数生成器である．つまり，真の乱数列を生成することができる．そのために，予測不能な物理現象やシステム外部のノイズなどを利用している．一方，PRNG は一定のアルゴリズムに従って擬似的な乱数列を生成するものである．つまり，確定的手法で乱数列を生成する．ここでの擬似的な乱数とは，一見乱数のように見えるが，極めて長い周期で観測すると規則性を持っている場合や，乱数を生成開始する際の初期条件を指定すると全く同じ乱数列を生成可能であるといった再現性を持つ．しかし，PRNG で生成された乱数列が擬似的ではあるとはいえ，統計的検定などを実施し，乱数を必要とするアプリケーションが求める性質を満たしていれば乱数として活用することができる．

次に，TRNG のハードウェア実装について述べる．TRNG は予測不可能な物理現象を用いることで乱数列を生成する．FPGA のハードウェアに実装されるものでは，メタスタビリティを用いたもの [34] やリングオシレータを用いたもの [35] などが存在する．さらに，身近な例として，コンピュータの OS である Linux ではキーボードやマウスの操作など，動作が予測不可能なデバイスからノイズを収集し，エントロピープールと呼ばれる領域に乱数として保存し，ユーザの要求に応じてこの乱数を提供している．このように TRNG では予測不可能な挙動を示す現象を用いて乱数列を生成するため，予測不可能，再現不可能な乱数を生成することができる．しかし，物理現象を利用する性質上，乱数の生成に時間を要する場合や，実験など乱数を用いるものの再現性も持たせたい場合には不向きである．

続いて，PRNG のハードウェア実装について述べる．基本的なものとして線形帰還シフトレジスタ（linear feedback shift register: LFSR）[36] がある．これは Fig. 2.7 に示すようにシフトレジスタを用意し，その一部の値を取り出し（取り出す部分をタップと呼ぶ），XOR 演算を施したのちに，シフトレジスタの末尾へ代入する．この時，初期値とタップの位置を適切に設定することで，M 系列疑似乱数と呼ばれる，設計上最も長い周期を持つ擬似乱数を生成することが可能である．例えば，n ビットの LFSR の場合は，$2^n - 1$ 周期の擬似乱数となる．ここで述べたものはフィボナッチ LFSR と呼ばれるもので，他には，シフトレジスタの途中に XOR 演算を挿入するガロア LFSR といった形式も存在する．また，xorshift[37] と呼ばれる乱数生成手

法も Code 2.1 に示すように，XOR とシフト演算だけで乱数を生成することができ，しばしば用いられる．xorshift は 32bit 整数の 4 つの内部状態を保持していて，この内部状態に対して XOR 演算とシフト演算を実行することで $2^{32}-1$ 周期の 32bit 乱数を生成することができる．この手法は XOR とシフト演算のみで実現されることから，FPGA などのディジタル回路では組合せ回路のみで実現することが可能であり，FPGA 内蔵のハードウェア乗算器（DSP ブロック）が不要であるため，実装時に求められる回路資源も少ない．

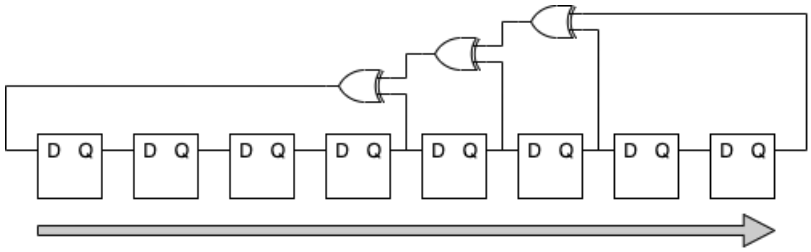

Fig. 2.7 8 bit フィボナッチ LFSR

Code 2.1 xorshift の例

```
1  int xorshift(void){
2      unsigned int tmp, x, y, z, w;
3      x=123456789, y=362436069, z=521288629, w=88675123;
4
5      tmp = x^(x<<11);
6      x = y;
7      y = z;
8      z = w;
9      w = (w^(w>>19))^(tmp^(tmp>>8));
10     return w;
11 }
```

ここで述べたように，FPGA などのハードウェア実装に向けた乱数生成器は多数提案されているが，乱数生成器が必要とする回路資源は必ずしも少ないものではない．例えば LFSR では長周期の乱数を生成するためにはシフトレジスタの規模が長大なものになる．また，アプリケーションのレイテンシを上げるために並列に複数の

乱数生成器を実装すれば，当然必要となる回路資源も増大する．このように，アプリケーションが乱数を必要とする限り，乱数生成器もまた必須の要素となり，どの程度の品質の乱数が必要なのか，どの程度並列化したいのかもしくは並列化可能なのかを考慮しなければならない．乱数生成器を実装する際には，FPGA などのハードウェアと実装したいアプリケーションが必要とする回路資源の規模から適切な実装方法を考えなければならない．時には，乱数生成器の規模が足枷となることもある．

第 3 章

切り捨てビットを用いた制限付きボルツマンマシンの実装法の提案

本章では FPGA などのディジタル回路向けの，切り捨てビットを用いた乱数生成器を用いない，省資源なニューラルネットワーク，特に RBM の実装手法を提案する．そのために，提案手法を解説する前に，本手法の基本となる固定小数点演算について述べ，乱数生成器の必要性と課題を指摘する．また，章末に提案手法の利点について言及する．

3.1 固定小数点演算

FPGA などのディジタル回路にアプリケーションを実装する場合は，固定小数点二進数が一般的には用いられる．CPU を搭載するパソコン上のソフトウェアでは一般的に浮動小数点表現による数値演算が実装されるが，これは数値の取扱が複雑であり，FPGA に実装する場合には，固定小数点二進数に比べて求められる回路資源が大きくなり，実装効率が下がってしまう．一方，固定小数点二進数を用いると，同じビット幅の場合，表現できる数値の幅や精度の面で劣る．そのため，ハードウェアになんらかの演算を実装する場合は，固定小数点化による演算精度の低下に注意する必要がある．

固定小数点数と浮動小数点数の数値の表現の違いを Fig. 3.1 に示す． 固定小数点表現では，小数点の位置を固定し，整数部 x bit 目の数値を 2^{x-1} の数値として表現し，小数部 y bit 目の数値を 2^{-y} の数値を表す．また，符号ビットは 1 を負の値とし

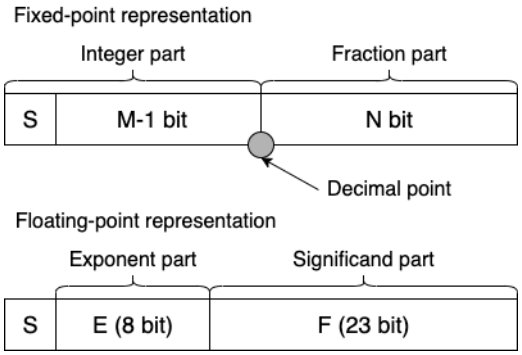

Fig. 3.1　固定小数点表現と単精度浮動小数点表現

て取り扱う．小数点の位置を固定しているので，ディジタル回路で演算回路を実装する際には，小数点の位置を合わせて演算すれば整数演算と同様に簡単な回路で実現することが可能である．

浮動小数点数は符号ビット（S），指数部（E），仮数部（F）で構成され，一般的には以下に示す式として，数値を表現している．なお，符号ビット S は 1 が負の値を表す．

$$(-1)^S \times F \times 2^E \tag{3.1}$$

浮動小数点表現では，数値を $1.x_2 \times 2^y$ のように，小数点の左側には 1 が一桁のみ存在するよう，数値の桁をずらし，その分を 2^y で表現するものである．この時，x を仮数（significand），y を指数（exponent）と呼ぶ．実際のコンピュータへの実装は，IEEE754 浮動小数点規格によって定められている．浮動小数点形式の数値は，広範囲な数値を表現することが可能である一方，表現方法が固定小数点表現と比較して複雑であるため，ディジタルハードウェアで演算器を実装する場合，回路が複雑になる．

3.2　乱数生成器の必要性と課題

RBM をはじめとする生成モデルは，データセットを生成している確率分布を学習する．ニューラルネットワークを構成する各ユニットはそれぞれ発火確率を持つ．RBM の場合は各ユニットがそれぞれ状態が 1 となる確率 $p(x = 1|\boldsymbol{\theta})$ を持っている．この式では x はそのユニットの持つ状態，$\boldsymbol{\theta}$ はネットワークのパラメタである．生成モデルはこのように確率を扱うネットワークであるため，発火確率からユニットの状態を決定する，サンプリングと呼ばれる処理が必要となる．このサンプリング時に乱

数が用いられる．具体的なサンプリング処理の流れを見ていく．まず，あるユニットが持つ発火確率を $p(x = 1|\boldsymbol{\theta})$ とおき，[0, 1] の範囲の乱数を rnd とおく．この際の乱数は，一様分布に従う乱数であり，乱数生成器や乱数生成アルゴリズムに従って生成される．C++ 言語の場合は，random ヘッダにてメルセンヌツイスタ法など様々な乱数生成手法が定義されている．発火確率からサンプリングする場合は，発火確率と乱数を比較して，以下のようにユニットの発火もしくは非発火を判定する．

$$rnd < p(x = 1|\boldsymbol{\theta}) \Rightarrow x = 1 \tag{3.2}$$
$$rnd \geq p(x = 1|\boldsymbol{\theta}) \Rightarrow x = 0 \tag{3.3}$$

つまり，確率に対して乱数の値の大小で判定を行う．そのため，サンプリングに使用する乱数の分布に偏りがあると，サンプリング結果にも偏りが生じることで，確率から正しく公平にデータをサンプリングしたとは言えなくなる．これが乱数に一様性が必要であることの理由である．これを Fig. 3.2 に図示する．

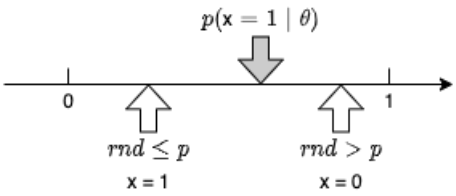

Fig. 3.2　乱数を用いた確率からのサンプリング例

以上より，RBM などの生成モデルにおいては，一様分布から生成される乱数は重要な役割を占めており，これを FPGA などのハードウェアに実装する際には乱数生成器の実装が求められる．

しかし，ハードウェアに実装される乱数生成器は多くの回路資源を必要とするため，例えば各ユニットごとに乱数生成器を配置するような高並列な回路設計では，この部分が消費する回路リソースを無視できない．具体的な事例を紹介すると，IBM が AI 向けチップとして開発した TrueNorth の場合，文献 [23] によると，一つのニューロンを実現するニューロンブロックのうち，メモリを除くと，全体で 1,272 個のゲートを使用している．これの内訳は，ニューロンモデルの計算部分に 924 個，乱数生成器に 348 個である．ニューロンの動作を実現するロジック全体に対して，約 27%，つまり 1/4 以上のリソースを乱数生成器のために消費していることになる．この乱数生成器を削減することができれば，ニューラルネットワークの実装をより省資源に実

現可能であると考えられる．そこで，次節では本研究における切り捨てビットを用いた乱数生成器を必要としないニューラルネットワークの実装手法について述べる．

3.3　切り捨てビットを用いた RBM の実装手法

本研究では固定小数点二進数を用いた演算の際に発生する切り捨てビットを乱数の代替として用いる手法 [38, 39] を提案する．何らかの演算を FPGA などのディジタルハードウェアに実装する際には，先に述べた通り，固定小数点二進数が用いられる．そこで，固定小数点で演算する際，整数部と小数部にそれぞれ M bit，N bit をもつ変数があった場合を考える．この値同士の乗算の結果は，Fig. 3.3 に示すように整数部が 2M bit，小数部が 2N bit となる．さらにニューラルネットワークでは乗算結果の総和が行われるので，整数部の bit 幅が桁のくり上がりによって増加する．図中では，このくり上がり部分を a bit と表現している．ハードウェアではこの演算結果をレジスタに格納する際，増加した分の bit を切り捨てることで元のビット幅を維持する．提案手法では，小数部で切り捨てられた bit を乱数の代替として利用する．整数部の切り捨てビットについては，本手法では使用しない．これは，整数部はもし

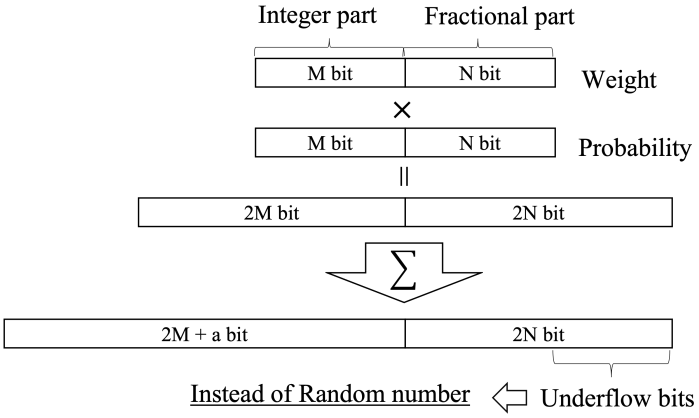

Fig. 3.3　固定小数点の積和演算におけるビット長の変化と切り捨ての様子

そのビット幅で表現可能な最大値を超える値を示した場合，数値はその最大値，つまり全ビットが 1 となる状態を維持するため，乱数の代替としてのランダム性を得られないと考えたためである．これは，過去に実施した SIDBA を用いた実験 [38] において，整数部と小数部の切り捨てビットのヒストグラムが取得され，整数部については

切り捨て値が大幅な偏りがある傾向がわかっている．この時の整数部の切り捨てビットのヒストグラムを Fig. 3.4 に，小数部の切り捨てビットのヒストグラムを Fig. 3.5 に示す．

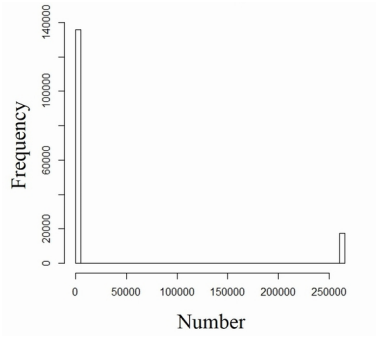

Fig. 3.4　整数部の切り捨てビットに関するヒストグラムの参考値

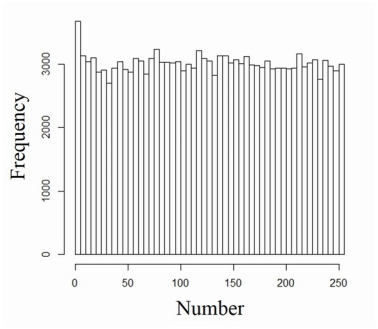

Fig. 3.5　小数部の切り捨てビットに関するヒストグラムの参考値

そこで，本研究では，本手法を RBM に適用し，ユニットの発火確率からその状態をサンプリングするための乱数の代替としてこの切り捨てビットを用いる．この手法を用いるために，隠れユニットの発火確率算出時とパラメタ更新量算出の式で可視ユニットの状態を用いている部分を，可視ユニットの発火確率 $p(v_i = 1|\boldsymbol{\theta})$ に置き換えて学習を行う [40]．以下に提案手法を用いた場合の学習の流れを示す．

1. 学習データ画像の画素値を 0 から 1 に正規化し可視ユニットの発火確率として入力
2. 隠れユニットの発火確率を計算

3. 発火確率計算時の積和演算で生じた切り捨てビットを保存し，0 から 1 の値に正規化（rnd とおく）
4. 隠れユニットを発火確率に基づきサンプリングする（$rnd \leq p(h_j = 1|\boldsymbol{\theta})$ でユニットの状態を 1 とし，それ以外では 0 とする）
5. 可視ユニットの発火確率を計算
6. パラメタの更新

以上の手順でパラメタの更新を行うことで，RBM で使用する乱数を切り捨てビットから生成されたもので置き換えることができる．RBM の FPGA 実装については従来もさまざまな研究が行われてきた [41, 42, 43, 44, 45] が，切り捨てビットを乱数の代替に用いることで，従来必要とされていた乱数生成器を取り除くことが可能となり，FPGA などのハードウェアに実装した際，乱数生成器で占められていたハードウェア領域を他の回路に活用することができる．次に，本手法の利点についてまとめる．

3.4　本手法の利点

乱数生成器のハードウェア実装では，乱数生成器をシリアルに配置するか，パラレルに配置するかで，レイテンシと消費回路資源のトレードオフが発生する．この関係性を Fig. 3.6 に示す．乱数を必要とするユニット複数につき，乱数生成器を共有する場合をシリアルな配置（serial RNG）と表記している．この場合，最初のタイミング T=1 で一つ目のユニットに乱数を提供し，次のタイミング T=2 で二つ目のユニットに乱数を提供する，といった繰り返しで乱数を各ユニットに配布する．そのため乱数生成器の個数は少ないが，全ユニットに乱数が行き渡るまでに時間がかかる（レイテンシが大きい）．一方，乱数生成器を各ユニットごとに持たせる場合を，並列な配置（parallel RNG）と表記している．図に示すように，全てのユニットに同じタイミングで乱数が行き渡るため，レイテンシは小さいが，乱数生成器を並列配置する分，必要となる回路資源は大きくなることがわかる．乱数を用いるアプリケーションをハードウェア実装する際には，このような乱数生成器をどのように配置するかが一つの大きな問題であり，一つの乱数生成器を共有するアーキテクチャを有するもの [46] や並列に実装するもの [23] など設計思想や回路実装時の要求仕様などから，さまざまな選択がなされている．

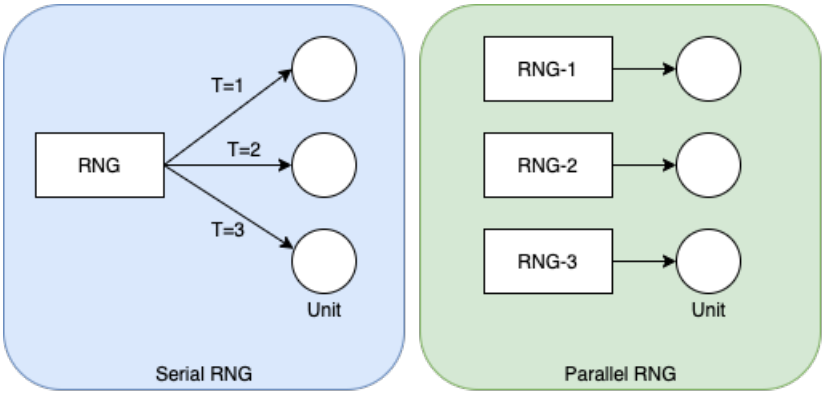

Fig. 3.6　乱数生成器のシリアルな配置とパラレルな配置

そこで，本手法を用いることで，専用の乱数生回路を実装する必要がなくなることはこれまでに述べた通りである．さらに，ユニット内での入力値と結合荷重の積和演算時に切り捨てビットを取得することができるため，

1. データの入力
2. 積和演算
3. 切り捨てビット発生
4. 積和演算結果と切り捨てビットを取得
5. サンプリング

とデータの入力からサンプリングまでを円滑に実施することが可能である．つまり，途中で乱数生成器の応答待ち時間の発生がない．この流れを Fig. 3.7 に示す．この図において，x_i は入力，w_i は結合荷重である．それぞれの乗算を計算し，その後それぞれの和を取り，切り捨てビットと演算結果を取得する．

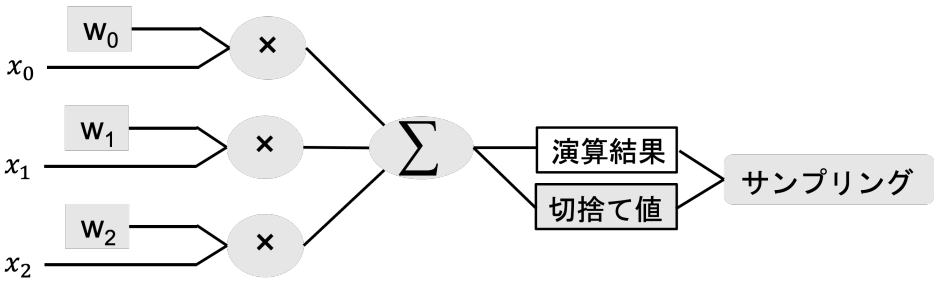

Fig. 3.7　データ入力からサンプリングまでの流れ

第 4 章

評価ハードウェア基盤の構築

本章では，まず初めに，FPGA を用いたシステムの例として，FPGA に搭載された SNN 向け SIMD プロセッサに SNN を構成するニューロンモデル 1 個を実装し，シミュレータ上で動作させる．これに対し，回路のモジュール単位での動作検証や，x86 プロセッサを搭載したコンピュータとの協調動作を実現するための新しいプラットフォームを提案する．このプラットフォームに関しては，FPGA 上に構築されるハードウェアの制御の仕組みを述べた後，コンピュータ上で動作するソフトウェアの設計について述べる．

4.1 FPGA におけるニューラルネットワークの実装例

第 1 章でも述べたとおり，FPGA に専用回路を構築し，データ処理などを高速化，低消費電力化する手法が取られる場合がある．その一例として，J.Madrenas らによって開発されている SNAVA と呼ばれる SNN 向けマルチコアプロセッサ [47] が挙げられる．これは FPGA に実装された single instruction multiple data（SIMD）プロセッサであり，Fig. 4.1（[47] より引用したものに PE 部分を強調するため加筆）に示すように，100 個の processor element（PE）が配置されている．この PE がそれぞれ 1 個のニューロンの挙動を計算する．本アーキテクチャではそれぞれの PE は AER（address event representation）バスによって接続され，SNN のスパイクを他のニューロンを表現する PE に対して伝達する． また，それぞれのレジスタには演算で使用するデータを保存するための 16 bit 幅のレジスタが 8 つ，その他値の保存に使用される 16 bit 幅のレジスタが 8 つ存在する．それぞれ Active register と

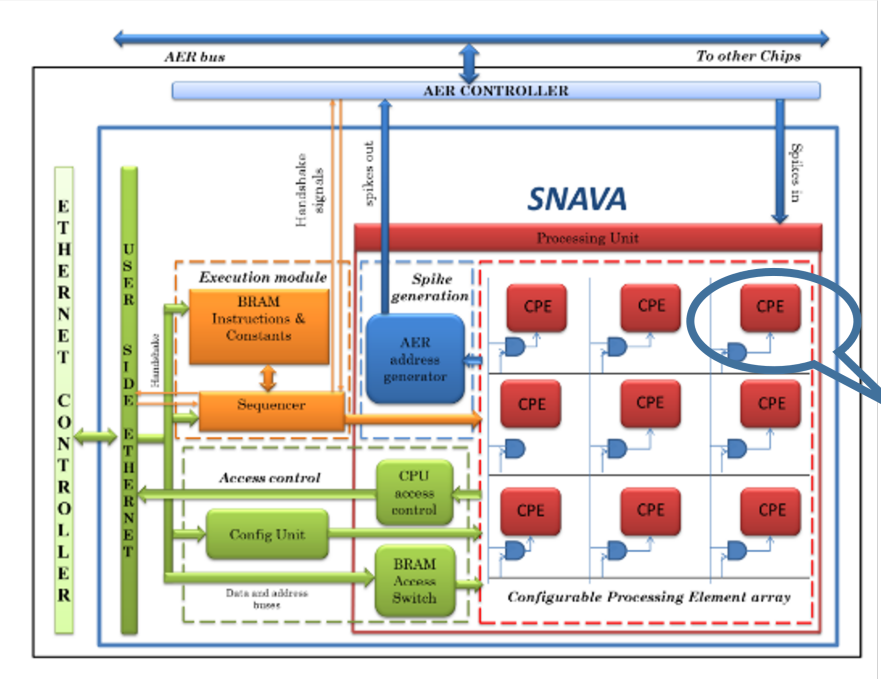

Fig. 4.1　SNAVA アーキテクチャ

Shadow register と呼ばれ，それぞれのレジスタ間でデータを入れ替える（swap 命令）ことで，演算用データと長期保存用データの切り替えを行う仕組みとなっている．このデータのやり取りを Fig. 4.2 に示す．

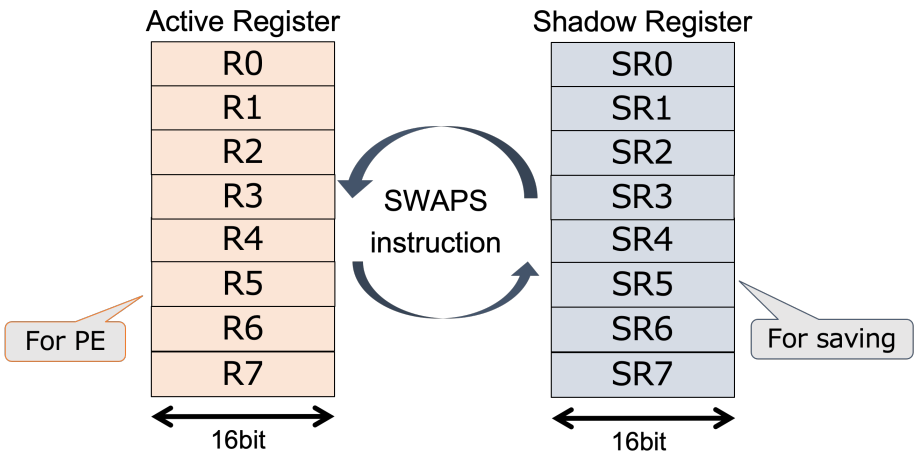

Fig. 4.2　Active register と shadow register

この SIMD プロセッサを用いたアプリケーションの実装には，専用のアセンブリ命令セットを用いる．ユーザはこのアセンブリ命令を用いてプログラミングを行い，シミュレータで動作を確認し，実機実装を行う，といった開発工程を経る．

次に，実例として，簡単な SNN ニューロンモデルをアセンブリで構築し，シミュレーション上で動作させた [48]．ここでは SNN のモデルとして，digital spiking silicon neuron（DSSN）モデル [49, 50] を実装した．このモデルは SNN を回路で実装することを目指したモデルであり，ニューロンの膜電位は，次の式で定義される．

$$\frac{dv}{dt} = \frac{\varphi}{\tau}\left(f(v) - n + I_0 + I_{stim}\right) \tag{4.1}$$

$$\frac{dn}{dt} = \frac{1}{\tau}\left(g(v) - n\right) \tag{4.2}$$

$$f(v) = \begin{cases} a_n(v + b_n)^2 - c_n & (v < 0) \\ -a_p(v - b_p)^2 + c_p & (v \geq 0) \end{cases} \tag{4.3}$$

$$g(v) = \begin{cases} k_n(v - p_n)^2 + q_n & (v < r) \\ k_p(v - p_p)^2 + q_p & (v \geq r) \end{cases} \tag{4.4}$$

ここで，この式の各変数の意味は次の通りである．v: 膜電位, n: イオンチャネルの活動を簡易表現する変数, φ, τ: 時定数, I_0: 一定の刺激電流, I_{stim}: 入力電流, その他の変数はニューロンの動作を決定するパラメタである．

また，シナプスの出力電流 I_s は以下の式で表される．この式における α, β は時定数である．

$$\frac{dI_s}{dt} = \begin{cases} \alpha(1 - I_s) & (v \geq 0) \\ -\beta I_s & (v < 0) \end{cases} \tag{4.5}$$

この例では，一つのニューロンの動作を SNAVA 用に開発されたアセンブリ命令セットを用いて実装し，ハードウェア記述言語である VHDL によるハードウェアシミュレーションで SNAVA の動作をシミュレーションした．また，比較用にソフトウェア上でにも同様のニューロンを実装した．この時，実装したニューロンに Fig. 4.3 に示すようなステップ上の入力を与え，その出力を，SNAVA シミュレーションの結果とソフトウェアシミュレーションの結果を比較した．この時の出力の波

形を Fig. 4.4 に示す．グラフ中のハードウェアと書かれた橙色の線がハードウェアシミュレーションで得られた結果である．

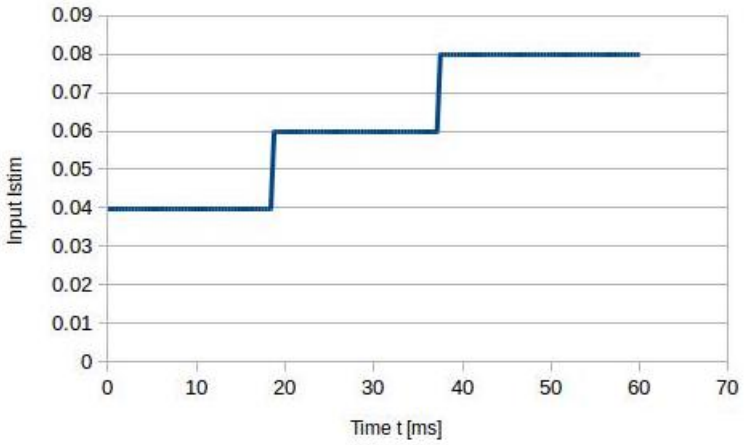

Fig. 4.3　入力波形

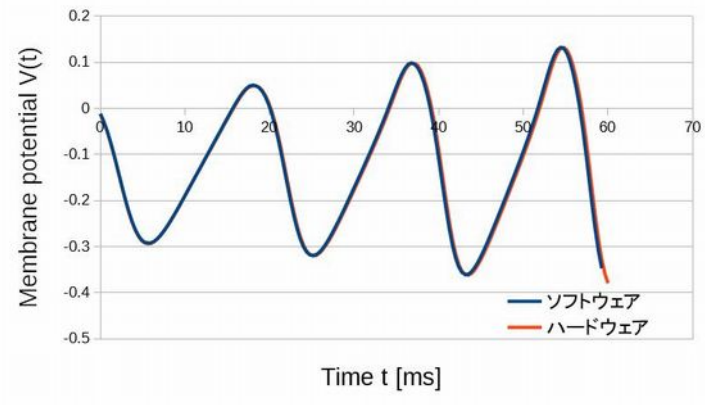

Fig. 4.4　ソフトウェアとハードウェアによる出力波形

このグラフからわかるように，SNAVA のハードウェアシミュレーションとソフトウェアによるシミュレーションが一致し，SNN 向け SIMD プロセッサが正しく動作していることが示された．

しかし，ここでの結果は SNN 向けの SIMD プロセッサを用いたものである．実装するアプリケーションが SNN であり，提供されているアセンブリ命令セットで実装可能であれば，このような専用プロセッサに専用の命令セットを用いて実装する

手段が効率的である．一方で，新しいハードウェア実装の仕組みの検証や独自に開発した IP 単体での検証といった用途は専用プロセッサでは実現できない．また，近年のアプリケーションでは FPGA とソフトウェアが協調し，互いが得意とする処理をそれぞれに割り当てて動作させる仕組みが構築されてきている．そこで，ハードウェアとソフトウェアが協調して動作し，なおかつ開発者が独自に作製した回路を簡単に FPGA に実装し，簡単にソフトウェアと連携できる仕組みが必要である．このような仕組みを用意することで，ユーザが作製した回路を簡単に検証することができれば，将来的に SNAVA のようなシステムに円滑に独自に作製した回路を追加し，機能のアップデートが可能になる．次節では，既存のハードウェアとソフトウェアの連携システムについて述べる．

4.2 近年のハードウェアソフトウェア連携システム

データ処理を FPGA にオフロードし，処理を高速化する場合は，システム全体は FPGA のみでは完結しない．FPGA を制御し，データの送受信を行うためのホストとなる存在が必要である．このホストは一般的には Linux などのオペレーティングシステムが稼働したサーバやパソコンなどのノイマン型コンピュータである．このような FPGA とノイマン型コンピュータによるホストとの協調システムとして，hw/sw（ハードウェア/ソフトウェア）複合体 [51] も提唱されている．hw/sw 複合体では FPGA の持つ動的再構成性を利用して，FPGA 上のハードウェアも含めて，オブジェクト指向プログラミングを可能とするものである．FPGA の活用においては，このようなソフトウェアとハードウェアの円滑な協調動作が重要となっている．

近年では，ハードウェアとソフトウェアが協調動作するシステムを実現するために，２種類のアプローチがとられている．まず第一のアプローチは，Linux などの OS が稼働し，ソフトウェアを実行できるプロセッサと FPGA を一つのチップに搭載し，SoC として完結させたものである．Xilinx 社の場合は Zynq と呼ばれるチップでこの構成を実現している．Zynq の場合は，ARM プロセッサを搭載しソフトウェアを実行する部分を processing system（PS）と呼び，FPGA 部分を programmable logic（PL）と呼ぶ．PS と PL は同じチップに統合されており，それぞれでデータの転送が可能である．また，PS 側から PL 側に回路を書き込むことも可能である．第２のアプローチは，アクセラレータカードと呼ばれる FPGA とメモリを搭載したデバイスを PCI-Express（PCIe）インタフェースによりサーバなどのホスト PC に接

続する方法である．Xilinx 社の場合は Alveo と呼ばれ，Fig. 4.5 に示すようなものである．これと同時にホスト PC と FPGA の通信をシームレスに実現できるような開発環境も提供されている．このようにして，ソフトウェアとハードウェアが強調するシステムが現在提供されつつある状況である．

しかし，これらのシステムにはいくつかの問題がある．まず，Zynq においては搭載される CPU が ARM であるため，この CPU 上で実行されるソフトウェアは ARM 向けにコンパイルされたものである必要があり，従来の x86 対応のバイナリでは動作しない．さらに，SoC である性質上，CPU のカスタマイズなどは不可能である．また，CPU の演算能力も大規模なサーバやワークステーション向けの CPU と比較すると非力である．組込み用途には的しているが，大規模演算に耐えうるものではない．

一方，Alveo のようなアクセラレータカードの場合，ホスト PC はサーバやワークステーションといった一般的な x86 互換アーキテクチャの CPU を搭載したマシンである．さらに，アクセラレータカードに搭載される FPGA は大規模なものが用いられるため，回路資源のを多く必要とするアプリケーションの実装も可能である．しかし，Alveo は数 10 万〜100 万円程度と高価であり，新しい世代のデバイスでの対応となる．さらに，ホスト PC に求められる性能も高く，ワークステーションやサーバのような高性能なマシンが必要となる．そのため，実用的なアプリケーションを実装し FPGA の恩恵を受ける場合には適するが，比較的小規模な回路の検証にはコストがかかりすぎる．

そこで，本研究では，x86 プロセッサを搭載したコンピュータと一般的な FPGA 評価ボードを接続し，協調動作させるためのプラットフォームを構築する [52]．このシステムにより，ユーザは必要に応じて自由に FPGA 評価ボードを選択することが可能となるとともに，ホストマシンとして，x86 互換の CPU を搭載した Linux や Windows といった OS が稼働する環境でソフトウェア開発を行うことができる．

4.3　システム構成

本節では本研究で構築したシステムの構成を解説する．本システムの全体像を Fig. 4.6 に示す．本システムでは PCIe によりホスト PC に FPGA 評価ボードを接続している．この PCIe 接続を実現するために，Xillybus[53] と呼ばれる IP コアを使用した．Xillybus を用いることで，ユーザやハードウェア開発者はホスト PC

Fig. 4.5　Alveo アクセラレータカード

と FPGA のインタフェースを自ら整備する必要がなくなる．また，デバイスドライバも提供されるため，Windows，Linux 問わず使用することができる．FPGA 内部では，Xillybus Core と呼ばれる Xillybus が提供する回路により，PCIe による通信を AXI-Stream と呼ばれるインタフェースに変換し，ユーザに提供している．AXI-Stream は近年の FPGA 内部バスとして用いられることの多いプロトコルのインタフェースである．この他に AXI や AXI-Lite も FPGA の内部バスとして用いられることが多いが，AXI-Stream はストリームデータの転送に特化しており，アドレスの概念がなく簡便なハンドシェイクによってのみ通信が実現される．また，ユーザ側には 32 bit 幅と 8 bit 幅の AXI-Stream ポートが用意されている．

Xillybus Core 以下に接続される各種回路が本研究で構築したホストマシンと FPGA の協調動作を実現するための各種コンポーネントである．本プラットフォームは，AXI-Stream to AXI Bridge，Special Function Register（SFR），AXI Interconnect，BRAM Controller，User Logic で構成されている．ここで，User Logic とは実際にユーザが FPGA 上で実現したい処理を実装した回路のことである．User Logic はホスト PC と直接通信するための 32 ビット幅の AXI-Stream インタフェースをもち，主にデータの送受信に用いる．また，FPGA 内部へは AXI インタフェースによって各種モジュールと接続されている．この AXI インタフェースによりユーザロジックはホスト PC から制御される．User Logic 以外の回路は Host PC から User Logic を制御したり FPGA の状態をホスト PC に伝達するといったホスト PC と FPGA の協調動作のための機能を提供するモジュールである．これらのモジュールや具体的な User Logic の制御方法については次節で解説する．

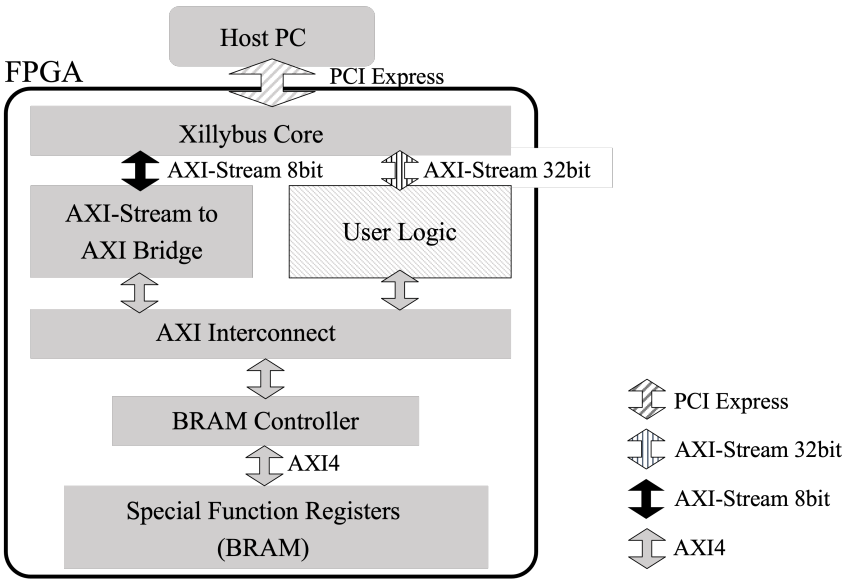

Fig. 4.6　Overview of the hardware system

4.4　User Logic 制御方法

本節ではホスト PC から FPGA 内に書き込んだ User Logic の制御方法について述べる．Fig. 4.7 にホスト PC からユーザロジックを制御する信号の流れを示す．制御信号は 8 bit 幅の Xillybus インタフェースを用いてホスト PC と送受信される．ユーザロジックの制御は全て，SFR と呼ばれるレジスタを経由して行われる．例えば，ホスト PC から SFR になんらかのコマンドを書き込むと，これを User Logic が読み込み，そのコマンドに従った処理を実施する．一方，User Logic からホスト PC へ回路の処理状態といったステータスなどの情報を送信する場合は，SFR にユーザロジックが書き込みを行い，ホスト PC がこれを読み込んで User Logic からの情報を取得することができる．User Logic からホスト PC への通信は，主に，ホスト PC からの実行開始命令に対して，User Logic が実行完了を通知する際などに用いられる．なお，SFR は FPGA 内のブロック RAM（block RAM: BRAM）を用いて実装されている．

SFR はそこに格納される 8 bit の値がどのような意味を示すか，ユーザが自由に定義することができる．SFR の構造とその定義例を Fig. 4.8 に示す．この図に示した

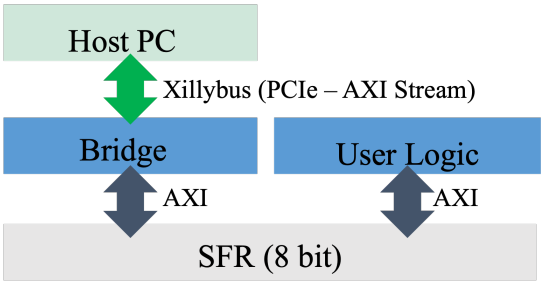

Fig. 4.7 ホスト PC からユーザロジックへの制御信号の流れ

ように SFR は 8 bit 幅で複数のレジスタを保持している．アドレス空間は 7 bit 幅で定義されているので，128 個のレジスタを用意することができる．図中の Data に示す部分が実際に各アドレスの SFR が保持する値である．この例では，LSB1 bit が実行開始フラグ E（execution）を意味し，これがアサートされるとユーザロジックの動作開始を意味する．また，Mode 部分は User Logic の動作を指定するための数値である．

Addr	Data (8 bit)		
0x00	Don't care	Mode	E*

Fig. 4.8 SFR のレイアウト例

Fig. 4.8 に示したレイアウトの SFR で User Logic が制御される流れを以下に列挙する．

1. FPGA に User Logic を含むシステムが書き込まれる
2. User Logic が常に SFR の E フラグを監視する
3. ホスト PC が Xillybus を介して Mode 部分の数値をセットする
4. ホスト PC が Xillybus を介して E フラグをアサートする
5. E フラグを監視している User Logic が動作を開始する
6. User Logic が Mode で指定された処理を行う
7. User Logic が処理を完了次第 E フラグをデアサートする
8. ホスト PC が Xillybus を介して SFR の E フラグを監視しでアサートを検知する

9. ホスト PC が User Logic の動作完了を認識する

以上が最も基本的な SFR を用いた User Logic の制御方法である．なお，User Logic には 32 bit 幅の Xillybus を用いたデータ専用バスが接続されている．このバスを用いたデータ転送は User Logic が受付可能状態に設計されていれば SFR による制御と関係なくデータ転送が可能である．しかし，一般的な使用方法では，SFR の E フラグがアサートされ，User Logic が動作開始した後，このデータ専用パスによってホストとのデータの送受信を実施する．

4.5　SFR アクセス制御

SFR は FPGA 内部の User Logic と Host PC 両方からアクセスされる．SFR は Xilinx の提供する BRAM Controller によって AXI インタフェースとしてアクセスすることが可能である．そこで，SFR，User Logic，Host PC との接続を AXI インタフェース同士を接続するための AXI Interconnect で相互に接続する．バスの調停や接続先の選択などは AXI Interconnect が AXI プロトコルに基づいて実施する．

しかし，ホスト PC と FPGA の接続を実現している Xillybus Core は AXI-Stream インタフェースを FPGA 内部に提供しているので，AXI-Stream による通信と AXI による通信を相互に変換する必要がある．そこで，AXI-Stream to AXI Bridge とする回路を実装した．この回路はアドレスのデータが存在しないストリームとしての AXI-Stream によるデータ通信をアドレスの概念が存在する AXI によるデータ通信に変換する．また，ホスト PC が SFR にデータを書き込もうとしているのか，SFR からデータを読み出そうとしているのかを判定し，通信方向を制御する．

この機能を実現するために，ホスト PC は Fig. 4.9 に示すような二つのトランザクションを生成し，Xillybusu に転送する．最初のトランザクションで転送されるデータは，ホストの SFR に対するデータの書き込み要求か読み出し要求かを判定するためのビットを MSB に配置し，図中では Read or Write から R/W と表現している．また，SFR の書き込みもしくは読み出し対象アドレスを R/W 以下に 7 bit で付随させている．ホスト PC から送られてきたこのデータを AXI-Stream to AXI Bridge が受け取ると，R/W ビットにより次の動作を決定する．R/W ビットが 0 の場合は SFR からの読み出し要求と定義しているので，AXI-Stream to AXI Bridge は最初のトランザクションでホスト PC から送られてきたアドレスを SFR から参照し，その

値をホスト PC へ返信する．一方，R/W ビットが 1 の場合は，ホスト PC から SFR への書き込み要求と定義している．この時は，ホスト PC は最初のトランザクションに続けて第 2 のトランザクションとして SFR に書き込みたいデータを Xillybus 経由で FPGA へ送信する．すると，AXI-Stream to AXI Bridge はこのデータを第 1 のトランザクションで指定されたアドレスの SFR へと書き込む．

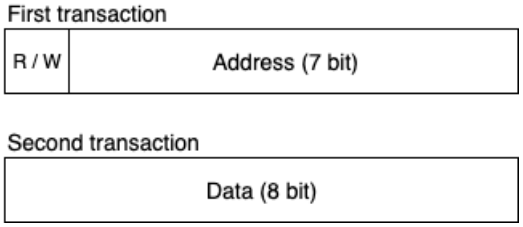

Fig. 4.9 SFR アクセスのための通信形式

4.6 ホスト PC 側プログラム

ホスト PC 上で動作するソフトウェアでの本システムの取り扱い方法について述べる．ホスト PC 内での Xillybus アクセス時の経路概観を Fig. 4.10 に示す．この図において，Software applcation とはユーザがホスト PC 上で実行するアプリケーションのことであり，C++ や Python などのプログラミング言語で記述されたソフトウェアのことである．Device files とは，Linux が外部機器との通信のために用意している仕組みであり，外部機器とのデータのやり取りをあたかもファイルへの読み書きを行うように実行できるものである．Xillybus を書き込んだ機器でを接続すると，Linux がこれを検知し，自動的に，“/dev” ディレクトリ以下に，“xillybus_write_32”, “xillybus_read_32”, “xillybus_write_8”, “xillybus_read_8” の 4 つのデバイスファイルを生成する．それぞれのデバイスファイルの用途は Table 4.1 に示すとおりである．図中の FPGA 側の Internal Logics は Fig. 4.6 で示された各種回路を省略して表記したものである．

Xillybus によるデータの送受信は Linux の場合はこれらのデバイスファイルを経由して行われる．ユーザが作成したソフトウェアは “/dev/xillybus*”（*は対応する xillybus のデバイスファイル名末尾を示す）に対してファイルと同様の処理を行うことでデータのやりとりが可能である．具体的には以下のような手順を踏む．

Table 4.1　Xillybus に関するデバイスファイル一覧

デバイスファイル名	用途	対応する FPGA 側インタフェース
xillybus_write_32	32 bit 幅データ送信用	AXI-Stream (32 bit)
xillybus_read_32	32 bit 幅データ受信用	AXI-Stream (32 bit)
xillybus_write_8	8 bit 幅データ送信用	AXI-Stream (8 bit)
xillybus_read_8	8 bit 幅データ受信用	AXI-Stream (8 bit)

1. ソフトウェアで通信に使用したいデバイスファイルを開く
2. デバイスファイルに対して書き込みもしくは読み込みコマンドを実行する（システムコール）
3. 通信が終了後デバイスファイルを閉じる

4.7　Xillybus アクセス用 API の構築

デバイスファイルに対するデータの読み書きには C++ 言語の場合，書き込みは write，読み込みは read システムコールを用いる．しかし，SFR へのアクセスでは R/W ビットやアクセス先アドレスの指定，データの送受信を生成する必要がある．また，32 bit の AXI-Stream バスを用いたデータ転送でも送信するデータを C++ 環境であれば unsigned int 型などの 32 bit の変数に変換して転送する必要がある．これらの処理は煩雑になるため，本研究では C++ 開発環境において，デバイスファイルやシステムコールの取り扱いを隠蔽できるような，Xillybus および SFR アクセス専用のクラスを作成し，application programming interface（API）とした．作成したクラス群とそのメソッドについて述べる．

Xillybus と SFR へのアクセス用にそれぞれ，“xillybus_tools” と “sfr_tools” とした二つの API を作成した．前者の xillybus_tools はデバイスファイルへの書き込みや読み込みを隠蔽し，配列や standard template library（STL）の vector などより扱いやすいデータ型を用いて Xillybus による通信を実現するクラスを提供する．具体的には，“xillybus8” と “xillybus32” なる二つのクラスを提供している．Xillybus を用いてデータの送受信を行うだけであればこのクラスを利用するだけで良い．な

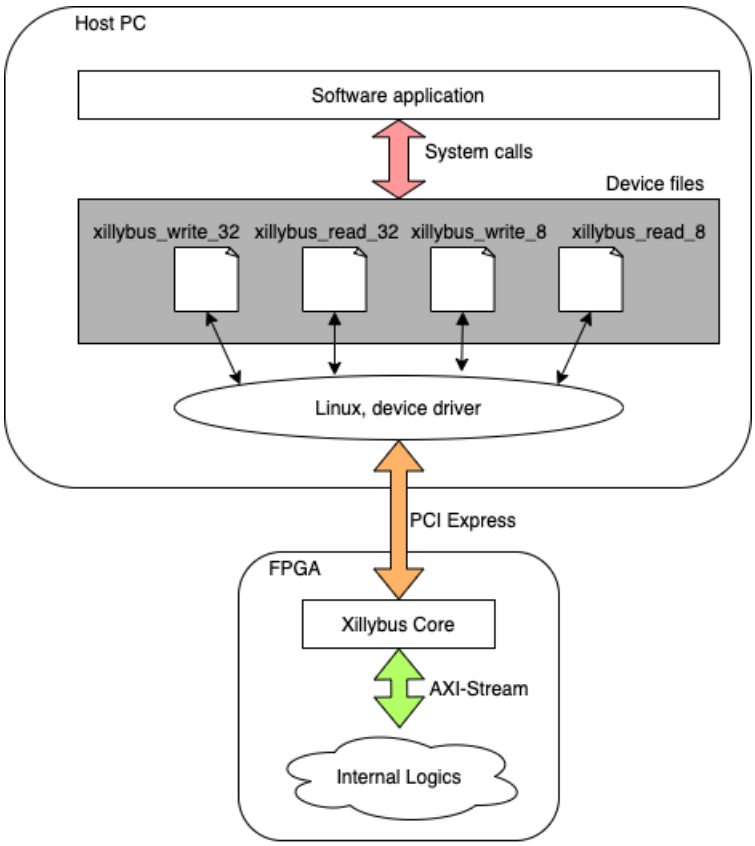

Fig. 4.10　ホスト PC 上のアプリケーションから FPGA 内部への Xillybus によるアクセス経路外観

お，Xillybus32 は 32 bit 幅のデータを転送するためのクラスであり，FPGA 側では 32 bit の AXI-Stream インタフェースに接続される．一方，xillybus8 は 8 bit 幅のデータを転送するためのクラスであり，FPGA 側では 8 bit の AXI-Stream インタフェースに接続される．

sfr_tools では “sfr” なるクラスを提供している．これは xillybus_tools の xillybus8 クラスオブジェクトに依存している．これは Xillybus 通信を xillybus_tools で提供されるクラスで一元管理するためである．sfr クラスには write と read メソッドが用意されており，先に述べた SFR へのアクセスのための各種手続きを隠蔽している．write メソッドでは，書き込み先の SFR アドレスと書き込みデータ，書き込み開始フラグを与えると xillybus8 クラスを用いて SFR の指定アドレスへデータの書き込みが行われる．また，read メソッドでは SFR のアドレスを与えると指定したアドレス

に格納されているデータを読み出すことができる．

以上のように，システムコールや SFR アクセスのためのトランザクションの生成など低レイヤの処理を隠蔽することで，本システムのユーザの利便性を高めている．なお，このアクセス用 API は GitHub でインターネット上に公開している [54, 55]．

第 5 章

性能評価と実装

本章では，まず提案した切り捨てビットを用いた確率的ニューラルネットワーク実装手法について，性能評価を行い，その結果について述べる．次に，提案した検証用ハードウェア基盤について，動作検証を行うためのアプリケーションを実装し，実際に FPGA 上で動作させた結果について述べる．

5.1　切り捨てビットを用いた手法の評価

本節では提案手法である切り捨てビットを用いた手法の性能評価を行い，その結果について述べる．まず提案手法を用いた RBM で MNIST と Fashion-MNIST データセットを学習させ，その結果について評価した．さらに，提案手法で取得できた切り捨てビットについて，その値が一様性を持つかどうか検証するために，統計的検定法であるカイ二乗適合度検定 [56] を実施した．

なお，提案手法の実装には固定小数点二進数の演算をソフトウェア上でシミュレーションする必要がある．これは，ソフトウェア上では通常は浮動小数点型の変数が用いられるためである．固定小数点二進数を使用するために，本実験では，Xilinx 社が提供している高位合成（high-level synthesis: HLS）ツール Vivado HLS，Vitis HLS に含まれる ap_fixed と呼ばれるライブラリを使用した．このライブラリは任意のビット幅の固定小数点二進数を取り扱うことができ，四則演算や各種数学関数への対応もなされている．また，FPGA に実装するために論理合成可能なライブラリとして提供されている．固定小数点型を用いる際は，Vitis HLS 上でプロジェクトを作成し，C-Sim と呼ばれる，プログラムの動作を C++ のコードとしてシミュレーショ

ンする手法を用いた.

5.1.1　提案手法を用いた RBM の学習

切り捨てビットによるハードウェア実装手法の実用性を検証するため，この手法を RBM に適用し，MNIST[57] と Fashion-MNIST[58] の 2 つのデータセットで学習を実施し，その学習結果のパラメタを使用して，入力画像の想起を RBM で行い，入力画像と出力画像の交差エントロピー誤差を計測した．また，比較対象として，通常の C++ 環境が提供する乱数生成ライブラリ，random を用いた RBM の学習も行った．学習時の条件は以下の通りである．

Table 5.1　学習条件

	提案手法	従来手法
可視ユニット数	784	784
隠れユニット数	150	150
変数型	固定小数	浮動小数
精度	小数部 18 bit, 整数部 14 bit	C++ double 型
学習率	0.12	0.12

学習に用いたデータセットについて述べる．MNIST は手書きの 0〜1 までの数字の画像を集めたデータセットであり，学習用データが 60,000 枚，テスト用データが 10,000 枚用意されている．画像サイズは縦横 28 ピクセルである．Fashion-MNIST は MNIST データセットをそのまま置き換え可能に作られたデータセットで，内容は靴や服など MNIST に比べて複雑な内容となっている．画像枚数，画像サイズは共に MNIST と同じである．それぞれのデータセットの例を Fig. 5.1 と Fig. 5.2 に示す．

交差エントロピー誤差の計測は次の手順で行った．なお，学習後のパラメタを取り込んで，画像の想起実験を行うプログラムは C++ 環境で作成し，double 精度で計算を行なっている．また，この時のサンプリングには C++ で提供されている乱数生成関数を用いている．また，データセットを入力する際に，各画素を 2 値化して入力している．

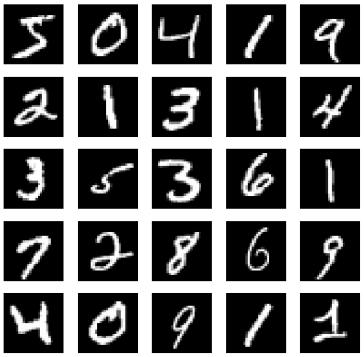

Fig. 5.1　MNIST データセットの例

Fig. 5.2　Fashion-MNIST データセットの例

1. 学習時に 1,000 イテレーションごとにパラメタをファイルとして全て保存
2. 学習完了後に，C++ 環境で構築した RBM プログラムで読み込み
3. データセットを 1 枚入力し，隠れ層，可視層と順にサンプリングを行い可視層の状態を出力値として取得
4. 入力した画像と出力として取得した画像間の交差エントロピー誤差を計測
5. データセット全てにおいて誤差を計測した後，入力した画像枚数で平均値を求める
6. 全イテレーションで上記処理を実施して記録

以上の実験を MNIST と Fashion-MNIST それぞれについて行なった時のイテレーションごとの交差エントロピー誤差の値を Fig. 5.3 と Fig. 5.4 に示す．なお，学習にはそれぞれのデータセットの学習用データ（train-set）を使用し，交差エントロピー誤差の計測時には，この学習用データセットを入力して計測したものと，学習では使用していないテスト用データセット（test-set）を用いた． また，グラフ中の Type が fixed と表記されているものが固定小数点二進数と切り捨てビットを用いたもので，double と表記されているものが，double 型整数と C++ が提供する乱数生成関数を用いたものである．

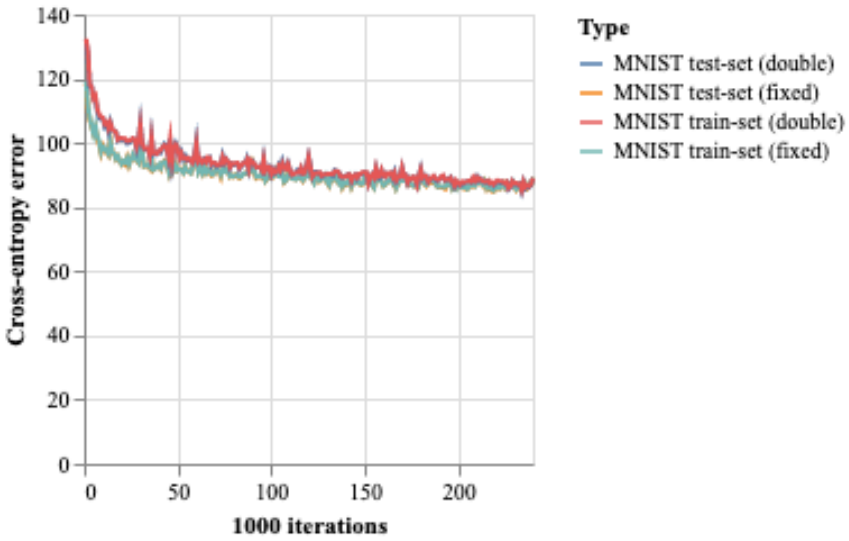

Fig. 5.3　MNIST 学習時の交差エントロピー誤差の推移

学習後のパラメタを用いて，RBM で入力データの再現を行なった．入力には

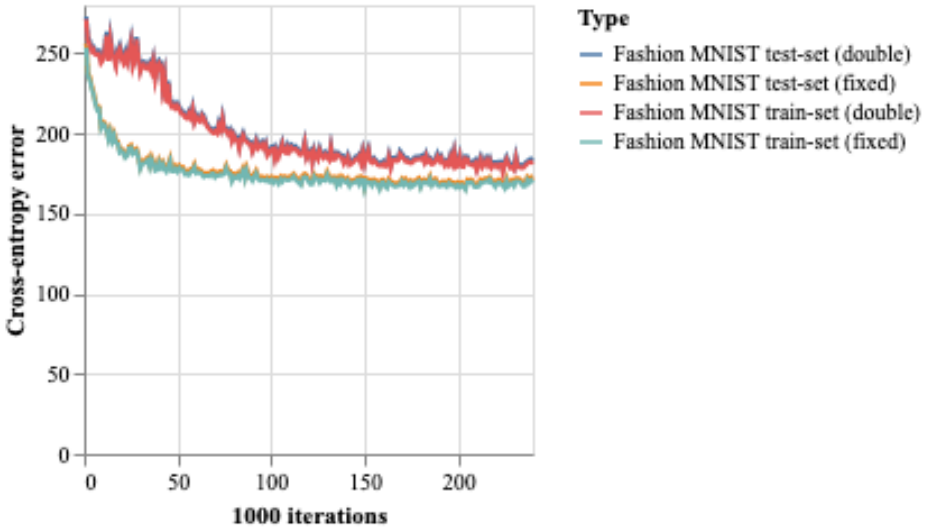

Fig. 5.4　Fashion-MNIST 学習時の交差エントロピー誤差の推移

MNIST と Fashion-MNIST の学習では用いていないテスト用データセットを入力し，可視層，隠れ層，可視層と発火確率の計算及びサンプリングを実施した時に，可視層に現れた画像を可視化した．この時は再現画像の可視化が目的であるので，テストデータセットからそれぞれ 100 枚の画像を抽出し，入力している．その結果を Fig. 5.5〜Fig. 5.10 に示す．

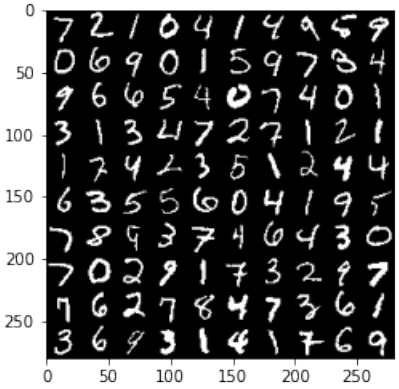

Fig. 5.5　入力した MNIST データセット 100 枚

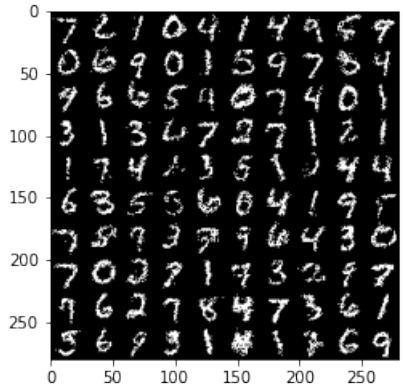

Fig. 5.6　提案手法で学習した場合の MNIST 出力画像

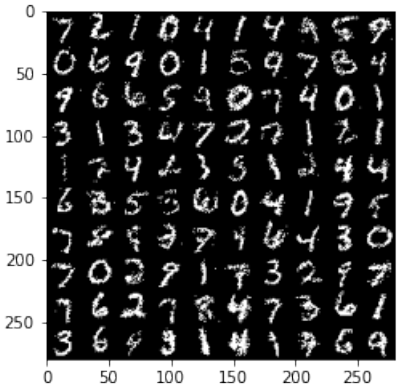

Fig. 5.7　C++ random を用いて学習した場合の MNIST 出力画像

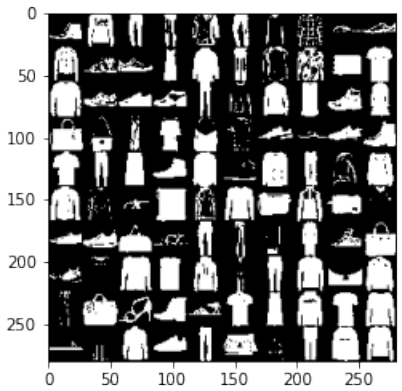

Fig. 5.8　入力した Fashion-MNIST データセット 100 枚

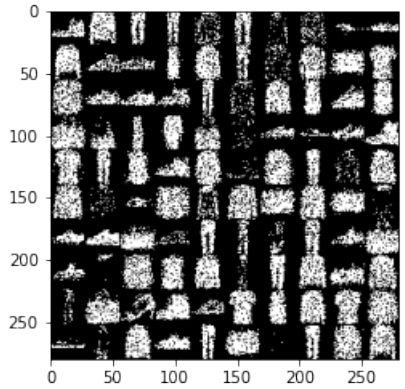

Fig.　5.9　提案手法で学習した MNIST 出力画像

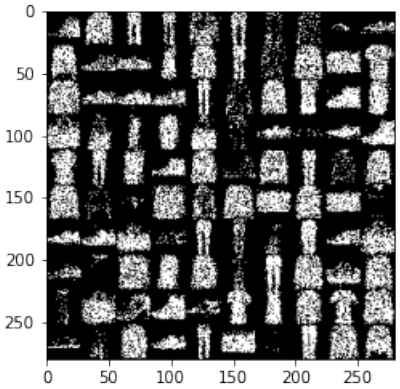

Fig.　5.10　C++ random を用いて学習した Fashion-MNIST 出力画像

5.1.2　切り捨てビットの一様性の検定

実験で得られた切り捨てビットの分布の一様性を検証した．ここで必要とされる性質は理想的には偏りのない一様な分布である．そこで，カイ二乗適合度検定を実施し，提案手法により取得された切り捨てビットが一様分布と一致しているかどうか仮説検定を行う．なお，ここで用いているカイ二乗適合度検定による一様性の検定は，乱数の一様性検定に用いられる手法の一つであり，日本工業規格である JIS Z9013「乱数生成及びランダム化の手順」の付属書 3（参考）乱数の特性及び検定方法 [59] で示されているものである．さらに，擬似乱数の検定に広く用いられている，アメリカ国立標準技術研究所（NIST: national institute of standards and technology）が定める，SP.800-22[60] の一部にも Frequency Test within a Block として頻度検定が組み込まれ，検定対象のビット列が理想的な分布に合致しているかどうかを統計的検定手法によって判定している．このように，ランダムと思われる数値の分布に対して統計的検定を用いる手法は一般的なものである．

まず，カイ二乗適合度検定について述べる．この検定は，統計的検定法の一つであり，乱数の頻度検定に用いられる．乱数の頻度検定とは，取得された乱数と思われる数値の分布がある確率分布に従うかどうかを調べるものである．ここでは，一様分布から切り捨てビットが取得されているかどうかを調べるために用いている．

カイ二乗適合度検定の手順について述べる．ここでは，検定対象の数値がある確率分布 $F(z)$ に適合していると帰無仮説をたて，検定を実施する．まず，検定対象となる，生成された n 個の値を，$z_1, z_2, \cdots, z_n$ とする．また，取得される数値が取りうる値の範囲を l 個の区間 $(p_0, p_1), (p_1, p_2), \cdots, (p_{l-1}, p_l)$ に分割する．つまり，区間を l 分割されたヒストグラムを生成することと同じである．なお，ここで分割した区間は互いに重ならないような区間でなければならない．この時，生成された n 個の値のうち i 番目（ただし，$i = 1, 2, \cdots, l$ である）の区間に入る値の個数を $f_i, (i = 1, 2, \cdots, l)$ とし，これを実現度数と呼ぶ．

ここで，帰無仮説に基づくと，i 番目の区間に入る個数は理論的に次式で表される数でなければならない．

$$F_i = n \times (F(p_i) - F(p_{i-1})), (i = 1, 2, \cdots, l) \tag{5.1}$$

この時，F_i を理論度数と呼び，$F(p_i)$ は実現度数と理論度数の差の平方の理論度数に対する比を χ^2_{l-1} とあらわし，次式で計算される．

$$\chi^2_{l-1} = \sum_{i=1}^{l} \frac{(f_i - F_i)^2}{F_i} \tag{5.2}$$

この式からわかるように，実現度数と理論度数の差が大きければ χ^2_{l-1} の値が大きくなる．この値によって帰無仮説を棄却するかどうか決めることになる．この時，F_i がある程度大きい時，χ^2_{l-1} は自由度 $l-1$ のカイ二乗分布に従うため，設定した有意水準と自由度におけるカイ二乗分布の棄却域を定めることで，統計的仮設検定が可能となる．

以上の手法で，一様分布に従う乱数であるかどうかを検定する場合の手順を以下に示す．ただし，乱数の範囲は $[0, 1]$ であるとする．

1. 検定対象とする n 個の乱数 $z_1, z_2, \cdots, z_n$ を用意する
2. 乱数の範囲 $[0, 1]$ を l 個の区間に分割し，i 番目の区間 (p_{i-1}, p_i) に入る乱数の実現度数 f_i を計数する
3. 理論度数 F_i を求める
4. 実現度数と理論度数から χ^2_{l-1} を求める
5. 有意水準を 5% ないし 1% に設定し，自由度 $(l-1)$ のカイ二乗分布の棄却域 (χ^2_0, ∞) を設定する（カイ二乗分布表などから求める）
6. $\chi^2_0 \leq \chi^2_{l-1}$ の真偽を判定する

最後のステップで行われた，$\chi_0^2 \leq \chi_{l-1}^2$ の判定結果が真であるならば，帰無仮説は棄却され，一様乱数としては不適当であると結論づけられる．一方，判定結果が偽である場合，乱数の一様性検定手法においては，検定に合格したとし，仮定した分布（一様分布）に従うものと判断される．

そこで，このカイ二乗検定を提案手法で取得された切り捨てビットに対しても適用する．MNIST および Fashion-MNIST 学習時に各ユニットで得られた切り捨てビットを全て収集し，これらが一様分布に従うと言えるか検定を行う．まず基本的な事柄であるが，学習画像を 1 枚入力した時に各ユニットでは一組の切り捨てビットが得られる．今回の学習では 240,000 イテレーションの学習を行なっているので，各ユニットあたり全学習を通して，240,000 個の切り捨てビットが生成されることとなる．そこで，以下の二つの検証を行なった．

まず初めに，全学習を通して得られた切り捨てビットを各ユニットごとに分割し，検定を実施し，それぞれのユニットで生成された切り捨てビットが一様分布に従うものであると言えるのか，判定する．

次に，1,000 イテレーションごとに切り捨てビットを区切り，そこでそれぞれのユニットで生成された切り捨てビットが一様分布に従うものとして認められるのか検定を実施し，この検定に合格したユニットが全ユニット数のうちどの程度の割合を占めているのか，検証する．

全学習を通して得られた切り捨てビットに対してユニットごとに検定を実施した結果を Fig. 5.11 と Fig. 5.12 に示す．MNIST データセットの学習時に得られた切り捨てビットを対象としたものを，Fig. 5.11 に示し，Fashion-MNIST の場合を Fig. 5.12 に示す．これらのグラフの横軸は隠れユニットの番号であり，150 個存在する．また，縦軸は計算された，χ_{l-1}^2 を表す．グラフ内の赤線は棄却域の下限を表し，これを超えたものについては，この検定においては，一様分布に従うものとして取り扱うことができないと結論づけられる．なお，本実験では切り捨てビットのとりうる値の範囲を 20 等分しているので，自由度は 19 となり，この時の有意水準 5% における棄却域は，$(30.1, \infty)$ である．ここで，棄却域に入ったユニットの数は 150 個中 9 個であった．つまり，全隠れユニット中 6% のユニットが棄却域に入っていた．これは MNIST と Fashion-MNIST どちらの場合も同じであった．

また，1,000 イテレーションごとに区切り，それぞれのユニットで生成された切り捨て値について，検定を実施し，検定に合格したユニットの割合をグラフにプロットした．MNIST 学習時に得られた切り捨てビットに対する結果を，Fig. 5.13 に示し，

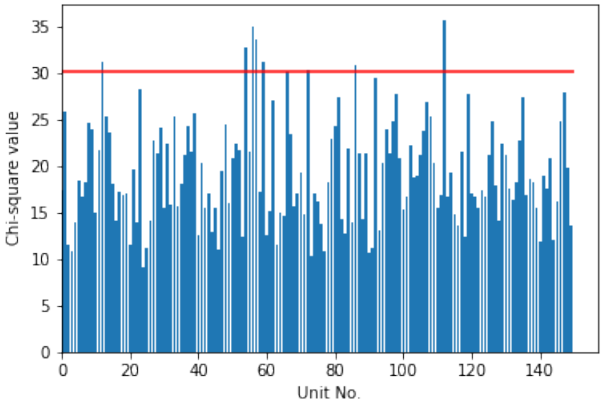

Fig. 5.11　MNIST データセット学習時の切り捨てビットのユニットごとの検定結果

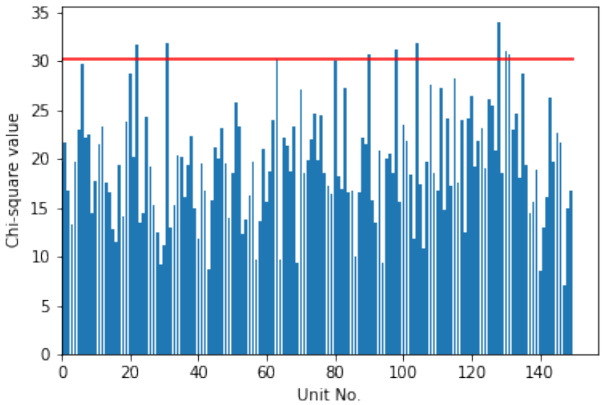

Fig. 5.12　Fashion-MNIST データセット学習時の切り捨てビットのユニットごとの検定結果

Fashion-MNIST に対する結果を，Fig. 5.14 に示す．横軸はイテレーションを表し，縦軸は 1,000 イテレーションごとに乱数の検定を実施し，合格したユニットの割合を表す．グラフに示す通り，検定に合格したユニットの割合は平均して 95% 程度であった．

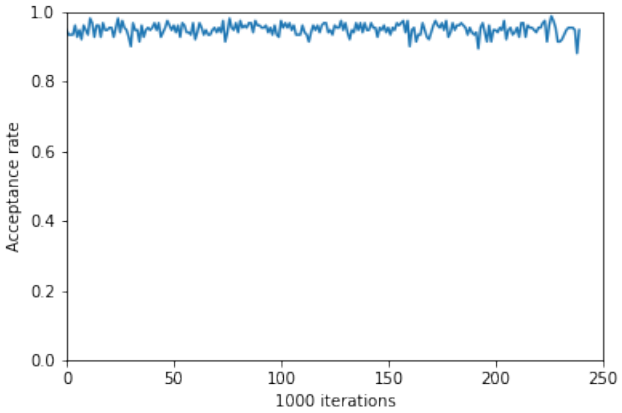

Fig. 5.13　MNIST 学習時の検定合格割合

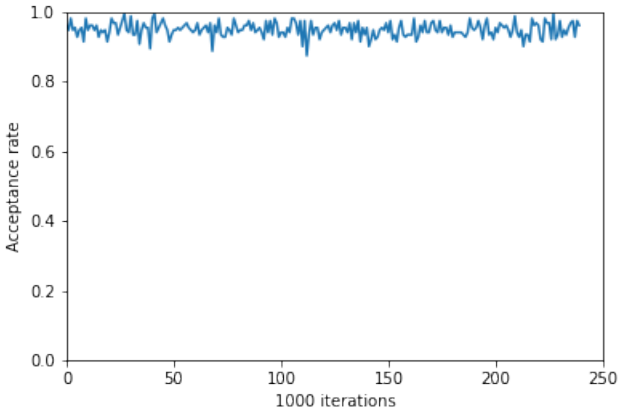

Fig. 5.14　Fashion-MNIST 学習時の検定合格割合

5.1.3　擬似乱数生成器と提案手法のハードウェアとしての比較

ここでは，擬似乱数生成器として，xorshift と LFSR を選び，この二つと，本論文で提案した切り捨てビットを取得する手法をハードウェア化した際に消費する回路資源を比較するため，実際に論理合成を実施した．なお，ここでは，擬似乱数生成器と提案手法を Verilog HDL で実装し，検証用ハードウェアプラットフォーム用の各種インタフェースは一切含まれないようにした．

まず，xorshift について述べる．これは，x, y, z, w, tmp の５つの変数を持ち，高位合成に対応した C++ では以下のようなコードで示されるものである．このコードで用いられている，“ap_uint” は高位合成用のクラスであり，ここでは 32bit 符号なし整数を表す．

Code 5.1　高位合成に対応した xorshift

```
1   ap_uint<32> xorshift() {
2       // initial values
3       static ap_uint<32> x = 123456789;
4       static ap_uint<32> y = 362436069;
5       static ap_uint<32> z = 521288629;
6       static ap_uint<32> w = 88675123;
7       static ap_uint<32> tmp;
8       // xorshift
9       tmp = x ^ (x << 11);
10      x = y;
11      y = z;
12      z = w;
13      return w = (w ^ (w >> 19)) ^ (tmp ^ (tmp >> 8));
14  }
```

このコードで表される xorshift 回路を Fig. 5.15 に示すような回路として Verilog HDL で実装した．図中の x, y, z, w, out は全て 32bit のレジスタである．また，楕円形で表記された部分はシフト演算を表し，それぞれ組み合わせ回路で構成されている．さらに，x, y, z, w はそれぞれ初期値が与えられているので，出力は Out レジス

タに値が格納される 1 クロックで得られる．

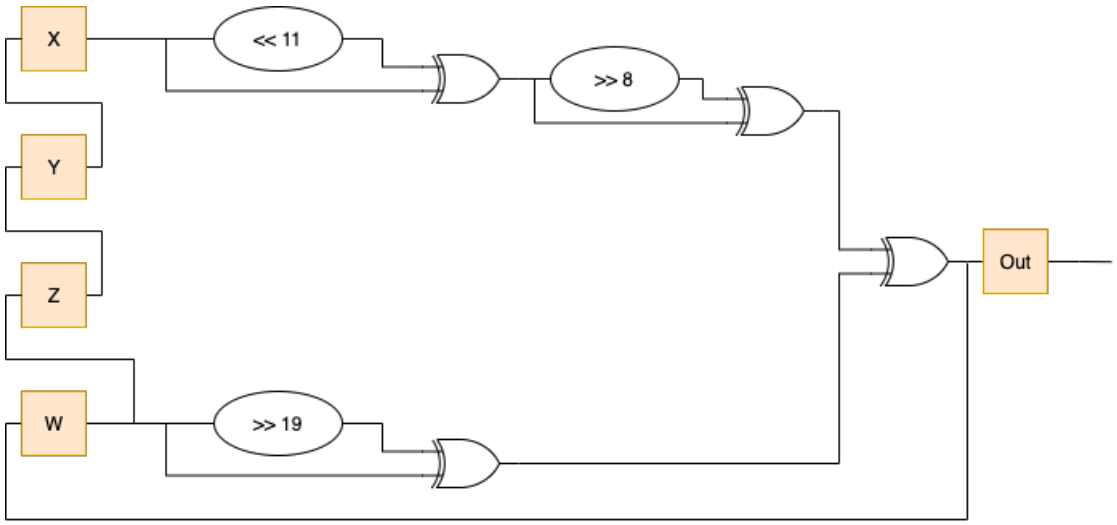

Fig. 5.15　実装した xorshift の回路図

この回路をハードウェアシミュレーションした際の波形を Fig. 5.16 に示す．図中の，rClk はクロックを，rnRst は負論理リセットを表し，wXorshiftOut が 32 ビットの xorshift の出力である．毎クロック出力が得られていることがわかる．

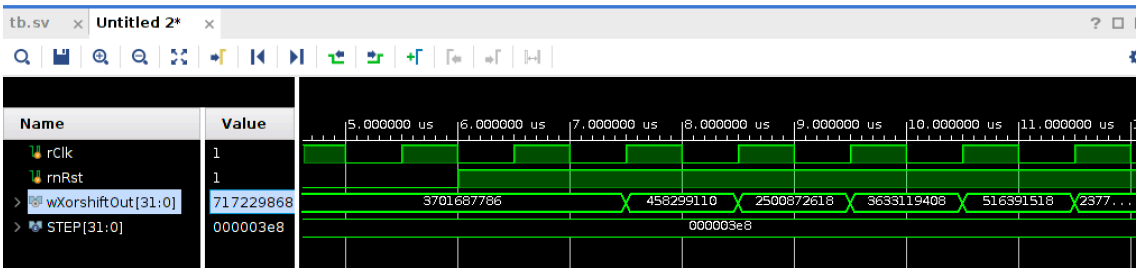

Fig. 5.16　xorshift 回路のハードウェアシミュレーション結果

次に，LFSR を Verilog HDL で記述した結果について述べる．LFSR は 32 ビットのシフトレジスタと XOR 演算から構成されるフィボナッチ LFSR である．この XOR を計算するために取り出すビットの位置（タップ）は，32，22，2，1 ビット目の値である．このタップ位置は 32bit の LFSR では最長周期が得られる組み合わせの一つである [61]．この LFSR の構成を Fig. 5.17 に示す．

LFSR は 1bit ずつシフトするごとにランダムな値がシフトレジスタの最下位ビッ

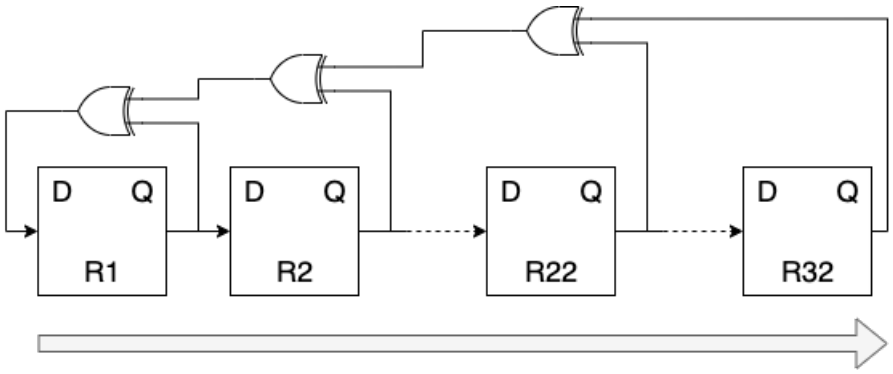

Fig. 5.17　32bit LFSR の構造

トに入力されるので，32bit 全てがランダムな値で満たされるまで待つ必要がある．毎クロックシフトするため，32bit の乱数を取得するためには，32 クロックの待ち時間が必要となる．Verilog HDL で記述した回路を Fig. 5.18 に示す．Verilog HDL では 5bit のカウンタを用意し，32bit のシフトレジスタが乱数列で満たされた時に Valid 信号を出力する構成とした．また，oValaid を出力するレジスタにデータが与えられると同時に，oData を出力するレジスタの書込みイネーブル iEn を与え，シフトレジスタの 32bit の値を出力レジスタに格納する．この挙動によって，図中の oValid 信号が立ち上がったタイミングで oData に 32bit の数値を得られる．

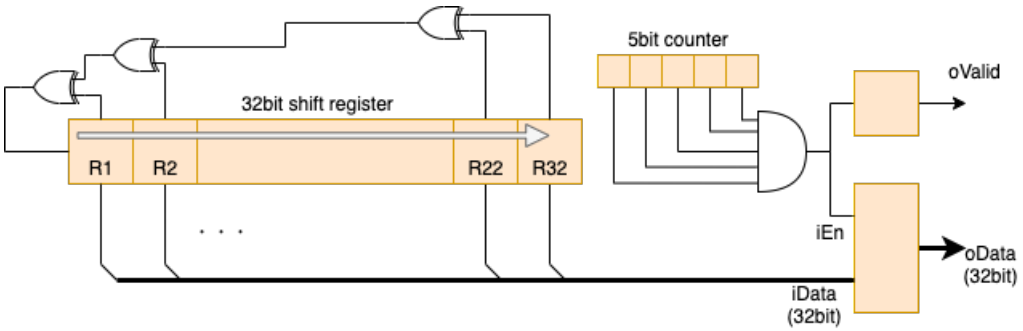

Fig. 5.18　実装した 32bit LFSR 回路

この回路の動作をハードウェアシミュレーションした際の波形を Fig. 5.19 に示す．wRnd が LFSR 回路の出力であり，wValid が立ち上がるタイミングで数値が出力されていることがわかる．また，xorshift と同様に，rnRst は負論理のリセット信号で，rClk はクロックを表す．

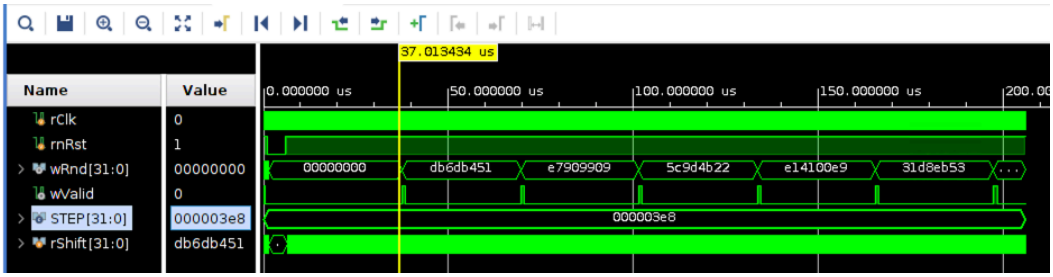

Fig. 5.19　32bit LFSR 回路のハードウェアシミュレーション結果

最後に，提案手法の切り捨てビット取得回路について述べる．この回路は入力された 64bit の数値から 32bit 固定小数点数の少数部である 18bit を取り出す回路である．xorshift や LFSR と異なり，単に切り捨てられるビットをレジスタに格納するだけの回路である．実装されたこの回路を Fig. 5.20 に示す．

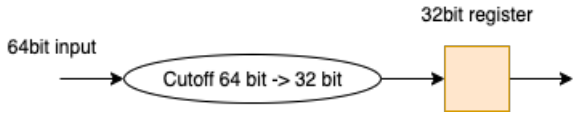

Fig. 5.20　実装した提案手法回路

切り捨てビット取得回路の動作をハードウェアシミュレーションした際の波形を Fig. 5.21 に示す．図中の rIn が入力される値であり，wOut が出力値である．出力値はレジスタに保存されるため，入力から 1 クロック遅れて出力される．リセット信号とクロックは他と同様である．この結果から，入力された 64bit の数値の下位 18bit が切り取られて，出力されていることがわかる．

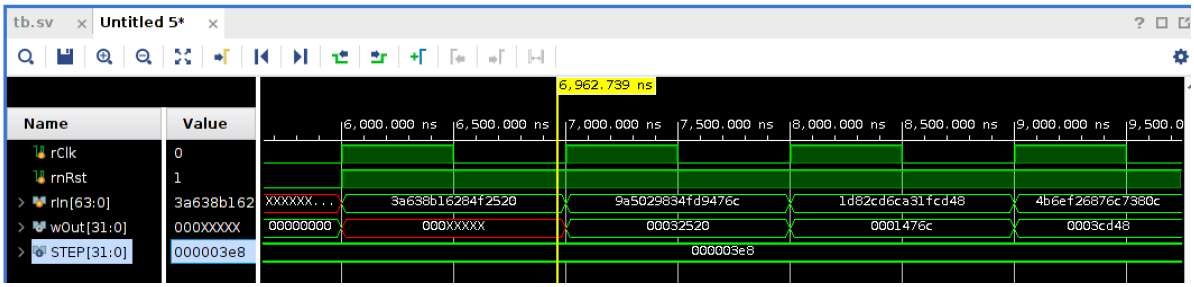

Fig. 5.21　提案手法のハードウェアシミュレーション結果

ここまで述べた，xorshift，LFSR，提案手法の切り捨てビット取得回路の 3 種類を Verilog HDL で記述し，トップモジュールとして，論理合成を実施した際の消費回路資源について Table 5.2 にまとめる．提案手法の回路が LUT と FF の消費量が最も少なく合成できていることがわかる．

Table 5.2　xorshift と LFSR，提案手法単体での回路資源使用量

Resource type	LUT	FF	IO
xorshift	33	160	34
LFSR	6	70	35
提案手法	1	18	52

xorshift, LFSR, 提案手法それぞれを論理合成した際の消費電力について, Table 5.3 に示す．消費電力は，FPGA の論理合成ツールである Vivado を用いて論理合成した際に生成される消費電力レポートから取得した．この際，クロックの周波数は 100[MHz] に設定した．表中の Clocks，Signals，Logic，I/O はダイナミックな消費電力と呼ばれ，書き込まれたロジックの動作により発生する消費電力である．これらの括弧内の割合は，ダイナミックな消費電力に占める割合を表す．さらにこれらの合計が Total Dynamic 欄に記載されている電力である．表中の Device Static 欄に記載されている数値は，スタティックな消費電力と呼ばれ，書き込まれる回路の動作に関わらず FPGA の動作のために必要な電力である．Total Dynamic と Device Static の値に記載されている割合は FPGA 全体におけるそれぞれが占める消費電力の割合である．この結果では，論理合成した回路が非常に小規模であったため，スタティックな消費電力が大半を占め，FPGA に書き込まれた回路が消費する電力は非常に小さなものとなった．

5.2　評価ハードウェア基盤の検証

本論文で提案した FPGA による評価用ハードウェア基盤の動作検証実験について述べる．本システムはホスト PC に接続された FPGA とこれを制御するためのホスト PC で構成される．そのため，動作制御を行うためには，FPGA 上に書き込む検証用回路とこれを制御し，データの送受信を行うためのホストプログラムの作成が必要

Table 5.3 xorshift, LFSR, 提案手法回路の消費電力見積もり（単位 [W]）

手法	xorshift	LFSR	提案手法
Clocks	0.003(9%)	0.002(51%)	0.001(20%)
Signals	0.002(6%)	<0.001(12%)	<0.001(2%)
Logic	0.001(2%)	<0.001(4%)	<0.001(<1%)
I/O	0.030(83%)	0.001(33%)	0.005(77%)
Total Dynamic	0.036(19%)	0.003(2%)	0.006(4%)
Device Static	0.158(81%)	0.158(98%)	0.158(96%)
Total On-Chip Power	0.195	0.162	0.164

である．ここでは，まず最小限のデータ転送及び FPGA 上でのデータ処理を検証するために，8 ビット階調のグレースケール画像を入力とし，この画像の各画素のピクセル値を反転させ，ネガ画像を生成する回路を実装した．この簡単なアプリケーションによって，ホスト PC からの FPGA 制御および，データ転送，FPGA 内でのデータ処理，FPGA からのデータ受信を検証することができる．次に，応用的なアプリケーションの実装例として，従来手法の RBM を FPGA 上に実装し，MNIST データセットをホスト PC から送信し，FPGA 上で RBM の学習を行なった．本検証で用いた FPGA 回路は全て C++ で記述し，Xilinx 社の高位合成ツール Vivado HLS, Vitis HLS によってモジュール化し，FPGA に実装した．本実験で使用した FPGA は Fig. 5.22 に示す Kintex-7 FPGA KC705 評価ボードであり，これを Ubuntu がインストールされた PC に PCIe で接続して使用した．

5.2.1 画像反転回路の実装

本検証実験では 1 枚の 8 bit 階調グレースケール画像を構築した FPGA システムに転送し，各画素値を反転させ，ネガ画像を作成し，ホスト PC に再度読み込んだ．ネガ画像変換は，入力画素値を x，出力画素値を y とすると，$y = 255 - x$ で計算される．この機能を実現する回路をまた，FPGA へ転送された画像は一旦，FPGA 内の BRAM に保存され，その後画像変換処理を行なったのち，その結果も BRAM に一旦保存する．これをホスト PC がデータ読み込みコマンドを発行し，処理結果を取

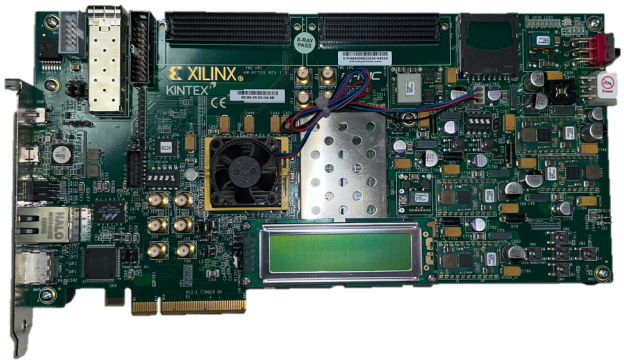

Fig. 5.22　Kintex-7 FPGA KC705 評価キット

得する手順である．この回路におけるデータの流れを Fig. 5.23 に示す．

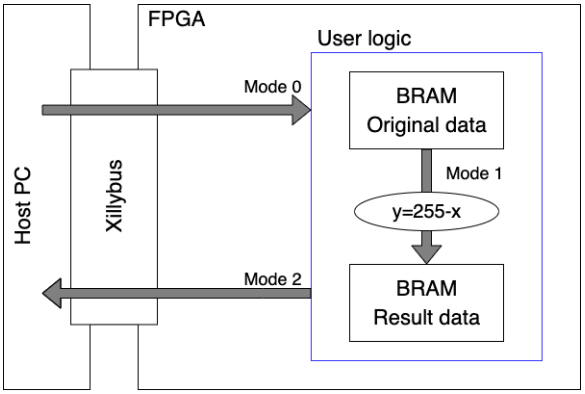

Fig. 5.23　画像処理回路におけるデータの流れ

この回路における SFR のレイアウトを Fig. 5.24 に示す．アドレス 0x00 には画像処理回路の動作モードを指定するための 4 bit の Mode 部分と，実行開始フラグである E が設定されている．アドレス 0x01 と 0x02 には回路に処理させる画像の横幅と高さを指定するためのパラメタを設定する部分である．

この回路においては，動作モードを Mode 0，Mode 1，Mode 2 の 3 つが実装され

Address	Data (8 bits)		
0x00	Unused	Mode (4 bits)	E
0x01	Width (8 bits)		
0x02	Height (8 bits)		

Fig. 5.24 画像反転回路における SFR のレイアウト

ている．それぞれ，4 bit の Mode 部分にそれぞれ，0x01，0x02，0x03 を書き込み，E フラグをアサートする事で指定したモードの処理が実行される．なお，それぞれのモードの動作は以下に示すとおりである．

Mode 0 Host PC から FPGA への画像転送を行い FPGA 内の BRAM に保存（Fig. 5.23 中の BRAM Original data）

Mode 1 FPGA 内で BRAM Original data からデータを読み出し画像処理を実施し，BRAM Result data に格納

Mode 2 FPGA から Host PC へ処理結果の画像を読み込み

この時，FPGA システムに転送する画像は，SIDBA データセット [62] から選んだ画像を縮小して転送した．入力画像を Fig. 5.18 に，出力として得られた画像を Fig. 5.19 に示す．画像からわかるようにネガ画像変換に成功している．

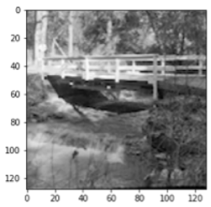

Fig. 5.25 SIDBA 入力画像

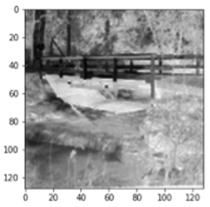

Fig. 5.26 出力画像

5.2.2 RBM の実装

提案したシステム上に，より規模の大きなアプリケーション例として従来手法の RBM の学習を行う回路を実装した．ここで実装する RBM は Xilinx Vivado HLS

による高位合成を用いて，C++ 言語により記述している．また，通常，HLS で回路を合成する際には，より効率的かつ高速に動作できるようにするために，各種最適化指示子を明示的にコードに書き込み，コンパイラへ最適化を支持していく．しかし，本実験は，ハードウェアプラットフォームの動作検証を目的とするため，RBM 回路への各種最適化などは行なっていない．ここで実装した RBM の仕様は以下の通りである．

可視層ユニット数　784
隠れ層ユニット数　150
固定小数点整数部ビット幅　14 bits（符号ビット含む）
固定小数点小数部ビット幅　18 bits

この回路をハードウェアプラットフォーム上に配置して，論理合成した際の合成結果を Table. 5.4 に示す．

Table 5.4　RBM を実装した提案ハードウェアプラットフォームの回路資源使用率

Resource type	Utilization	Available	Utilization %
LUT	13202	203800	6.48
LUTRAM	580	64000	0.91
FF	16279	407600	3.99
BRAM	276.5	445	62.13
DSP	108	840	12.86
IO	5	500	1.00
GT	8	16	50.00
MMCM	2	10	20.00

また，MNIST の学習データセット 60,000 枚を 24 epoch 学習させた際に，最終的に得られた重み行列を可視化したものを Fig. 5.27 に示す．この画像は，得られた重みを 0〜255 の 8 bit 階調の値に正規化したものを画像として表示している． この画像が取得できている通り，本研究で構築した FPGA プラットフォームでも学習データセットを転送し，ディジタル回路上で学習させ，その結果をホスト PC で取得することが可能であることがわかる．

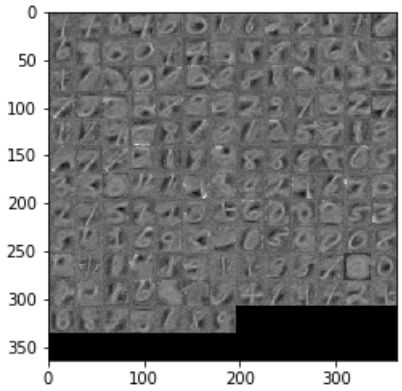

Fig. 5.27 FPGA システム上で MNITS を学習させた際に得られた重み行列

5.3 提案手法の FPGA 実装と動作検証

本節では，実際に提案手法を用いた積和演算とサンプリングを行うユニットを 1 個 FPGA に実装し，その動作を検証した．FPGA に実装する際には，提案ハードウェアプラットフォームに対応させ，ホスト PC からデータ転送及び回路の制御ができるような構成としている．

擬似乱数生成器を用いた積和演算とサンプリングを行う回路は，Fig. 5.28 に示すように，内部に擬似乱数生成器（PRNG）を保持する．このような構成とすることで，消費する回路資源量は増加するが，各ユニットで並列に乱数を生成することが可能である． しかし，提案手法の場合，Fig. 5.29 に示すように，切り捨てビットを取得する回路のみで構成されるので，PRNG が不要である．本検証では，この提案手法を用いた回路を実際に FPGA に実装し，その動作を確認した．

ここで実装した回路は，一つの隠れユニットの挙動を再現するものであり，入力として，ユニットの入力である x とユニットの重みである w の二つを持ち，ユニットの状態を出力するものである．この回路の構成要素は，積和演算回路（multiply-accumulate: MAC），活性化関数であるシグモイド関数 $\sigma(x)$，Comparator，切り捨てビットの取得回路である．MAC は入力 x と w の積和演算を計算する部分であり，乗算器とその結果を積算するアキュムレータ（Acc.）で構成される．MAC 計算後の数値は 32bit 固定小数点として，拡張された部分のビットが切り捨てられ，活性化関

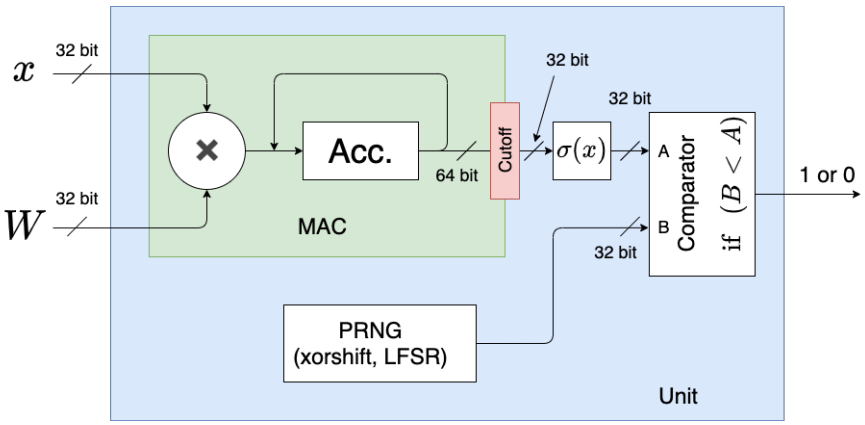

Fig. 5.28　擬似乱数生成器を搭載した場合の回路

数であるシグモイド関数に入力される．シグモイド関数によって積和演算結果が 0〜1 の数値へと変換され，Comparator へ入力される．この部分でシグモイド関数の出力が乱数生成器や切り捨てビットによって生成された値と比較され，ユニットの出力として出力される．

この回路の入出力はここまで述べた FPGA による検証用プラットフォームに実装することを前提として設計しているため，入力 x と w は一つの 32 ビット AXI-Stream バスから供給される．実装した回路をハードウェアプラットフォームに実装する時の接続図を Fig. 5.30 に示す．図に示すように，検証用 FPGA プラットフォームではホスト PC とのデータ通信用に 32bit 幅の AXI-Stream が提供されている．実装されるユニットの動作を検証する回路は入力が x と w の二つ存在し，それぞれ

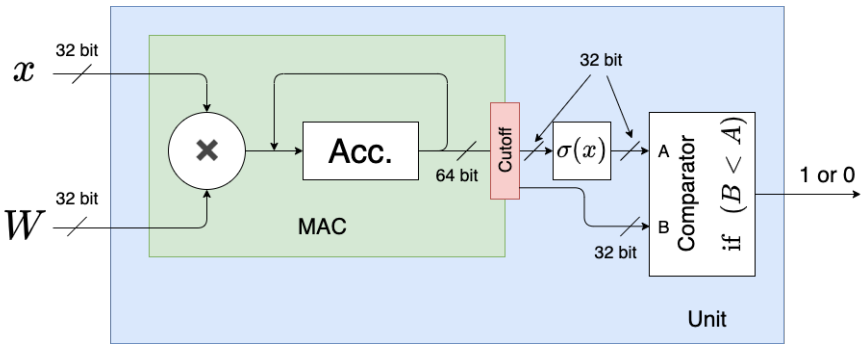

Fig. 5.29　提案手法を用いた場合の回路

32bit の数値であるため，本検証用回路では入力はシーケンシャルに Xillybus を介してホスト PC から入力されることを想定している．つまり，x と w がストリームとして，$\{x_1, w_1, x_2, w_2, \cdots, x_n, w_n\}$ のように転送される．これをユーザロジック側で x と w に分けて，積和演算を実施している．この方法は，あくまで提案手法である切り捨てビットの取得回路の動作検証であるため，ストリームでデータの転送を行なっている．実際にニューラルネットワークを構築する場合は，入力を並列化するなどの工夫が行われる．また，ホスト PC からのコマンドやユーザロジックの処理完了の通知など制御用の信号として，SFR に接続される AXI4 ポートが用意されている．

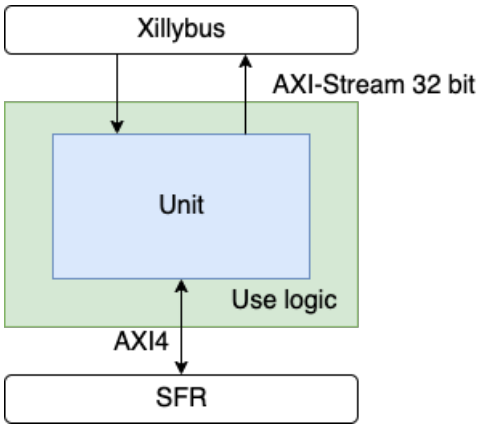

Fig. 5.30　検証用 FPGA プラットフォームへの接続図

まず，ユーザロジックとして，Fig. 5.29 を Vitis HLS により高位合成した際の回路資源使用量を Table. 5.5 に示す．データ転送や制御用のポートのためのロジックが組込まれるため，切り捨てビット取得回路単体で論理合成した際に比べ，使用する回路資源量が増えている．

Table 5.5　プラットフォーム実装のための提案手法を用いた回路の回路資源使用量

Resource	Utilization
BRAM	7
DSP	3
FF	2125
LUT	2113

つづいて，提案ハードウェアプラットフォームへの実装結果について述べる．本プラットフォームのユーザロジック部分に Fig. 5.29 に示した回路を実装した．この時の回路資源使用量は Table. 5.6 に示す通りであった．これはユーザロジックのみならず，提案プラットフォームが使用する回路資源（Xillybus や SFR など）を全て含んだ状態での論理合成結果である．なお，この時のターゲットデバイスは，Kintex7 評価ボード KC705 である．

Table 5.6　提案手法検証用回路を FPGA 実装した際の回路資源使用量

Resource	Utilization	Available	Utilization %
LUT	9436	203800	4.63
LUTRAM	470	64000	0.73
FF	12182	407600	2.99
BRAM	31	445	6.97
DSP	31	840	3.69
IO	5	500	1.00
GT	8	16	50.00
MMCM	2	10	20.00

Fig. 5.29 に示した回路を実際に今回構築した FPGA プラットフォームに実装し，ホスト PC から入力を与え，サンプリング後の値を取得することで動作検証を行なった．まず，回路への入力はそれぞれ図に示した x と w が 10 要素ずつとし，その値はホスト PC 側の C++ プログラムが提供するメルセンヌツイスタによる乱数生成で生成した値を $[-1, 1]$ の範囲に整形して入力している．なお，ホストプログラムで乱数を生成する際のメルセンヌツイスタの初期値は 1 としている．実験を繰り返す度に値が変わらないようにするためである．次に，ここで生成した x と w を FPGA へと転送し，演算を行い，切り捨てビットとの比較を行なった結果である，1 もしくは 0 の値をホスト PC へと返している．ホスト PC へと返された結果は，高位合成ツールである Vitis HLS におけるシミュレーションによって得られた理想的な計算結果と等しくなっていることをもって，FPGA に実装した回路の動作及び，提案プラットフォームでの動作が正しく行われていることを確認する．

ここでは，実装した回路に対して値を入力し，その出力を取得する動作を 10 回繰

り返している．入力値はその都度乱数を取得するので，それぞれ異なるものとなる．この時，FPGA から得られた結果を以下に示す．

Code 5.2 FPGA 上で動作させた提案回路の戻り値

```
1  ----------
2      start
3  ----------
4  Prepare data...
5  FPGA Result: 1
6  FPGA Result: 1
7  FPGA Result: 0
8  FPGA Result: 0
9  FPGA Result: 0
10 FPGA Result: 1
11 FPGA Result: 1
12 FPGA Result: 0
13 FPGA Result: 1
14 FPGA Result: 1
```

また，Vitis HLS で実装したハードウェア関数を C++ 関数としてシミュレーションした結果を以下に示す．ここで，answer となっている値はテスト用に C++ で記述した関数の計算結果であり，retval が実際にハードウェアに論理合成される関数の戻り値である．つまり，先に示した FPGA の戻り値と以下の retval の値が一致していれば設計した回路が正しく動作していることがわかる．

Code 5.3 提案回路のシミュレーション結果

```
1  ----------
2  Execute the test
3  ----------
4  answer: 1, retval: 1
5  answer: 1, retval: 1
6  answer: 0, retval: 0
7  answer: 0, retval: 0
```

```
8       answer: 0, retval: 0
9       answer: 1, retval: 1
10      answer: 1, retval: 1
11      answer: 0, retval: 0
12      answer: 1, retval: 1
13      answer: 1, retval: 1
```

以上の結果から，FPGA に実装した際の戻り値と Vitis HLS 上でのシミュレーションの結果が一致しているため，本検証実験でハードウェアに実装した回路は正しく動作していると考えられる．

第 6 章

考察と今後の課題

本章では 5 章で実施した提案手法の性能評価と実装で得られた結果について，考察を行い，今後の課題について述べる．

6.1 提案手法による RBM 学習結果に関する考察

まず，MNIST と Fashion-MNIST ともに提案手法を用いて RBM の学習を実行することができた．また，交差エントロピー誤差の値も固定小数点精度かつ提案手法を用いたものと，double 精度かつ従来手法の乱数生成法を用いたものとで，学習結果は最終的に近い数値となっている．具体的な数値を Table 6.1 に示す．この数値と合わせて，RBM での画像の想起結果からもわかるように，提案手法を用いた場合でも学習が可能であることがわかった．

Table 6.1 交差エントロピー誤差の最小値

データセット	実装手法	誤差（学習データセット）	誤差（テストデータセット）
MNIST	提案手法	85.18	84.87
MNIST	C++ random	85.36	85.12
Fashion-MNIST	提案手法	165.07	166.69
Fashion-MNIST	C++ random	175.81	177.68

Fashion-MNIST の結果が MNIST よりも悪い原因は，データセットの複雑さにあると考えられる．MNIST が 0〜1 の数字であるのに対して，Fashion-MNIST はカテゴリは 10 種類であるが，同じカテゴリに含まれる物体の形状は大きく異なる場合が多い．例えば，サンダルのカテゴリでは，かかとのないタイプとあるタイプのように形状が全く異なるものが含まれる．そのため，MNIST と全く同じユニット数の RBM を用いれば，学習精度が悪くなることは十分に考えられる．これは隠れユニット数が RBM の表現可能なモデルの柔軟性に大きく関わるためである．

6.2　提案手法とソフトウェアによる乱数の分布の比較

カイ二乗適合度検定の結果より，MNIST および Fashion-MNIST の学習時に発生する切り捨てビットは 9 割以上の隠れユニットにおいて，一様性の検定に合格した．しかし，一部のユニットについては検定による一様性が認められなかったものの，学習が進行していることから，隠れユニット数が十分に存在すれば提案手法を用いて，学習が可能であると考えられる．また，学習が進行するにつれ，切り捨てビットの検定通過割合が減少するといったことも認められなかった．

本考察では，取得された切り捨てビットのデータのばらつきを確認するため，Fig. 6.1 に箱髭図としてプロットした．さらに，比較対象として C++ 環境が提供するメルセンヌツイスタ法によって生成された乱数の分布も同じ図にプロットした．このメルセンヌツイスタ法による乱数は C++ で記述したソフトウェアで $[0, 1]$ の範囲で一様分布に従うように生成している．この時に生成する乱数の個数は，実験時に提案手法によって生成された乱数と同数の 36,000,000 個である．図の縦軸は乱数の値を表し，三角形の点が平均値を表す．

図からもわかるように，MNIST 学習時，Fashion-MNIST 学習時，C++ random それぞれの分布に大きな差は見られず，提案手法で生成された切り捨てビットの値については，学習時に取得されるもの全体で見ると，大きな偏りがないことが概観できる．

6.3　切り捨てビットの一様性の検定結果に関する解釈

本研究では，切り捨てビットが一様分布に従うとの仮説をたて，カイ二乗適合度検定を実施した．一部のユニットにおいては，検定の結果，有意水準を 5% とした際の

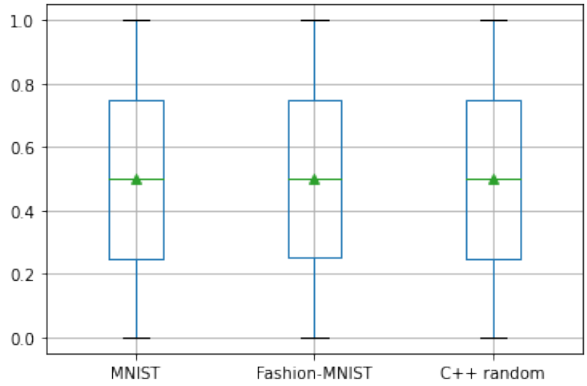

Fig. 6.1　切り捨てビットと C++ で生成した乱数の分布

棄却域に計算した統計量が入り，一様分布には従わないことがわかった．一方で，棄却域に入らなかったものについては，統計的検定手法の観点では，帰無仮説を採択するかどうか判断を保留することとなる．しかし，一般的に用いられている乱数の検定手法においては，カイ二乗適合度検定に合格したと扱われる．この場合，日本工業規格 JIS の Z9031 「付属書 3（参考）乱数の特性及び検定方法」では，仮説とした確率分布を採択するとしている．さらに，擬似乱数の検定に広く用いられている，アメリカ国立標準技術研究所（NIST: national institute of standards and technology）が定める，SP.800-22 の一部にも Frequency Test として頻度検定が組み込まれており，これはビット列に対する検定ではあるものの，検定対象の数値が理想的な分布に合致しているかどうかを判定し，統計量が棄却域に入らなかった場合は，テストに合格したものとしている．NIST の乱数検定である SP.800-22 はさまざまな乱数生成手法の検定に実際に利用されているものである [63, 64].

そこで，本研究における結果でも棄却域に入らなかったものについては，今回の検定において，合格とし，一様分布に従うと考える．しかしながら，一様分布に従う可能性が本検定により見出されたことは事実であるが，検定対象とした切り捨てビットが乱数であるかどうかの判断は今回の実験だけでは不十分であるため，乱数として切り捨てビットを扱えるかどうかについての判断は行えない．乱数として用いることができるかどうか判断する場合は，NIST SP.800-22 のような複数の乱数検定手法で構成される検定の枠組みに一定程度合格する必要がある．さらに，NIST の検定手法についても，結果の解釈で議論が存在 [65, 66] するので，仮に乱数検定を実施する場合は，非常に慎重な取り組みが必要であると考える．

6.4　他の乱数を使用する手法への応用

本論文では RBM を用いて切り捨てビットを用いた乱数生成器削減手法の検証を行なった．しかし，本提案手法は RBM に限ったものではなく，切り捨てビットが発生する回路への応用が可能である．ニューラルネットワークであれば，過学習の抑制手法であるドロップアウト [67] や，DBM の事前学習などに利用できる可能性がある．このように，本提案手法は，ランダム性を必要とするものへの応用可能性があると考えられる．

6.5　提案ハードウェア基盤における外部メモリの必要性

本稿では単純な画像処理回路と RBM をアプリケーション例として提案したハードウェアプラットフォーム上に実装し，動作させた．画像処理回路では単純なデータ転送と SFR による簡単な回路の制御，データ受信をテストした．また，より応用的な例として RBM を実装し，大量の学習データを転送させながら回路を制御した．どちらの実験でも提案したハードウェアプラットフォームは正常に動作することができた．しかし，RBM を実行した際，本稿で使用した回路ではデータセットの画像データを 1 枚ずつ 1 イテレーションごとに PCIe で転送する手法をとった．そのため，FPGA 上でアプリケーションを実行しているにもかかわらず，処理に時間を要する構造となった．このような構造となったのは，FPGA 内部に配置されている BRAM リソースに限りがあるためであり，データセットを FPGA 内部に保持し続けるのは難しいと判断したためである．今回構築したシステムのメモリ配置は Fig. 6.2 に示す通りである．この時，FPGA 内に実装したユーザロジックとのデータ転送速度は一般的に，BRAM，FPGA ボード上の SDRAM，PCIe を経由したホスト PC 上の SDRAM となる．そこで，FPGA ボード上の SDRAM を活用することができればデータ転送の問題を改善することができると考えられる．

6.6　提案ハードウェア基盤と既存のシステムの比較

本研究で提案したハードウェアプラットフォームのように，PC に FPGA を搭載し，ソフトウェアとハードウェアが協調して動作し，処理を高速化するシステムは

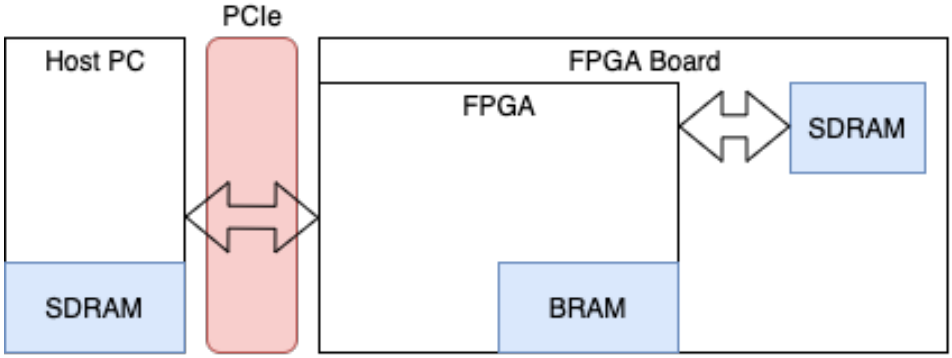

Fig. 6.2　提案ハードウェア基盤のメモリ配置

いくつか存在する．Xilinx 社の製品では Zynq と呼ばれる SoC や Alveo と呼ばれるアクセラレータカードである．前者は ARM CPU と FPGA が内蔵されており，ワンチップで OS の稼働から FPGA による処理まで完結することができる．しかし，CPU が ARM アーキテクチャであるため，x86 プロセッサが主流の PC 向けにコンパイルされたソフトウェアが使用できず，ARM アーキテクチャにクロスコンパイルする必要がある．また，PC と比較すると，CPU や RAM などの性能で劣る場合が多い．一方，Alveo は PC の PCIe スロットに接続し，利用するものである．非常に広帯域な PC との通信経路と Alveo 上に配置された専用の RAM が存在するため，大容量かつ大規模な演算を実行するプラットフォームとして用いられる．また，Vitis と呼ばれるソフトウェアとハードウェアを一体的に開発する環境も提供されている．しかし，Alveo を用いる場合，これを接続するホスト PC にも非常に高い性能が要求され，開発向けではホスト PC に 80GB 以上の RAM 容量が要求されている．

一方，提案システムでは x86 アーキテクチャを搭載した PC に接続可能でありながら，既存の FPGA 評価ボードに実装可能であり，ホスト PC の要求仕様も一般的な OS を実行するものと変わらない．よって，提案システムでは，既存のボードを活用しながら，ハードウェアとソフトウェアが協調するシステムを構築し，運用できる．

よって，組込み用途などでは Zynq が有効に活用でき，パフォーマンスが求められるような用途やサーバなどで実際のアプリケーションとしての用途を想定したものは Alveo が適する一方，独自設計のハードウェア IP の検証や設計したアーキテクチャの動作実験などの用途には本研究でのプラットフォームが適すると考える．

6.7　擬似乱数生成器と提案手法のハードウェア実装の比較

ここでは，擬似乱数生成器と提案手法をハードウェア実装した際の使用回路資源の量について考察する．Table 6.2 に Verilog HDL で xorshift と LFSR，提案手法を実装したときの回路資源量とレイテンシを示す．回路資源量については，実験結果で述べたものの再掲である．

まず，xorshift についてであるが，32bit のレジスタを 5 つ内部に持つ構造となっているため，使用されるフリップフロップ（FF）の個数が 160 個となるのは妥当であると考えられる．次に LFSR では，32bit のシフトレジスタ及び，出力用の 32bit レジスタ，カウンタとしての 5bit レジスタ，valid 信号を保存するために 1bit のレジスタを宣言している．合計で 70 個のフリップフロップが消費されることになり，これは論理合成結果と一致しているため，妥当な数値である．最後に，提案手法についてであるが，これは FF が 18 個となっている．64bit の入力を 32bit に切り捨てているため，32bit の出力を保持するレジスタが Verilog HDL のコード上では宣言されている．しかし，本実験においては，整数部の切り捨てビットは使用しないので，切り捨てた値は 64bit の固定小数点数のうち，小数部の LSB 側 18bit である．そのため，論理合成ツールの最適化により，値が代入されない部分のレジスタの使用が省かれたものと考えられる．よって，この数値も妥当なものであると考えられる．

LUT については，それぞれの回路で実装されている各種組合せ回路やリセットなどのために使用されていると考えられる．特に，配線とレジスタのみで実現できる提案手法において，LUT が用いられている点は，Verilog HDL で記述した回路が負論理のリセットを採用しているためであると考えられる．

なお，IO については，擬似乱数生成器及び提案手法回路を論理合成のトップモジュールとして設定したため，モジュールの入出力ピンが自動的に回路資源に割り当てられたものであると考えられる．実際にニューラルネットワークなどのアプリケーションに実装する際には，FPGA 内部での回路同士の接続となる．

以上の結果より，提案手法を用いた場合，擬似乱数生成器で使用していた回路資源を大幅に削減することが可能であることがわかった．

さらに，レイテンシについては，Verilog HDL で記述したハードウェアシミュレーションの結果より，xorshift と提案手法は 1 クロックで出力が得られたのに対して，

LFSR では 32bit シフトレジスタが乱数列で満たされるまで待機する必要があり，出力は 32 クロックごとにしか得ることができない．このように LFSR では擬似乱数の取得に時間がかかることがシミュレーションからもわかった．しかし，このレイテンシについては，積和演算処理と並列に実行するなど，工夫をすることで無視できるものにすることもできるため，大きな問題にはなりにくいと考えられるが，提案手法ではそのような工夫が不要である．

よって，提案手法は使用する回路リソース及びレイテンシの面で今回比較対象とした xorshift 及び LFSR に対して有利であることがわかった．

Table 6.2 xorshift と LFSR，提案手法単体での回路資源使用量とレイテンシ

Resource type	LUT	FF	IO	Latency
xorshift	33	160	34	1
LFSR	6	70	35	32
提案手法	1	18	52	1

6.8 PRNG と提案手法の消費電力量の比較

擬似乱数生成器である xorshift 及び LFSR と提案手法である切り捨てビット取得回路を Verilog HDL で記述し，論理合成を行なった．その際に，論理合成後のレポートとして，回路の消費電力の見積もりが論理合成ツールによって行われた．これは第５章で結果を示したとおりである．また，それぞれの手法において，出力が得られるまでのレイテンシは先述のとおりである．そこで，動作クロックを 100[MHz] として，出力を一つ得るために必要な消費電力量を見積もる．消費電力の見積もりには，デバイスが消費するトータルでの消費電力である Total On-Chip Power を用いたものと，実装される回路部分で使用される電力である Dynamic Power を用いたものの２種類を計算した．前者は FPGA に実装した際にチップ全体が消費する電力である．後者は，実装される回路が消費する電力である．

消費電力量 E の求め方について述べる．論理合成の結果からえられる回路の消費電力を P[W]，出力を得るまでにかかるクロック数を N[クロック]，１クロックの周期を T[s] とする．その時の，消費電力量は $E = P \times N \times T$[Ws] である．1[Ws]=1[J]

である．まず，論理合成時の動作クロックの設定は 100[MHz] であるから，1 クロックの周期は，10[ns] である．さらに，xorshift と提案手法は出力が得られるまでに 1 クロック，LFSR は 32 クロックを要する．以上の条件から消費電力量を見積もると，以下の Table 6.3 のようになる．この結果から，提案手法の消費電力量が最も少ないことがわかる．一方で，出力を得るために複数クロック実行する必要のある LFSR は xorshift，提案手法両者と比べても大きな数値となった．

Table 6.3　各手法における出力を一つ得るための消費電力量見積もり結果

手法	xorshift	LFSR	提案手法
Total On-Chip Power [W]	0.195	0.162	0.164
Dynamic Power [W]	0.036	0.003	0.006
Latency [Clock]	1	32	1
消費電力量 (Total On-Chip Power) [nJ]	1.95	51.84	1.64
消費電力量（Dynamic Power) [nJ]	0.36	0.96	0.06

6.9　切り捨てビットを用いた手法の提案ハードウェア基盤への実装

本研究では，固定小数点二進数演算における切り捨てビットを用いた乱数生成器を使用しない RBM 実装手法を提案した．一方で，ハードウェア開発者が開発した回路を FPGA に実装し，ホスト PC から制御し，それぞれが協調動作できるプラットフォームを構築した．実験では積和演算とサンプリングを実施する基本的なユニットの実装を行い，動作を確認した．

また，ハードウェアプラットフォームの動作検証のため，切り捨てビットを用いない従来型の RBM を実装し，動作を確認した．今後，提案手法を用いた RBM を実機で動作検証を行うためには，提案手法を用いた RBM を一つの IP として合成し，提案プラットフォームに接続する必要がある．実装は register transfer level（RTL）もしくは C++ による高位合成（HLS）が考えられるが，近年では HLS 環境が充実しつつあるため，まずは高位合成で実装することが動作検証のステップとしては望ましい．特に，今回提案したハードウェアプラットフォームでは AXI-Stream や AXI プ

ロトコルによる通信の実装が求められ，高位合成環境で実装することで，これらの複雑なプロトコルの実装を開発環境に任せることができる．

その上で検証すべき事柄としては，以下の点が挙げられる．

- 提案プラットフォームにおける動作速度
- 実機上で生成された切り捨てビットの解析
- RBM などの実装したアプリケーションの動作確認（学習の可否など）
- HLS による最適化の効果検証

ここに挙げたように，実装するアプリケーションは HLS で記述されるため，HDL で回路を記述する場合と違い，並列化やパイプライン化などを C++ の言語に最適化指示子（pragma）と呼ばれる形で明示的に埋め込む必要がある．この最適化指示子を適切に指定することで効率の良い回路を C++ から合成することができる．そのため，RBM などのアプリケーションを実装する際に，HLS による最適化がどの程度効果を発揮するのかを確認することも，今後，ソフトウェアとハードウェアが協調して動作するシステムの構築には不可欠であると考える．

第 7 章

結論

本論文では，近年の深層学習の発展に伴う計算資源の需要増大とそれに伴う消費電力の増大に対する一つの解決法として，FPGA へのアプリケーションの実装に焦点を当てた．まず，RBM に代表される，乱数生成器を必要とするアプリケーションを，乱数生成器を用いずにディジタルハードウェアに実装する手法を提案した．さらに，FPGA ユーザが自身の設計した回路をできるだけ簡単に実機上で動作させられる，FPGA による評価用ハードウェア基盤を提案した．これらの提案した手法とプラットフォームについて，その有効性や動作について検証を行なった．

まず，アプリケーションの FPGA への効率的な実装法として，特にハードウェア上に実装する乱数生成器に注目した．乱数生成器はハードウェア化する際に大きな回路資源を要求するコンポーネントの一種であるとともに，ランダム性を必要とするアプリケーションには必須の存在である．そこで，本研究では，ディジタルハードウェアで一般的に用いられる固定小数点二進数による演算時に発生する切り捨てビットを乱数の代替として利用してアプリケーションを実装する手法を提案した．また，乱数を必要とするアプリケーション例として深層学習の基礎であり，確率的ニューラルネットワークの一つである RBM を選び，RBM の学習時に得られる切り捨てビットの性質を分析し，その一様性を検証した．また，提案手法を RBM に適用することで，実際にアプリケーションに応用できることを示した．加えて，ディジタルハードウェアで用いられることの多い，xorshift 及び LFSR と提案手法単体を Verilog HDL で実装することで，提案手法が必要とする回路資源が非常に少ないものであることを確認した．さらに，論理合成を実施することで得られる消費電力見積もりから，各手法が出力を生成するまでに消費する消費電力量の見積もりを行い，提案手法が最も小さ

な消費電力量となることを示した．

次に，FPGA を用いたハードウェア評価用基盤についてまとめる．このプラットフォームでは，Xillybus を用いて，ユーザが簡単に FPGA アプリケーションを検証できる環境を整備した．ホスト PC から FPGA にアクセスし，データの送受信と回路の制御を行う部分を用意し，ユーザは自らが設計したアプリケーションを実現する回路（User logic）のインタフェースに AXI-Stream と AXI を用意し，本ハードウェア基盤に接続するだけで利用できるようにした．本研究では，このプラットフォームに簡単な画像処理回路と RBM をアプリケーションの例として実装することで，実際にホスト PC から FPGA に実装した User logic を制御し，データの送受信が可能であることを検証した．さらに，ホスト PC 上で Xillybus による通信と FPGA を制御する際のデバイスファイル取扱といった低レイヤの処理を隠蔽する API を C++ クラスとしてまとめ，GitHub 上に公開した．さらに，提案手法である切り捨てビットを用いた積和演算およびサンプリング回路を本プラットフォーム上に実装することで，両者の応用可能性を確認した．

本稿の成果により，これまで固定小数点演算で捨てられていた数値を活用し，乱数生成器を実装することなく RBM をはじめとする，乱数を必要とするアプリケーションをディジタルハードウェアに実装できる可能性を示した．また，近年の専用ハードウェアによる演算システムを開発する上で，ソフトウェアとハードウェアが協調動作するシステムを簡単に実機で検証できる環境が整備された．このハードウェア基盤を用いることで，様々なアプリケーションを FPGA に実装し，ソフトウェアと FPGA が協調して処理を実行するシステムを実現できる．

以上の通り，本研究は，RBM に代表される確率的に動作するアプリケーションの乱数生成器を用いない実装方法と，ソフトウェアと FPGA が協調して処理を実行するシステムを構築できるハードウェア基盤の構築，の 2 点を実現したことで，目的に設定した「低消費電力かつ高速な FPGA の特性に焦点を当て，RBM をはじめとする確率的ニューラルネットワークの効率的な FPGA 実装法の提案と，アプリケーションを実装するための FPGA プラットフォームの実現」に大きく寄与できるものであると考える．

謝辞

本研究をまとめるにあたり，長きにわたり，昼夜休日問わず，非常に多くのご指導をいただくとともに，様々な議論を交わし，研究上大切な多くの経験，充実した研究環境，アイデアを授けてくださった九州工業大学大学院生命体工学研究科 人間知能システム工学専攻の田向権教授に心より感謝いたします．また，同分野の研究者として本論文をまとめるにあたり，様々なアドバイスをいただきました，九州工業大学大学院生命体工学研究科，人間知能システム工学専攻の森江隆教授，堀尾恵一教授，東海大学の大川猛准教授に心より感謝いたします．さらに，スペイン Universitat Politècnica de Catalunya の Jordi Madrenas 准教授にはバルセロナ留学時に私を温かく歓迎していただき，異国の地での貴重な経験と，興味深い研究の議論，共同研究をさせていただきました．バルセロナでの経験は一生忘れることのないものだと確信しております．また，同研究室の皆様にも研究のみならず，日々の生活など様々な形でサポートしていただきましたこと，感謝いたします．また，九工大博士後期課程進学後にも，たびたび北九州へ足をお運びいただき，議論を交わしました，東京農工大学名誉教授，故 関根優年先生に感謝いたします．関根先生とは北九州でプログラミングの手法などについて議論を交わしたことをよく覚えております．また，日頃から様々な意見やアドバイス，相談に乗っていただいた，大江至流氏，宮野あゆみ氏，鈴木章央氏に感謝いたします．また，常に快く接してくださった田向研究室の皆様に感謝いたします．

最後に，ここまで様々な形で私を支えてくれた両親，妹に心から感謝し，謝辞といたします．

参考文献

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, Vol. 521, No. 7553, pp. 436–444, 2015.

[2] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, Vol. 313, No. 5786, pp. 504–507, 2006.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012.

[4] Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, p. 1237–1242. AAAI Press, 2011.

[5] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, Vol. 1, No. 4, pp. 541–551, 1989.

[6] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, Vol. 9, No. 1, pp. 147–169, 1985.

[7] Garrison W. Cottrell and Paul Munro. Principal Components Analysis Of Images Via Back Propagation. In T. Russell Hsing, editor, *Visual Communications and Image Processing '88: Third in a Series*, Vol. 1001, pp. 1070 – 1077. International Society for Optics and Photonics, SPIE, 1988.

[8] Google 翻訳. https://translate.google.co.jp, accessed: 2021-11-16.

[9] Deepl. https://www.deepl.com/translator, accessed: 2021-11-16.

[10] Mehdi Gheisari, Guojun Wang, and Md Zakirul Alam Bhuiyan. A survey on deep learning in big data. In *2017 IEEE international conference on computational science and engineering (CSE) and IEEE international conference on embedded and ubiquitous computing (EUC)*, Vol. 2, pp. 173–180. IEEE, 2017.

[11] Xiaoqiang Ma, Tai Yao, Menglan Hu, Yan Dong, Wei Liu, Fangxin Wang, and Jiangchuan Liu. A survey on deep learning empowered iot applications. *IEEE Access*, Vol. 7, pp. 181721–181732, 2019.

[12] NVIDIA. *CUDA C++ Programming Guide, PG-02829-001_v11.5*, 2021. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, accessed: 2021-11-16.

[13] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, Vol. 6, No. 2, p. 40–53, March 2008.

[14] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS 2017 Workshop on Autodiff*, 2017. accessed: 2021-11-16.

[15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[16] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. A survey of deep learning on cpus: Opportunities and co-optimizations. *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2021.

[17] 国立研究開発法人科学技術振興機構低炭素社会戦略センター. 情報化社会の進展がエネルギー消費に与える影響（vol. 1) - IT 機器の消費電力の現状と将来予測-, 2019.

[18] 国立研究開発法人科学技術振興機構低炭素社会戦略センター. 情報化社会の進展がエネルギー消費に与える影響（vol. 2) -データセンター消費エネルギーの現状と将来予測及び技術的課題-, 2021.

[19] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

[20] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Vol. 5, No. 4, pp. 115–133, 1943.

[21] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, Vol. 18, No. 7, pp. 1527–1554, 2006.

[22] Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, Vol. 14, No. 8, pp. 1771–1800, 2002.

[23] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, Vol. 345, No. 6197, pp. 668–673, 2014.

[24] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, Vol. 38, No. 1, pp. 82–99, 2018.

[25] Intel. *Technology Brief: Taking Neuromorphic Computing to the Next Level with Loihi 2*, 2021. `https://download.intel.com/newsroom/2021/new-technologies/neuromorphic-computing-loihi-2-brief.pdf`,

accessed: 2021-11-16.

[26] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

[27] Google I/O 2021. `https://io.google/2021/?lng=ja`, accessed: 2021-11-16.

[28] NVIDIA. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*, 2020.

[29] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, Vol. 2, No. 11, pp. 1–4, 2015.

[30] Xilinx. Versal: 初の ACAP (Adaptive Compute Acceleration Platform) (WP505) . `https://japan.xilinx.com/support/documentation/white_papers/j_wp505-versal-acap.pdf`, accessed: 2021-11-17.

[31] Xilinx. ザイリンクスの AI エンジンとそのアプリケーション (WP506) . `https://japan.xilinx.com/support/documentation/white_papers/j_wp506-ai-engine.pdf`, accessed: 2021-11-17.

[32] Apple. *Apple Silicon*. `https://developer.apple.com/documentation/apple-silicon`, accessed: 2021-11-16.

[33] Intel. Press Kit: 12th Gen Intel Core, 2021. `https://www.intel.com/content/www/us/en/newsroom/resources/press-kit-12th-gen-core-procesors.html`, accessed: 2021-11-16.

[34] Ihor Vasyltsov, Eduard Hambardzumyan, Young-Sik Kim, and Bohdan Karpinskyy. Fast digital trng based on metastable ring oscillator. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 164–180. Springer, 2008.

[35] Paul Kohlbrenner and Kris Gaj. An embedded true random number generator for fpgas. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pp. 71–78, 2004.

[36] Robert C Tausworthe. Random numbers generated by linear recurrence

modulo two. *Mathematics of computation*, Vol. 19, No. 90, pp. 201–209, 1965.

[37] George Marsaglia, et al. Xorshift rngs. *Journal of Statistical Software*, Vol. 8, No. 14, pp. 1–6, 2003.

[38] Sansei Hori, Takashi Morie, and Hakaru Tamukoh. Restricted boltzmann machines without random number generators for efficient digital hardware implementation. In *International Conference on Artificial Neural Networks*, pp. 391–398. Springer, 2016.

[39] Sansei Hori and Hakaru Tamukoh. A random number generation method for hardware implemented neural networks. *IEICE Tech. Rep.*, Vol. 119, No. 78, pp. 1–4, 2019.

[40] Geoffrey E Hinton. A practical guide to training restricted boltzmann machines. Technical Report UTML TR 2010-003, Department of Computer Science, University of Tronto, 2010.

[41] S. K. Kim, L. C. McAfee, P. L. McMahon, and K. Olukotun. A highly scalable restricted boltzmann machine fpga implementation. In *International Conference on Field Programmable Logic and Applications*, pp. 367–372, 2009.

[42] S. K. Kim, P. L. McMahon, and K. Olukotun. A large-scale architecture for restricted boltzmann machines. In *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 201–208, 2010.

[43] D. L. Ly and P. Chow. High-performance reconfigurable hardware architecture for restricted boltzmann machines. *IEEE Transactions on Neural Networks*, Vol. 21, No. 11, pp. 1780–1792, 2010.

[44] Kodai Ueyoshi, Tetuya Asai, and Masato Motomura. Scalable and highly parallel architecture for restricted boltzmann machines. In *2015 RISP International Workshop on Nonlinear Circuits, Communications and Signal Processing*, pp. 369–372, 2015.

[45] Kodai Ueyoshi, Takao Marukame, Tetsuya Asai, Masato Motomura, and Alexandre Schmid. FPGA Implementation of a Scalable and Highly Parallel Architecture for Restricted Boltzmann Machines. *Circuits and Systems*, Vol. 7, No. 9, pp. 2132–2141, 2016.

[46] Masato Hayashi, Masanao Yamaoka, Chihiro Yoshimura, Takuya Okuyama, Hidetaka Aoki, and Hiroyuki Mizuno. Accelerator chip for ground-state searches of ising model with asynchronous random pulse distribution. *International Journal of Networking and Computing*, Vol. 6, No. 2, pp. 195–211, 2016.

[47] Giovanny Sánchez Rivera. *Efficient multiprocessing architectures for Spiking Neural Network emulation based on configurable devices*. PhD thesis, 2014.

[48] 堀三晟, Mireya Zapata, Jordi Madrenas, 森江隆, 田向権. FPGA を用いた SIMD プロセッサによる Digital Spiking Silicon Neuron の実装. In *IEICE Conferences Archives*. The Institute of Electronics, Information and Communication Engineers, 2017.

[49] Takashi Kohno, Munehisa Sekikawa, Jing Li, Takuya Nanami, and Kazuyuki Aihara. Qualitative-modeling-based silicon neurons and their networks. *Frontiers in neuroscience*, Vol. 10, p. 273, 2016.

[50] Jing Li, Yuichi Katori, and Takashi Kohno. An fpga-based silicon neuronal network with selectable excitability silicon neurons. *Frontiers in neuroscience*, Vol. 6, p. 183, 2012.

[51] 田向権, 関根優年. ニューラルネットワークのハードウェア実装とそのシステム化へのアプローチ. 日本神経回路学会誌, Vol. 20, No. 4, pp. 166–173, 2013.

[52] Sansei Hori and Hakaru Tamukoh. A hardware-oriented random number generation method and a verification system for fpga. *Proc. 2021 Int. Conf. on Artificial Life and Robotics (ICAROB 2021)*, pp. 122–125, 2021.

[53] xillybus. `http://www.xillybus.com`, accessed: 2021-11-16.

[54] xillybus_tools. `git@github.com:HoriThe3rd/xillybus_tools.git`, accessed: 2021-11-16.

[55] sfr_tools. `git@github.com:HoriThe3rd/sfr_tools.git`, accessed: 2021-11-16.

[56] 宮武修, 脇本和昌. 数学ライブラリー 47　乱数とモンテカルロ法. 森北出版株式会社, 1988.

[57] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, Vol. 86, No. 11, pp. 2278–2324, 1998.

[58] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[59] 財団法人日本規格協会. JIS Z 9031 乱数生成及びランダム化の手順, 2011.

[60] Lawrence Bassham, Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Stefan Leigh, M Levenson, M Vangel, Nathanael Heckert, and D Banks. A statistical test suite for random and pseudorandom number generators for cryptographic applications, 2010-09-16 2010.

[61] Xilinx. *Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators (XAPP052).* `https://www.google.com/search?client=safari&rls=en&q=xilinx+efficient+shift+registers+lfsr+counters&ie=UTF-8&oe=UTF-8`, accessed: 2021-11-30.

[62] Standard image data base. `http://www.ess.ic.kanagawa-it.ac.jp/app_images_j.html`, accessed: 2021-11-16.

[63] Marcin M Jacak, Piotr Jóźwiak, Jakub Niemczuk, and Janusz E Jacak. Quantum generators of random numbers. *Scientific Reports*, Vol. 11, No. 1, pp. 1–21, 2021.

[64] Weitao Xu, Junqing Zhang, Shunqi Huang, Chengwen Luo, and Wei Li. Key generation for internet of things: A contemporary survey. *ACM Computing Surveys (CSUR)*, Vol. 54, No. 1, pp. 1–37, 2021.

[65] 奥富秀俊, 中村勝洋. NIST 乱数検定を用いた合理的なランダム性の判定法に関する考察. 電子情報通信学会論文誌. A, 基礎・境界, Vol. 93, No. 1, pp. 11–22, jan 2010.

[66] 吉田等明, 村上武, 川村暁. NIST SP800-22 rev.1a による疑似乱数の検定に関する一考察. 電子情報通信学会技術研究報告. NLP, 非線形問題, Vol. 112, No. 301, pp. 13–18, nov 2012.

[67] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, Vol. 15, No. 1, pp. 1929–1958, 2014.

研究実績等

論文誌（筆頭著者，査読有)

- Sansei Hori, Hakaru Tamukoh, “A Control and Data Transfer Platform for FPGA Applications,” Journal of Robotics, Networking and Artificial Life. (Accepted)

特許

- 堀三晟，田向権，森江隆，出願人：九州工業大学，“乱数生成器が不要なニューラルネットワークのハードウェア実装の方法及び乱数生成器が不要なニューラルネットワーク,”特願 2016-156471, 特開 2018-25920, 特許第 6831990 号.

学会発表（筆頭著者，査読有）

- Sansei Hori, Hakaru Tamukoh, “A Hardware-Oriented Random Number Generation Method and A Verification System for FPGA,”The 2021 International Conference on Artificial Life and Robotics (ICAROB2021), OS19-3, Online, January 21-24 (22), 2021.
- Sansei Hori, Mireya Zapata, Jordi Madrenas, Takashi Morie and Hakaru Tamukoh, “An Implementation of a Spiking Neural Network Using Digital Spiking Silicon Neuron Model on a SIMD Processor,” 26th International Conference on Artificial Neural Networks (ICANN2017), Lecture Notes in Computer Science, Vol. 10613, pp. 437-438, Alghero, Sep. 11-14(12), 2017.
- Sansei Hori, Takashi Morie, Hakaru Tamukoh, “Restricted Boltzmann Ma-

chines Without Random Number Generators for Efficient Digital Hardware Implementation," Proc. Of the 25th Int. Conf. on Artificial Neural Networks (ICANN2016), Lecture Notes in Computer Science, Vol. 9886, pp.391-398, 2016.

学会発表（筆頭著者，査読無）

- 堀 三晟，田向 権，“ニューラルネットワークのハードウェア実装に向けた乱数生成手法の提案と検証," 電子情報通信学会スマートインフォメディアシステム研究会 (SIS)，SIS2019-1，2019 年 6 月 13-14 日 (13)，長崎，福江文化会館.
- Sansei Hori, Takashi Morie, Hakaru Tamukoh, “A hardware-oriented random number generation method for restricted Boltzmann machines,” Abstract Book, The 5th RIEC Int. Symposium on Brain Functions and Brain Computer，pp.19, Feb., 2017，Miyagi，Japan.
- 堀 三晟，Mireya Zapata，Jordi Madrenas，森江隆，田向 権，“FPGA を用いた SIMD プロセッサによる Digital Spiking Silicon Neuron の実装," 電子情報通信学会 2017 総合大会,「神経回路ハードウェア研究の最前線」，DS-2-9, 2017 年 3 月 22 日-25 日 (22)，愛知，名城大学天白キャンパス .
- 堀三晟，田向権，森江隆，“hw/sw 複合体への実装を目指したハードウェア指向の制限付きボルツマンマシン”, LSI とシステムのワークショップ 2016, 5 月, 2016.

学会発表（共著者，査読無）

- 石田裕太郎，堀三晟，森江隆，田向権，“FPGA による ROS 向け高速分散処理システムの実装,”第 60 回システム制御情報学会研究発表会講演会，111-5，京都，5 月，2016.

講演

- 堀 三晟，石田 裕太郎，奥村 弘治，木山 雄太，楠根 穣，田中 悠一朗，辻 湧弥，土田 崇 弘，土谷 諒，藤本 武，山崎 裕太，佐藤 寧，森江 隆，田向 権，“Hibikino-Musashi@Home チームにおけるロボット開発,” 第 4 回インテリジェントホームロボティクス研究会, 大阪，5 月，2016 (招待講演).

その他

- RoboCup Japan Open 2020 @Home リーグ (オンライン開催)，実行委員
- RoboCup Japan Open 2019 Nagaoka @Home リーグ，実行委員
- RoboCup Japan Open 2018 Ogaki @Home リーグ，実行委員
- RoboCup 2017 Nagoya Japan（世界大会）, Team Description Paper (OPL 及び DSPL) 執筆
- RoboCup 2017 Nagoya Japan（世界大会）@Home リーグ, OPL: 5 位, DSPL: 優勝
- RoboCup Japan Open 2017 @Home リーグ，実行委員，OPL: 3 位，DSPL: 準優勝

投稿中（論文誌，筆頭著者，査読有）

- Sansei Hori, Hakaru Tamukoh, “An implementation method using cut-off bits for restricted Boltzmann machines without random number generators,” IEEE Access.