# Algorithm Optimization and Hardware Acceleration for Machine Learning Applications on Low-energy Systems 

Jianchi Sun<br>jianchs@clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations
Part of the Digital Circuits Commons, Systems and Communications Commons, and the VLSI and Circuits, Embedded and Hardware Systems Commons

## Recommended Citation

Sun, Jianchi, "Algorithm Optimization and Hardware Acceleration for Machine Learning Applications on Low-energy Systems" (2022). All Dissertations. 3145.
https://tigerprints.clemson.edu/all_dissertations/3145

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

# Algorithm Optimization and Hardware Acceleration for Machine Learning Applications on Low-energy Systems 

A Dissertation<br>Presented to the Graduate School of<br>Clemson University

$\qquad$

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Electrical Engineering

by<br>Jianchi Sun<br>July 2022

Accepted by:
Dr. Yingjie Lao, Committee Chair
Dr. Deborah Kunkel
Dr. Yongkai Wu
Dr. Linke Guo

## Abstract

Machine learning (ML) has been extensively employed for strategy optimization, decision making, data classification, etc. While ML shows great triumph in its application field, the increasing complexity of the learning models introduces neoteric challenges to the ML system designs. On the one hand, the applications of ML on resource-restricted terminals, like mobile computing and IoT devices, are prevented by the high computational complexity and memory requirement. On the other hand, the massive parameter quantity for the modern ML models appends extra demands on the system's I/O speed and memory size. This dissertation investigates feasible solutions for those challenges with software-hardware co-design.

In many emerging wireless IoT systems, the captured latency-sensitive data and the channel dynamics are governed by stochastic processes that are unknown a priori, which introduces the necessity of a self-learning system that can dynamically adapt to such unknown dynamics and statistical information. To this end, we find reinforcement learning (RL) a promising approach. However, current RL technologies are either too slow-converged (like Q-learning) for real-time learning or too complex (like deep Q-learning) for resource-constrained wireless IoT systems that cannot satisfy the learning requirement of the wireless IoT systems.

To address the limitations of the existing approaches described above, we design an novel RL technique, post decision states (PDS) learning and the corresponding hardware accelerator. In PDS learning, the learning problem is decomposed into known and unknown components, which significantly accelerates the learning convergence rate compared to Q -learning with the cost of additional computational complexity to integrate the known components into the algorithm. Solving this problem, we then exploit efficient hardware architectures for PDS learning. We first implement an arithmetic accelerator with paralleled structures and customized look-up table with state encoding so that it is $5.3 \times$ faster than Q -learning. Then we propose a stochastic computing (SC) based
reconfigurable hardware architecture to estimate the probability distribution instead of calculating the true value. Ultimately, the proposed SC-based architecture further reduces the critical path of the arithmetic accelerator by $87.9 \%$.

In order to minimize the parameter sizes for ML models, we study a novel number system called posit number, which delivers better value accuracy and dynamic range compared to floating point. Those advantages are yielded from a varying-length segment, regime bits, which lead to the size variations for all rest components except the sign bit. Consequently, it requires an extra decoding process to extract the numerical value of a posit number. The current posit decoder is designed based on a leading one/zero detector. However, we find that this conventional method holds implicit redundancy when dealing with binary numbers. Based on that, we design a novel hardware architecture, i.e., the leading difference detector, to optimize the circuit operation by eliminating the redundancy. The experimental results show that the proposed architecture can decrease the delay and power consumption by over $41 \%$ compared to the conventional designs for 16 -bit, 32 -bit, and 64 -bit posit decoders.

Recent studies show that the current machine learning models perform poorly in tracking the implied uncertainty of real-world problems. Improving this weakness, Bayesian neural network use probability distributions instead of single value numbers as its parameter to represent the involved uncertainty. However, the computational complexities for the current Bayesian neural networks are unacceptably high, it limits the application scenarios of the Bayesian neural networks. To this end, we proposed Bayesian optimization for neural networks based on the piecewise probability distributions, which can perform efficient Bayesian updates on the current hardware to improve the neural network's performance. In addition, we proposed a hardware accelerator that generates samples based on the piecewise probability distributions. The simulation result shows that it burns about half of the energy when generating the same amount of samples compared to the basic hardware structure.

## Dedication

I am dedicating this thesis to Mr. Steve Jobs, who have meant and continue to mean so much to me. You and your great idea and products were the initial spark for me to study electricity, to travel to United State, and to always, think different.

## Acknowledgments

I feel grateful to all the people in my life who showed kindness and would like to give my best wish. I would never have a chance to reach this point without your help and support.

Especially, I would like to express my sincere gratitude to my academic advisor, Dr. Yingjie Lao. He helped me grow from an abecedarian to a PhD candidate with his professional skills and patient. I could not image how can I complete the PhD program without his help. Beside the academic support, I am very thankful to him for providing me a not only fruitful but also cheerful journey. He tried his best to make the dull and stressful PhD life smooth and enjoyable to me.

Secondly, I would like to thank all my committee members: Dr. Deborah Kunkel, Dr. Linke Guo, and Dr. Yongkai Wu. for offering me many comments, reviews, and advices of great value on my research and dissertation.

I would also like to thank Dr. Nicholas Mastronarde and Dr. Jacob Chakareski. They provided many meritorious academic suggestions on my papers and dissertation.

I also want to appreciate my lab colleagues for their self-giving support on both of my research and life. I will never forget the wonderful days we spent together.

Finally, I want to express my appreciation to my family, my mother P. Ma, my father H. Sun for their unconditional love and support, and also my girlfriend Y. Sun, my irreplaceable family member Meng Meng and Hei Hei for their accompanying and sustaining.

## Table of Contents

Title Page ..... i
Abstract ..... ii
Dedication ..... iv
Acknowledgments ..... v
List of Tables ..... viii
List of Figures ..... ix
1 Introduction ..... 1
1.1 Motivation ..... 1
1.2 Objective ..... 2
2 Hardware Acceleration for Next-Generation Real-Time Reinforcement Learn- ing in Emerging IoT Systems ..... 4
2.1 Motivation ..... 4
2.2 Related Work ..... 6
2.3 Proposed Hardware Architecture ..... 8
2.4 Experimental Results ..... 12
2.5 Summary ..... 16
3 Stochastic Computing Based Programmable Hardware Accelerator for Post- Decision State Reinforcement Learning in IoT Systems ..... 17
3.1 Motivation ..... 17
3.2 Related Works ..... 19
3.3 Proposed Hardware Architecture ..... 30
3.4 Experimental Results ..... 39
3.5 Summary ..... 48
4 Efficient Data Extraction Circuit for Posit Number System: LDD-based Posit Decoder ..... 49
4.1 Motivation ..... 49
4.2 Background ..... 51
4.3 LDD-based Posit Decoder ..... 53
4.4 Hardware Cost Estimation ..... 57
4.5 Experimental Results ..... 59
4.6 Conclusion ..... 60
5 Bayesian Optimization for Neural Network ..... 63
5.1 Motivation ..... 63
5.2 Bayesian Update for Peicewise Probability Distributions ..... 64
5.3 Bayesian Optimization Algorithm ..... 65
5.4 Hardware Acceleration ..... 68
6 Future Work ..... 71
6.1 LDD-based Posit Arithmetic Core and Neuron ..... 71
6.2 Bayesian Optimization for Neural Network ..... 71
Bibliography ..... 73

## List of Tables

2.1 Optimized vs. Baseline Architectures (32-Bit) ..... 15
2.2 PDS vs. Q-learning on Hardware (32-Bit) ..... 15
3.1 Arithmetic vs. Baseline Hardware vs. Q-Learning (32-Bit) ..... 45
3.2 Comparison with our prior work [107] (32-Bit) ..... 47
3.3 4-way parallel AE (32-Bit) ..... 48
4.1 Regime Bits Decoding Example ..... 52
4.2 Hardware Cost Estimation ..... 59
4.3 Comparison: LDD vs. LOD [59, 57, 24, 119, 36] ..... 60
4.4 Comparison for Extremely Small Data Size ..... 60
5.1 Comparison for Sample Generator ..... 70

## List of Figures

$$
\begin{aligned}
& \text { 2.1 Top-level architecture of the proposed hardware accelerator for action evaluation. It } \\
& \text { comprises two main blocks: Known Cost and State Value Expectation. . . . . . . . . } 9
\end{aligned}
$$

2.2 An example of case encoding, where the input bit-width is compressed from 6 to 2 ,
and unused cases are decreased over 61 times.
2.3 The proposed parallel structures for (a) power tree and (b) multi-sum tree. . . . . . 11
2.4 Ordered storage array (left) vs. random storage array (right). . . . . . . . . . . . . . 12
2.5 Comparison between PDS learning and Q-learning. . . . . . . . . . . . . . . . . . . . 13
2.6 Comparison of convergence speed. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.7 Comparison of different bit-widths. All results are normalized to those for 16 -bit,
whose delay is 49.89 ns , power is 1.87 mW , and cell number is 32,030 cells. . . . . 16
3.1 Wireless IoT system model. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2 Stochastic computing circuit. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.3 Action evaluation hardware accelerator designs for the example system model. The
SVE block is illustrated in Fig. 3.3(a) assuming that up to 10 packets can be trans-
mitted in each time step, i.e., $\mathcal{S}_{z}=\{1,2, \ldots, 10\}$. . . . . . . . . . . . . . . . . . . 33
3.4 An example of state encoding where the input bit-width is compressed from 3 to 1. . 34
3.5 Programmable lookup table with memory and SCM. . . . . . . . . . . . . . . . . . . 34
3.6 The logic circuit of state encoding (SE) module (4 states example). . . . . . . . . . . 35
3.7 Lookup table with state encoding. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
3.8 The framework of the stochastic sample generator. . . . . . . . . . . . . . . . . . . . 37
3.9 TPDE for the binomial distribution family. . . . . . . . . . . . . . . . . . . . . . . . 38
3.10 Programmable 4-way action evaluation structure. . . . . . . . . . . . . . . . . . . . . 39
3.11 Comparison between PDS learning, Q-learning, and deep Q-learning. . . . . . . . . . 41
3.12 Effect of stochastic process from SSG. . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.13 Convergence for a single sample. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.14 Error-tolerance. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.15 The layout of the arithmetic hardware design. . . . . . . . . . . . . . . . . . . . . . . 46
3.16 Replaced circuit from the arithmetic accelerator. . . . . . . . . . . . . . . . . . . . . 47
4.1 Generic posit format for finite, nonzero values. . . . . . . . . . . . . . . . . . . . . . 50
4.2 Circuit design for LOD. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.3 LDD output format example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.4 Example circuit for 4-bit LDD. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.5 3-bit output 'en' stage. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.6 Example circuit for 4-bit shifter. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.7 LDD-based decoder vs. LOD-based decoder. . . . . . . . . . . . . . . . . . . . . . . 62
5.1 Bayesian Optimization without and with Backpropagation. . . . . . . . . . . . . . . 68
5.2 Sample generator for piecewise probability distribution. . . . . . . . . . . . . . . . . 69
5.3 The logic circuit of comp array (4-sample example). . . . . . . . . . . . . . . . . . . 70

## Chapter 1

## Introduction

### 1.1 Motivation

Since invented, machine learning (ML) has been dramatically promoted by researchers with its extraordinary potential for strategy optimization, decision making, data classification, etc. As one of the most popular ML models, reinforcement learning (RL) [109, 70] trains agents to optimize the decision-making strategies for maximizing the reward. Additionally, artificial deep neural networks (DNN) have become one of the most successful ML models that outperform the conventional statistical models and regression [4]. In recent studies, RL and DNN are combined in the way that DNN holds the underlying state values for RL model. This combination forms many deep RL models like deep Q-learning (DQL) [75], deep deterministic policy gradient (DDPG) [63], proximal policy optimization (PPO) [94], and asynchronous advantage actor critic (A3C) [74]. By introducing the deep models, deep learning methodologies show great ability to capture the non-linear features of complex problems.

With all the advantages of deep learning, it comes with the cost of the extremely high hardware requirement, which forces most of the deep learning models to be trained offline on powerful GPUs. For example, as a relatively "old" middle-size deep learning model, VGG-16 [102] has $1.38 \times 10^{8}$ parameters and performs $1.55 \times 10^{10}$ multiplications for each iteration. This high cost limits the application of deep learning on mobile devices like cell phones and Internet of Things (IoT) systems. Targeting on this weakness, [53] designed a light weighted DNN for mobile and embedded vision applications, [97] proposed a survey on Mobile Edge Computing where the resource-hungry
tasks were transferred to the near servers. Despite all the works on low-energy machine learning systems, there is still not a high-performance solution for the real-time learning on wireless IoT devices, where both efficient training and inference need to be executed online, at run time, since the energy support on those devices is insufficient to run small-size neural networks, and each device need to be trained individually for its own environment dynamics.

At the same time, growing ML model sizes also bring challenges to memory access with the enormous number of parameters [26]. As mentioned by [49], a 32-bit DRAM operation burns approximately $173 \times$ energy compared to a 32 -bit float multiplication with 45 nm CMOS process. Driven by this, recent studies have been trying to decrease the parameter sizes for ML models. $[108,128]$ reduced the parameter sizes of DNN to 8 bits, and binary neural networks were proposed and studied by $[125,64,113]$ to further shrink the parameter size to only 1 bit. However, we found the training accuracy of the mentioned studies inevitably suffered distinctly from the lost parameter dynamic ranges.

### 1.2 Objective

In this dissertation, we aim to develop a hardware friendly ML methodology for real-time learning on wireless IoT systems and its hardware accelerators. In addition, we try to find a way to reduce parameter sizes without sacrificing the dynamic range for ML. Our contributions can be summarized as follows:

- We proposed a network-free novel RL model called post-decision state (PDS) learning to capture the latency-sensitive data and the channel dynamics for wireless IoT systems and optimize the system performance. Our simulations show that PDS learning delivers similar converging performance compared to DQL even without the costly deep learning structures.
- We design an efficient optimized hardware accelerator for the action evaluation of PDS learning, which computes the action to select given the present state. We implement many novel structures like look-up tables with state encoding, highly parallel multi-sum tree, and ordered storage register array with component auto-disable. The accelerator optimize the speed and power consumption for applying PDS learning on wireless IoT systems.
- We then propose a novel stochastic computing (SC)-based hardware architecture, referred
to as the transition probability distribution estimator (TPDE), for calculating the known transition probability from the state to the PDS without using multipliers. Leveraging the PDS learning's robustness to stochastic perturbations, TPDE further accelerates the required computation and reduces the induced power consumption, and introduces extra error tolerance to the system.
- We study the novel number system named posit number, which holds better accuracy and dynamic range compared to floating point and shows great prospect for ML. We design a brand new structure called leading difference detector (LDD) and the corresponding posit number decoder, which outperforms the current decoding circuits on both speed and power consumption.


## Chapter 2

## Hardware Acceleration for

## Next-Generation Real-Time

## Reinforcement Learning in

## Emerging IoT Systems

### 2.1 Motivation

A variety of emerging applications spanning autonomous driving, mobile augmented and virtual reality, remote multi-view sensing, personalized healthcare, virtual teleportation, UAV-IoT, $360^{\circ}$ video streaming, remote robot navigation, cooperative video delivery, and telemetry [19, 96, 22, $23,114,21,71,18,20,30$, rely on computing and communication limited Internet of Things (IoT) devices and sensors $[69,5,123]$. The stochastic processes governing the captured latency-sensitive data and the channel dynamics, arising in such emerging settings, are not known a priori. This necessitates learning the respective desired optimal transmission policies online, during operation, to adapt to the experienced traffic and channel dynamics.

To this end, reinforcement learning (RL) [109, 70] has been shown to be an extremely effective tool, with Q-learning being its most widely-used method [121]. For instance, Q-learning
has been employed to maximize the throughput of an energy-harvesting transmitter [13]. While Qlearning can solve problems with small state/action spaces, it exhibits poor convergence rates, which makes it inappropriate for problems involving large state/action spaces. Additionally, this approach is purely data-driven, which does not incorporate any useful information about the underlying system dynamics.

Recently, we explored and advanced the concept of post-decision states (PDS) [73, 70, 109, 84, 117, 99, 98, 72], which exploits basic system knowledge to considerably advance the RL learning rate. PDS capture the system state after an action is taken, but before the unknown dynamics take place, which allows us to decompose the problem into known and unknown components, where only the latter must be learned. Though using PDS can speed-up the convergence to the optimal policy, it introduces the cost of increased action-selection complexity [70], which brings challenges to realtime applications. Moreover, the limited computing and power of wireless IoT systems [?] represent further challenges to actual deployment. Thus, hardware acceleration is a promising direction to enable real-time IoT applications of PDS based learning [83, 95].

In this chapter, we design an efficient architecture for action evaluation, which computes the action to select given the present state. This step is the computational bottleneck in PDS based RL systems, as it is involved in greedy action selection and state value updating in each iteration. The key novelty of our design includes i) re-structuring the action evaluation of PDS based RL for hardware optimization, which yields a speed up of over 49 times, compared to the software counterpart; and ii) further optimizing the hardware accelerator's performance by efficiently computing the transmission power costs $\left(P_{t x}\right)$ and packet loss rates $(P L R)$ using lookup tables (LUTs), re-ordering the register array for the value function $V(s)$, and parallelizing the computation with two dedicated trees. As a result, the computational delay of our hardware accelerator is further reduced by $66.3 \%$, while the power consumption and cells number are also decreased by $85 \%$ and $86 \%$, respectively. Meanwhile, when compared to Q-learning, our optimized accelerator achieves a $83 \%$ delay reduction and a $59 \%$ power consumption reduction.

### 2.2 Related Work

### 2.2.1 PDS based Reinforcement Learning

We consider a time-slotted wireless IoT sensor and aim at improving the wireless power management, with the specific objective to minimize the sensor's energy consumption, subject to an operational delay constraint.

To implement RL for the wireless power management problem, we first formulate it into a constrained MDP. We assume that time is divided into slots with length $\Delta T$ (seconds) and that the system's state in the $n$-th time slot is denoted by $s^{n} \triangleq\left(b^{n}, h^{n}, x^{n}\right) \in \mathcal{S}$, with packet buffer state $b^{n}$ (i.e., the number of packets stored in the buffer), channel fading state $h^{n}$, and power management state $x$ (radio on/off). At the beginning of each time slot, the IoT sensor observes its state $s^{n}$ and takes an action $a^{n}=\left(B E P^{n}, y^{n}, z^{n}\right)$, where $B E P^{n}$ is its target bit-error probability, $y^{n}$ is its power management action (turn on/off the radio), and $z^{n}$ is its packet throughput (number of transmitted packets). We aim to determine the action in each state to minimize the cost $c\left(s^{n}, a^{n}\right)=$ $\rho\left(s^{n}, a^{n}\right)+\lambda g\left(s^{n}, a^{n}\right)$ over time, where $\rho(s, a)$ is the power cost, $g(s, a)$ is the delay cost, and $\lambda$ is a Lagrange multiplier to set the delay constraint.

The sequence of states $s^{n}:\{n=0,1, \ldots\}$ can be modeled as a controlled Markov chain with transition probabilities equal to the product of individual state transitions, as in Equation (3.10), where $b^{\prime}$ is defined by Equation (2.2). Here $f$ is the packet goodput (correctly received packets), $l$ is the number of packet arrivals, and $N_{b}$ is the buffer's capacity.

$$
\begin{equation*}
P\left(s^{\prime} \mid s, a\right)=P^{b}\left(b^{\prime} \mid[b, h], a\right) P^{h}\left(h^{\prime} \mid h\right) P^{x}\left(x^{\prime} \mid x, a\right) \tag{2.1}
\end{equation*}
$$

$$
\begin{equation*}
b^{\prime}=\min \left(b-f+l, N_{b}\right) \tag{2.2}
\end{equation*}
$$

From Equation (2.2), it can be concluded that $P^{b}$ depends on the goodput distribution $P^{f}$. Assuming independent packet losses, $P^{f}(f \mid B E P, z)=\operatorname{binomial}(z, 1-P L R)$, where $P L R=$ $1-(1-B E P)^{L}$ is the packet loss rate for a packet with size of $L$ (bits).

A post-decision state (PDS), represented by $\widetilde{s} \triangleq(\widetilde{b}, \widetilde{h}, \widetilde{x}) \in \mathcal{S}$, denotes a state of the system after all known/controllable dynamics have occurred but before the unknown dynamics occur [90,

70, 109]. In our problem,

$$
\begin{equation*}
\widetilde{s}^{n}=\left(\left[b^{n}-f^{n}\right], h^{n}, x^{n+1}\right) . \tag{2.3}
\end{equation*}
$$

We can formulate our problem in terms of PDSs instead of conventional states by decomposing the transition $s \rightarrow s^{\prime}$ into two parts: a known transition $s \rightarrow \widetilde{s}$ with cost $c_{k}(s, a)$ and transition probability $P_{k}(\widetilde{s} \mid s, a)$, and an unknown transition $\widetilde{s} \rightarrow s^{\prime}$ with cost $c_{u}(\widetilde{s})$ and transition probability $P_{u}\left(s^{\prime} \mid \widetilde{s}\right)$. We can define two optimal value functions $V^{*}(s)$ and $\widetilde{V}^{*}(\widetilde{s})$ over the conventional states and PDSs, respectively. The two value functions are related by the following equations:

$$
\begin{align*}
\widetilde{V}^{*}(\widetilde{s}) & =c_{u}(\widetilde{s})+\gamma \sum_{s^{\prime} \in \mathcal{S}} P_{u}\left(s^{\prime} \mid \widetilde{s}\right) V^{*}\left(s^{\prime}\right)  \tag{2.4}\\
V^{*}(s) & =\min _{a \in \mathcal{A}(s)}\left\{c_{k}(s, a)+\sum_{\widetilde{s} \in \mathcal{S}} P_{k}(\widetilde{s} \mid s, a) \widetilde{V}^{*}(\widetilde{s})\right\} . \tag{2.5}
\end{align*}
$$

Knowing $\widetilde{V}^{*}(\widetilde{s})$, the optimal policy $\pi^{*}$ can be found by taking the action in each state that minimizes the right-hand side of Equation (3.20). To solve the problem online, we use the PDS learning algorithm $[90,70,73]$, which is a stochastic iterative algorithm. PDS learning takes the greedy action in each time slot and updates the value of the present state $\widetilde{s}^{n}$ by using a weighted average of (i) the current PDS value function estimate $\widetilde{V}\left(s^{n}\right)$, and (ii) a new sample estimate of the PDS value function based on the next state's estimated value as:

$$
\begin{equation*}
\widetilde{V}^{n+1}\left(\widetilde{s}^{n}\right)=\left(1-\alpha^{n}\right) \widetilde{V}^{n}\left(\widetilde{s}^{n}\right)+\alpha^{n}\left[c_{u}^{n}\left(\widetilde{s}^{n}\right)+\gamma V^{n}\left(s^{n+1}\right)\right] \tag{2.6}
\end{equation*}
$$

Since the unknown system dynamics are not dependent on the action taken, using PDSs obviates the need for action exploration. Algorithm 2 presents the pseudo-code for the PDS learning algorithm using an adaptive learning rate $\alpha^{n} \in[0,1]$, where action evaluation requires computing $\left\{c_{k}\left(s^{n}, a\right)+\sum_{\widetilde{s}} P^{k}\left(\widetilde{s} \mid s^{n}, a\right) \widetilde{V}^{n}(\widetilde{s})\right\}$ in Equations (3.22) and (2.8).

### 2.2.2 Conventional Q-Learning

For the algorithmic comparison, we also briefly introduce Q-learning. The key step in Qlearning is performing an update at the end of every time slot according to the current experience

```
Algorithm 1 Post-Decision State Learning
    initialize \(\tilde{V}^{0}(\widetilde{s})=0\) for all \(\widetilde{s} \in \mathcal{S}\)
    for time slot \(n=0,1,2, \ldots\) do
        Take the greedy action:
```

$$
\begin{equation*}
a^{n}=\underset{a \in \mathcal{A}}{\arg \min }\left\{c_{k}\left(s^{n}, a\right)+\sum_{\widetilde{s}} P^{k}\left(\widetilde{s} \mid s^{n}, a\right) \widetilde{V}^{n}(\widetilde{s})\right\} \tag{2.7}
\end{equation*}
$$

Observe PDS $\widetilde{s}^{n}$, next state $s^{n+1}$, unknown $\operatorname{cost} c_{u}^{n}$
Evaluate the state value function at time $n+1$ :

$$
\begin{equation*}
V^{n}\left(s^{n+1}\right)=\min _{a \in \mathcal{A}}\left\{c_{k}\left(s^{n+1}, a\right)+\sum_{\widetilde{s}} P^{k}\left(\widetilde{s} \mid s^{n+1}, a\right) \widetilde{V}^{n}(\widetilde{s})\right\} \tag{2.8}
\end{equation*}
$$

Calculate $\widetilde{V}^{n+1}\left(\widetilde{s}^{n}\right)$ using Equation (3.23)
end for
tuple: $\left(s^{n}, a^{n}, c^{n}, s^{n+1}\right)$. The update can be expressed as:

$$
\begin{align*}
& Q^{n+1}\left(s^{n}, a^{n}\right) \leftarrow \\
& \left(1-\alpha^{n}\right) Q^{n}\left(s^{n}, a^{n}\right)+\alpha^{n}\left[c^{n}+\gamma \min _{a^{\prime} \in \mathcal{A}} Q^{n}\left(s^{n+1}, a^{\prime}\right)\right] \tag{2.9}
\end{align*}
$$

where $s^{n+1}$ is distributed based on the transition probability distribution $P\left(s^{n+1} \mid s^{n}, a^{n}\right) ; a^{\prime}$ is the greedy action in time slot $n+1$; $\alpha^{n}$ represents the time-varying learning rate parameter; and $Q^{0}(s, a)$ can be initialized arbitrarily for all $(s, a) \in \mathcal{S} \times \mathcal{A}$. In the literature, many researchers have explored various Q-learning based RL hardware accelerator structures for better performance and lower power consumption $[40,8,54,87]$. However, these hardware optimization techniques are not, at least directly, applicable to our PDS learning algorithm, as PDS based methods are uniquely optimized for emerging wireless IoT systems to reduce the convergence time. Therefore, it is important to exploit dedicated hardware accelerators for the PDS based learning algorithms.

### 2.3 Proposed Hardware Architecture

Here, we present an optimized hardware accelerator for the action evaluation step to improve the efficiency and hence facilitate real-world deployment of next-generation RL techniques. The proposed hardware accelerator is mainly composed by two components: Known Cost (KC) block and State Value Expectation (SVE) block, as shown in Fig. 3.3. Specifically, we optimize the lookup table (green), tree structure (blue), and data selection (orange), according to the unique
characteristics of the PDS based RL algorithm to speedup the computation and reduce the power consumption. We present the detailed design and optimization approaches below.


Figure 2.1: Top-level architecture of the proposed hardware accelerator for action evaluation. It comprises two main blocks: Known Cost and State Value Expectation.

### 2.3.1 Lookup Table Reduction and State Encoding for RL

To avoid an infinite number of channel states in the proposed module, all analog states are quantized to discrete values. In order to further reduce the computational complexity, we design a lookup table reduction structure with state encoding. This reduces execution time and lowers the power consumption of the learning system, which are critical aspects for real-time wireless IoT systems [60].

At the beginning stages of our module, most computations are complex and computationallyintensive with heavy multiplications and power operations (e.g., $\binom{z}{i}$ when computing the Binomial goodput distribution, $P L R=1-P R R=1-(1-B E P)^{L}$, and $P_{t x}$ defined by Equation (2.10),
where $\beta$ is proportional to $z$, and $\operatorname{erf}()$ denotes the error function). However, the combinations of the inputs are limited by the size of state and action spaces. When the number of states is small, a lookup table is proved to be a promising choice for the implementation [55, 2, 115]. Therefore, we pre-process most computations at the input stage, which are then implemented as lookup tables, as shown in Fig. 3.3.

$$
\begin{equation*}
P_{t x}=\frac{\sqrt{2} N_{0}\left(2^{\beta}-1\right) \operatorname{erf}^{-1}\left(1-\frac{\beta * B E P}{4}\right)}{3 * h} . \tag{2.10}
\end{equation*}
$$

However, in the PDS learning algorithm, a large number of state values are not used after quantization (e.g. There are only 8 valid channel states, but $2^{32}$ possible inputs for a 32 -bit system), which introduces redundant input space for the lookup table and negatively impacts the performance. To this end, the lookup tables for $B E P$ and $h$ are further optimized by state encoding. Discrete values are encoded into successive binary addresses to compress the input bit-width and unused cases, as shown in Fig. 2.2, which achieved a reduction of $61 \times$ for unused case numbers. As a result, the circuit cost, speed, and power consumption are all improved by using a smaller input size. In our implementation, the bit-widths of both $B E P$ and $h$ are reduced from 32 -bit to 3 -bit for the 32 bit system. Furthermore, the encoded case input makes the circuit more re-programmable friendly across different applications [37, 111]. The inputs can be encoded similarly based upon the resolution used for the channel state and BEP (or any other continuous parameter), while the lookup tables can be easily updated for a different environment.


Figure 2.2: An example of case encoding, where the input bit-width is compressed from 6 to 2 , and unused cases are decreased over 61 times.

### 2.3.2 State Value Expectation (SVE)

Tree Structure: When calculating the SVE, all probabilities and state values for possible PDSs have to be collected and calculated (3.22), which makes the SVE block in general much
slower than the KC block. Inspired by the parallel designs in recent works of efficient hardware implementation $[120,58,104]$, we propose a parallelized structure for the SVE block with two tree structures: power tree (Fig. 2.3(a)) and multi-sum tree (Fig. 2.3(b)). The power tree takes a probability $p$ as input and outputs all of the $p^{0}-p^{10}$ simultaneously (all the outputs will be read out at the same time when the circuit finishes switching, while the multi-sum tree collects all $P L R^{i}$ (packet loss rate), $P R R^{i}$ (packet receive rate), $V(s)$, and chooses values $\binom{z}{i}$ based on the current state and action ( 77 values in total), then computes $\mathbb{E}(V(\widetilde{s}))$ with only 3 multipliers and 5 adders. Besides accelerating the computation, the parallel design can also reduce power consumption since it decreases the critical path and eliminates the need for extra registers for data buffering or redundant computation.


Figure 2.3: The proposed parallel structures for (a) power tree and (b) multi-sum tree.

Data Selection: During the AE step, a set of state values for each possible PDS needs to be selected among all the state values (i.e., $\widetilde{V}(\widetilde{s})$ for all $\operatorname{PDS} \widetilde{s}$ such that $\left.P_{k}(\widetilde{s} \mid s, a) \neq 0\right)$. This process introduces two challenges to the hardware design: 1) the total number of states could change significantly based on the complexity of the system model; and 2) the number of possible PDSs may vary, for instance, when the current buffer state $b$ is smaller that the maximum value of the transmission action $z$ in our example system model. We propose to use an ordered state value array and a component auto-disable mechanism to simplify the computation.

In all cases, the range of possible PDSs is near the current state $b_{0}$, i.e., the PDS buffer state range $\left\{b_{0}-z, b_{0}-z+1, \ldots, b_{0}\right\}$ in our system model is just like the area around a player's location in video game that can be reached within one step. Therefore, we reorder the storage array such that
all candidates of the PDSs for each possible case are stored consecutively, as shown in Fig. 2.4. At the same time, we design the selection module to always output PDS values for $\widetilde{b}=b_{0}$ to $\left(b_{0}-z_{\max }\right)$ for both $\widetilde{x}=\mathrm{ON}$ and OFF , since redundant state values will be canceled by the 0s from the Choose Lookup. With all the designs above, the selection module needs to find only the location for $\widetilde{V}\left(b_{0}\right)$ then outputs it with its very next 21 state values. As a result, by implementing this for our wireless model, the selection module is reduced from 416-to- (2~22) selection (total 416 states and possible $2 \sim 22$ PDSs) to 52 -to- 1 selection, which only finds $b_{0}(26$-to- 1$)$ and $x$ (2-to- 1 ).


Figure 2.4: Ordered storage array (left) vs. random storage array (right).

### 2.3.3 Known Cost (KC)

The computation of transmission power $P_{t x}$ dominates the complexity of the KC block, which includes multiplications, power options, and the inverse error function, as expressed by Equation (2.10). To speed up the computation, we decompose $P_{t x}=g(z, B E P) * 1 / h$, where $g(z, B E P)$ can be given by:

$$
\begin{equation*}
g(z, B E P)=\frac{\sqrt{2} N_{0}\left(2^{\beta}-1\right) \operatorname{erf}^{-1}\left(1-\frac{\beta * B E P}{4}\right)}{3} \tag{2.11}
\end{equation*}
$$

Consequently, we construct a lookup table for $g(z, B E P)$ of size $\operatorname{size}(z) * \operatorname{size}(B E P)=10 \times 5$, which helps avoid integral and power computations.

### 2.4 Experimental Results

### 2.4.1 Learning Algorithm Comparison

Fig. 3.11 compares the simulated performance between our PDS learning implementation (Algorithm 2) and Q-learning. All results are generated by a MATLAB based simulator over
$3,000,000$ time slots. It can be seem from Fig. 3.11 that the PDS learning algorithm outperforms Q-learning in terms of both cumulative average delay and power consumption.


Figure 2.5: Comparison between PDS learning and Q-learning.

Besides power and delay, we further analyse the convergence speed of our algorithm in Fig. 2.6. The red curve (circle markers) denotes the cumulative average cost incurred up to time slot $n$ by Q-learning (where the cost is defined in Section 2.2.1 as a weighted sum of the power cost and delay cost) and the blue curve (+ markers) denotes the cumulative average cost for PDS learning. While PDS learning approximately converges in 250,000 time slots, Q-learning has still not converged after
$3,000,000$ time slots, so it is at least 12 times slower than PDS learning. This shows that PDS learning is a better candidate for real-time IoT systems, where fast learning is needed to adapt to the real environment.


Figure 2.6: Comparison of convergence speed.

### 2.4.2 Hardware Implementation

We implemented and evaluated the following four approaches: Our proposed efficient actionevaluation architecture, a baseline straightforward hardware design without employing the proposed optimization, a software implementation with C++, and a Q-learning circuit using Verilog HDL. For a fair comparison, all common intrinsic variables and state values $V(s)$ use a bit-width of 32 . They were all mapped to a 32 nm technology node using a Synopsys Design Compiler. The software is coded and tested with $\mathrm{C}++$ on macOS, with 2.6 GHz 6 -core Intel i7 processor and 16GB RAM. No multi-threaded optimization is added to the code, which means the software runs with only a single core under the limitation of macOS. As wireless IoT systems usually have less computing resources, we consider this setting as a guaranteed upper bound for the software implementation's speed.

We evaluate and compare the execution delays and average runtime for our two hardware designs and the software implementation of PDS learning. Furthermore, the power and area consumption of the optimized hardware accelerator and the baseline design are compared to illustrate the effectiveness of the proposed hardware optimization techniques. These results and comparisons are shown in Table 3.1, where the execution times and power/area consumptions are also shown
normalized to the optimized hardware design, for the baseline hardware design and software implementation. According to the experimental results, our optimized hardware accelerator is $3 \times$ faster than the baseline circuit, while achieving a 49 times acceleration over the software implementation. The power consumption and cells number are also decreased by $85 \%$ and $86 \%$ respectively, compared to the baseline hardware design.

Table 2.1: Optimized vs. Baseline Architectures (32-Bit)

|  | Optimized <br> Hardware <br> $($ PDS $)$ | Baseline <br> Hardware <br> (PDS) | Software |
| :--- | :--- | :--- | :--- |
| Delay $(n s)$ | 86.97 | $258.31(3 \times)$ | $4240(49 \times)$ |
| Power $(m W)$ | 6.17 | $41.21(7 \times)$ | - |
| $\#$ of Cells | 93448 | $666543(7 \times)$ | - |

The comparison between our proposed architecture for PDS learning and Q learning is presented in Table 2.2. The implementation of Q-learning is based on Equation (3.13). According to the simulation results in Section 3.4.2, Q-learning converges over an order of magnitude slower than PDS based learning. Therefore, since the hardware will be activated once for each time slot, we normalize the hardware cost with respect to the convergence time for a fair comparison. These results show that the proposed PDS based learning accelerator achieves reductions of $83 \%$ and $59 \%$ in delay and power consumption, respectively, compared to Q-learning. Therefore, we can conclude that the proposed PDS learning architecture is faster and consumes less energy than Q-learning.

Table 2.2: PDS vs. Q-learning on Hardware (32-Bit)

|  | Optimized <br> Hardware <br> $(\mathrm{PDS})$ | Normalized <br> Q-learning |
| :--- | :--- | :--- |
| Delay $(n s)$ | 86.97 | $521.9(6 \times)$ |
| Power $(m W)$ | 6.17 | $15(2.4 \times)$ |

In addition, to achieve better performance according to the data range of a certain application scenario, designers vary the bit-width of the implementation[116, 15]. Thus, we also studied the hardware cost of our PDS learning accelerator for different bit-widths (i.e., 16, 32, and 64) as shown in Fig. 2.7. We normalize all the results to those for 16 -bit. It can be observed that the hardware complexity increases approximately linearly with the increase of the bit-width.


Figure 2.7: Comparison of different bit-widths. All results are normalized to those for 16-bit, whose delay is 49.89 ns , power is 1.87 mW , and cell number is 32,030 cells.

### 2.5 Summary

We presented an efficient hardware accelerator for action evaluation of PDS based realtime RL for next generation wireless communication systems. By algorithmic and hardware cooptimization of the PDS learning implementation, we achieved a significant speedup for the action evaluation process of PDS, while simultaneously reducing its power consumption.

## Chapter 3

## Stochastic Computing Based

## Programmable Hardware

## Accelerator for Post-Decision State

## Reinforcement Learning in IoT

## Systems

### 3.1 Motivation

In many emerging wireless IoT systems, the captured latency-sensitive data and the channel dynamics are governed by stochastic processes that are unknown a priori. This introduces the necessity of a self-learning system that can dynamically adapt to such unknown dynamics and statistical information. To this end, reinforcement learning (RL) [109, 70] has proven to be a promising approach. For example, in recent studies, the well-known Q-learning algorithm [121] has been employed to maximize the throughput [13] of energy harvesting transmitters, to minimize the sum of data compression and transmission energy of energy harvesting transmitters [48, 27], and to optimally trade off power and delay in IoT edge computing [65, 32]. Although Q-learning is
lightweight enough to be implemented on resource-constrained IoT devices, it converges too slowly to effectively adapt to the experienced information source and channel dynamics.

In parallel, deep RL has received increasing attention for its ability to solve difficult decisionmaking problems with large (and possibly continuous) state and action spaces, both from the machine learning community $[93,94,63,67,74]$ and from the wireless networking community $[45,28,14]$. However, deep RL algorithms have complex deep neural network architectures that make them infeasible to implement on resource-constrained wireless IoT systems where power, memory, and computational resources are limited [62, 112]. ${ }^{1}$ Worse still, deep RL algorithms are typically trained offline; therefore, they are not suitable for real-time learning where both training and decisionmaking need to be performed online, at run-time. For these reasons, none of the previously cited papers [45, 28, 14] deploy deep RL algorithms directly on end-devices and all of them train the algorithms offline. For instance, [45] investigates buffer-aware video streaming in a small-cell wireless network, [28] studies uplink scheduling for multiple energy-harvesting user equipments in a small-cell IoT system, and [14] demonstrates scheduling control in sliced 5G networks through an open radio access network (O-RAN). All of these deploy the trained deep RL agent at the base station or in the RAN, where sufficient computational resources are available.

To address the limitations of the existing approaches described above, our prior work advanced the concept of post-decision states (PDS) [73, 70, 117, 99, 100, 72] , as have others [109, 84, 90]. PDSs allow us to exploit basic system knowledge to improve the learning performance. Concretely, the learning problem is decomposed into known and unknown components, by identifying the transitory system state after the execution of an action (hence the name PDS) and prior to the unknown system dynamics taking place. With this property, PDS-based RL is capable of significantly accelerating the learning convergence rate compared to Q-learning, but this comes at the cost of additional computational complexity to integrate the known components into the algorithm. Although PDS learning is far less complex than deep RL, its complexity may still hinder its real-time implementation on resource-constrained IoT devices.

On the other hand, although software is a remarkable option in most use cases due to its great flexibility, recent literature demonstrates that hardware acceleration is essential for various machine learning methods to enable real-time and lightweight applications in resource-constrained

[^0]wireless IoT systems $[103,34,43,33,35,107]$. Following this direction, this chapter exploits efficient hardware architectures for PDS learning. We first design a hardware accelerator for the action evaluation (AE) step of PDS learning in Chapter 2, which evaluates the value of a prospective action. In this chapter, we propose a stochastic computing (SC) based and reconfigurable hardware architecture for the PDS learning algorithm. Specifically, by adopting SC, we eliminate the costly multiplications involved in the AE and replace them with sample estimation, which hence simultaneously reduces the hardware area and power consumption. Thanks to the resiliency of PDS learning to stochastic perturbations, we can further improve the computational efficiency by using extremely short stochastic representations (i.e., each signal is represented by a very small number of stochastic samples) without sacrificing arithmetic performance. To differentiate from the SC-based accelerator, we refer to the arithmetic accelerator as the arithmetic circuit in the rest of this chapter.

### 3.2 Related Works

### 3.2.1 System Model

We assume that a resource-constrained wireless IoT sensor must transmit delay-sensitive data over a fading channel to a receiver, while minimizing its power consumption. The system operates over discrete time steps indexed by $n \in\{0,1, \ldots\}$, with fixed length $\Delta T$ seconds.

Fig. 3.1 illustrates the considered wireless IoT system. At the beginning of time step $n$, the RL module observes the system's state $s^{n} \triangleq\left(b^{n}, h^{n}, x^{n}\right) \in \mathcal{S}$, where $b^{n} \in \mathcal{S}_{b}=\left\{0,1, \ldots, N_{b}\right\}$ is the finite buffer state, which represents the number of packets waiting in the buffer to be transmitted; $h^{n} \in \mathcal{S}_{h}$ is the channel state, which represents the discretized channel gain between the transmitter and receiver; $x^{n} \in \mathcal{S}_{x}$ is the binary power management state, which indicates if the radio is "on" and ready to transmit, or "off" in a power-saving state; and $\mathcal{S}=\mathcal{S}_{b} \times \mathcal{S}_{h} \times \mathcal{S}_{x}$ is the discrete and finite set of states. Subsequently, the RL module takes an action $a^{n}=\left(B E P^{n}, y^{n}, z^{n}\right) \in \mathcal{A}$, where $B E P^{n} \in \mathcal{A}_{B E P}$ is the target maximum bit-error probability (BEP) at the receiver; $y^{n} \in \mathcal{A}_{y}$ is the binary power management action, which indicates whether to turn "on" or "off" the radio; $z^{n} \in \mathcal{A}_{z}$ is the packet throughput, which specifies the number of packets to transmit; and $\mathcal{A}=\mathcal{A}_{B E P} \times \mathcal{A}_{y} \times \mathcal{A}_{z}$ is the discrete and finite set of actions. In our specific model implementation (see Section 3.3), there are a total of 416 states and 110 actions, which is relatively complex for resource-constrained wireless IoT devices.


Figure 3.1: Wireless IoT system model.

In the remainder of this subsection, we describe the channel, physical layer, transmission power, power management, transmission buffer, and traffic models in detail.

Channel model: We consider a frequency non-selective block fading channel with channel gain $h^{n} \in \mathcal{S}_{h}$ in time step $n$. As in prior work $[100,48,129,13,90,127]$, we assume that the set of channel states $\mathcal{S}_{h}$ is discrete and finite, that the channel state $h^{n}$ is known and constant in each time step, and that it evolves over time according to a discrete-time Markov chain with transition probability function $P^{h}\left(h^{\prime} \mid h\right)$. We determine the discretized channel state by defining fixed thresholds $0=\tau_{0}<\tau_{1}<\cdots<\tau_{N_{h}}$, where $N_{h}$ denotes the number of channel states. Then, we define the discretized channel state to be $h_{k}$ if the channel gain falls in the interval $\left[\tau_{k}, \tau_{k+1}\right)$.

Physical layer model: We consider a single-carrier single-input single-output physical layer with a fixed symbol period of $T_{s}$ seconds. The physical layer supports $M$ modulation schemes that achieve data rates $\beta^{n} / T_{s} \mathrm{bits} / \mathrm{s}$, where $\beta^{n} \in\left\{\beta_{1}, \beta_{2}, \ldots, \beta_{M}\right\}$ and $\beta_{m}$ is the number of bits per symbol used by the $m$ th modulation scheme. Therefore, to transmit $z^{n}$ packets of size $L$ bits in $\Delta T$ seconds, we must have

$$
\begin{equation*}
\beta^{n}=\left\lceil z^{n} L T_{s} / \Delta T\right\rceil \text { bits/symbol, } \tag{3.1}
\end{equation*}
$$

where $\lceil x\rceil$ denotes the ceiling operator, which rounds $x$ up to the nearest integer. In time step $n$, the transmission scheduler module in Fig. 3.1 takes as input the maximum bit-error probability $B E P^{n}$ and the desired packet throughput $z^{n}$, and then selects the modulation scheme according
to Equation (3.1).
Transmission power model: Let $P_{t x}(h, B E P, z)$ watts denote the power required to transmit $z \in \mathcal{A}_{z}$ packets in channel state $h \in \mathcal{S}_{h}$ with maximum bit-error probability $B E P \in \mathcal{A}_{B E P}$. The transmission power $P_{t x}(h, B E P, z)$ depends on the physical layer modulation scheme and is typically 1) convex increasing in the number of transmitted packets, 2 ) higher for lower bit-error probabilities, and 3) higher in worse channel states. These assumptions hold for typical modulation schemes, such as $M$-ary PSK and $M$-ary QAM [42, Table 6.1], and under information-theoretic bounds on the minimum power required for error-free communication [12]. Note that, as in [129, 100], we do not consider coding, but it can be introduced by appropriately modifying Equation (3.1) and defining $P_{t x}(h, B E P, z)$. In the rest of this chapter, we consider $M$-ary QAM for illustration; however, our learning algorithm and hardware accelerator can be modified to consider other modulation schemes and transmission power models. Under $M$-ary QAM, the transmission power can be expressed as follows [42, Table 6.1]:

$$
\begin{equation*}
P_{t x}(h, B E P, z)=\frac{\sqrt{2} N_{0}\left(2^{\beta}-1\right) \operatorname{erf}^{-1}\left(1-\frac{\beta \cdot B E P}{4}\right)}{3 h} \tag{3.2}
\end{equation*}
$$

where $N_{0}$ denotes the noise power spectral density, $\operatorname{erf}^{-1}(\cdot)$ denotes the inverse error function, and $\beta$ is the number of bits per symbol determined using Equation (3.1).

Power management model: To trade off power and delay, the wireless transmitter can be in one of two power management states, $\mathcal{S}_{x}=\{$ on, off $\}$, and can be switched "on" and "off" using one of two power management actions, $\mathcal{A}_{y}=\left\{\mathrm{s} \_\right.$on, s_off $\} .{ }^{2}$ We let $P_{\text {on }}$ and $P_{\text {off }}$ watts denote the power consumed by the wireless transmitter in the "on" and "off" states, respectively, and $P_{\mathrm{tr}}$ watts denote the power required to transition between the "on" and "off" states. We assume that $P_{\mathrm{tr}}>P_{\mathrm{on}}>P_{\mathrm{off}}>0$; therefore, there is a high cost for switching between the states, but less power is consumed in the "off" state than in the "on" state. Importantly, packets can only be transmitted if $x=$ on and $y=$ s_on; otherwise, $z=0$.

The total power cost $\rho$ incurred by taking action $a=(B E P, y, z) \in \mathcal{A}$ in channel state $h \in \mathcal{S}_{h}$ and in power management state $x \in \mathcal{S}_{x}$, can be expressed as a sum of the transmission power

[^1]and the system power: i.e.,
\[

\rho([h, x], B E P, y, x)= $$
\begin{cases}P_{\mathrm{on}}+P_{t x}(h, B E P, z), & \text { if } x=\mathrm{on}, y=\mathrm{s} \_ \text {on }, \\ P_{\mathrm{off}}, & \text { if } x=\mathrm{off}, y=\mathrm{s} \_ \text {off }, \\ P_{\mathrm{tr}}, & \text { otherwise }\end{cases}
$$
\]

As in prior work [11], we assume that the power management state $x^{n}$ evolves over time according to a discrete-time controlled Markov chain with the following transition probability function:

$$
\begin{align*}
P^{x}\left(x^{\prime} \mid x, y=\text { s_on }\right)= & \begin{array}{rc}
\text { on } & \text { off } \\
\text { off }\left(\begin{array}{cc}
1 & 0 \\
\theta & 1-\theta
\end{array}\right) \\
P^{x}\left(x^{\prime} \mid x, y=\text { s_off }\right)= & \left.\begin{array}{c}
\text { on }\left(\begin{array}{cc}
1-\theta & \theta \\
0 & \text { off } \\
0
\end{array}\right.
\end{array}\right),
\end{array}, \tag{3.3}
\end{align*}
$$

where the row and column labels represent the current power management state $x$ and the next power management state $x^{\prime}$, respectively, and $\theta \in(0,1]$ denotes the probability of a successful power management transition (from "off" to "on" or from "on" to "off"). For simplicity of exposition, we assume that the power management state transition is deterministic, i.e., $\theta=1$; however, our learning algorithm and hardware accelerator can be extended to the non-deterministic case.

Transmission buffer and traffic model: At the end of the time step $n, l^{n}$ new packets arrive into the IoT sensor's transmission buffer from the information source, where $l^{n}$ is distributed according to the packet arrival distribution $P^{l}(l) .{ }^{3}$ The buffer state evolves according to the following Lindley recursion:

$$
\begin{equation*}
b^{n+1}=\min \left(b^{n}-f^{n}\left(B E P^{n}, z^{n}\right)+l^{n}, N_{b}\right) \tag{3.5}
\end{equation*}
$$

where $N_{b}$ is the maximum number of packets that can be stored in the buffer and $f^{n}\left(B E P^{n}, z^{n}\right)$ is the packet goodput (i.e., the number of packets successfully delivered to the receiver). Note that $z^{n} \leq b^{n}$ because it is not possible to transmit more packets than are in the buffer and $f^{n}\left(B E P^{n}, z^{n}\right) \leq z^{n}$

[^2]because it is not possible to receive more packets than are transmitted. We assume that the value of $f^{n}$ is sent to the transmitter over the feedback channel at the end of time step $n$.

Assuming that bit-errors are independent, the packet loss rate (PLR) can be expressed as

$$
\begin{equation*}
P L R=1-(1-B E P)^{L} \tag{3.6}
\end{equation*}
$$

where $L$ is the packet size in bits, and the goodput $f$ has the following binomial distribution:

$$
\begin{align*}
P^{f}(f \mid B E P, z) & =\operatorname{Bin}(z, 1-P L R) \\
& =\binom{z}{f}(1-P L R)^{f}(P L R)^{z-f} \tag{3.7}
\end{align*}
$$

where $\binom{z}{f}=z!/ f!(z-f)$ !. Importantly, since packets arrive at the end of each time step, packets that arrive in time step $n$ cannot be transmitted until time step $n+1$ or later. Moreover, any packets that are not successfully delivered to the receiver in time step $n$ remain in the buffer to be retransmitted in a future time step. Based on the above discussion, the buffer state $b^{n}$ evolves over time according to a discrete-time controlled Markov chain with the following transition probability function:

$$
\begin{equation*}
P^{b}\left(b^{\prime} \mid b, B E P, z\right)=\sum_{l=0}^{\infty} \sum_{f=0}^{z} P^{f}(f \mid B E P, z) P^{l}(l) \mathbb{I}_{\left\{b^{\prime}=\min \left(b-f+l, N_{b}\right)\right\}} \tag{3.8}
\end{equation*}
$$

where $\mathbb{I}_{\{\cdot\}}$ is an indicator function that is set to 1 when the condition in $\{\cdot\}$ is true and is set to 0 otherwise.

Recall that our goal is to transmit delay-sensitive data while minimizing the IoT sensor's power consumption. We already defined the power cost in Equation (3.3). Now, we need to define the expected buffer cost, which we introduce to penalize buffer delays and overflows. Please note that here we do not put transmission delay into the consideration. The expected buffer cost incurred when transmitting $z \in \mathcal{A}_{z}$ packets with target maximum bit-error probability $B E P \in \mathcal{A}_{B E P}$ in buffer state $b \in \mathcal{S}_{b}$ can be expressed as

$$
\begin{equation*}
g(b, B E P, z)=\sum_{l=0}^{\infty} \sum_{f=0}^{z} P^{f}(f \mid B E P, z) P^{l}(l) \times\left\{[b-f]+\eta \min \left(b-f+l-N_{b}, 0\right)\right\} \tag{3.9}
\end{equation*}
$$

where the holding cost $b-f$ penalizes large buffer states, the overflow cost $\eta \min (b-f+$ $\left.l-N_{b}, 0\right)$ penalizes each packet overflow by $\eta>0$, and the expectation is taken with respect to the
packet arrival distribution $P^{l}$ and goodput distribution $P^{f}$.

### 3.2.2 Markov Decision Process Formulation

The problem described above can be formulated as a Markov decision process (MDP) with discrete and finite state space $\mathcal{S}=\mathcal{S}_{b} \times \mathcal{S}_{h} \times \mathcal{S}_{x}$ and discrete and finite action space $\mathcal{A}=\mathcal{A}_{B E P} \times$ $\mathcal{A}_{y} \times \mathcal{A}_{z}$. The state $s^{n}$ evolves over time according to a discrete-time controlled Markov chain with transition probability function

$$
\begin{equation*}
P\left(s^{\prime} \mid s, a\right)=P^{b}\left(b^{\prime} \mid b, B E P, z\right) P^{h}\left(h^{\prime} \mid h\right) P^{x}\left(x^{\prime} \mid x, y\right) \tag{3.10}
\end{equation*}
$$

and cost function defined as a weighted sum of the power and buffer costs: i.e.,

$$
\begin{equation*}
c(s, a)=\rho(s, a)+\lambda g(s, a), \tag{3.11}
\end{equation*}
$$

where $\lambda \geq 0$ can be used to set the buffer cost constraint. The goal is to determine the optimal policy $\pi: \mathcal{S} \rightarrow \mathcal{A}$, which specifies the optimal action to take in each state to minimize the average power cost subject to an average buffer cost constraint.

For a given $\lambda$, the optimal solution satisfies the following Bellman equation:

$$
\begin{equation*}
V^{*}(s)=\min _{a \in \mathcal{A}} \underbrace{\left\{c(s, a)+\gamma \sum_{s^{\prime} \in \mathcal{S}} P\left(s^{\prime} \mid s, a\right) V^{*}\left(s^{\prime}\right)\right\}}_{Q^{*}(s, a)}, \forall s \in \mathcal{S} \tag{3.12}
\end{equation*}
$$

where $V^{*}(s)$ is the optimal value function, which indicates how good it is to be in each state when following the optimal policy $\pi^{*}(s)$, and the related optimal action-value function $Q^{*}(s, a)$ indicates how good it is to take an arbitrary action in each state and then follow the optimal policy thereafter. The optimal policy $\pi^{*}(s)$ can be determined by taking the action that minimizes the right-hand side of Equation (3.12) in each state.

If the cost and transition probability functions are known, then the optimal value function can be computed numerically using dynamic programming (e.g., value iteration or policy iteration [109]) and the optimal value of $\lambda$ that satisfies the buffer cost constraint can be computed using the subgradient method. In the considered problem, however, the cost function in Equation (3.11) is only partially known because the buffer cost in Equation (3.9) depends on the unknown packet arrival
distribution $P^{l}(l)$. Moreover, the transition probability function $P\left(s^{\prime} \mid s, a\right)$ defined in Equation (3.10) is only partially known because the buffer state transition probabilities $P^{b}\left(b^{\prime} \mid b, B E P, z\right)$ defined in Equation (3.8) depend on the unknown packet arrival distribution $P^{l}(l)$, and the channel state transition probabilities $P^{h}\left(h^{\prime} \mid h\right)$ are unknown. Hence, the optimal value function and policy cannot be computed using dynamic programming; instead, they must be learned online, based on experience. Q-learning is a popular approach for this task, as described next.

### 3.2.3 Q-Learning

In each time step $n$, Q-learning updates an estimate of the action-value function based on the observed experience tuple $\left(s^{n}, a^{n}, c^{n}, s^{n+1}\right)$, which comprises the current state, selected action, incurred cost, and next state. The update is performed as follows:

$$
\begin{equation*}
Q^{n+1}\left(s^{n}, a^{n}\right) \leftarrow\left(1-\alpha^{n}\right) Q^{n}\left(s^{n}, a^{n}\right)+\alpha^{n}\left[c^{n}+\gamma \min _{a^{\prime} \in \mathcal{A}} Q^{n}\left(s^{n+1}, a^{\prime}\right)\right] \tag{3.13}
\end{equation*}
$$

where $s^{n+1} \sim P\left(\cdot \mid s^{n}, a^{n}\right)$ and $E\left[c^{n}\right]=c\left(s^{n}, a^{n}\right) ; a^{\prime}$ is the greedy action in state $s^{n+1} ; \alpha^{n} \in[0,1]$ is a time-varying step size parameter; and $Q^{0}(s, a)$ can be initialized arbitrarily $\forall(s, a) \in \mathcal{S} \times \mathcal{A}$.

In the literature, many researchers have explored various Q-learning-based RL hardware accelerator structures for better performance and lower power consumption [105, 31, 32]. However, due to the limited training data and learning time for real-time learning, these hardware optimization techniques are not, at least directly, applicable in emerging wireless IoT systems because of Qlearning's slow convergence speed. In real-time learning, training data is generated or observed over time, which means that the agent has to wait for the new data no matter how fast each iteration is. Under these circumstances, slow convergence speed means that Q-learning will spend a relatively long period of time to reach the anticipated optimization level, during which energy and time is wasted. Different from Q-learning, PDS-based methods are uniquely optimized for the underlying wireless IoT system to increase the learning convergence speed.

### 3.2.4 Deep Q-Learning

Unlike tabular Q-Learning, deep Q-learning (DQL) estimates action values with a deep Qnetwork (DQN [75]). By updating the weights of the DQN based on mini-batches of experience tuples, DQL learns successful policies directly from (possibly high-dimensional) sensory inputs and
optimizes its action selection policy to fit the unknown dynamics.
In recent studies, DQL showed great potential in IoT wireless network optimization [85, $9,38,45]$. Nevertheless, all their DQL agents run on powerful platforms like network servers, base stations, and satellites. In [86], the authors realized that deep learning was not suitable for low-power wireless applications and optimized their model, but it still required at least one hidden layer with 128 units to achieve relatively good performance, and only the inference phase could be performed on a low-power platform.

### 3.2.5 Post-Decision State Learning

Before we can describe PDS learning, we need to formally introduce the PDS concept. A PDS denotes a state of the system after all known and controllable effects of the action have occurred but before the unknown dynamics occur [73, 90, 109]. In our wireless IoT system, the PDS in time step $n$ is defined as follows:

$$
\begin{equation*}
\widetilde{s}^{n} \triangleq\left(\widetilde{b}^{n}, \widetilde{h}^{n}, \widetilde{x}^{n}\right)=\left(\left[b^{n}-f^{n}\right], h^{n}, y^{n}\right) \in \mathcal{S} \tag{3.14}
\end{equation*}
$$

where $\widetilde{b}^{n}=b^{n}-f^{n}$ denotes the buffer state after packets are successfully delivered to the receiver, but before new packets arrive; ${ }^{4} \widetilde{h}^{n}=h^{n}$ since we do not know anything about the channel state transition; and $\widetilde{x}^{n}=y^{n}$ since we assume that the power management state transition is deterministic. Given the PDS in time step $n$, we can express the state in time step $n+1$ as follows:

$$
\begin{align*}
s^{n+1} & =\left(b^{n+1}, h^{n+1}, x^{n+1}\right) \\
& =\left(\min \left(\widetilde{b}^{n}+l^{n}, N_{b}\right), h^{n+1}, \widetilde{x}^{n}\right) \tag{3.15}
\end{align*}
$$

where $l^{n} \sim P^{l}(\cdot)$ and $h^{n+1} \sim P^{h}\left(\cdot \mid \widetilde{h}^{n}\right)$ denote the realizations of the packet arrivals and next channel state, respectively.

We formulate our problem in terms of PDSs by decomposing the transition $s \rightarrow s^{\prime}$ into two parts: a known transition $s \rightarrow \widetilde{s}$ with expected cost $c_{k}(s, a)$ and transition probabilities $P_{k}(\widetilde{s} \mid s, a)$, and an unknown transition $\widetilde{s} \rightarrow s^{\prime}$ with expected cost $c_{u}(\widetilde{s})$ and transition probabilities $P_{u}\left(s^{\prime} \mid \widetilde{s}\right)$,

[^3]such that:
\[

$$
\begin{align*}
P\left(s^{\prime} \mid s, a\right) & =\sum_{\widetilde{s}} P_{k}(\widetilde{s} \mid s, a) P_{u}\left(s^{\prime} \mid \widetilde{s}\right) \text { and },  \tag{3.16}\\
c(s, a) & =c_{k}(s, a)+\sum_{\widetilde{s}} P_{k}(\widetilde{s} \mid s, a) c_{u}(\widetilde{s}) . \tag{3.17}
\end{align*}
$$
\]

Each of these factors can be easily derived based on the transition probability and cost functions defined in Equation (3.10) and Equation (3.11), respectively. For example, the unknown cost is nothing more than the expected overflow cost, i.e.,

$$
\begin{equation*}
c_{u}(\widetilde{s})=\eta \sum_{l=0}^{\infty} P^{l}(l) \min \left(\widetilde{b}+l-N_{b}, 0\right) \tag{3.18}
\end{equation*}
$$

because the arrival distribution $P^{l}$ is the only unknown component of the cost function defined in Equation (3.11).

To map traditional RL to PDS learning, we define two value functions $V(s)$ and $\widetilde{V}(\widetilde{s})$ over the conventional states and PDSs, respectively. The corresponding optimal value functions are related by the following two Bellman equations:

$$
\begin{align*}
& \widetilde{V}^{*}(\widetilde{s})=c_{u}(\widetilde{s})+\gamma \sum_{s^{\prime} \in \mathcal{S}} P_{u}\left(s^{\prime} \mid \widetilde{s}\right) V^{*}\left(s^{\prime}\right),  \tag{3.19}\\
& V^{*}(s)=\min _{a \in \mathcal{A}}\left\{c_{k}(s, a)+\sum_{\widetilde{s} \in \mathcal{S}} P_{k}(\widetilde{s} \mid s, a) \widetilde{V}^{*}(\widetilde{s})\right\} . \tag{3.20}
\end{align*}
$$

Given the PDS value function $\widetilde{V}^{*}(\widetilde{s})$, the optimal policy $\pi^{*}(s)$ can be found by taking the action in each state that minimizes the right-hand side of Equation (3.20).

To solve the problem online, we use the PDS learning algorithm presented in Algorithm 2 [70, 73]. First, the PDS value function $\widetilde{V}^{0}(\widetilde{s})$ is initialized to 0 for all $\widetilde{s} \in \mathcal{S}$ (line 1 ). In each time step $n$, PDS learning takes the greedy action defined in Equation (3.22) using the known cost function $c_{k}(s, a)$, the known transition probability function $P_{k}(\widetilde{s} \mid s, a)$, and the current estimate of the PDS value function $\widetilde{V}^{n}(\widetilde{s})$ (line 3). Subsequently, PDS learning updates the estimated PDS value function as in Equation (3.23) based on the observed experience tuple ( $\widetilde{s}^{n}, c_{u}^{n}, s^{n+1}$ ) (lines 4 and 5), where the PDS $\overparen{s}^{n} \sim P_{k}\left(\cdot \mid s^{n}, a^{n}\right)$ is defined in Equation (3.14); the realization of the unknown cost

$$
c_{u}^{n}=\eta \min \left(\widetilde{b}^{n}+l^{n}-N_{b}, 0\right)
$$

satisfies $E\left[c_{u}^{n}\right]=c_{u}\left(\widetilde{s}^{n}\right)$, where $c_{u}\left(\widetilde{s}^{n}\right)$ is defined in Equation (3.18); and the next state $s^{n+1} \sim$ $P_{u}\left(\cdot \mid \widetilde{s}^{n}\right)$ is defined in Equation (3.15). In [100], we proved that the sequence of PDS value functions $\tilde{V}^{n}$ generated by the PDS learning algorithm converges to $\tilde{V}^{*}$ with probability 1 as $n \rightarrow \infty$.

PDS learning has several advantages over Q-learning. First, only the unknown information in the transition $\widetilde{s} \rightarrow s^{\prime}$ needs to be learned. Second, by updating the value of one PDS, we learn about all state-action pairs that can precede it due to the expectation over the known transition probabilities in both Equation (3.22) and Equation (3.23). Third, in RL, there is a trade-off between exploiting actions that currently have the best estimated value and exploring other actions that might be better. However, if the unknown transition probabilities do not depend on the action (as in the considered problem), then PDS learning does not require exploration.

Together, the above three features significantly increase PDS learning's convergence speed compared to Q-learning; however, this comes at the cost of increased action selection and learning update complexity. In Q-learning, the action selection and update steps both require optimizing $Q^{n}(s, a)$ over the actions, so they have complexity $O(\mathcal{A})$. In PDS learning, in addition to optimizing over the actions, both Equation (3.22) and Equation (3.23) require calculating the action-value estimate $Q^{n}(s, a)$ for each prospective action based on the known cost and transition probability functions: ${ }^{5}$

$$
\begin{equation*}
Q^{n}(s, a)=c_{k}(s, a)+\sum_{\widetilde{s}} P_{k}(\widetilde{s} \mid s, a) \widetilde{V}(\widetilde{s}) \tag{3.21}
\end{equation*}
$$

Therefore, both steps have complexity $O(\mathcal{S} \times \mathcal{A})$. We will refer to the calculation in Equation (3.21) as the action evaluation step. In Section 3.3, we present efficient methods to calculate the known cost $c_{k}(s, a)$ and the state value expectation $\sum_{\widetilde{s}} P_{k}(\widetilde{s} \mid s, a) \widetilde{V}(\widetilde{s})$, which appear in the action evaluation step.

### 3.2.6 Stochastic Computing

To further optimize our hardware circuit, we design Transition Probability Distribution Estimator based on stochastic computing (SC). SC [39] enables complex computations to be performed using simple bit-wise operations on streams of random bits. SC has recently been exploited for various low-energy or low-area applications, such as neural networks acceleration and 5G decoding [7, 88, 44, 80]. In particular, SC is highly suitable for error-tolerant applications where approximated

[^4]```
Algorithm 2 Post-Decision State Learning
    initialize \(\tilde{V}^{0}(\widetilde{s})=0\) for all \(\widetilde{s} \in \mathcal{S}\)
    for time slot \(n=0,1,2, \ldots\) do
        Take the greedy action:
\[
\begin{equation*}
a^{n}=\underset{a \in \mathcal{A}}{\arg \min }\left\{c_{k}\left(s^{n}, a\right)+\sum_{\widetilde{s}} P_{k}\left(\widetilde{s} \mid s^{n}, a\right) \widetilde{V}^{n}(\widetilde{s})\right\} \tag{3.22}
\end{equation*}
\]
Observe PDS \(\widetilde{s}^{n}\), cost \(c_{u}^{n}\), andnext state \(s^{n+1}\). Update \(\widetilde{V}^{n+1}\left(\widetilde{S}^{n}\right)\) :
\[
\begin{equation*}
\widetilde{V}^{n+1}\left(\widetilde{s}^{n}\right)=\left(1-\alpha^{n}\right) \widetilde{V}^{n}\left(\widetilde{s}^{n}\right)+\alpha^{n}\left[c_{u}^{n}+\gamma V^{n}\left(s^{n+1}\right)\right], \tag{3.23}
\end{equation*}
\]
where
\[
V^{n}\left(s^{n+1}\right)=\min _{a \in \mathcal{A}}\left\{c_{k}\left(s^{n+1}, a\right)+\sum_{\widetilde{s}} P_{k}\left(\widetilde{s} \mid s^{n+1}, a\right) \widetilde{V}^{n}(\widetilde{s})\right\}
\]
end for
```

results are acceptable or certain errors in the intermediate stages are not perceivable by the end-used [122, 6]. Moreover, SC enables very lightweight hardware implementations for resource-constraint devices. One example of an SC circuit is shown in Fig. 3.2(a). It can be seen that stochastic multiplication can be easily realized by an AND gate on the two bit-streams, as the probability to get a ' 1 ' as the output equals to the product of the equivalent probabilities for each of the inputs. In a typical SC architecture, stochastic number generators (SNGs) and comparators are also needed to convert binary signals to stochastic representations and stochastic bit-streams back to binary signals, respectively. To this end, a linear feedback shift register (LFSR) has been widely used as the SNG to generate stochastic bit-streams, as shown in Fig. 3.2(b), while a counter can effectively perform the stochastic-to-binary conversion, as illustrated in Fig. 3.2(c). Note that the goal of adopting stochastic computing is to accelerate the hardware computation, which is qualitatively different from Bayesian-based methods.

Although SC offers simpler hardware for complex operations, it requires a long sequence of stochastic bits to obtain a precise result [7]. As a result, stochastic systems suffer from high latency or require a large number of processing elements (e.g., AND gates for multiplication) to operate on the bit-streams in parallel. Thus, it is imperative to exploit ways for reducing the length of the bit-streams while maintaining the arithmetic performance. In Section 3.3.2, we develop an SC-based accelerator to efficiently estimate the known transition probability function $P_{k}(\widetilde{s} \mid s, a)$ rather than compute it arithmetically.


Figure 3.2: Stochastic computing circuit.

### 3.3 Proposed Hardware Architecture

To address the high computational complexity of PDS learning, we design an optimized hardware accelerator framework for the critical action evaluation (AE) step in Equation (3.21). As noted earlier, this step is performed once for each prospective action in both the action selection step (Equation (3.22)) and the learning update step (Equation (3.23)). For our accelerator framework, it consists of two main components: the Known Cost (KC) block for computing $c_{k}(s, a)$ and the State Value Expectation (SVE) block for computing $\sum_{\widetilde{s}} P_{k}(\widetilde{s} \mid s, a) \widetilde{V}(\widetilde{s})$. To realize a hardware accelerator for a specific system, we design the programmable lookup table (PLUT) (green) with state encoding (wathet), transition probability distribution estimator (TPDE) (grey), state value array (orange), and tree structure (blue), according to the unique characteristics of both the system and the PDSbased RL algorithm.

For illustration, in the remainder of this chapter, we consider an instance of the example system model in Section 3.2 .1 with 26 buffer states $\left(b \in \mathcal{S}_{b}=\{0,1, \ldots, 25\}\right.$ packets), 8 channel states $\left(h \in \mathcal{S}_{h}=\{-18.82,-13.79,-11.23,-9.37,-7.80,-6.30,-4.98,-2.08\} \mathrm{dB}\right), 2$ power management
states $\left(x \in \mathcal{S}_{x}=\{\mathrm{ON}, \mathrm{OFF}\}\right), 2$ power management actions $\left(y \in \mathcal{A}_{y}=\{\right.$ SWITCH_ON, SWITCH_OFF $\left.\}\right)$, 5 target BEPs $\left(B E P \in \mathcal{A}_{B E P}\right.$ yielding PLRs of $0.01,0.02,0.04,0.08$, and 0.16 for packets of size $L=5000$ bits), and 11 transmission scheduling actions ( $z \in \mathcal{S}_{z}=\{0,1, \ldots, 10\}$ ). Therefore, there are a total of 416 system states and 110 possible actions. Although we consider this specific parameter configuration, the PDS learning algorithm and hardware acceleration architectures can be applied for any values of these parameters.

Fig. 3.3(a) illustrates an instance of the hardware accelerator design for the example system model in Section 3.2.1, which is extended from our prior work [107]. Recall that we do not have complete information about our model because we do not know the data arrival probability distribution $P^{l}(l)$ or the channel state transition probabilities $P^{h}\left(h^{\prime} \mid h\right)$. We briefly introduce the circuit functions below, while detailed circuit designing can be found in Sections ??, 3.3.1, and 3.3.2.

The bottom KC block in Fig. 3.3(a) calculates the known buffer cost and transmission cost, and then combines them to calculate the known components of Equation (3.11). The known buffer cost only includes the known components of Equation (3.9), which do not depend on $P^{l}(l)$, i.e.,

$$
\begin{equation*}
g_{k}(s, a)=\sum_{f=0}^{z} P^{f}(z \mid B E P, z)[b-f] \tag{3.24}
\end{equation*}
$$

and is computed with an arithmetic circuit. For the transmission cost, the dominant part is the computation of $P_{t x}$ defined in Equation (3.2), where we implement two lookup tables to simplify the calculation. By multiplying $P_{t x}$ by $h, P_{t x} * h$ lookup cancels the existence of $h$ and stores the results for all the combinations of $B E P \mathrm{~s}$ and $z \mathrm{~s}$. Then, with another lookup table outputting values for $1 / h, P_{t x}$ is calculated with very minimal cost.

The top block in Fig. 3.3(a) computes the SVE as:

$$
\begin{equation*}
\sum_{\widetilde{s} \in \mathcal{S}} P_{k}(\widetilde{s} \mid s, a) \widetilde{V}(\widetilde{s})=\sum_{\widetilde{x} \in \mathcal{S}_{x}} \sum_{f=0}^{z} P^{x}(\widetilde{x} \mid x, y) P^{f}(f \mid B E P, z) \widetilde{V}(b-f, \widetilde{x}, h) \tag{3.25}
\end{equation*}
$$

where $P^{f}$ is the goodput distribution defined in Equation (3.7). The SVE block includes the following components:

- The BEP Lookup block takes as input the BEP's address and outputs both $P L R$ and $1-P L R$, where $P L R$ is defined in Equation (3.6).
- The Power Tree block takes as input $p=P L R$ and $q=1-P L R$ and outputs $p^{0}, p^{1}, \ldots, p^{10}$
and $q^{0}, q^{1}, \ldots, q^{10}$, which are used to calculate the goodput distribution in Equation (3.7).
- The Choose Lookup block takes as input the transmission action $z$ and outputs the values $c(f)=\binom{z}{f}$ when $f \leq z$ and $c(f)=0$ when $f>z$, for $f=0,1, \ldots, 10$. The combinations $c(f)$ are also used to calculate the goodput distribution in Equation (3.7).
- The State Value Selection block takes as input the current state $S$ and all state values, then outputs the state values for possible PDSs.
- Finally, the Multi-Sum Tree block takes as input the outputs of the State Value Selection, Choose Lookup, and Power Tree blocks, and outputs the SVE.

More details about the Power Tree and State Value Selection blocks are provided in Chapter 2.
Fig. 3.3(b) illustrates the proposed novel alternative SC-based SVE module, which we describe further in Section 3.3.2.

### 3.3.1 Programmable Lookup Table with State Encoding for RL

The channel state in the PDS learning algorithm is quantized into discrete state values. Since the number of states is typically limited to simplify the learning process and save energy in IoT applications, we implement lookup tables for the input stages to further accelerate the computation. For a directly implementation, there will be $2^{32}$ possible input values (for a 32 -bit system) from the channel sensor, which corresponds to a 'costly' 32-bit lookup table. However, since many input cases share the same output and there are only eight channel fading states $h$ in our model, we introduce state encoding (SE) to compress the input space of the lookup table. It encodes the input values into successive binary state addresses to compress the input bit-width, as illustrated in Fig. 3.4, where a 3 -bit input is mapped into two states with ' 100 ' as the boundary. With state encoding applied, its input width is compressed by 3 (from 3 bits to 1 bit). Additionally, in order to adapt the same IoT circuit to various environments and use cases, the lookup table and state encoding are designed to be programmable with a memory module controlled by a SCM (single chip micro-controller). The functionality and state encoding of the lookup table are defined by the corresponding values from memory, which can be modified by the SCM, as shown in Fig. 3.5.

The circuit design for state encoding is shown in Fig. 3.6. Each block illustrates the basic SE unit, where port in takes the input value of the lookup table and std indicates the boundary

(b) Alternative SVE module with TPDE.

Figure 3.3: Action evaluation hardware accelerator designs for the example system model. The SVE block is illustrated in Fig. 3.3(a) assuming that up to 10 packets can be transmitted in each time step, i.e., $\mathcal{S}_{z}=\{1,2, \ldots, 10\}$.
value between the neighboring states that can be defined by the memory. The SE unit will compare in with std and then set one of the 1 -bit outputs large or small to ' 1 ' and another to ' 0 '. Besides, when en is ' 0 ', both large and small will be set to ' 0 ', which can be simply implemented by logical AND operations.

By connecting multiple SE unit blocks as a binary tree structure and making all ins share the same input value as the input of the lookup table, we can easily obtain a programmable state encoding circuit for arbitrary state numbers. A four-state circuit design is demonstrated in Fig. 3.6,

| Boundary 100 |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Case | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| Out | State0 | State0 | State0 | State0 | State1 | State1 | State1 | State1 |
| State Encoding |  |  |  |  |  |  |  |  |
|  |  | Case |  | 0 | 1 |  |  |  |
|  |  |  | Out | State0 | State1 |  |  |  |

Figure 3.4: An example of state encoding where the input bit-width is compressed from 3 to 1.


Figure 3.5: Programmable lookup table with memory and SCM.
which has the function:

$$
\text { State }= \begin{cases}0, & \text { if } \text { in } \in(0, b 1]  \tag{3.2}\\ 1, & \text { if } \text { in } \in[b 1, b 0) \\ 2, & \text { if } \text { in } \in[b 0, b 2) \\ 3, & \text { if } \text { in } \in[b 2,+\infty)\end{cases}
$$

The circuit for our lookup table is designed based on the SE unit, as shown in Fig. 3.7. $S_{0}$ to $S_{n-1}$ are outputs of the state encoding circuit that correspond to $n$ states. Then the desired value can be quickly selected using AND gates, where $D_{0}$ to $D_{n-1}$ are the corresponding output values from memory. To reconfigure the function for different use cases, we only need to update the boundary values and output values in the memory.


Figure 3.6: The logic circuit of state encoding (SE) module (4 states example).

### 3.3.2 Transition Probability Distribution Estimator and Stochastic Sample Generator

Transition Probability Distribution Estimator (TPDE): The TPDE estimates the distribution of the PDS based on the current state and action as $P_{k}(\widetilde{s} \mid s, a)$, where $\widetilde{s}$ denotes the PDS and $s, a$ are the current state and action, respectively. In PDS RL, this distribution is crucial as it needs to be computed at least two times in each time step (once for action selection and once for the learning update). However, calculating the entire transition probability distribution can be computationally expensive. For example, the transition probability distribution from the buffer state $b$ to the post-decision buffer state $\widetilde{b}=b-f$ depends on the goodput distribution $P^{f}$ defined in Equation (3.7).

It can be seen that costly operations, including multiplications and powers, are involved in Equation (3.7), which are not suitable for resource-constraint IoT systems. To tackle this challenge, we design a novel SC-based TPDE that can significantly reduce complexity while lowering power consumption. Based on the Monte Carlo sampling method, which is widely adopted for estimating expectations, in order to get:

$$
\begin{equation*}
E[f(x)]=\sum_{x} f(x) p(x), \tag{3.27}
\end{equation*}
$$



Figure 3.7: Lookup table with state encoding.
we can sample $L$ data points $\left\{x^{1}, \ldots, x^{L}\right\}$ then establish an unbiased estimator for $E[f(x)]$ :

$$
\begin{equation*}
\hat{f}=\frac{1}{L} \sum_{i=1}^{L} f\left(x^{i}\right) \tag{3.28}
\end{equation*}
$$

The variance can be given by $\operatorname{var}(\hat{f})=\frac{1}{L} E\left[(f-E[f])^{2}\right]$, which indicates that the estimation accuracy improves with the sample size $L$. The goal of the TPDE is to estimate the transition probability distribution:

$$
\begin{equation*}
P\left(S_{i} \mid S, A\right)=\sum_{S^{\prime}} f\left(S_{i}, S^{\prime}\right) P\left(S^{\prime} \mid S, A\right) \tag{3.29}
\end{equation*}
$$

where $S_{i}$ is one specific case of the next state, $f\left(S_{i}, S^{\prime}\right)=1$ when $S^{\prime}=S_{i}$ and 0 when $S^{\prime} \neq S_{i}$. By gathering $L$ samples $S_{1}^{\prime}, \ldots, S_{L}^{\prime}$ for the PDSs from distribution $P\left(S^{\prime} \mid S, A\right)$, based on Equation (3.27) and Equation (3.28), we can obtain $\hat{P}\left(S_{i} \mid S, A\right)$ as the unbiased estimator for $P\left(S_{i} \mid S, A\right)$, which is expressed as:

$$
\begin{equation*}
\hat{P}\left(S_{i} \mid S, A\right)=\frac{1}{L} \sum_{j=1}^{L} f\left(S_{i}, S_{j}^{\prime}\right) \tag{3.30}
\end{equation*}
$$

Thus, based on Equation (3.29) and Equation (3.30), we construct a TPDE with a sample generator $\left(P\left(S^{\prime} \mid S, A\right)\right)$ and a discriminator $\left(f\left(S_{i}, S^{\prime}\right)\right)$.

Stochastic Sample Generator (SSG): To obtain an accurate estimation for the transition probability distribution, it is also crucial to design a sample generator that can generate samples based on the specific distribution. The design of our stochastic sample generator is shown in Fig. 5.2, which consists of three main structures: stochastic number generator (SNG), distribution tuner,
and accumulative discriminator array.


Figure 3.8: The framework of the stochastic sample generator.

We use the same SNG as in most prior stochastic computing designs, which is composed of LFSRs and a comparator that can generate a random bit-stream with a probability of $P$ to be 1. After the SNG, the distribution tuner turns the bit-stream into samples based on the target distribution. For example, the tuner directly outputs each $n$ bits as one sample for the binomial distribution in our PDS learning algorithm:

$$
\begin{equation*}
S_{i} \sim \operatorname{Bin}(P, n) \tag{3.31}
\end{equation*}
$$

It is shown in prior works $[25,106]$ that the binomial distribution can be used to fit many other common distributions, such as Poisson distribution with $\lambda=n P$ and standard distribution with $\mu=n P$ and $\sigma^{2}=n P(1-P)$. It is also possible to design a tuner for a logically descriptive distribution, similar to the distribution on the check node of the LDPC decoding [118].

Finally, the accumulative discriminator array will gather all the samples. Each discriminator will count the number of samples $N_{i}$ that belong to the specific state $S_{i}$. The output of the $S_{i}$ discriminator is an estimate of $L * P\left(S_{i}\right)$, i.e.,

$$
\begin{equation*}
P\left(S_{i}\right) \approx \frac{N_{i}}{L} \tag{3.32}
\end{equation*}
$$

Although a larger $L$ will increase the accuracy of this estimation, we find that the PDS learning method implies remarkable tolerance to the random error, which means a small $L$ can be
adopted for acceleration and energy saving. This property is further discussed in Section 3.4.2.
TPDE Circuit Design for Binomial Distribution: The circuit design of the TPDE for the binomial distribution family $(\operatorname{Bin}(n, p))$ is shown in Fig. 3.9, where the controlled counter is implemented as a distribution tuner. It takes a stochastic bit-stream and the throughput $z$ (corresponding to the $n$ of the binomial distribution), generates one sample for each $z$-bit, and informs the accumulative discriminator array when one sample is ready. For the accumulative discriminator array, each discriminator will count the number of received samples that belong to its state.


Figure 3.9: TPDE for the binomial distribution family.

### 3.3.3 Programmable Parallel Greedy Action

In PDS-based RL, the AE step defined in Equation (3.21) must be performed twice for every action in each time step (i.e., once for every action during the action selection step and once for every action during the learning update step). This presents challenges to wide applicability since the length of one time step can be small due to the high communication frequency, which brings the requirement of high-speed computation. On the other hand, in scenarios such as smart homes, saving energy becomes more important. Therefore, programmability is desired to enable a trade-off between speed and power consumption for different applications. A 4-way example of the proposed programmable parallel structure is shown in Fig. 3.10. Here AE represents the action evaluation module as described above. MC is the minimum comparator module that takes two numbers as input and compares them, then outputs the smaller one. By connecting the MC module in series, we can then realize the arg min function. With the MUX gate at the output node, the parallelism can be configured by the control signals.


Figure 3.10: Programmable 4-way action evaluation structure.

### 3.4 Experimental Results

### 3.4.1 Experiments Setup

For software simulation, all algorithms are coded and tested with MATLAB on Windows 11, with a 3.80 GHz i7-10700K processor and 32GB RAM. As wireless IoT systems usually have fewer computing resources, we consider this setting as a guaranteed upper bound for the software implementation's speed. For hardware testing, we implement our circuits with Verilog HDL, and then map them into a 32 nm technology node using Synopsys Design Compiler. All simulations are conducted using the state and action sets defined at the beginning of Section 3.3 and with packet size $L=5000$ bits.

### 3.4.2 Algorithmic Performance

Fig. 3.11 compares the simulated performance of our PDS learning implementation (Algorithm 2), Q-learning, and DQL. DQL is implemented with MATLAB's deep RL toolbox. We examine two architectures with one and two fully connected hidden layers. The activation function is ReLU. The feature input layer for our model inputs current state, $\left(b^{n}, h^{n}, x^{n}\right)$, to DQL and applies data normalization. And the output layer is designed with the same size as the action space $\mathcal{A}$, so
that each output corresponds to one possible action. In order to minimize the cost function, the reward for each action selection is defined as $-c(s, a)$. Consistent with the network size of a recent study [86] on low power wireless applications and the output layer's size for our model (110), we set the output size for each fully connected layer to be 128 . The learning step size for DQL is $1 \times 10^{-3}$. All results are averaged over at least 75,000 time slots. It can be seen from Fig. 3.11 that our PDS learning algorithm outperforms Q-learning and DQL in terms of both cumulative average delay and power consumption. Moreover, we find that DQL with one hidden layer (marked as 'DQL 1 * 128 ') performs much worse than DQL with two hidden layers (marked as 'DQL $2^{*} 128$ '), which further proves that DQL requires a relatively complex network in order to achieve acceptable performance.


Figure 3.11: Comparison between PDS learning, Q-learning, and deep Q-learning.

We also evaluate the convergence speed of our algorithm in Fig. 3.11(c) with $3 \times 10^{6}$ time slots. The red curve (circle markers) denotes the cumulative average cost incurred up to time slot $n$ by Q-learning (where the cost is defined in Equation (3.11) as a weighted sum of the power cost and delay cost, which makes it the best representative of the overall performance) and the blue curve (+ markers) denotes the cumulative average cost for PDS learning. While PDS learning approximately converges in 250,000 time slots, Q-learning has still not converged after 3,000,000 time slots, and hence is at least 12 times slower than PDS learning.

We now evaluate the algorithmic performance when using the TPDE. As discussed in Section 3.3.2, the randomness introduced by the TPDE is highly dependent on the sample number $L$. By decreasing the sample number for each estimation, the delay and energy consumption of the TPDE can be reduced. However, the convergence of the learning algorithm may suffer from the estimator's high variance. To study the impact of this randomness on the learning process of our PDS model and to select the best sample number for the hardware test, we also evaluate the arithmetic performance of the SSG model. The same learning simulation processes are executed for sample numbers per estimation of a single PDS of $1,10,100,1000$, and 10000 . The results are shown as Fig. 3.12, which show that all learning processes with different sample numbers converge similarly. Please note that the differences between each curve are caused by the combination of the stochastic channel model, stochastic arrivals, and randomness from the TPDE. We further repeat the simulation of the learning process five times with only a single sample per estimation and compare the results with arithmetic PDS learning in Fig. 3.13, where we print the best and worst cumulative average cost among all five learning episodes for each time slot. It can be seen that all the learning curves have similar convergence speeds. Thus, we conclude that PDS learning is very resilient to the randomness introduced by stochastic computing, which can be leveraged to optimize the hardware cost by using a single sample without sacrificing the arithmetic performance.


Figure 3.12: Effect of stochastic process from SSG.


Figure 3.13: Convergence for a single sample.

### 3.4.3 Fault Tolerance

Fault tolerance is another advantage of stochastic computing, which indeed is also a desired characteristic for wireless IoT systems under noisy and low-energy environments. Many studies have shown that bit-flip errors are very common in those environments [91], while SC is inherently resilient to these soft transient errors [77,51, 92]. Based on that, we verify the error-tolerance of our proposed method in Fig. 3.14, where we randomly flip the bits of all the outputs from multipliers in the power tree and multi-sum tree based on the error rate. The results show that our PDS learning accelerator achieves a high degree of error tolerance as all learning processes converge similarly.


Figure 3.14: Error-tolerance.

### 3.4.4 Hardware Performance

We implement our proposed efficient architecture, a straightforward baseline design without employing the proposed optimization, and Q-learning using Verilog HDL. For a fair comparison, all common intrinsic variables and state values $V(s)$ use a bit-width of 32 .

We evaluate and compare the execution delays and average runtime for our two hardware designs and the software implementation of PDS learning. The power and area consumption of the arithmetic hardware accelerator and the baseline design is also compared to illustrate the effectiveness of the proposed hardware optimization techniques. These results and comparisons are shown in Table 3.1, where the execution times and power/area consumption are normalized with respect to those of the arithmetic hardware design. It can be observed that our arithmetic hardware accelerator is $2.6 \times$ faster than the baseline circuit while achieving a $1 \times 10^{4}$ times acceleration over the software implementation. Besides, the power and area consumptions are also decreased by $85.7 \%$ and $86.1 \%$, respectively, compared to the baseline hardware design.

We use Synopsis IC compiler to generate the layout of the arithmetic hardware design with 32 nm technology, as shown in Fig. 3.15, where the post-layout area (not \# of cells) and power are $0.38 \mathrm{~mm}^{2}$ and 5.72 mW , respectively.

Table 3.1: Arithmetic vs. Baseline Hardware vs. Q-Learning (32-Bit)

|  | Arithmetic <br> Hardware <br> $($ PDS $)$ | Baseline <br> Hardware <br> $(P D S)$ | Normalized <br> Q-learning | Software |
| :--- | :--- | :--- | :--- | :--- |
| Delay $(n s)$ | 98.76 | 258.31 | $521.9(5.3 \times)$ | $(2.6 \times)$ <br> Power $(m W)$ |
| Area $(\#$ of cells $)$ | 5.87 | $41.21(7 \times)$ <br> 666543 <br> $(7.2 \times)$ | $15(2.6 \times)$ | - |

The implementation of Q-learning is based on Equation (3.13). According to the simulation results in Section 3.4.2, Q-learning converges over an order of magnitude slower than PDS-based learning. We normalize the hardware cost with respect to the convergence time for a fair comparison. These results show that even though Q-learning costs less for a single iteration compared to PDS learning, when considering the convergence time, the proposed PDS-based learning accelerator yields reductions of $81 \%$ and $61 \%$ in delay and power consumption, respectively, compared to Q-learning. Therefore, we can conclude that the proposed PDS learning architecture achieves much superior


Figure 3.15: The layout of the arithmetic hardware design.
hardware performance than Q-learning.

### 3.4.5 TPDE vs. Arithmetic Circuit

From the experimental results, we find that the delay of the Know Cost module is only $39.8 \%$ of the SVE module and the SVE module's delay takes $100 \%$ of the total delay (which means it is the critical path of the accelerator), indicating that the optimization for the SVE module is more crucial for speeding up the overall accelerator. This further confirms the motivation to adopt stochastic computing (i.e., TPDE) in the proposed architecture.

For a fair comparison, we implement TPDE and the corresponding circuit from the arithmetic accelerator (Fig. 3.16) that performs the same function as the TPDE. Here the corresponding circuit is the SVE module without the state value selection module (as it is not included in the criti-
cal path) or adders at the output stage that perform the sum function. Both circuits are individually implemented under the same 32-bit input setting. The comparison of the arithmetic hardware architecture in our prior work [107] and the proposed TPDE is summarized in Table 3.2, where the time per result for TPDE is defined by $\frac{z \times \text { SampleNumber }}{\text { ClkFreq }}(z \in[1,10])$. We set the sample number for one estimation as 1. It can be seen that the TPDE is $86.7 \%$ faster while consuming only $0.74 \%$ energy compared to the optimized arithmetic hardware architecture even with the largest packet throughput $z$.


Figure 3.16: Replaced circuit from the arithmetic accelerator.

From the results, we can see that the TPDE significantly reduces the energy consumption and circuit area as most stochastic circuits do. Besides that, the TPDE is 8.3 x faster compared to the corresponding arithmetic circuit that executes the same function thanks to the resiliency of the PDS learning algorithm to the stochastic errors as shown in Fig. 3.12.

Table 3.2: Comparison with our prior work [107] (32-Bit)

|  | Arithmetic <br> Hardware <br> $(\mathrm{PDS})$ | TPDE |
| :--- | :--- | :--- |
| Delay $(n s)$ | 75.54 | 0.79 |
| Power $(u W)$ | 3695 | 206 |
| Area | 132134 | 1095 |
| Clk Freq $(\mathrm{MHz})$ | 12 | 1000 |
| Latency | $83 n s$ | $1-10 n s$ |
| Power-delay Product | $1 \times$ | $.00067-.0067 \times$ |

### 3.4.6 Programmable Parallel Greedy Action

To adapt our learning accelerator to broader application scenarios, we introduce programmable parallel greedy action in Section 3.3.3. The comparison of non-parallel and 4-way parallel AE (Fig. 3.10) is shown in Table 3.3. In the worst-case (i.e., all four paths are activated), the additional MC modules and 4-to-1 MUX only incur an additional delay of 3.69 ns and 0.17 mW extra power consumption, which correspond to only $3.7 \%$ and $2.9 \%$ overhead, respectively.

Table 3.3: 4-way parallel AE (32-Bit)

|  | Non-Parallel | 4-Way Parallel |
| :--- | :--- | :--- |
| Delay $(n s)$ | 98.76 | 102.45 |
| Power $(m W)$ | 5.87 | $6.04 * 4$ |

### 3.5 Summary

This chapter presented efficient hardware architectures for accelerating PDS learning in IoT applications. We first designed a hardware accelerator for the most costly computation, i.e., the action evaluation step in Chapter ??. Then, built upon this architecture, we developed a SC-based hardware architecture, which can further simplify the computation while simultaneously reducing the power consumption. The effectiveness of the proposed methods is comprehensively verified from both arithmetic and hardware perspectives. Future work will be directed towards the generalization of the proposed architecture to various wireless and IoT settings.

## Chapter 4

## Efficient Data Extraction Circuit for Posit Number System: LDD-based Posit Decoder

### 4.1 Motivation

Since the universal number was proposed, IEEE 754 Standard floating-point format [1] has become one of the most commonly used number formats. In searching for higher accuracy and dynamic range to better serve modern applications, [47] designed posit number in 2017, a drop-in replacement for IEEE 754, as claimed by the developers.

With the same bit size as floating-point, posit number offers a more flexible trade-off than floating-point between decimal accuracy and dynamic range. Compared with floating-point, posit shows many advantages such as larger dynamic range, higher accuracy, better closure, and overflow resistance. Besides, [24] found that posit can save the hardware cost such that an $n$-bit IEEE 7542008 adder and multiplier can be safely replaced by an $m$-bit Posit Arithmetic Units adder and multiplier where $m<n$. In addition, posit number achieves superior performance in computing some special functions. For example, it only requires simple bit shifting and flipping to estimate the value of the sigmoid function $\left(1 /\left(1+e^{-x}\right)\right)$ with posit number.

Recent works have been exploring its applications by leveraging the advantages of posit
numbers. For instance, $[57,24,119]$ designed ASIC architectures for posit arithmetic core generator, [82, 56, 52] exploited the implementations of posit system on FPGA, [79] applied approximate computing to the posit system, [126] designed efficient multiplier for posit number, and [29, 17, 78, 61] adapted posit number system to deep neural networks (DNN). For instance, one of the biggest challenges for DNN is the DRAM capacity and speed limits due to its massive trainable parameters [89, 66]. Alleviating the challenge, techniques like low-precision arithmetic [124,50] are studied to lessen the data size. Enlightened by this approach, researchers found posit number a great fit to neural network applications due to its high dynamic range [68], which means the users can either have higher dynamic range with the same number size, or similar dynamic range with smaller number size, compared to the floating-point.

A posit number is composed by four parts: sign bit $(s)$, regime bits $(r)$, exponent bits $(e)$, and fraction bits $(f)$, as shown in Fig. 4.1. The size of regime bits varies, which can even take over the space of fraction bits and exponent bits for different number values. This key property yields the trade-off between decimal accuracy and dynamic range. However, it requires an extra decoding/data extraction process to obtain the sizes and values for each component before arithmetic calculation.


Figure 4.1: Generic posit format for finite, nonzero values.

To perform the decoding process for posit, the state-of-the-art posit decoder designs [59, 57, $24,119,36]$ are based on hardware structures named leading one detector (LOD) or/and leading zero detector (LZD) [3] (some papers call them leading one/zero counter), whose function is to detect the size of the regime bits. After regime size is obtained, the decoder then 'flush out' the specific values for all parts and get them ready for the subsequent arithmetic calculations. However, we find that this design does not fully utilize the hardware when encoding the regime's size into a binary number and decoding it for bit shifting, and the implied redundancy introduces extra delay and power consumption. In this paper, to address this weakness, we design a novel circuit structure, leading difference detector (LDD). Then we implement a posit number decoder based on the LDD. Our experimental results show that the proposed LDD-based posit decoder can reduce the delay
and energy consumption by about $60 \%$ and $50 \%$, respectively, compared to the conventional LOD decoder for 16 -bit, 32 -bit, and 64 -bit posit numbers.

The rest of the paper is organized as follows: Section 4.2 reviews the basic principle of the posit number system, the current decoding methodology, and the corresponding circuit design. Then, our proposed efficient LDD-based posit number decoder is presented in Section 4.3. In Section 4.5, we present the experimental results to verify the advantages of our design. Finally, Section 4.6 concludes this paper.

### 4.2 Background

### 4.2.1 Posit Number System

The universal number (unum) has several types. The "type I" unum is a superset of IEEE 754 Standard floating-point format, which is widely used today, but it requires extra management to activate variable length. Unlike the "type I" unum that is used for expressing interval arithmetic, the "type II" unum is designed based on the projective reals, which means it becomes a pointer to the values instead of the value itself. Although having many ideal mathematical properties, the "type II" unum has exaggerated hardware cost since it requires a bigger lookup table for most operations [46]. As a representative of the "type III" unum, posit number system is designed to create a hardware-friendly version of the "type II" unum.

As shown in Fig. 4.1, a posit number is composed by: sign bit (s), regime bits ( $r$ ), exponent bits $(e)$, and fraction bits $(f)$, together with two pre-known parameters: number size $(N)$ and exponent size (es).

The highest bit will always be the sign bit, where ' 0 ' stands positive and ' 1 ' stands negative. When negative, we need to take the 2's complement before decoding the rest parts. The very next part is the regime bits. To decode it, we need to count the number of consecutive 0 s or 1 s after the sign bit, and the last bit of regime bits will be the first different bit. For $m$ consecutive 0 s, regime $r=-m$, while for $m$ consecutive 1 s , regime $r=m-1$. If all the bits except the sign bit are the same, they will all be counted as $m$. One 4 -bit decoding example is shown in Table 4.1.

After the regime bits, the very next es bits will be $e$. If there are not enough bits left, $e$ equals the remaining bits or just 0 if no bit is left. After decoding all the parts mentioned above, the rest of the bits are all $f$, and $f=0$ when there is no bit left. With all the extracted data, the

Table 4.1: Regime Bits Decoding Example

| Regime Bits | 000 | $001 X$ | $01 X X$ | $10 X X$ | $110 X$ | 111 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $r$ | -3 | -2 | -1 | 0 | 1 | 2 |

value of a posit number can be expressed as:

$$
\begin{equation*}
\left(2^{2^{e s}}\right)^{r} \times 2^{e} \times 1 . f \tag{4.1}
\end{equation*}
$$

Due to the variable bit sizes for each component of a posit number, an extra data extraction/decoding process is necessary to perform arithmetic operations.

### 4.2.2 Leading One/Zero Detector

To decode a posit number and perform data extraction, LOD and LZD are employed by the state-of-the-art studies [59,57, 24, 119, 36]. Those hardware structures detect and output the location for the first $0 / 1$ for a binary number. The circuit design for fast LOD used by, to the best of our knowledge, all the recent studies, is shown in Fig. 4.2. A LOD/LZD has two outputs, $K=(i-1)$ indicates the first $0 / 1$ occurs at the $i-t h$ bit (counting from left), and $V l d=0$ when no $0 / 1$ is detected. For example, an LOD will output $K=101$ to indicate that the first 1 occurs at $6 t h$ bit when the input number starts with ' $000001 \ldots$ '. Then, based on the outputs of LOD, the posit decoder can obtain the value for regime bits and flush out the rest parts of the posit number with a shifter. Since the posit's decoding process typically finishes within one clock cycle, the 'shifter' here is actually a selector, which selects the correct output from all possible shifting results that are pre-defined. As LOD and LZD have a similar circuit design and current posit decoders only use one of those combined with inverters to handle all the input patterns (as shown in lines 7 and 8 of Algorithm 3), we implement an LOD-based decoder as the baseline comparison in this paper. The detailed decoding process with LOD can be found in Algorithm 3.

Although this design is intuitive, there are several places that we can further optimize in the hardware implementation. During the decoding process of the example we mentioned above, the LOD encodes the first 1's location into a binary number ' 101 ', then the 'shifter' decodes this number and makes the selection. Such redundancy introduced by the encoding and decoding of binary numbers will consume extra power and circuit area.


Figure 4.2: Circuit design for LOD.

### 4.3 LDD-based Posit Decoder

### 4.3.1 Leading Difference Detector

In this section, we present the design of a novel posit decoding circuit based on a leading difference detector (LDD), which eliminates the redundant binary decoding process of the conventional decoder.

The decoding process with LDD is shown in Algorithm 4. In essence, the LDD generates a binary indicator ' $L D D$ ' instead of a binary number based on the location of the first different bit. This indicator has the property that its $(i-1)-t h$ bit will be ' 1 ' and the rest will be ' 0 ' if the input's first difference occurs at the $i-t h$ bit. An example is shown as Fig. 4.3, where the output of LDD ‘ 00010000000 ' indicates that the first difference occurs at the 5 th bit. Please note that the output size of LDD will be 1 bit smaller than its input size, as the difference will never occur at the very first bit. Then, based on the obtained value of $L D D$, the corresponding output for each component will be generated by a customized selection circuit.

For a better illustration, we provide an example of the circuit design with a 4 -bit input in Fig. 4.4. There are 3 stages in the LDD circuit:

- The 'dif' stage (Fig. 4.4(a)) checks the differences for all adjacent bits in the way that dif[i]= ' 0 ' when in $[i+1] \neq i n[i]$ (Algorithm 4, line 6 ).
- The 'en' stage (Fig. 4.4(b)) implements a priority arbiter [16] to examine the existence of the

```
Algorithm 3 Posit Data Extraction with LOD
    Input: \(I N[N-1: 0]\)
    Outputs:
        \(\operatorname{Sign}(s), \operatorname{Regime}(r)\), Exponent \((e)\), Fraction \((f), \quad\) Zero(z), Infinity \((i n f)\)
    Known: Input_Size \((N)\), Exponent_Size(ES)
    \(z \leftarrow \operatorname{NOR}(I N[N-1: 0])\)
    \(\inf \leftarrow I N[N-1] \&(\mathbf{N O R}(I N[N-2: 0]))\)
    \(X I N \leftarrow I N[N-1] ?(\sim I N[N-2: 0]+1): I N[N-2: 0]\) (Take 2's complement if \(I N[N-1]=1\) )
    \(L I N \leftarrow X I N[N-2] ?(\sim X I N[N-2: 0]): X I N[N-2: 0]\)
    \(K \leftarrow\) Leading One Detector \((L I N)\)
    \(r \leftarrow X I N[N-2] ?(K-1): \sim(K-1)\)
    temp \(\leftarrow X I N \ll(K+1)\)
    if \(N-K-2>E S\) then
        \(e \leftarrow\) Highest \(E S\) bits of \(t e m p\)
    else
        \(e \leftarrow\) Highest \((N-K-2)\) bits of temp
    end if
    \(f \leftarrow\) temp \(\ll E S\)
```

```
Algorithm 4 Posit Data Extraction with LDD
    Input: \(I N[N-1: 0]\)
    Outputs:
        \(\operatorname{Sign}(s), \operatorname{Regime}(r), \operatorname{Exponent}(e), \operatorname{Fraction}(f), \quad \operatorname{AllZero}(z), \operatorname{AllOne}(o)\)
    Known: Input_Size( \(N\) ), Exponent_Size(ES)
    \(X I N \leftarrow I N[N-1] ?(\sim I N[N-2: 0]+1): I N[N-2: 0]\) (Take 2's complement if \(I N[N-1]=1)\)
    for \(i=0:(N-3)\) do
        \(d i f[i] \leftarrow X I N[i] \odot X I N[i+1]\)
    end for
    for \(i=0:(N-4)\) do
        \(e n[i] \leftarrow \mathbf{A N D}(\operatorname{dif}[(N-3): i])\)
    end for
    \(L D D[N-3] \leftarrow \sim \operatorname{dif}[N-3]\)
    \(L D D[N-4] \leftarrow \operatorname{dif}[N-3] \& \sim \operatorname{dif}[N-4]\)
    for \(i=0:(N-5)\) do
        \(L D D[i] \leftarrow \sim \operatorname{dif}[i] \& e n[i+1]\)
    end for
    \(z \leftarrow X I N[N-2] \& e n[0]\)
    \(o \leftarrow \sim X I N[N-2] \& e n[0]\)
    \(s \leftarrow I N[N-1]\)
    \(r, e, f \leftarrow\) Corresponding values from NAND selection
        arrays based on current \(L D D\). Follow the principle
        that introduced in Section 4.2.1. Circuit design is
        introduced in Section 4.3.2.
```


## Input 111101101011 LDD 00010000000

Figure 4.3: LDD output format example.
differences among the higher bits with AND logic (Algorithm 4, line 9). When a difference is detected among the higher bits, the current en $[i]$ will be locked at ' 0 ' ignoring the value of $d i f[i]$. A straightforward implementation is shown as Fig. 4.5(a), which uses fewest logic gate but have largest delay. To balance the cell number and circuit delay, we start from implementing a large tree-structured AND gate for en $[0]=A N D(\operatorname{dif}[(N-2): 0])$, and then add 2-to-1 AND gates onto that large AND gate to obtain the rest en. Fig. 4.5(b) illustrates the design for a 3-bit output 'en' stage, where the red AND gate is added to generate en[1].

- The output stage computes the final decision of $L D D$ based on the en (Algorithm 4, line 14). Besides, it uses en $[0]$ to check if all the bits are 0 s or 1 s , as en $[0]=0$ only when no difference is detected.

By removing the process of 'encoding the first 1's location into a binary number' introduced by LOD, the LDD circuits utilize the AND-gate tree instead of the multiplexer (MUX) tree for LOD (Fig. 4.2) to identify the first difference's location. Since the tree sizes for LDD and LOD are similar, better performance on LDD with simplified logic gates can be expected. Specific comparisons are shown in Section 4.5.

### 4.3.2 Bit Shifter

As we mentioned above, a 'bit shifter' is typically implemented as a selection circuit, which selects the corresponding output from all possibilities based on the input of ' $\#$ of bits to be shifted'. The conventional posit decoders with LZD/LOD utilize MUX for the shifter [76] as illustrated in Fig. 4.6(a), where ' $o_{-} i[j]$ ' indicates the corresponding output value for out $[i]$ when left shifting $j$ bits. As the first difference will never occur at the first bit, the $o_{-} i[1]$ is always unused (marked as red) for all cases.

(c) Output stage

Figure 4.4: Example circuit for 4-bit LDD.

In contrast, with LDD, suppose the input size for LDD is $N$ bits, we can simply express the out $[i]$ as:

$$
\begin{equation*}
\operatorname{out}^{2}[i]=\overline{\left(\overline{o_{-} i[N] L D D[0]}\right) \ldots\left(\overline{o_{-} i[j] L D D[N-j]}\right) \ldots} \tag{4.2}
\end{equation*}
$$

which can be implemented as a tree-structured NAND selection array. A 4-bit example is shown in Fig. 4.6(b), where the LDD's input size $N=5$. With the 'en' stage from LDD module, only $L D D[N-j]$ will be ' 1 ' and the rest bits of $L D D$ will be ' 0 ' when the first difference appears at in $[N-j]$. With this characteristic, the NAND gate that takes $L D D[N-j]$ as input will output $\sim o_{i}[j]$, while all the rest of NAND gates in the input stage will output 1 since the rest bits of LDD are ' 0 '. Then with another stage of NAND operation, we will have out $[i]=o_{i}[j]$, according to Eq. 4.2, which achieves the same selection function with a conventional 'shifter'. In addition, since LDD has the bit-to-bit flexibility, no input bit will be unused here.

Similar with LDD, the 'shifter' for LDD replaces the MUX tree of conventional shifter with NAND tree by removing the redundancy introduced by the binary numbers, which further optimizes the hardware cost of posit decoder.

(b) Balanced design. The red AND gate is added to generate en[1].

Figure 4.5: 3-bit output 'en' stage.

### 4.4 Hardware Cost Estimation

In this section, we theoretically analyze the hardware cost and compare our LDD based decoder with the conventional design. We evaluate the circuits with respect to two parameters, i.e., the total logic gates number $(T)$ and the longest-path logic gates number $(L)$. Here, every 2 -input basic logic gate (like NAND, AND, XNOR...) is considered as one logic gate, which means that each 2-to-1 MUX will be counted as 3 logic gates and 2 longest-path logic gates based on the CMOS MUX design. Based on the definitions, $T$ will have a positive correlation to the circuit's area, and $L$ can be used as an estimation to the circuit's delay.

Suppose the input size of the posit decoder is $N=2^{i}$, and we use $T^{i}$ and $L^{i}$ to represent


Figure 4.6: Example circuit for 4-bit shifter.
the $T$ and $L$ with $2^{i}$ bits input.
For LOD circuit shown in Fig. 4.2, we know that $T_{L O D}^{1}=2$ and $T_{L O D}^{i+1}=2 \times T_{L O D}^{i}+4$, from which we can get $T_{L O D}^{i}=3 \times 2^{i}-4$. It is obvious that $L_{L O D}^{i}=2 i-1$.

For LDD design shown in Algorithm 4 and Fig. 4.4, it is easy to tell there are $2^{i}-1$ gates in 'dif' stage and $2^{i}$ gates in output stage. In the 'en' stage, our design aims at minimizing the delay, by adopting an AND gate tree to obtain en[0] and then adding AND gates to get the rest ens based on lines 8 and 9 of Algorithm 4. With mentioned design strategies, we have $T_{L D D}^{i}=(i+4) 2^{i-1}-3$, $L_{L D D}^{i}=i+2$.

Regarding the shifter circuits, the conventional shifter and LDD-based shifter are composed by tree structured MUX gates and NAND gates, respectively, as illustrated in Fig. 4.6. Assume the total size of the decoder's output is $2^{i}$ bits (the output size for the decoder may vary with different $E S$, but it will always close to $2^{i}$ ), and we use $C S$ and $D S$ to represent the conventional LODbased shifter and LDD-based shifter, respectively. It is easy to get: $T_{C S}^{i}=3\left(2^{i}-1\right) 2^{i}, L_{C S}^{i}=2 i$, $T_{D S}^{i}=\left(2^{i+1}-1\right) 2^{i}, L_{D S}^{i}=i+1$.

All the estimations are summarized in Table 4.2. To better demonstrate the advantage of our design over, we calculate the total $T^{i}$ and $L^{i}$ for conventional LOD-based decoder and LDD-based decoder as:

$$
\begin{equation*}
R_{T}=\frac{T_{L D D}^{i}+T_{D S}^{i}}{T_{L O D}^{i}+T_{C S}^{i}} \tag{4.3}
\end{equation*}
$$

and

$$
\begin{equation*}
R_{L}=\frac{L_{L D D}^{i}+L_{D S}^{i}}{L_{L O D}^{i}+L_{C S}^{i}} \tag{4.4}
\end{equation*}
$$

Fig. 4.2 plots the estimation results for $i=2$ to 7 . It can be seen that the proposed LDDbased decoder outperforms the current technology on both $T$ and $L$ for all tested data sizes. Besides, we can observe that the LDD-based decoder performs even better with a larger input size.

Table 4.2: Hardware Cost Estimation

| $N=2^{i}$ | LOD-Based Decoder | LDD-Based Decoder |  |  |
| :--- | :--- | :--- | :--- | :--- |
|  | LOD | Shifter | LDD | Shifter |
| $T^{i}$ | $3 \cdot 2^{i}-4$ | $3\left(2^{i}-1\right) 2^{i}$ | $(i+4) 2^{i-1}-3$ | $\left(2^{i+1}-1\right) 2^{i}$ |
| $L^{i}$ | $2 i-1$ | $2 i$ | $i+2$ | $i+1$ |
| $T_{\text {Decoder }}^{i}$ | $T_{L O D}^{i}+T_{C S}^{i}$ |  | $T_{L D D}^{i}+T_{D S}^{i}$ |  |
| $L_{\text {Decoder }}^{i}$ | $L_{L O D}^{i}+L_{C S}^{i}$ |  | $L_{L D D}^{i}+L_{D S}^{i}$ |  |

### 4.5 Experimental Results

Our experimental results are presented in this section. We implement the LDD-based posit decoder and the LOD-based posit decoder proposed by recent studies [59, 57, 24, 119, 36] using Verilog HDL. Each decoder has three specific circuit designs that are compatible with the 16 -bit number system with $E S=1$, the 32 -bit number system with $E S=3$, and the 64 -bit number system with $E S=4$, according to the posit inventor's recommendation [47]. All the designs are then mapped into a 32 nm technology node using Synopsys Design Compiler. To make the comparison fair, all designs are synthesized with exact same synthesis setting and optimization effort.

All of our Verilog codes can be found in the GitHub link: https://github.com/JSCooode/ Posit_Decoder_LDD. The modules for 16 -bit and 64 -bit LDD-based decoders are parameterized so that they can be easily configured for any posit system with different number sizes.

We compare the hardware complexities of these decoders to show the advantages of the proposed LDD-based decoder. For a fair comparison, all circuits are optimized with the same effort level and are driven by identical inverters. Our experimental results are summarized in Table 4.3, where 'P-D Product' stands for 'power-delay product', which represents the average energy consumption under the same throughput. The result shows that the delay of the LDD-based decoder is decreased by $47.6 \%, 60.1 \%$, and $61.2 \%$ for 16 -bit, 32 -bit, and 64 -bit designs, respectively, compared

Table 4.3: Comparison: LDD vs. LOD [59, 57, 24, 119, 36]

|  | 16-Bit |  | 32 -Bit |  | 64-Bit |  |
| :--- | :--- | :---: | :---: | :---: | :---: | :---: |
|  | LDD | LOD | LDD | LOD | LDD | LOD |
| Delay $(n s)$ | 1.44 | 2.75 | 1.8 | 4.61 | 1.88 | 4.85 |
| Power <br> $(\mu W)$ | 22.2 | 19.6 | 66.8 | 56.1 | 231.7 | 167.8 |
| Area |  |  |  |  |  |  |
| $\left(\mu m^{2}\right)$ | 369 | 489 | 1201 | 1461 | 4474 | 4733 |
| P-D Prod. | $1 \times$ | $1.7 \times$ | $3.8 \times$ | $8.1 \times$ | $13.6 \times$ | $25.5 \times$ |

to the LOD-based decoder. Meanwhile, the average energy consumption of the LDD-based decoder is also about $50 \%$ smaller than the LOD-based decoder for 16 -bit, 32 -bit, and 64 -bit posit numbers. In addition, the LDD-based decoders also have smaller area consumption. For a better illustration, we plot the changes of delay, area, and P-D product for both LDD and LOD under different bit sizes in Fig. 4.7, from which we can see that LDD-based decoder outperforms LOD-based decoder for all input sizes, and an increasing advantage of LDD-based decoder on P-D product when expanding the input size can be observed in Fig. 4.7(c). This indicates that the LDD-based decoder will be even more applicable for modern computer systems that work with bigger data size (like the upgrade from 32 -bit systems to 64 -bit systems).

To evaluate the novelty of our design in small-data-size use cases like 8-bit Neural Networks [10], we also implement the LDD-based decoder for extremely small sized posit number (8-bit, $E S=1$ ) and compare it with state-of-the-art decoder as shown in Table 4.4, which shows that LDD still outperforms in all the aspects.

Table 4.4: Comparison for Extremely Small Data Size

|  | Delay $(n s)$ | Power $(\mu W)$ | Area $\left(\mu m^{2}\right)$ | P-D Prod. |
| :--- | :--- | :---: | :--- | :---: |
| LDD- <br> based | 1.16 | 9.11 | 118 | $1 \times$ |
| LOD- <br> based | 1.28 | 10.21 | 136 | $1.24 \times$ |

### 4.6 Conclusion

In this paper, we presented an efficient circuit structure named leading difference detector (LDD) and designed a novel decoder based on that to perform data extraction for posit numbers.

By eliminating the redundant binary number encoding and decoding processes, our proposed LDDbased posit decoder approximately halves the delay and energy consumption with a smaller hardware cost for 16 -bit, 32 -bit, and 64 -bit decoders compared to the conventional design. Future work will be directed towards the design of an efficient posit arithmetic core based on the proposed LDD and the corresponding evaluation of the overall performance on a wide range of applications.


Figure 4.7: LDD-based decoder vs. LOD-based decoder.

## Chapter 5

## Bayesian Optimization for Neural

## Network

### 5.1 Motivation

Deep learning (DL) has been showing incredible successes in solving challenging problems like decision making and data classification. However, it also has many inevitable weaknesses like overfitting, which limits the generalization capabilities [110]. One popular explanation for this weakness suggested by recent studies is that the DL model cannot handle the problem's uncertainty, which is ubiquitous in the real world [81, 41]. To this end, Bayesian statistics show great potential to express and quantify the uncertainty implied under the deep learning processes.

Recent studies implement the Bayesian statistics into DL by replacing the single-valuerepresented parameters of conventional neural networks with the parameters that are expressed by probabilistic distributions. By doing so, Bayesian neural networks (BNN) become uncertainty-aware and are able to train and analyze the uncertainty of the problems.

Suppose all trainable parameters for DNN are $\theta \sim P(\theta), D_{x}$ is the network input data and $D_{y}$ is the data label. The Bayesian update for BNN can then be expressed as:

$$
\begin{equation*}
P(\theta \mid D)=\frac{P\left(D_{y}, D_{x} \mid \theta\right) P(\theta)}{P\left(D_{y}, D_{x}\right)} \tag{5.1}
\end{equation*}
$$

where $P(\theta)$ is the trainable parameters' prior that stands for the prior knowledge of the
parameters, $P(\theta \mid D)$ is the posterior of $\theta$ after the training process. As calculating $P\left(D_{y}, D_{x}\right)$ is practically impossible, researchers adopt $P(\theta \mid D) \propto P\left(D_{y}, D_{x} \mid \theta\right) P(\theta)$ and apply normalization to get the final results. In addition, to simplify the computational complexity to an acceptable level, Variational inference [101] is widely studied and applied for Bayesian neural networks. However, the current solutions are still relatively computationally expensive for some unavoidable steps even with finite solution space (piecewise distributions), like generating samples from specific distribution equations and calculating the Kullback-Leibler divergence for two distributions $\left(D_{K L}(P \| Q)=\right.$ $\left.\sum P(x) \log \left(\frac{P(x)}{Q(x)}\right)\right)$. Besides, it is hard to track whether the selection of prior fits the real distribution of parameters is also queried, and a bad fitness can cause avoidless converging issues. The mentioned problems affect the feasibility of the BNN in practical application.

In this chapter, we propose a method to generate samples and execute Bayesian update for piecewise probability distributions with only simple arithmetic operations. In addition, we further proposed a way in 5.3 .2 to potentially approximate infinite distribution ranges with finite piecewise settings.

### 5.2 Bayesian Update for Peicewise Probability Distributions

We are looking for a piecewise solution for the BNN. Suppose $\theta_{i}$ is a single trainable parameter that $\theta_{i} \in \theta$. We divide $\theta_{i}$ into $N$ discrete values so that $\theta_{i}=\left\{\theta_{i 0} \ldots, \theta_{i j \ldots,}, \theta_{i(N-1)}\right\}$ and $\theta_{i j} \in \theta_{i}$. Then we can assign specific probability to each value point that satisfies $\sum_{j} P\left(\theta_{i j}\right)=1$. With this property, we can then update the Equation 5.1 to:

$$
\begin{equation*}
P\left(\theta_{i j} \mid D\right) \propto P\left(D_{y}, D_{x} \mid \theta\right) P\left(\theta_{i j}\right) \tag{5.2}
\end{equation*}
$$

where $P\left(\theta_{i j}\right)$ is the prior of the $\theta_{i j}$ and $P\left(D_{y}, D_{x} \mid \theta\right)$ is the probability to receive a correct estimation based on current $P(\theta)$ and training data.

Now, consider the scenario that for each iteration, the value for each $\theta_{i}$ will be sampled from the discrete distribution $P\left(\theta_{i}\right)$, and then execute the regular forward and backward propagation with the sampled value. In this case, there will be:

$$
\begin{equation*}
P\left(D_{y}, D_{x} \mid \theta\right) P\left(\theta_{i j}\right)=P\left(D_{y}, D_{x} \mid \theta, \theta_{i j}\right) \propto \sum P\left(D_{y}, D_{x} \mid \theta, \theta_{i j} \text { Sampled }\right) \tag{5.3}
\end{equation*}
$$

For the $P\left(D_{y}, D_{x} \mid \theta\right)$, it equals to the softmax of the correct training label based on the property of the neural network. Combining all equations mentioned above, we can get that in our piecewise scenario:

$$
\begin{equation*}
P\left(\theta_{i j} \mid D\right) \propto \sum \operatorname{softmax}\left(D_{y}, D_{x} \mid \theta_{i j} \text { Sampled }\right) \tag{5.4}
\end{equation*}
$$

where $\operatorname{softmax}\left(D_{y} \mid \theta_{i j}\right.$ Sampled) means the value of softmax on the correct training label when $\theta_{i j}$ gets sampled.

With all the transforms, the original Bayesian updating problem is simplified into a plain arithmetic problem, which is solvable for modern computer systems.

The main advantages of adopting a piecewise solution for BNN are: (1) Computational complexity is massively simplified as no distribution is involved. (2) The final result will not be limited by the prior distribution family as piecewise solution can be trained into any distribution by Bayesian update.

### 5.3 Bayesian Optimization Algorithm

### 5.3.1 Bayesian Optimization for Pre-trained Neural Network

We call our algorithm Bayesian optimization, which is appended after the normal neural network training. The basic idea is that we map the value for each trainable parameter $\theta_{i}$ of Neural network into $(2 N+1)$ pieces $\left(\theta_{i 0}, \ldots, \theta_{i(2 N)}\right)$, every piece will be representing one of the $\theta_{i}$ 's neighbour value. Then, a piecewise prior probability distribution $P\left(\theta_{i}\right)$ will be assigned to all the trainable parameters with $\sum_{j} P\left(\theta_{i j}\right)=1$. During each iteration of the inference, the $P\left(\theta_{i j}\right)$ will be updated based on the equations we discussed in Section 5.2. The detailed process is shown below in Algorithm 5. To avoid the occasionally missing sampling for $\theta_{i j}\left(\theta_{i j}\right.$ is not sampled between two Bayesian updates), we add an offset to all the $P\left(\theta_{i j}\right)$ during the Bayesian updating, as presented in Algorithm 5, line 10, 12.

After some iterations, the $P\left(\theta_{i j}\right)$ will be updated by the Bayesian optimization based on the validation accuracy as shown in Fig. 5.1 (a).

We run our simulation test on a pre-trained ResNet-18 over the CIFAR-10 database. We set $N=3, c=20, T=64, \alpha=1 / 16$. As the primary results, we improved the validation accuracy from $89.4 \%$ to $91.24 \%$.

```
Algorithm 5 Bayesian Optimization for Pre-trained Neural Network
    Optimization Parameters: Piecewise number: \(N\), piecewise coefficient: \(c\), Bayesian update
    period: \(T\), Bayesian update offset: \(\alpha\).
    For each trainable parameter \(\theta_{i}\) :
    for \(j=0: 2 N\) do
        \(\theta_{i j}=\theta_{i}+(j-N) *\left(\theta_{i} / c\right)\)
        \(P\left(\theta_{i j}\right)=1 /(2 N+1)\)
    end for
    counter \(=1\)
    for \(D_{y}, D_{x}\) from Data Loader do
        if counter \(\% T==0\) then
            \(S M_{o f f s e t}=\alpha * \max \left[\operatorname{softmax\_ sum}\left(\theta_{i 0}\right), \ldots, \operatorname{softmax} \_\operatorname{sum}\left(\theta_{i 2 N}\right)\right]\)
            for \(j=0: 2 N\) do
                Update \(P\left(\theta_{i j}\right)\) with \(P\left(\theta_{i j}\right) \propto \operatorname{softmax\_ sum}\left(\theta_{i j}\right)+S M_{o f f s e t}\)
                \(\operatorname{softmax} \_\)sum \(\left(\theta_{i j}\right)=0\)
            end for
        end if
        Sample value for \(\theta_{i}\) from \(P\left(\theta_{i}\right)\)
        Perform forward propagation
        for \(j=0: 2 N\) do
            if \(\theta_{i j}\) get sampled then
                softmax_sum \(\left(\theta_{i j}\right)+=\operatorname{softmax}\left(D_{y}\right)\)
            end if
        end for
        counter \(+=1\)
    end for
```


### 5.3.2 Bayesian Optimization for Pre-trained Neural Network with Backpropagation

One major problem for basic Bayesian optimization is the limitation of the piecewise number. To make the $P\left(\theta_{i j}\right)$ cover a wider range with higher resolution for $\theta_{i}$, we need to increase the piecewise number. However, a large piecewise number means more parameters are required, which can cause a considerable cost on memory access.

To further improve the capability of our Bayesian optimization algorithm to handle preciser probability distributions, we combine it with the original backpropagation of the neural network as shown in Algorithm 6, the extra steps over the basic Bayesian optimization are marked with red color. Generally, it updates both the data value and the probability distribution of the sampled $\theta_{i j}$, while the basic Bayesian optimization only update the probability distribution, as illustrated in Fig. 5.1.

```
Algorithm 6 Bayesian Optimization for Pre-trained Neural Network with Backpropagation
    Optimization Parameters: Piecewise number: \(N\), piecewise coefficient: \(c\), Bayesian update
    period: \(T\), Bayesian update offset: \(\alpha\).
    For each trainable parameter \(\theta_{i}\) :
    for \(j=0: 2 N\) do
        \(\theta_{i j}=\theta_{i}+(j-N) *\left(\theta_{i} / c\right)\)
        \(P\left(\theta_{i j}\right)=1 /(2 N+1)\)
    end for
    counter \(=1\)
    for \(D_{y}, D_{x}\) from Data Loader do
        if counter \(\% T==0\) then
            \(S M_{o f f s e t}=\alpha * \max \left[\right.\) softmax_sum \(\left.\left(\theta_{i 0}\right), \ldots, \operatorname{softmax\_ sum}\left(\theta_{i 2 N}\right)\right]\)
            for \(j=0: 2 N\) do
                Update \(P\left(\theta_{i j}\right)\) with \(P\left(\theta_{i j}\right) \propto \operatorname{softmax\_ sum}\left(\theta_{i j}\right)+S M_{o f f s e t}\)
                \(\operatorname{softmax\_ sum}\left(\theta_{i j}\right)=0\)
            end for
        end if
        Sample value for \(\theta_{i}\) from \(P\left(\theta_{i}\right)\)
        Perform forward propagation
        Perform backpropagation
        for \(j=0: 2 N\) do
            if \(\theta_{i j}\) get sampled then
                    \(\operatorname{softmax} \_\)sum \(\left(\theta_{i j}\right)+=\operatorname{softmax}\left(D_{y}\right)\)
                    Update the value of \(\theta_{i j}\) with the backpropagation's result
            end if
        end for
        counter \(+=1\)
    end for
```


(a) Basic Bayesian Optimization.

(b) Bayesian Optimization with Backpropagation.

Figure 5.1: Bayesian Optimization without and with Backpropagation.

### 5.4 Hardware Acceleration

The circuit design for our sample generator for piecewise probability distribution is shown in Fig. 5.2. It takes a uniformly distributed random number $I N$ as random input, then by comparing it with the boundary values, which are set based on the piecewise probability distribution, the generator can select the corresponding sample for $\theta_{i}$ and send it to the output with two stages of AND gates.

The key architecture of the sample generator is the comp (comparator) array. A four-sample example is shown in Fig. 5.3. The comp array is composed of tree-structured comp units, where port in takes the input value of the uniformly distributed random number, and std indicates the boundary values defined by the piecewise probability distribution. The SE unit will compare in with $s t d$ s and then set one of the 1 -bit outputs large or small to ' 1 ' and another to ' 0 '. Besides, when en is ' 0 ', both large and small will be set to ' 0 '.

By connecting multiple comp unit blocks as a binary tree structure and making all ins share the same random input, we can easily obtain a comp array circuit for arbitrary sample numbers. A four-sample circuit design is demonstrated in Fig. 5.3, which has the function:


Figure 5.2: Sample generator for piecewise probability distribution.

$$
\text { Sample } \#= \begin{cases}0, & \text { if } i n \in(0, b 1]  \tag{5.5}\\ 1, & \text { if } \text { in } \in[b 1, b 0) \\ 2, & \text { if } \text { in } \in[b 0, b 2) \\ 3, & \text { if } \text { in } \in[b 2,+\infty)\end{cases}
$$

In this example, for an $i$-bit random input, to generate sample 1 with probability $P_{1}$, we only need to make the boundary values satisfy $(b 0-b 1) / 2^{i}=P_{1}$.

To check the novelty of our design, we implement our design together with a basic design that performs a similar logic to the GPU when generating samples based on specific probability (check the conditions one by one). All the circuits are designed with Verilog HDL and mapped into a 32 nm technology node using Synopsys Design Compiler. The synthesized results are tabulated in Table 5.1, where 'P-D Product' stands for 'power-delay product,' which represents the average energy consumption under the same throughput. The results show that our design is $24.1 \%$ faster than the basic design, and it only consumes about half the energy of the basic design under the same throughput. With all the advantages, it only costs $19 \%$ more chip area compared to the basic design as the cost.


Figure 5.3: The logic circuit of comp array (4-sample example).

Table 5.1: Comparison for Sample Generator

|  | Delay $(n s)$ | Power $(\mu W)$ | Area $\left(\mu m^{2}\right)$ | P-D Prod. |
| :--- | :--- | :---: | :--- | :---: |
| Our Design | 1.32 | 154.4 | 5190 | $1 \times$ |
| Basic Design | 1.74 | 238.7 | 4358 | $2.04 \times$ |

## Chapter 6

## Future Work

### 6.1 LDD-based Posit Arithmetic Core and Neuron

An arithmetic core is the hardware structure that can perform all arithmetic calculations and return the result for a specific number system. With all the great properties of leading difference detector (LDD) over the convention technology, it is necessary to design a complete arithmetic core based on LDD. Leveraging the bit-wise indicator of LDD and the corresponding efficient selection circuit, we believe further optimization is promising for LDD-based posit arithmetic core.

In addition, to implement posit numbers for neural network applications, a well-designed posit neuron is crucial. Designing an LDD-based neuron that can execute multiply-accumulate operations efficiently can be an option for the future work.

### 6.2 Bayesian Optimization for Neural Network

There is great potential to further study the Bayesian Optimization. For example, is there any better way to tune the training parameters like Piecewise number: $N$, piecewise coefficient: $c$, Bayesian update period: $T$, and Bayesian update offset: $\alpha$ to receive a better validation accuracy? In addition, the current prior is simply set as uniform distribution, which means there is potential to improve the performance with specific prior distributions.

Besides that, more mathematical analysis can be added to the problem, like the error introduced by the Monte Carlo process. Is softmax on the correct label a suitable representation for the
$P\left(D_{y}, D_{x} \mid \theta\right) ?$
On the hardware side, the current hardware accelerator only considers the sample generation for the piecewise distribution. A larger accelerator that can take over more tasks during the Bayesian optimization can be a good direction for the future work.

## Bibliography

[1] IEEE standard for floating-point arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008), pages 1-84, 2019.
[2] Mohamed S Abdulnabi and Hisham Ahmed. Design of efficient cyclic redundancy check32 using FPGA. In Proc. IEEE Int'l Conf. Computer, Control, Electrical, and Electronics Engineering, pages 1-5, 2018.
[3] Khalid H Abed and Raymond E Siferd. Vlsi implementations of low-power leading-one detector circuits. In Proceedings of the IEEE SoutheastCon 2006, pages 279-284. IEEE, 2006.
[4] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: A survey. Heliyon, 4(11):e00938, 2018.
[5] Shadi Al-Sarawi, Mohammed Anbar, Kamal Alieyan, and Mahmood Alzubaidi. Internet of things (IoT) communication protocols. In Proc. IEEE Int'l Conf. Information Technology (ICIT), pages 685-690, 2017.
[6] Armin Alaghi and John P Hayes. Survey of stochastic computing. ACM Trans. on Embedded computing systems (TECS), 12(2s):1-19, 2013.
[7] Armin Alaghi, Weikang Qian, and John P Hayes. The promise and challenge of stochastic computing. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 37(8):1515-1531, 2017.
[8] Anvesha Amravati, Saad Bin Nasir, Sivaram Thangadurai, et al. A 55nm time-domain mixedsignal neuromorphic accelerator with stochastic synapses and embedded reinforcement learning for autonomous micro-robots. In Proc. IEEE Int'l Solid-State Circuits Conference, pages 124126, 2018.
[9] Alia Asheralieva and Dusit Niyato. Distributed dynamic resource management and pricing in the IoT systems with blockchain-as-a-service and UAV-enabled mobile edge computing. IEEE Internet of Things Journal, 7(3):1974-1993, 2019.
[10] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. Advances in neural information processing systems, 31, 2018.
[11] L. Benini, A. Bogliolo, G. Paleologo, and G. De Micheli. Policy optimization for dynamic power management. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 18(6):813-833, 1999.
[12] Randall A Berry and Robert G Gallager. Communication over fading channels with delay constraints. IEEE Trans. on Information theory, 48(5):1135-1149, 2002.
[13] P. Blasco, D. Gunduz, and M. Dohler. A learning theoretic approach to energy harvesting communication system optimization. IEEE Trans. Wireless Commun., 12(4):1872-1882, 2013.
[14] Leonardo Bonati, Salvatore D'Oro, Michele Polese, Stefano Basagni, and Tommaso Melodia. Intelligence and learning in o-ran for data-driven nextg cellular networks. IEEE Communications Magazine, 59(10):21-27, 2021.
[15] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In Proc. IEEE Int'l Symp. High-Performance Computer Architecture, pages 13-22, 1999.
[16] Alex Bystrov, David John Kinniment, and Alexandre Yakovlev. Priority arbiters. In Proceedings of Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 128-137. IEEE, 2000.
[17] Zachariah Carmichael, Hamed F Langroudi, Char Khazanov, Jeffrey Lillie, John L Gustafson, and Dhireesha Kudithipudi. Deep positron: A deep neural network using the posit number system. In 2019 Design, Automation \& Test in Europe Conference \& Exhibition (DATE), pages 1421-1426. IEEE, 2019.
[18] J. Chakareski. Uplink scheduling of visual sensors: When view popularity matters. IEEE Trans. Commun., 2(63), February 2015.
[19] J. Chakareski. UAV-IoT for next generation virtual reality. IEEE Trans. Image Process., 28(12):5977-5990, December 2019.
[20] J. Chakareski, R. Aksu, X. Corbillon, G. Simon, and V. Swaminathan. Viewport-driven ratedistortion optimized $360^{\circ}$ video streaming. In Proc. IEEE Int'l Conf. Communications, May 2018.
[21] J. Chakareski and P. Frossard. Distributed collaboration for enhanced sender-driven video streaming. IEEE Trans. Multimedia, 10(5):858-870, August 2008.
[22] J. Chakareski and B. Girod. Rate-distortion optimized packet scheduling and routing for media streaming with path diversity. In Proc. IEEE Data Compression Conference, pages 203-212, Snowbird, UT, March 2003.
[23] J. Chakareski, V. Velisavljević, and V. Stanković. User-action-driven view and rate scalable multiview video coding. IEEE Trans. Image Process., 22(9):3473-3484, September 2013.
[24] Rohit Chaurasiya, John Gustafson, Rahul Shrestha, Jonathan Neudorfer, Sangeeth Nambiar, Kaustav Niyogi, Farhad Merchant, and Rainer Leupers. Parameterized posit arithmetic hardware generator. In 2018 IEEE 36th International Conference on Computer Design (ICCD), pages 334-341. IEEE, 2018.
[25] Louis HY Chen. On the convergence of poisson binomial to poisson distributions. The Annals of Probability, pages 178-180, 1974.
[26] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. ACM SIGARCH Computer Architecture News, 44(3):27-39, 2016.
[27] Michele Chincoli and Antonio Liotta. Self-learning power control in wireless sensor networks. Sensors, 18(2):375, 2018.
[28] Man Chu, Hang Li, Xuewen Liao, and Shuguang Cui. Reinforcement learning-based multiaccess control and battery prediction with energy harvesting in iot systems. IEEE Internet of Things Journal, 6(2):2009-2020, 2018.
[29] Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, and Sergio Saponara. A fast approximation of the hyperbolic tangent when using posit numbers and its application to deep neural networks. In International Conference on Applications in Electronics Pervading Industry, Environment and Society, pages 213-221. Springer, Cham, 2019.
[30] X. Corbillon, A. Devlic, G. Simon, and J. Chakareski. Viewport-adaptive navigable 360-degree video delivery. In Proc. IEEE Int'l Conf. Communications, Paris, France, May 2017. IEEE.
[31] Lucileide M. D. Da Silva, Matheus F. Torquato, and Marcelo A. C. Fernandes. Parallel implementation of reinforcement learning Q-learning technique for fpga. IEEE Access, 7:27822798, 2019.
[32] Yvan Debizet, Guénolé Lallement, Fady Abouzeid, Philippe Roche, and Jean-Luc Autran. Q-learning-based adaptive power management for IoT system-on-chips with embedded power states. In IEEE International Symposium on Circuits and Systems (ISCAS), pages 1-5, 2018.
[33] Li Du, Yuan Du, Yilei Li, Junjie Su, Yen-Cheng Kuan, Chun-Chen Liu, and Mau-Chung Frank Chang. A reconfigurable streaming deep convolutional neural network accelerator for Internet of Things. IEEE Trans. on Circuits and Systems I: Regular Papers, 65(1):198-208, 2017.
[34] Shahriar Ebrahimi, Siavash Bayat-Sarmadi, and Hatameh Mosanaei-Boorani. Post-quantum cryptoprocessors optimized for edge and resource-constrained devices in iot. IEEE Internet of Things Journal, 6(3):5500-5507, 2019.
[35] Arash Fayyazi, Mohammad Ansari, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. An ultra low-power memristive neuromorphic circuit for Internet of Things smart sensors. IEEE Internet of Things Journal, 5(2):1011-1022, 2018.
[36] Luc Forget, Yohann Uguen, and Florent de Dinechin. Comparing posit and ieee-754 hardware cost. 2021.
[37] Robert J Francis, Jonathan Rose, and Kevin Chung. Chortle: A technology mapping program for lookup table-based field programmable gate arrays. In Proc. ACM/IEEE Design Automation Conference, pages 613-619, 1991.
[38] Xiaoyuan Fu, F Richard Yu, Jingyu Wang, Qi Qi, and Jianxin Liao. Service function chain embedding for NFV-enabled IoT based on deep reinforcement learning. IEEE Communications Magazine, 57(11):102-108, 2019.
[39] Brian R Gaines. Stochastic computing. In Proc. of Spring Joint Computer Conference, pages 149-156, 1967.
[40] Pranay Reddy Gankidi. FPGA accelerator architecture for Q-learning and its applications in space exploration rovers. PhD thesis, Arizona State University, 2016.
[41] Jakob Gawlikowski, Cedrique Rovile Njieutcheu Tassi, Mohsin Ali, Jongseok Lee, Matthias Humt, Jianxiang Feng, Anna Kruspe, Rudolph Triebel, Peter Jung, Ribana Roscher, et al. A survey of uncertainty in deep neural networks. arXiv preprint arXiv:2107.03342, 2021.
[42] Andrea Goldsmith. Wireless Communications. Cambridge University Press, 2005.
[43] Hector A. Gonzalez, Shahzad Muzaffar, Jerald Yoo, and Ibrahim Abe M. Elfadel. An inference hardware accelerator for eeg-based emotion detection. In IEEE Int'l Symposium on Circuits and Systems (ISCAS), pages 1-5, 2020.
[44] Warren J Gross, Vincent C Gaudet, and Aaron Milner. Stochastic implementation of LDPC decoders. In Proc. of Conference Record of the Thirty-Ninth Asilomar Conference on Signals, Systems and Computers, 2005., pages 713-717. IEEE, 2005.
[45] Yashuang Guo, F Richard Yu, Jianping An, Kai Yang, Ying He, and Victor CM Leung. Bufferaware streaming in small-scale wireless networks: A deep reinforcement learning approach. IEEE Transactions on vehicular technology, 68(7):6891-6902, 2019.
[46] John L. Gustafson. A radical approach to computation with real numbers. In Supercomputing Frontiers and Innovations, page 3(2):38-53, 2016.
[47] John L Gustafson and Isaac T Yonemoto. Beating floating point at its own game: Posit arithmetic. Supercomputing frontiers and innovations, 4(2):71-86, 2017.
[48] Vesal Hakami, Seyedakbar Mostafavi, Nastooh Taheri Javan, and Zahra Rashidi. An optimal policy for joint compression and transmission control in delay-constrained energy harvesting IoT devices. Computer Communications, 160:554-566, 2020.
[49] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. Advances in neural information processing systems, 28, 2015.
[50] Soheil Hashemi, Nicholas Anthony, Hokchhay Tann, R Iris Bahar, and Sherief Reda. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In Design, Automation $8 \mathcal{T}$ Test in Europe Conference $\xi^{6}$ Exhibition (DATE), 2017, pages 14741479. IEEE, 2017.
[51] John P Hayes. Introduction to stochastic computing and its challenges. In Proc. of the 52nd Annual Design Automation Conference, pages 1-3, 2015.
[52] Junjie Hou, Yongxin Zhu, Sen Du, and Shijin Song. Enhancing accuracy and dynamic range of scientific data analytics by implementing posit arithmetic on fpga. Journal of Signal Processing Systems, 91(10):1137-1148, 2019.
[53] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861, 2017.
[54] Hantao Huang, Rai Suleman Khalid, Wenye Liu, and Hao Yu. Work-in-progress: A fast online sequential learning accelerator for IoT network intrusion detection. In Proc. IEEE Int'l Conf. Hardware/Software Codesign and System Synthesis, pages 1-2, 2017.
[55] Mohsen Imani, Daniel Peroni, and Tajana Rosing. Nvalt: Nonvolatile approximate lookup table for GPU acceleration. IEEE Embedded Systems Letters, 10(1):14-17, 2017.
[56] Manish Kumar Jaiswal and Hayden K-H So. Universal number posit arithmetic generator on fpga. In 2018 Design, Automation \& Test in Europe Conference $\mathcal{G}$ Exhibition (DATE), pages 1159-1162. IEEE, 2018.
[57] Manish Kumar Jaiswal and Hayden K-H So. Pacogen: A hardware posit arithmetic core generator. IEEE Access, 7:74586-74601, 2019.
[58] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. Zena: Zero-aware neural network accelerator. IEEE Design \& Test, 35(1):39-46, 2017.
[59] Zep Kleijweg. Hybrid posit and fixed point hardware for quantized dnn inference. 2021.
[60] Hermann Kopetz. Internet of things. In Real-time systems. Springer, 2011.
[61] Seyed Hamed Fatemi Langroudi, Tej Pandit, and Dhireesha Kudithipudi. Deep learning inference on embedded devices: Fixed-point vs posit. In 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2), pages 19-23. IEEE, 2018.
[62] He Li, Kaoru Ota, and Mianxiong Dong. Learning IoT in edge: Deep learning for the Internet of Things with edge computing. IEEE Network, 32(1):96-101, 2018.
[63] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971, 2015.
[64] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. Advances in neural information processing systems, 30, 2017.
[65] Xiaolan Liu, Zhijin Qin, and Yue Gao. Resource allocation for edge computing in IoT networks via reinforcement learning. In Proc. of IEEE Int'l Conf. on Communications (ICC), pages 1-6, 2019.
[66] Yun Long, Daehyun Kim, Edward Lee, Priyabrata Saha, Burhan Ahmad Mudassar, Xueyuan She, Asif Islam Khan, and Saibal Mukhopadhyay. A ferroelectric fet-based processing-inmemory architecture for dnn acceleration. IEEE Journal on Exploratory Solid-State Computational Devices and Circuits, 5:113-122, 2019.
[67] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. Advances in neural information processing systems, 30, 2017.
[68] Jinming Lu, Chao Fang, Mingyang Xu, Jun Lin, and Zhongfeng Wang. Evaluations on deep neural networks training using posit number system. IEEE Transactions on Computers, $70(2): 174-187,2020$.
[69] Nguyen Cong Luong, Dinh Thai Hoang, Ping Wang, Dusit Niyato, Dong In Kim, and Zhu Han. Data collection and wireless communication in Internet of Things (IoT) using economic analysis and pricing models: A survey. IEEE Communications Surveys $\mathcal{F}$ Tutorials, 18(4), 2016.
[70] N. Mastronarde and M. van der Schaar. Joint physical-layer and system-level power management for delay-sensitive wireless communications. IEEE Trans. Mobile Comput., 12(4):694709, 2013.
[71] N. Mastronarde, F. Verde, D. Darsena, A. Scaglione, and M. van der Schaar. Transmitting important bits and sailing high radio waves: A decentralized cross-layer approach to cooperative video transmission. IEEE J. Selected Areas in Communications, 30(9), October 2012.
[72] Nicholas Mastronarde, Jalil Modares, Changcan Wu, and Jacob Chakareski. Reinforcement learning for energy-efficient delay-sensitive CSMA/CA scheduling. In Proc. IEEE Global Telecommunications Conf., December 2016.
[73] Nicholas Mastronarde and Mihaela van der Schaar. Fast reinforcement learning for energyefficient wireless communication. IEEE Trans. Signal Process., 59(12):6262-6266, 2011.
[74] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In International Conference on Machine Learning, pages 1928-1937. PMLR, 2016.
[75] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. nature, 518(7540):529-533, 2015.
[76] Yehdhih Ould Mohammed Moctar, Nithin George, Hadi Parandeh-Afshar, Paolo Ienne, Guy GF Lemieux, and Philip Brisk. Reducing the cost of floating-point mantissa alignment and normalization in FPGAs. In Proceedings of international symposium on Field Programmable Gate Arrays, pages 255-264, 2012.
[77] Bert Moons and Marian Verhelst. Energy-efficiency and accuracy of stochastic computing circuits in emerging technologies. IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 4(4):475-486, 2014.
[78] Raul Murillo, Alberto A Del Barrio, and Guillermo Botella. Deep pensieve: A deep learning framework based on the posit number system. Digital Signal Processing, 102:102762, 2020.
[79] Raul Murillo, Alberto Antonio Del Barrio Garcia, Guillermo Botella, Min Soo Kim, Hyunjin Kim, and Nader Bagherzadeh. Plam: a posit logarithm-approximate multiplier. IEEE Transactions on Emerging Topics in Computing, 2021.
[80] Ali Naderi, Shie Mannor, Mohamad Sawan, and Warren J. Gross. Delayed stochastic decoding of LDPC codes. IEEE Trans. on Signal Processing, 59(11):5617-5626, 2011.
[81] Tim Pearce, Felix Leibfried, and Alexandra Brintrup. Uncertainty in neural networks: Approximately bayesian ensembling. In International conference on artificial intelligence and statistics, pages 234-244. PMLR, 2020.
[82] Artur Podobas and Satoshi Matsuoka. Hardware implementation of posits and their application in fpgas. In IEEE International Parallel and Distributed Processing Symposium Workshops, pages 138-145, 2018.
[83] Andrii Polianytsia, Olena Starkova, and Kostiantyn Herasymenko. Survey of hardware IoT platforms. In Proc. IEEE Int'l Scientific-Practical Conf. Problmes of Infocommunications Science and Technology, pages 152-153, 2016.
[84] Warren B Powell. Approximate Dynamic Programming: Solving the curses of dimensionality, volume 703. John Wiley \& Sons, 2007.
[85] Chao Qiu, F Richard Yu, Haipeng Yao, Chunxiao Jiang, Fangmin Xu, and Chenglin Zhao. Blockchain-based software-defined industrial Internet of Things: A dueling deep Q-learning approach. IEEE Internet of Things Journal, 6(3):4627-4639, 2018.
[86] Sreeraj Rajendran, Wannes Meert, Domenico Giustiniano, Vincent Lenders, and Sofie Pollin. Deep learning models for wireless signal classification with distributed low-cost spectrum sensors. IEEE Transactions on Cognitive Communications and Networking, 4(3):433-445, 2018.
[87] Usman Raza, Parag Kulkarni, and Mahesh Sooriyabandara. Low power wide area networks: An overview. IEEE Comm. Surveys $\&$ Tutorials, 2017.
[88] Ao Ren, Zhe Li, Caiwen Ding, Qinru Qiu, Yanzhi Wang, Ji Li, Xuehai Qian, and Bo Yuan. Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing. ACM SIGPLAN Notices, 52(4):405-418, 2017.
[89] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In 249th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1-13. IEEE, 2016.
[90] N. Salodkar, A. Bhorkar, A. Karandikar, and V.S. Borkar. An on-line learning algorithm for energy efficient delay constrained scheduling over a fading channel. IEEE J. Sel. Areas Commun., 26(4), 2008.
[91] Behrooz Sangchoolie, Karthik Pattabiraman, and Johan Karlsson. One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors. In Proc. of 47 th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN), pages 97-108, 2017.
[92] John Sartori, Joseph Sloan, and Rakesh Kumar. Stochastic computing: Embracing errors in architecture and design of processors and applications. In Proc. of the 14 th Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES), pages 135-144. IEEE, 2011.
[93] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In International Conference on Machine Learning, pages 18891897. PMLR, 2015.
[94] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. Report from Open AI, arXiv preprint arXiv:1707.06347, 2017.
[95] Anirban Sengupta and Sandip Kundu. Guest editorial: Securing IoT hardware: threat models and reliable, low-power design solutions. IEEE Trans. Very Large Scale Integration Systems, 25(12), 2017.
[96] Alireza Seyedi and Biplab Sikdar. Energy efficient transmission strategies for body sensor networks with energy harvesting. IEEE Trans. Commun., 58(7):2116-2126, 2010.
[97] Ali Shakarami, Mostafa Ghobaei-Arani, and Ali Shahidinejad. A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective. Computer Networks, 182:107496, 2020.
[98] N. Sharma, N. Mastronarde, and J. Chakareski. Accelerated structure-aware reinforcement learning for delay-sensitive energy harvesting wireless sensors. IEEE Trans. Signal Processing, 68(1), December 2020.
[99] N. Sharma, N. Mastronarde, and J. Chakareski. Delay-sensitive energy-harvesting wireless sensors: Optimal scheduling, structural properties, and approximation analysis. IEEE Trans. Communications, 68(4):2509-2524, April 2020.
[100] Nikhilesh Sharma, Nicholas Mastronarde, and Jacob Chakareski. Accelerated structure-aware reinforcement learning for delay-sensitive energy harvesting wireless sensors. IEEE Transactions on Signal Processing, 68:1409-1424, 2020.
[101] Kumar Shridhar, Felix Laumann, and Marcus Liwicki. A comprehensive guide to bayesian convolutional neural network with variational inference. arXiv preprint arXiv:1901.02731, 2019.
[102] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.
[103] Arvind Singh, Nikhil Chawla, Jong Hwan Ko, Monodeep Kar, and Saibal Mukhopadhyay. Energy efficient and side-channel secure cryptographic hardware for iot-edge nodes. IEEE Internet of Things Journal, 6(1):421-434, 2018.
[104] Wei Song, Dirk Koch, Mikel Luján, and Jim Garside. Parallel hardware merge sorter. In Proc. IEEE Int'l Symp. Field-Programmable Custom Computing Machines, pages 95-102, 2016.
[105] Sergio Spanò, Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Marco Matta, Alberto Nannarelli, and Marco Re. An efficient hardware implementation of reinforcement learning: The Q-learning algorithm. IEEE Access, 7:186340-186351, 2019.
[106] SN Stamenković, V Lj Marković, AP Jovanović, and MN Stankov. Generalization of electron avalanche statistics based on negative binomial distribution-multielectron initiation and gaussian approximation. Journal of Instrumentation, 13(12):P12002, 2018.
[107] Jianchi Sun, Nikhilesh Sharma, Jacob Chakareski, Nicholas Mastronarde, and Yingjie Lao. Action evaluation hardware accelerator for next-generation real-time reinforcement learning in emerging IoT systems. In IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pages 428-433, July 2020.
[108] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Viji Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. Advances in neural information processing systems, 32, 2019.
[109] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. MIT Press, 2018.
[110] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199, 2013.
[111] Yasuhiko Takemura. Lookup table and programmable logic device including lookup table, February 14 2017. US Patent 9,571,103.
[112] Jie Tang, Dawei Sun, Shaoshan Liu, and Jean-Luc Gaudiot. Enabling deep learning on IoT devices. Computer, 50(10):92-96, 2017.
[113] Wei Tang, Gang Hua, and Liang Wang. How to train a compact binary neural network with high accuracy? In Thirty-First AAAI conference on artificial intelligence, 2017.
[114] N. Thomos, J. Chakareski, and P. Frossard. Randomized network coding for UEP video delivery in overlay networks. In Proc. IEEE Int'l Conf. Multimedia and Expo, pages 730-733, June/July 2009.
[115] Ye Tian, Ting Wang, Qian Zhang, and Qiang Xu. ApproxLUT: A novel approximate lookup table-based accelerator. In Proc. IEEE/ACM Int'l Conf. Computer-Aided Design, pages 438443, 2017.
[116] Jonathan Ying Fai Tong, David Nagle, and Rob A Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. IEEE Trans. Very Large ScaleIntegration Systems, 2000.
[117] Niloofar Toorchi, Jacob Chakareski, and Nicholas Mastronarde. Fast and low-complexity reinforcement learning for delay-sensitive energy harvesting wireless visual sensing systems. In Proc. of IEEE Int'l Conf. on Image Processing (ICIP), pages 1804-1808, 2016.
[118] Yeong-Luh Ueng, Chun-Yi Wang, and Mao-Ruei Li. An efficient combined bit-flipping and stochastic LDPC decoder using improved probability tracers. IEEE Trans. on Signal Processing, 65(20):5368-5380, 2017.
[119] Yohann Uguen, Luc Forget, and Florent de Dinechin. Evaluating the hardware cost of the posit number system. In 2019 29th International Conference on Field Programmable Logic and Applications (FPL), pages 106-113, 2019.
[120] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. DLAU: A scalable deep learning accelerator unit on FPGA. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, 2016.
[121] Christopher JCH Watkins and Peter Dayan. Q-learning. Machine learning, 8(3-4):279-292, 1992.
[122] Bo K Wong, Thomas A Bodnovich, and Yakup Selvi. Neural network applications in business: A review and analysis of the literature (1988-1995). Decision Support Systems, 19(4):301-320, 1997.
[123] Fan Wu, Christoph Rüdiger, and Mehmet Yuce. Real-time performance of a self-powered environmental IoT sensor network system. Sensors, 2017.
[124] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. arXiv preprint arXiv:1802.04680, 2018.
[125] Shimeng Yu, Zhiwei Li, Pai-Yu Chen, Huaqiang Wu, Bin Gao, Deli Wang, Wei Wu, and He Qian. Binary neural network with 16 mb ram macro chip for classification and online training. In 2016 IEEE International Electron Devices Meeting (IEDM), pages 16-2. IEEE, 2016.
[126] Hao Zhang and Seok-Bum Ko. Design of power efficient posit multiplier. IEEE Transactions on Circuits and Systems II: Express Briefs, 67(5):861-865, 2020.
[127] Q. Zhang and S. A. Kassam. Finite-state markov model for rayleigh fading channels. IEEE Trans. Commun., 47(11), 1999.
[128] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. Towards unified int8 training for convolutional neural network. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 1969-1979, 2020.
[129] D. Zordan, T. Melodia, and M. Rossi. On the design of temporal compression strategies for energy harvesting sensor networks. IEEE Trans. Wireless Commun., 15(2):1336-1352, Feb 2016.


[^0]:    ${ }^{1}$ For instance, in a recent study [86], even with optimizations to adapt deep neural networks to low-power spectrum sensing applications, their solution still required at least one 128 -output hidden layer to achieve relatively good performance, and the training phase of their model had to be executed on a powerful GPU.

[^1]:    ${ }^{2}$ The power management action s_on should be interpreted as "stay on" in the "on" state or "switch on" in the "off" state; and s_off should be interpreted as "stay off" in the off state and "switch off" in the "on" state.

[^2]:    ${ }^{3} \mathrm{We}$ assume that the arrivals in each time step are independent and identically distributed; however, the proposed system model can be extended to include Markovian traffic arrivals.

[^3]:    ${ }^{4}$ Although we do not know the realization of the goodput $f^{n}$ until the end of time step $n$, we know the goodput distribution defined in Equation (3.7). This is sufficient to include $f^{n}$ in the definition of the post-decision buffer state.

[^4]:    ${ }^{5}$ PDS learning's action selection and update steps are given in Equation (3.22) and Equation (3.23), respectively, and require calculating $Q^{n}\left(s^{n}, a\right)$ and $Q^{n}\left(s^{n+1}, a\right)$, respectively, using Equation (3.21) for each prospective action.

