Karlsruhe Institute of Technology

Towards Intelligent Data Acquisition Systems with Embedded Deep Learning on MPSoC

Zur Erlangung des akademischen Grades eines
DOKTOR-INGENIEURS (Dr.-Ing.)

von der KIT-Fakultät für Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie (KIT)
angenommene

Dissertation

von

Weijia Wang

Tag der Mündlichen Prüfung:	21.12.2020
Hauptreferent:	Prof. Dr. Marc Weber
Korreferent:	Prof. Dr.-Ing. Dr. h. c. Jürgen Becker

21.12.2021 21.12.2021

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, June 24, 2021

Abstract

Large-scale scientific experiments rely on dedicated high-performance data-acquisition systems to sample, readout, analyse, and store experimental data. However, with the rapid development in detector technology in various fields, the number of channels and the data rate are increasing. For trigger and control tasks data acquisition systems needs to satisfy real-time constraints, enable short-time latency and provide the possibility to integrate intelligent data processing. During recent years machine learning approaches have been used successfully in many applications. This dissertation will study how machine learning techniques can be integrated already in the data acquisition of large-scale experiments. A universal data acquisition platform for multiple data channels has been developed. Different machine learning implementation methods and application have been realized using this system.

On the hardware side, recent FPGAs do not only provide high-performance parallel logic but more and more additional features, like ultra-fast transceivers and embedded ARM processors. TSMC's 16nm FinFET Plus (16FF+) 3D transistor technology enables Xilinx in the Zynq UltraScale+ FPGA devices to increase the performance/watt ratio by 2 to 5 times compared to their previous generation. The selected main processor ZU11EG owns 32 GTH transceivers where each one could operate up to 16.3 Gb/s and 16 GTY transceivers where each of them could operate up to 32.75 Gb/s. These transceivers are routed to x16 lanes Gen $3/4$ PCIe, 12 lanes full-duplex FireFly electrical/optical data link and VITA 57.4 FMC+ connector.

The new Zynq UltraScale+ device provides at least three major advantages for advanced data acquisition systems: First, the 16nm FinFET+ programmable logic (PL) provides high-speed readout capabilities by high-speed transceivers; second, built-in quad-core 64-bit ARM Cortex-A53 processor enable host embedded Linux system. Thus, webservers, slow control and monitoring application could be realized in a embedded processor environment; third, the Zynq Multiprocessor System-on-Chip technology connects programmable logic and microprocessors. In this thesis, the benefits of such architectures for the integration of machine learning algorithms in data acquisition systems and control application are demonstrated.

On the algorithm side, there have been many achievements in the field of machine learning over the last decades. Existing machine learning algorithms split into several categories depending on how the learning phase is organized: Supervised Learning, Unsupervised Learning, Semi-Supervised Learning and Reinforcement Learning. Most commonly used in scientific applications are supervised learning and reinforcement learning.

Supervised learning learns from the labelled input and output, and generates a function that could predict the future different input to the appropriate output. A common application instance is a classification. They have a wide difference in basic math theory, training, inference, and their implementation.

One of the natural solutions is Application Specific Integrated Circuit (ASIC) Artificial Intelligence (AI) chips. A typical example is the Google Tensor Processing Unit (TPU), it could cover the training and inference for both supervised learning and reinforcement learning. One of the major issues is that such chip could not provide high data transferring bandwidth other than high compute power. As a comparison, the Xilinx UltraScale+ FPGA could also provide raw compute power and efficiency for all different data types down to a single bit.

From a deployment point of view, the training part of supervised learning is typically performed by CPU/GPU/TPU on a fixed dataset. For reinforcement learning, the training phase is more complex. The algorithm needs to periodically interact with the controlled system and execute a Markov Decision Process (MDP). There is no static training dataset, but it is obtained in real-time. The time slot between each step depends on the dynamics of the controlled system. The inference is also bound to this sampling time because the algorithm needs to interact with the environment and decide the appropriate action for a response, then a higher demand on time is proposed.

This thesis gives solutions for both training and inference of reinforcement learning. At first, the requirements are analyzed, then the algorithm is deduced from scratch, and training on the PS part of Zynq device is implemented, meanwhile the inference at FPGA side is proposed which is similar solution compared with supervised learning. The results for Policy Gradient show a lot of improvement over a CPU/GPU-based machine learning framework. The Deep Deterministic Policy Gradient also has improvement regarding both training latency and stability. This implementation method provides a low-latency approach for reinforcement learning on-field training process.

Zusammenfassung

Große wissenschaftliche Experimente basieren auf dedizierten Hochleistungsdatenerfassungssystemen, um experimentelle Daten auszuprobieren, auszulesen, zu analysieren und zu speichern. Mit der rasanten Entwicklung der Detektortechnik in verschiedenen Bereichen steigen jedoch die Anzahl der Kanäle und die Datenrate. Für Trigger- und Steuerungsaufgaben müssen Datenerfassungssysteme Echtzeit-Einschränkungen erfüllen, Kurzzeitlatenz ermöglichen und die Möglichkeit bieten, intelligente Datenverarbeitung zu integrieren. In den letzten Jahren wurden Machine Learning-Ansätze in vielen Anwendungen erfolgreich eingesetzt. In dieser Dissertation wird untersucht, wie maschinelle Lerntechniken bereits in die Datenerfassung von Großexperimenten integriert werden können. Es wurde eine universelle Datenerfassungsplattform für mehrere Datenkanäle entwickelt. Mit diesem System wurden verschiedene Implementierungsmethoden und Anwendungen für maschinelles Lernen realisiert.

Auf der Hardwareseite bieten die jüngsten FPGAs nicht nur eine leistungsstarke parallele Logik, sondern immer mehr zusätzliche Funktionen wie ultraschnelle Transceiver und eingebettete ARM-Prozessoren. Die 16nm FinFET Plus (16FF+) 3D-Transistor-Technologie von TSMC ermöglicht es Xilinx in den Zynq UltraScale+ FPGA-Geräten, das Leistungs-Watt-Verhältnis im Vergleich zur vorherigen Generation um das 2- bis 5-fache zu erhöhen. Der ausgewählte Hauptprozessor ZU11EG besitzt 32 GTH-Transceiver, bei denen jeder von ihnen bis zu 16,3 GB/s und 16 GTY-Transceiver betreiben kann, bei denen jeder von ihnen bis zu 32,75 USD Gb/s betreiben kann. Diese Transceiver werden an x16-Lanes Gen 3/4- PCIe, 12- und -duplex-Vollduplex-Verbindung FireFly und VITA 57.4 FMC+-Anschluss geroutet.

Das neue Zynq UltraScale+ Gerät bietet mindestens drei wesentliche Vorteile für fortschrittliche Datenerfassungssysteme: Erstens bietet die 16nm FinFET+ programmierbare Logik (PL) Hochgeschwindigkeits-Auslesefunktionen durch Hochgeschwindigkeits-Transceiver; zweitens, integrierter Quad-Core 64-Bit ARM Cortex-A53 Prozessor ermöglichen Host Embedded Linux System. So konnten Webserver, langsame Steuerungs- und Überwachungsanwendungen in einer eingebetteten Prozessorumgebung realisiert werden. drittens verbindet die Zynq Multiprozessor System-on-Chip-Technologie programmierbare Logik und Mikroprozessoren. In dieser These werden die Vorteile solcher Architekturen für die Integration von Machine Learning-Algorithmen in Datenerfassungssysteme und Steuerungsanwendungen demonstriert.

Auf der Algorithmusseite gab es in den letzten Jahrzehnten viele Erfolge auf dem Gebiet des maschinellen Lernens. Bestehende Machine Learning-Algorithmen gliederten sich in mehrere Kategorien, je nachdem, wie die Lernphase organisiert ist: Supervised Learning, Unsupervised Learning, Semi-Supervised Learning und Reinforcement Learning. Am häufigsten in wissenschaftlichen Anwendungen verwendet werden, sind überwachtes Lernen und Verstärkung lernen.

Überwachtes Lernen lernt von der beschrifteten Eingabe und Ausgabe und generiert eine Funktion, die die zukünftigen unterschiedlichen Eingaben für die entsprechende Ausgabe vorhersagen könnte. Eine gemeinsame Anwendungsinstanz ist eine Klassifizierung. Sie haben einen großen Unterschied in der grundlegenden mathematischen Theorie, Ausbildung, Rückschlüsse,

und ihre Umsetzung.

Eine der natürlichen Lösung ist Application Specific Integrated Circuit (ASIC) Artificial Intelligence (AI)-Chips. Ein typisches Beispiel ist die Google Tensor Processing Unit (TPU), die die Ausbildung und Rückschlüsse sowohl für das überwachte Lernen als auch für das Verstärkungslernen abdecken könnte. Eines der Hauptprobleme ist, dass ein solcher Chip keine hohe Datenübertragungsbandbreite bieten konnte, außer hoher Rechenleistung. Zum Vergleich: Der Xilinx UltraScale+ FPGA könnte auch eine rohe Rechenleistung und Effizienz für alle Datentypen bis auf ein einziges Bit bieten.

Aus Sicht der Bereitstellung wird der Trainingsteil des überwachten Lernens in der Regel von CPU/GPU/TPU für ein festes Dataset durchgeführt. Für das Verstärkungslernen ist die Trainingsphase komplexer. Der Algorithmus muss regelmäßig mit dem kontrollierten System interagieren und einen Markov Decision Process (MDP) ausführen. Es gibt kein statisches Trainings-Dataset, aber es wird in Echtzeit abgerufen. Das Zeitfenster zwischen den einzelnen Schritten hängt von der Dynamik des gesteuerten Systems ab. Die Schlussfolgerung ist auch an diese Samplingzeit gebunden, da der Algorithmus mit der Umgebung interagieren und die geeignete Aktion für eine Antwort entscheiden muss, dann wird eine höhere Zeitanforderung vorgeschlagen.

Diese These liefert Lösungen sowohl für die Ausbildung als auch für die Rückschlussung des Verstärkungslernens. Zuerst werden die Anforderungen analysiert, dann wird der Algorithmus von Grund auf neu abgeleitet, und die Ausbildung auf dem PS-Teil des Zynq-Geräts wird implementiert, während die Schlussfolgerung auf FPGA-Seite vorgeschlagen wird, die eine ähnliche Lösung im Vergleich zum überwachten Lernen ist. Die Ergebnisse für Richtliniengradient zeigen eine Menge Verbesserungen gegenüber einem CPU/GPU-basierten Machine Learning Framework. Der tiefer deterministischer Richtliniengradient hat auch Verbesserungen in Bezug auf Trainingslatenz und Stabilität. Diese Implementierungsmethode bietet einen Ansatz mit geringer Latenz für den Schulungsprozess für die Verstärkung des Lernens vor Ort.

Acknowledgments

Upon the completion of this thesis, I am grateful to those who have offered me encouragement and support during the 4 years of my study and work. Firstly, the acknowledgement is given to Prof. Marc Weber, who offered me the opportunity to start the Ph.D. at IPE KIT, and gave me a lot of suggestions during Ph.D. working and thesis writing. I would like to give my heartfelt gratitude to my supervisor Dr. Michele Caselle. During the four years of study, he gave me incalculable valuable guidance, and helped me improve a lot in knowledge, engineering practice ability and working ability. Working with him and debugging hardware system is always a nice experience. I wish to show my gratitude to Dr. Andreas Kopmann, who gave me a lot of suggestion and help during all of my conference, paper and thesis writing. The quality of work has improved a lot under his help. I also like to give my appreciation to my colleagues: Timo Dritschler, Meghana Patil, Luis Eduardo Ardila Perez, Nick Karcher, Oliver Sander and many other IPE colleagues. I like the mutual help atmosphere here at IPE and working together for the same projects. I also thank my colleagues from IBPT and LAS: Tobias Boltz, Dr. Miram Brosi, Edmund Blomley, Andrea Santamaria Garcia. They gave me a lot of support during the Ph.D. stage. I am also greatly indebted to the professors and teachers at the Department of Electrical Engineering and Information Technology. I have learned a lot at KIT.

Contents

Chapter 1

Introduction

With the success of deep learning algorithms and the sharply increasing number of applications, the competition on the ideal processing architectures has been re-opened again. For many years, the x86 architecture[1] was the de-facto standard. Other architectures have been presented several times but have not been considered. The effort and cost to migrate existing code was considered too high. Especially the validation of approved implementations made researchers stick to architectures and their corresponding programming languages of the last millennium. Artificial intelligence seems to be a game-changer now which provides an novel processing method and more clearly, the novel algorithms. Meanwhile, dedicated AI processors are proposed and integrated nearly everywhere: from mobile embedded processors to FPGAs[2] and GPUs[3].

In parallel to the spread on artificial intelligence in science, industry and society, the ARM platform[4] has reached a performance level that can compete with the traditional x86 CPU architecture. ARM Holdings develops the architecture and licenses it to other companies. ARM processors have been used in Apples mobile phones and tablets for years and increased drastically in performance. From next year on, ARM architectures will also be used in Apple computers. As a proof, in year 2020, the Fujitsu A64FX, an Armv8.2-A+SVE based architecture is used as supercomputer architecture and shows a relatively low execution time and high performance compared with Marvell (Cavium) ThunderX2 processor and Intel Xeon Skylake processor in [5].

Today's power consumption demands alternatives to the x86 architecture. The energy consumption of computing centres worldwide in the year 2017 has reached 350 billions of kWh per year [6]. The ARM and FPGA both show the potential of low power consumption but the same performance by comparison with CPU and GPU. The ARM cluster is more efficient for a large dataset than GPU [7]. [8] illustrate that FPGAs overperform GPUs in energy efficiency measured in Multiply-accumulate operation.

Dedicated AI-based algorithms, accelerator units like ARM processors and FPGAs are able to reduce the aforementioned demand by at least an order of magnitude. This thesis will investigate the impact of these probably disruptive technologies on Data Acquisition (DAQ) Systems. A promising platform for DAQ applications is enabled by recent System on Chips (SoCs), where the programmable logic combines with embedded ARM processors. They seem to be ideal to evaluate the architectures for AI-enabled applications in science and evaluate the estimated performance increase. The thesis will demonstrate how to implement different AI application on DAQ, how to allocate the AI tasks on different computing architectures within one SoC, and the results will be illustrated.

1.1 Advanced data acquisition system

The quality of data determines the quality of scientific results and thus the success of an experiment. Data acquisition (DAQ) systems are essential to realizing high-quality results. DAQ system span a wide bridge of technologies form front-end electronics to data processing and archival. The term "data acquisition" refers to many measurement applications, all of which require some form of characterization, monitoring, or control. Regardless of the specific application, the purpose of general data acquisition systems is to measure physical parameters or different kinds of signal (for example, optical, mechanical, thermal, electrical, magnetic, chemical, etc.), or to use actuators generate specific actions based on the data received. Two primary missions of the DAQ systems are *data transmission* and *data processing*. The competence of *data transmission* includes the DAQ system can receive the data and handle the data (e.g. high speed and high bandwidth data). The *data processing* means the DAQ system is able to interpret the data and react to it. The major challenges to recent DAQ systems are:

1) To support high data throughput acquisition and enable data processing with low-latencies;

2) To provide the flexibility to adopt to changing experimental phases and to cover the dynamical development process during the construction of large-scale experiments;

3) To realize complex data processing algorithms to make DAQ systems intelligent.

These three challenges will then be discussed in the following and solutions will be presented. First, the large scale physical experiment detectors have ever-increasing time, spatial, and energy resolution, and produce the data volume that previous generation sensors and DAQ system could not match. This requires that the DAQ systems in used need to support high-throughput data in the real-time range from front-end electronics to FPGAs, and also should include or connect to post-processing components (e.g. CPU/GPU clusters). The DAQ system could be configured as a transfer station, or process the data as an end-point. This requires the DAQ system owns different and multiple communication channels, and this could enable the DAQ system to be configured to the different topology. Some physics experiment need feedback within a short time frame, for example, the beam control system at the Karlsruhe Research Accelerator (KARA) [9, 10, 11]. Thus the DAQ needs to react to the incoming data with low-latency shorter than the sampling time.

- Which devices are able to provide high throughput and low latencies? How to optimize the latency?

Normally, the answer will be the Field Programmable Gate Array (FPGA). The leading edge FPGA [12] could provide dozens of high bandwidth configurable transceivers for different high-speed data bus protocols. The execution model at FPGA is free for construction: parallel, pipeline, and data flow, which is different from the instruction model on CPU/GPU. This enables FPGA a possibility to build low latency data processing.

For the beam diagnostics systems, the FPGA-GPU heterogeneous [13, 14] system developed at the Institute for Data Processing and Electronics (IPE) plays an important role, providing a multi-Gigabyte per second real-time data transmission. In this heterogeneous system, one of the key components is a Direct Memory Access (DMA) engine [15], so-called "KIT DMA". That is fully compatible with Xilinx FPGA families 6 and 7, UltraScale+ device. The DMA is PCIe Generation 2, 3, 4 compatible. The Xilinx Virtex-7 based PCIe FPGA board ("Hi-Flex") used in [14, 15]. Fig (1.1) demonstrated the data throughput measurement by KIT DMA and it has a throughput larger than 6.5 GB/s in these 3 different transfer modes. This also provides very low latency transmission between FPGA and GPU. By "GPUDirect (NVIDIA)", the average transfer latency is lower than 2 µs. By "DirectGMA" from AMD, the latency between Hi-Flex and AMD FirePro W9100 is lower than 1.3 µs.

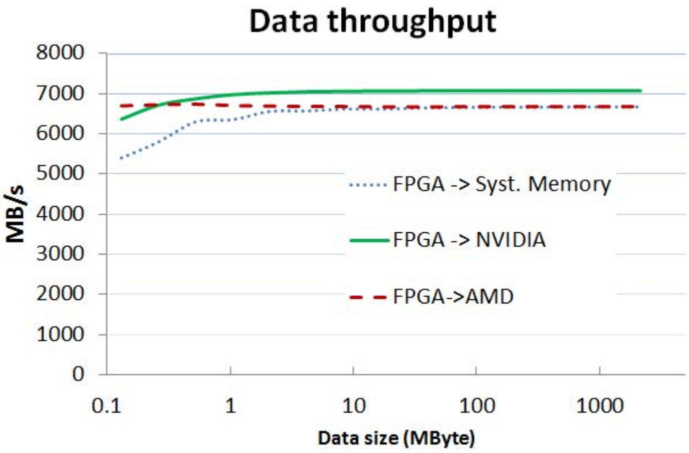


Figure 1.1: Data throughput measurements by KIT DMA to system memory, NVIDIA Tesla K40 and AMD FirePro W9100. [13]

Secondly, the demanded flexibility of DAQ systems means they could serve as a *general-purpose* tool, with dedicated processing units according to the problem to be solved. Because FPGA has the capacity for various hardware implementation structure, it could dynamically be reprogrammed with a data path that matches a dedicated processing/specific workload. Thus FPGA performs better for specific applications than general-purpose application processors like CPU/GPU. On the other hand, for general purpose applications, for example, a general calculation or task without strict time tolerance could be implemented on such an application processor. Some general embedded processors are also able to support a flexible control over front-end analogue-to-digital converter (ADC) or other peripherals. The state and data inside FPGA also demand a conveniently and flexibly control. Several questions could ask as follows for these additional functionalities:

- Which device could provide such flexibility and support embedded operating system?
- Is there another chip is required in the hardware design? Is there a solution to reduce the layout of the printed circuit board (PCB) if two chips on one board?
- Which tools are available to simplify the design effort and help to realize efficient communication between FPGA and it's embedded processor?

The third and also most demanding requirement is to realize complex tasks within DAQ systems. The first and second requirement serves as a prerequisite for the third requirement: complex task accomplishment. With the deepening of the research and the improvement of the front-end data volume, the new detector/accelerator may have different and more complex task than before. For some of these tasks, it is very hard to write the program or implement it on the hardware. For example, a problem like recognizing a multi-dimensional object in a cluttered scene. Such a problem could be the track finding through hundreds of hits at one event [16] at collider experiments to construct a program to recognize the track is difficult because one doesn't know how physics experts recognize the track in their brain. Or even there was already one solution but a *sophisticated algorithm* is required. One simple method may perform well for one single event but the track form and style are changing frequently per event, a generalized approach for recognization is required. Besides the recognization problem, it is a similar situation for the control problem. At KIT's research synchrotron KARA, a micro-bunching control loop is under development [17]. A formation of microstructures and CSR fluctuations caused by self-interaction is *tremendously complex*, and even **no** proper solution or traditional control methods to generate a longitudinal RF feedback to the RF cavity.

- Which device is suitable to cover many different kinds of machine learning approaches?
- Is there some general tool to transfer the Machine Learning (ML) model from ML experts to engineers?
- How to allocate the different machine learning tasks on DAQ?

Machine learning (ML) will not replace the traditional approach in every case. But in some applications, take the two above mentioned applications as examples, due to the complexity of the problem, it may be the only solution to the problem. In the field of High Energy Physics (HEP), machine learning becomes a novel data analysis approach and many applications have been reported by white paper [18].

For other situations, machine learning algorithms also could be regarded as a good alternative. The traditional data processing methods may need sophisticated professional knowledge of the application field, and a complex parameter adjustment process. Moreover, each method is targeting a dedicated application, therefore it has poor generalization ability and robustness. Deep learning is mainly based on data-driven feature extraction. Training on a large number of samples, deep and specific feature representation of the data set can be obtained, which makes the expression of the data set more efficient and accurate. That has lowered barriers to entry in some industries.

1.2 Machine Learning

As Deep Learning (DL) is a branch of Machine Learning (ML), this thesis, will only use the term *Machine Learning* because some of the method mentioned do not belong to deep learning.

Machine learning methods could be divided into three major paradigms: *Supervised Learning*, *Unsupervised Learning Semi-supervised Learning* and *Reinforcement Learning*. These algorithms are addressing different problems.

This thesis will focus on *Supervised Learning* and *Reinforcement Learning*, which is the most prevalently used approaches. The *Supervised Learning* (SL) is learning a model from the training dataset of labelled examples by machine learning engineer, and use the model for inference. Some of the typical tasks could be a classification or regression problem. On the other side, *Reinforcement Learning* (RL) is designed to maximize long-term future rewards. The algorithm aims to find an optimal control under the finite Markov Decision Processes (MDPs). Both SL and RL have *training* stage and *inference* stage. The *training stage* is how the model or algorithm becomes clever, no matter which approach it is to use. The *inference* is to use the **trained** model/algorithm to solve the reality problem.

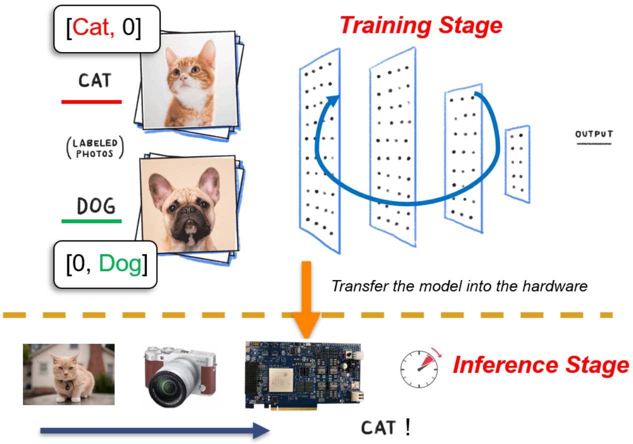


Figure 1.2: The training and inference stage in supervised learning are normally independent. The training stage is realized often on GPUs/CPUs due to a floating-point calculation and machine learning framework availability, while inference can be implemented also on hardware accelerators.

1.2.1 Supervised Learning

As shown in Fig (1.2), the label cat or dog needs to be prepared before the training stage. The model will try to extract the pattern hidden behind the training data set with its corresponding label. Once the training stage is finished and the model is verified, the parameters (e.g. the weights and biases in Neural Network, Convolutional Network or Recurrent Neural Network) is keep fixed. Then it validated in an inference stage, for instance, an image recognization task, only the forward pass of the network is required to implement on the hardware. After the verification, if the model satisfies the requirement, the model will be transferred to the hardware for the inference. The inference hardware could be FPGA/GPU/CPU/ARM and/or any other hardware devices. For realization in DAQ systems, several questions need to be answered:

- Which device is more suitable for the inference stage of SL?
- Is it obliged to deploy also the training process on the hardware for supervised learning applications?
- Are there any tools to help to decrease the developing time to transfer the model from the machine learning environment to the hardware platform?
- Is there a suitable device or platform that could cover as much supervised learning as possible?

1.2.2 Reinforcement Learning

As to Reinforcement Learning, the model is established through interaction with the system that needs to be controlled. Thus, in this case, there is no training data like supervised learning required. The training data is generated online by interaction with the system to be modelled. There are three basic components in RL: agent, action, and environment (normally a state signal representing the environment). An agent is a decision-maker, and that means generate proper action at each step. Then the agent action will be sent to the environment. The environment reacts to the actions and presents a new state to the agent shown in Fig (1.3). After this step, the agent will collect the interaction information and use such information to learn.

The rules of *Go-playing* are simple and elegant, but the ways of playing vary largely. It takes a lot of practice and studies to master its connotation. Go is considered to be one of the most complex board games in the world. Writing a program that possesses the capability of strategy, memory, observation, judgement and counting is extremely difficult. Therefore to set up the reinforcement learning framework and to let the agent learning directly from the interaction with the game is an attractive alternative. The immediate or long-term wining or failure indication will lead the machine learning to become more "clever". The agent AlphaGo defeated the human European Go champion by 5 games to 0 with reinforcement learning together with a tree search algorithm [19]. As mentioned earlier, reinforcement learning is acquiring training data during the interaction stage. This implies that the inference and training process needs to be considered as one unit. Several questions need to be answered:

- Which device is suitable to implement reinforcement learning with both the inference stage and training stage?
- Is it necessary to deploy the training process also on the hardware?
- If required, are there tools to help decrease the developing time that transferring the training process from reinforcement learning engineering to electronic engineering?
- Is there a suitable device or platform that is able to cover most of the advanced reinforcement learning algorithms?

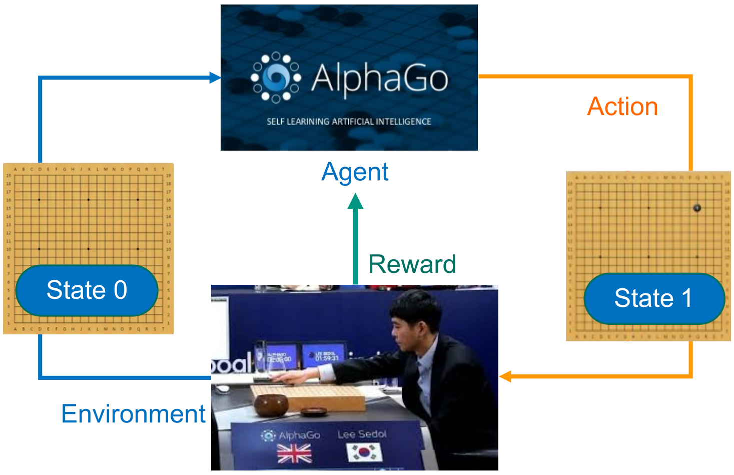


Figure 1.3: Go is an abstract strategy board game for two players [20]. Under the definition of reinforcement learning concept [19], one player act as an agent, the other one is its opponent, acting as the environment. The Go play continues in a way that agent and opponent give their movements one after each. Once the opponent settles the stone, then the state (the location of each stone on the board) is changed. After the agent receives the state signal, it will generate the action for itself. The environment will change according to such movement. During each step, the reward signal also generates from the environment, that guide the agent to learn from each step.

1.3 Machine Learining applications

Machine learning (ML) is the most common method in Artificial Intelligence (AI). In the field of High Energy Physics (HEP), machine learning methods have been used for data analysis. Many tools have been presented since 2010. The following examples demonstrate the success of those applications in SL and RL.

In High-energy Physics after the collision of high eneretic particles, beams of secondaries are generated. Jets denotes high-energetic, directed beams of particles. The main task of Jet Physics is to identity the various particles after collision. For jet multiclass classification problems at the Large Hadron Collider (LHC) at Conseil Européen pour la Recherche Nucléaire CERN [21]. Convolutional Neural Network (CNN) are used to discriminate quark and gluon jets The Compact Muon Solenoid (CMS) [22] collaboration at LHC considers using convolutional layers and Recurrent Neural Network (RNN). The Graphic Neural Network (GNN) are also used for the analysis of jet substructures [23].

The heavy-ion experiments follows the theory of Quantum Chromodynamics (QCD). The QCD have two different transition: a crossover [24] or first order [25]. The CNN is used to identify two different equation of state (EoS) [26]. The PANDA detector (antiProton ANnihilation at DArmstadt) [27] is a multi-purpose detector system. The task of track finding aims to identify particles belonging to the same track. The neural network is used for hit aggregation in [16], and a improved GNN method in [28].

The emission of Coherent Synchrotron Radiation (CSR) depends on the shortness of the electron bunches. The self-interaction of the bunch causes micro-bunching instability. The Karlsruhe Institute of Technology (KIT) storage ring Karlsruhe Research Accelerator (KARA) aims at the stabilization of the emitted THz radiation to tackle the problem of micro-bunching instability. A reinforcement learning (RL) based method is planned to be implemented through an RF feedback system located at storage [17].

1.4 Accelerating Machine Learning

To use machine learning in DAQ systems, often real-time requirements need to be fulfilled. Accelerated Machine Learning is a topic that reached out beyond scientific applications, including industrial use-cases. How the algorithm is been accelerated firstly depends on the algorithm itself, and its corresponding problem set to be solved. Secondly, the precision, power, throughput, latency and cost required by application [29].

If one combines two different stages (training and inference) with two different methods (reinforcement learning and supervised learning), there will be four components that have the possible requirement for implementation and acceleration:

1. supervised learning training process
2. supervised learning inference process
3. reinforcement learning training process
4. reinforcement learning inference process

Not always all four implementations are required. Each algorithm results in a different implementation strategy. Normally, for both supervised learning and reinforcement learning, the training process is not needed any more during the inference procedure cause the model has already been verified by the test data set. Especially for supervised learning, because the training process always happens off-line, therefore a natively long preparation time for data collection, model fine-tuning happens in the supervised learning based engineering.

For reinforcement learning, however, the consideration will be slightly more complex. First, the inference process part must be included because it will be the final solution. Whether the learning process is needed is depends on the problem set. For example, in autonomous driving, the tasks in [30] are mainly divided into two different tasks, the test task and the training task. During the test task, the episode is running with a fixed optimal policy, and that is a pure inference stage. But in the training task, it chooses a task with a noisy policy that will randomly choose some other non-optimal policy to explore the action space. Because the learning stage is required happening in the field, thus, for a hardware implementation, needs to deploy both the inference stage and the training stage. The training stage for supervised learning is already difficult, moreover, because some of the prevalent RL algorithm frameworks is more complex than supervised learning, the reinforcement learning training process needs much more computation and control than supervised learning. The training process required the single/double precision Floating Point Unit (FPU) to guarantee the computational accuracy of the training process. This again requires a capable application processor on the DAQ system.

For accelerated machine learning the questions listed above need to be refined.

1. Which Algorithm will be accelerated? Supervised Learning (SL) or Reinforcement Learning(RL)?

2. Is there any difference between SL acceleration and RL acceleration?

3. The weights of the model(SL or RL) is fixed or need to be updated to adapt to the changing environment?

4. Which part should be put into the hardware, training process, inference part or both?

5. How to accelerate the SL inference part?

6. How to accelerate the RL inference part?

7. Is there any similarity between SL and RL inference?

8. How to process the RL training process on application processor?

9. Which main chip will be selected for SL and RL? FPGA, GPU, CPU, ARM or SoC?

1.5 Hardware platform

High-performance DAQ systems are today based on FPGAs. They provide a variety of high-throughput links and an enormous amount of logic cells. In recent years, embedded processors have been added to FPGA devices. The combination of both units makes FPGA-based DAQ boards an ideal platform for machine learning integrated with high bandwidth data acquisition and control. But only the latest FPGAs using the ARM architecture possess the potential to own adequate ability of application performance. There is a wealth of portable software, drivers and modules under the ARM-based embedded Linux system. Thus it provides both relatively powerful computing power and control capability. Especially for machine learning, FPGA could achieve low latency and high bandwidth for inference. For some of the small networks, the ARM could undertake the responsibility of training where will show in Chapter 7, 8, and 9.

The first embedded processors have already presented in the early 2000 years. Among the first heterogeneous architecture are the Virtex-4 FX family [31], Virtex-5 FXT families [32]. The Virtex-5 FXT is a high-performance embedded system. The FPGA logic is one to two-speed grade improvement over Virtex-4 devices. It owns PowerPC 440 (PPC440) cores embedded with FPGA logic. One of the major drawbacks of this generation system-on-chip is it required a hardware engineer to program the FPGA part first. Then the peripherals, memory systems, including the data path across processor and logic, is required for the design. Finally, the processor could be programmed. Xilinx delivered the Zynq-7000 All Programmable SoC [33] in the winter of 2011. This enables the developer to jump directly to the hardware-software co-design, both hardware development and software development are independent and can happen simultaneously then they are combined by the system engineer.

It already improves the embedded processor from PPC440 to Coretex-A9 (Zynq-7000), thus resulting in a performance improvement from 1.0 to 2.5 DMIPS. Then comes the Zynq UltraScale+ equipped with Cortex-A53 which increase this performance to 3.45 DMIPS, enabling a powerful application processor on the DAQ.

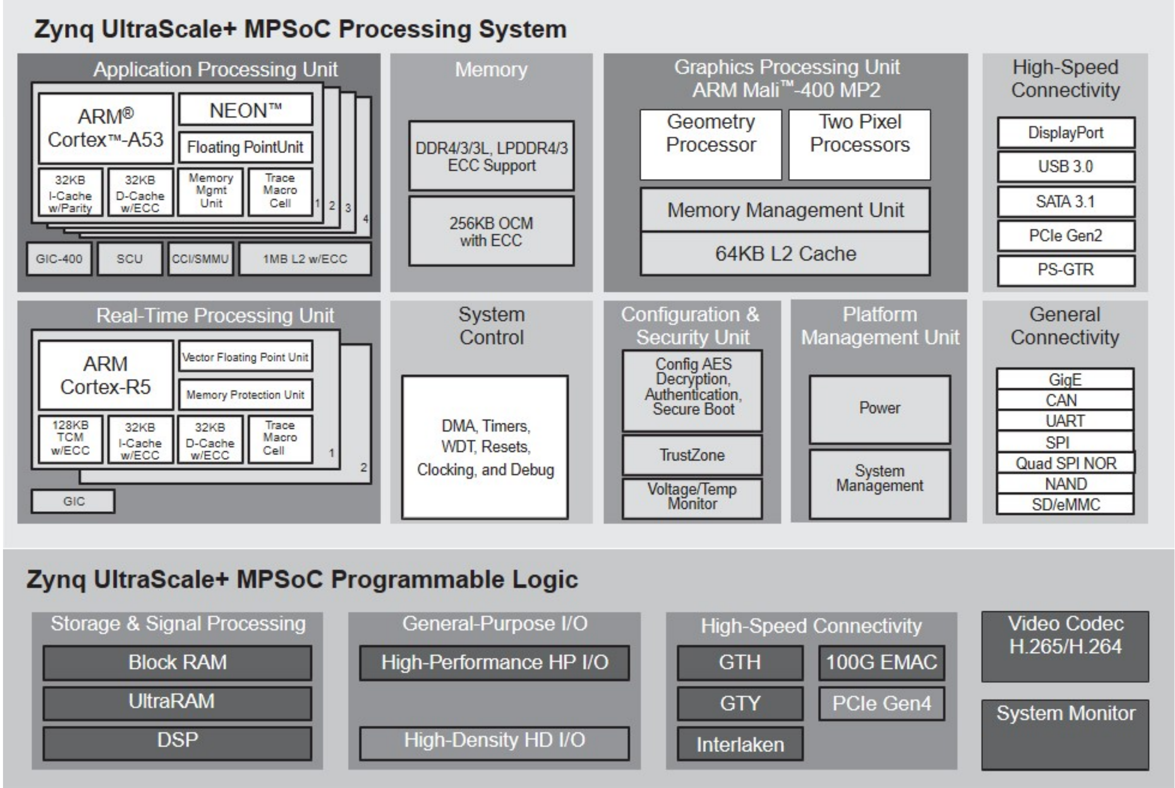


Figure 1.4: Zynq UltraScale+ MPSoC block design

Xilinx UltraScale+ MPSoC architecture family of products [12] integrates a feature-rich 64-bit quad-core or dual-core Arm Cortex-A53, dual-core Arm Cortex-R5 based processing system (PS) and Xilinx programmable logic (PL) UltraScale architecture in a single device. At the PS part, the embedded Linux system could be built and for data acquisition, processing and general applications. On the other hand, at the PL part, the DMA could be integrated into the FPGA and can communicate with GPU directly with high data bandwidth. As shown in Fig (1.4), at the PS part of the Zynq UltraScale+ device, the ARM Cortex-A53 and real-time ARM Cortex-R5 could provide a powerful embedded Linux system and/or bare mental application. These processors have multiple access method to soft intellectual property (IP) core located at PL side through rich internal buses. This heterogeneous MPSoC saves PCB wiring resources and DAQ system development time.

Demonstrated in Fig (1.4), the top part of the picture is the processing system of Zynq. It is consists of an application processing unit, real-time processing unit, graphics processing unit, memory controller, and connectivity for abundant peripherals. At PS part the Zynq UltraScale+ device, the ARM Corex-A53 and real-time ARM Cortex-R5 could provides a powerful embedded Linux system and/or bare mental application. These processors have multiple access method to soft intellectual property (IP) core located at PL side through rich internal buses. This heterogeneous MPSoC save PCB wiring resources and DAQ development time.

The bottom side of the picture of block design is the programmable logic of Zynq. It contains system logic cells, look-up-table (LUTs), Digital Signal Processing (DSPs), block RAM, UltraRAM, general-purpose inputs/outputs and many high-speed connections.

1.6 Chapter arrangement

This dissertation has two parts. The first part is the hardware design. Within which Chapter 2 is the design and test of the novel multipurpose PCIe readout card (HighFlex2); Chapter 3 demonstrates the works around system-on-chip and embedded Linux system. The second part of the dissertation covers artificial intelligence implementation methods and its application based on HighFlex2. Chapter 4 will give an introduction to machine learning, basic mathematic theory in supervised learning, and its acceleration methods; In Chapter 5, the Xilinx DPU is implemented as a solution for supervised learning inference on HighFlex2; Chapter 6 introduce the major knowledge required in the dissertation; Chapter 7 demonstrate the efforts to develop the reinforcement learning (Policy Gradient) training process on HighFlex2 and its testbench; Chapter 8 shows the Deep Deterministic Policy Gradient implementation method on HighFlex2; Chapter 9 is the overall reinforcement learning design at KARA; Chapter 10 is the conclusion.

Part I

Data Aquisition System Hardware Design

Chapter 2

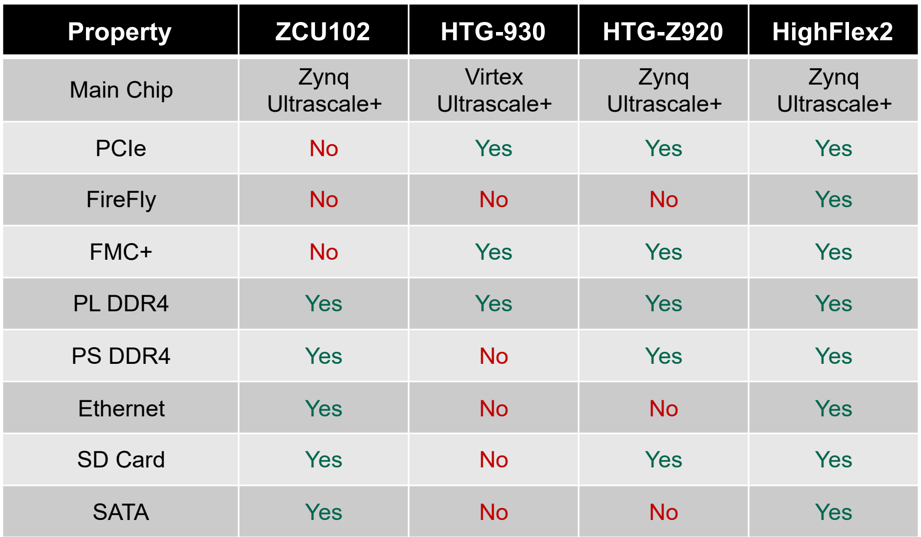
Hardware design of the readout card

This chapter focuses on the hardware design of a PCIe readout card based on the recent Zynq-MPSoC programmable family. For large-scale real-time data process, the FPGA-GPU combination is considered as one suitable solution [34]. For many years, the Institute for Data processing and Electronics (IPE) has been involved in the development of heterogeneous FPGA-GPU systems for the processing of the data in real-time. As an upgraded version of the existing readout card (so-called "HighFlex" [35]) to meet the needs of high-speed data processing and artificial intelligence, a major feature provided by HighFlex2 is its compatibility with the PCIe standard bus.

There are several commercial development platforms on the market, the HTG-930 [36] and HTG-Z920 [37] from HiTech Global, Xilinx ZCU102 [38], ZCU104 [39] and ZCU106 [40]. Unfortunately, none of the evaluation boards meets all the requirements required by the projects, such as electrical/optical interfacing, data throughput, flexibility, etc. Therefore, a custom readout card based on PCIe generation 3/4 with 16 lanes has been developed.

For example, the ZCU102 includes a PCIe Generation 2 with x4 lanes connected to the ARM Processor Subsystem (PS) by dedicated GTR transceivers which support data rate up to 6 Gb/s. While the custom readout card is based on PCIe generation 3/4 with 16 lanes, which allows managing a data throughput of over 200 Gb/s in continuous readout mode. The CPU/GPU platform is the data destination, therefore the readout card is a PCIe *Endpoint* device, the PCIe Gen3/4 also needs to connect with the PL part of the Zynq device where PCIe Integrated Block [41] is supported. The HTG-930 and HTG-Z920 support x8 PCIe Gen4 or x16 PCIe Gen3. However, the HTG-930 is mounted with Xilinx Virtex UltraScale+ [42] device, which is not an MPSoC. Therefore, it is not fully suitable to guest multiple machine learning implementations and an embedded operating system. The HTG-Z920 is based on a Xilinx MPSoC UltraScale+ PCIe platform but several important interfaces are not available, like the Serial Advanced Technology Attachment (SATA), Ethernet, and in terms of flexible high-speed data transmission, FireFly [43] connection.

The comparison between the available evaluation boards and the HighFlex2 is shown in Fig (2.1). The last reason is scalability. The chip mounted on the ZCU102 is equipped with Zynq UltraScale+ XCZU9EG-2FFVB1156E MPSoC and both ZCU104 and ZCU106 have Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC on top. But such devices do not integrate the PCIe hard-core and the necessary number of DSP slides that are required for the corrected implementation of the ML network on FPGA. The selected Zynq is UltraScale+ XCZU11EG-1FFVC1760 MPSoC. As shown in Fig (2.3), the direct comparison between the ZU7EG and the ZU9EG, mounted on the evaluation boards, and the selected Zynq device ZU11EG shows as the selected device integrates up to 4 PCIe hard-core logics, a large number of DSP slices



Property	ZCU102	HTG-930	HTG-Z920	HighFlex2
Main Chip	Zynq Ultrascale+	Virtex Ultrascale+	Zynq Ultrascale+	Zynq Ultrascale+
PCIe	No	Yes	Yes	Yes
FireFly	No	No	No	Yes
FMC+	No	Yes	Yes	Yes
PL DDR4	Yes	Yes	Yes	Yes
PS DDR4	Yes	No	Yes	Yes
Ethernet	Yes	No	No	Yes
SD Card	Yes	No	Yes	Yes
SATA	Yes	No	No	Yes

Figure 2.1: Comparison between HighFlex2 readout card and the available commercial boards.

(up to 3000), which are necessary for ML implementation, and a large number of both GTY and GTH serial high-speed transceivers, which are necessary for the implementation of the fast optical and electrical interfaces (e.g. PCIe x16). Moreover, the selected FPGA package FFVC1760 allows the migration that enables the re-mounting of the other high-end ZU17EG and ZU19EG devices on the HighFlex2 card.

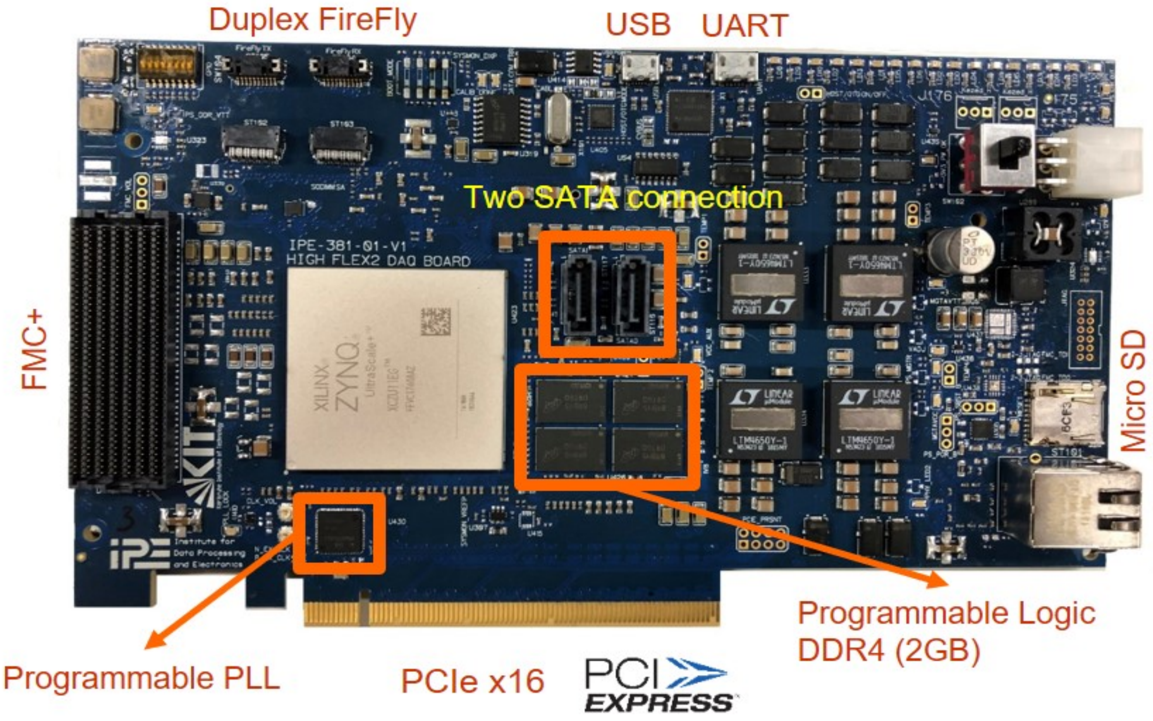
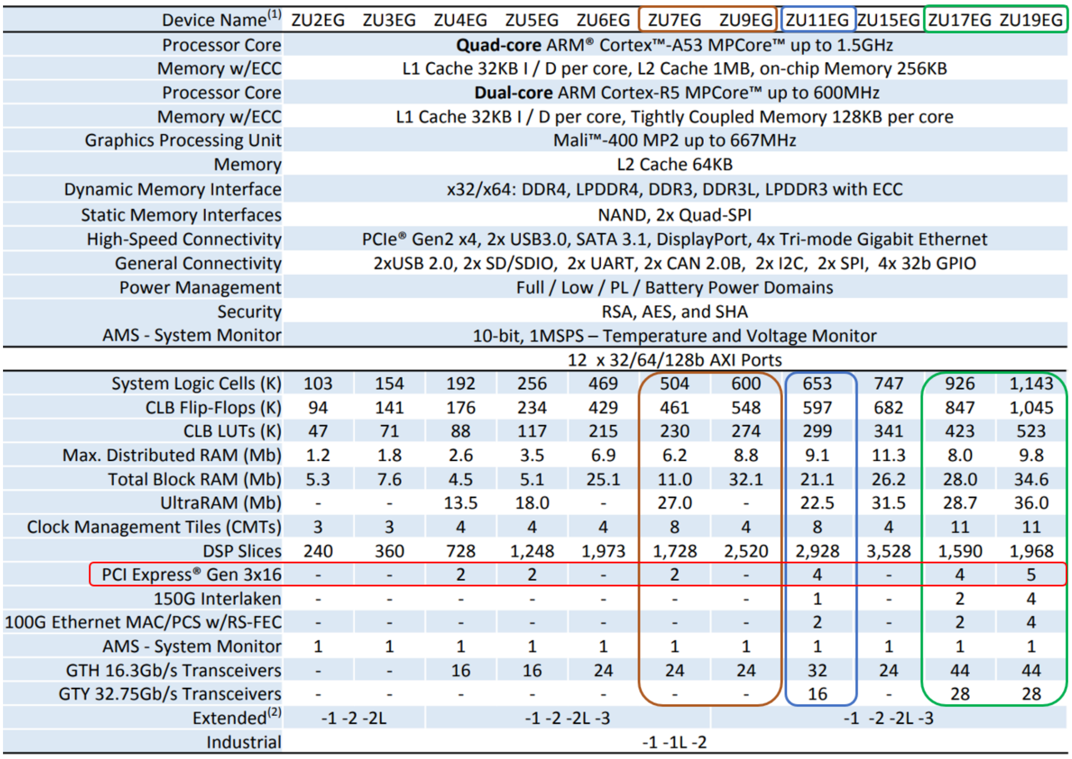


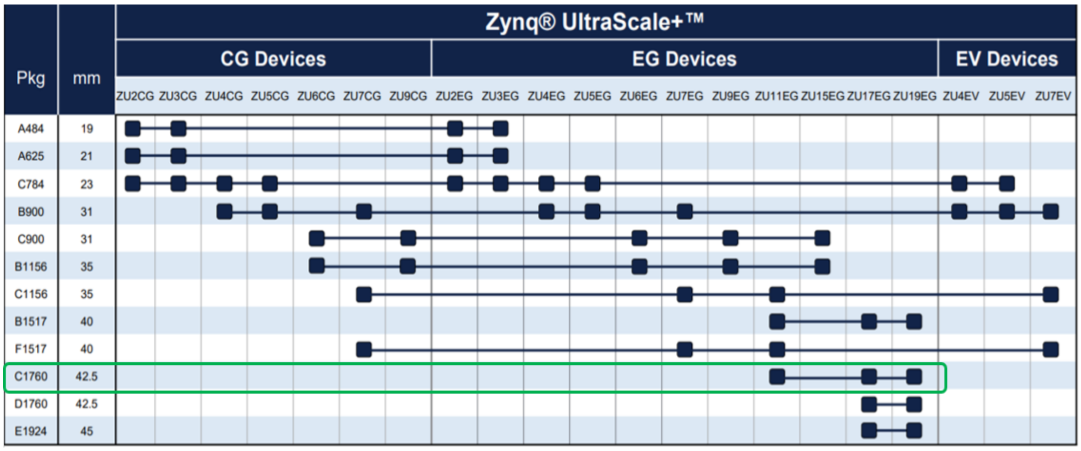
Figure 2.2: Top side of HighFlex2 DAQ card

In Fig (2.2), the fully assembled HighFlex2 DAQ card is shown. **All** components on the board are tested and verified. The main processor on the centre to the left of the board is Zynq UltraScale+ XCZU11EG-1FFVC1760 MPSoC. Located on the left side is the FMC+, which is a VITA 57.4 standard connector. On the top left side are two FireFly connectors. The top side is a USB and a UART connector for the PS part. On the right-bottom side are the Micro SD card and Ethernet connector. The bottom side is the PCIe x16 lanes slot. Also on the bottom left of the board is the Phase Locked Loop (PLL) chip, which has 10 programmable clock outputs. The middle part of the board has two SATA connectors where one could plug in an external storage device like SSD. Below the SATA connectors are four DDR4 chips, which contain 2 GB size in total. On the backside of the board, there is a 72 bits data width SODIMM which provides processing system with different DDR sizes compatibility(4GB, 8GB, and 16GB).



Device Name[1]	ZU2EG	ZU3EG	ZU4EG	ZU5EG	ZU6EG	ZU7EG	ZU9EG	ZU11EG	ZU15EG	ZU17EG	ZU19EG
Processor Core	**Quad-core** ARM® Cortex™-A53 MPCore™ up to 1.5GHz										
Memory w/ECC	L1 Cache 32KB I / D per core, L2 Cache 1MB, on-chip Memory 256KB										
Processor Core	**Dual-core** ARM Cortex-R5 MPCore™ up to 600MHz										
Memory w/ECC	L1 Cache 32KB I / D per core, Tightly Coupled Memory 128KB per core										
Graphics Processing Unit	Mali™-400 MP2 up to 667MHz										
Memory	L2 Cache 64KB										
Dynamic Memory Interface	x32/x64: DDR4, LPDDR4, DDR3, DDR3L, LPDDR3 with ECC										
Static Memory Interfaces	NAND, 2x Quad-SPI										
High-Speed Connectivity	PCIe® Gen2 x4, 2x USB3.0, SATA 3.1, DisplayPort, 4x Tri-mode Gigabit Ethernet										
General Connectivity	2xUSB 2.0, 2x SD/SDIO, 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO										
Power Management	Full / Low / PL / Battery Power Domains										
Security	RSA, AES, and SHA										
AMS - System Monitor	10-bit, 1MSPS – Temperature and Voltage Monitor										
	12 x 32/64/128b AXI Ports										
System Logic Cells (K)	103	154	192	256	469	504	600	653	747	926	1,143
CLB Flip-Flops (K)	94	141	176	234	429	461	548	597	682	847	1,045
CLB LUTs (K)	47	71	88	117	215	230	274	299	341	423	523
Max. Distributed RAM (Mb)	1.2	1.8	2.6	3.5	6.9	6.2	8.8	9.1	11.3	8.0	9.8
Total Block RAM (Mb)	5.3	7.6	4.5	5.1	25.1	11.0	32.1	21.1	26.2	28.0	34.6
UltraRAM (Mb)	-	-	13.5	18.0	-	27.0	-	22.5	31.5	28.7	36.0
Clock Management Tiles (CMTs)	3	3	4	4	4	8	4	8	4	11	11
DSP Slices	240	360	728	1,248	1,973	1,728	2,520	2,928	3,528	1,590	1,968
PCI Express® Gen 3x16	-	-	2	2	-	2	-	4	-	4	5
150G Interlaken	-	-	-	-	-	-	-	1	-	2	4
100G Ethernet MAC/PCS w/RS-FEC	-	-	-	-	-	-	-	2	-	2	4
AMS - System Monitor	1	1	1	1	1	1	1	1	1	1	1
GTH 16.3Gb/s Transceivers	-	-	16	16	24	24	24	32	24	44	44
GTY 32.75Gb/s Transceivers	-	-	-	-	-	-	-	16	-	28	28
Extended[2]	-1 -2 -2L		-1 -2 -2L -3				-1 -2 -2L -3				
Industrial	-1 -1L -2										

Figure 2.3: Zynq UltraScale+ EG devices [44]



Pkg	mm	Zynq® UltraScale+™																				
		CG Devices							EG Devices												EV Devices		
		ZU2CG	ZU3CG	ZU4CG	ZU5CG	ZU6CG	ZU7CG	ZU9CG	ZU2EG	ZU3EG	ZU4EG	ZU5EG	ZU6EG	ZU7EG	ZU9EG	ZU11EG	ZU15EG	ZU17EG	ZU19EG	ZU4EV	ZU5EV	ZU7EV
A484	19	●	●						●	●												
A625	21	●	●						●	●												
C784	23	●	●	●	●				●	●	●	●								●	●	
B900	31			●	●		●				●	●		●						●	●	●
C900	31					●		●					●		●		●					
B1156	35					●		●					●		●		●					
C1156	35						●							●		●						●
B1517	40															●		●	●			
F1517	40						●							●		●						●
C1760	42.5															●		●	●			
D1760	42.5																	●	●			
E1924	45																	●	●			

Figure 2.4: MPSoC Device Migration Table [44]

2.1 FPGA resources allocation

Zynq UltraScale devices provide various I/O and offer: several Giga-bit transceivers, High-performance, (HP), high-density (HD), and high-range (HR) I/O banks. The HP banks could support different voltage level of I/O standard, LVDS, and many other kind of chip-to-chip interfaces. The HD I/O banks are designed for low-speed purpose.

The Xilinx high-speed transceivers on FPGA are designed to cover many high-speed protocols that is available nowadays. The UltraScale+ device integrates two different types of high-speed transceivers: the GTH and GTY [45]. GTH has a 32.75 Gbps speed and GTY owns 16.3 Gbps respectively. XCZU11EG-1FFVC1760 have 32 number of GTH transceivers and 16 GTY transceivers.

To optimize the performance of the optical data link, 12 full-duplex lanes of the FireFly are routed on 12 GTY transceivers. Therefore, the left 4 GTY resources are routed to FMC+ gigabit transceiver data pairs (DP). PCIe connection requires x16 transceivers, thus 16 lanes of GTH are employed for this purpose. The left 16 GTH are routed to FMC+ DP. Therefore, FMC+ contains 4 GTY and 16 GTH, providing a total 391.8 Gbps connection from/to a FMC+ mezzanine card. In Fig (2.5), the resources allocation plan map are shown.

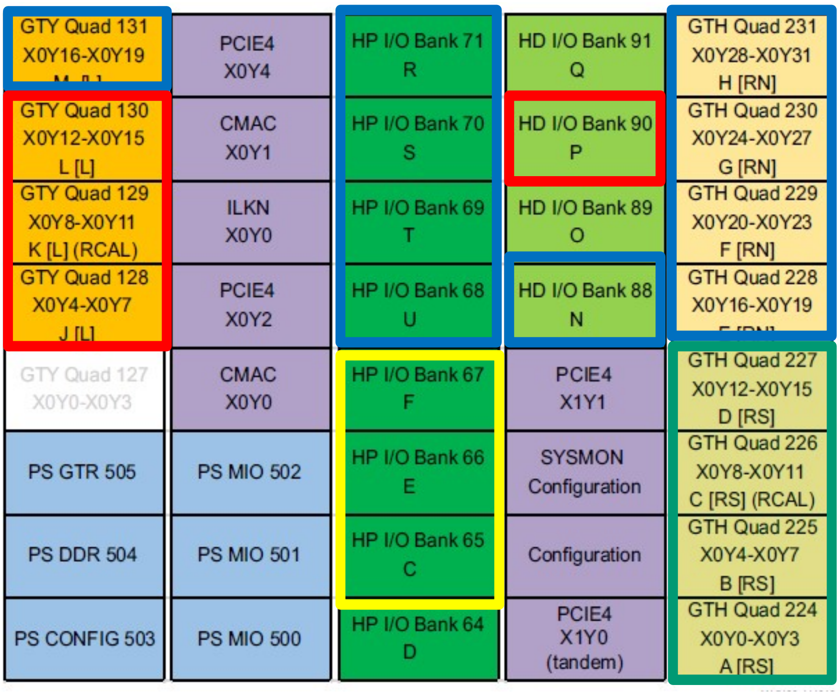


Figure 2.5: The Zynq XCZU11EG-1FFVC1760 packaging diagram [46]

On the top-left side of the plan map, the *red* colour rectangular frames three GTY Quads (GTY Quad 128-130) for FireFly. In addition to these, one HD I/O Bank 90 has been routed to manage the I2C interface required from the FireFly optical connection. On the bottom-right side of the plan map, the GTH Quad (GTH Quad 224-227) (*green* colour) shows PCIe transceivers allocation. The *blue* colour part are the Quads and Banks that connected with FMC+. The *yellow* rectangular frames the Banks (HP I/O Bank 65-67) that employed to connect the FPGA to the DDR4 memory device integrated on the HighFlex2 card.

2.2 Clock distribution

The clock distribution is fundamental to enable digital communication. The overall clock distribution on HighFlex2 is shown in Fig (2.6). The clocks which are provided to the Processing System are PS_REF_CLK and PS-GTR. The PS-GTR clock is employed for PS-GTR high-speed Transceivers. These transceivers are connected to 2 SATA connectors and operating in raid configuration. The GTR requires a clock of 150 MHz.

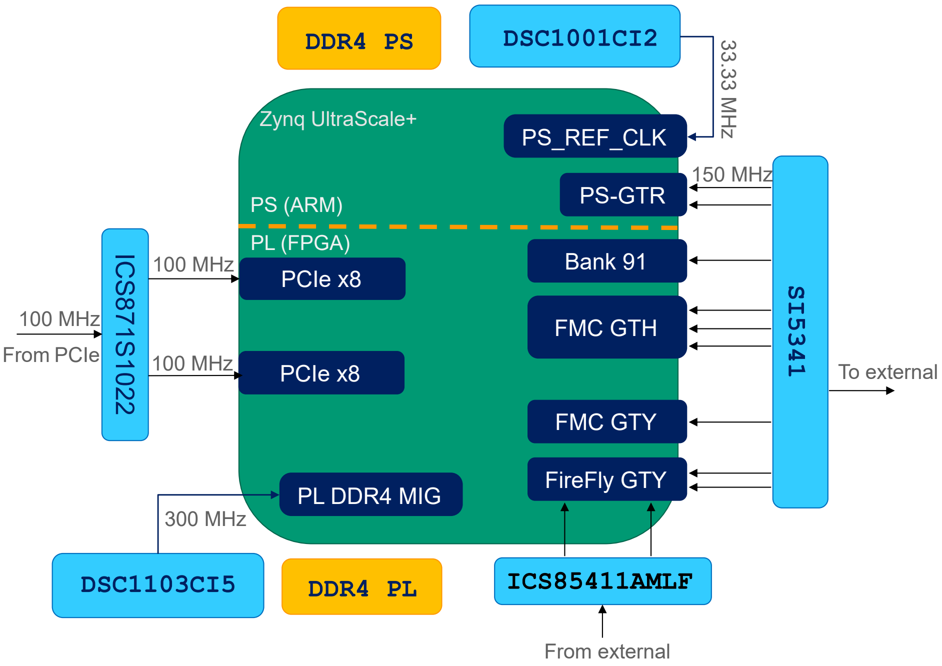


Figure 2.6: HighFlex2 clock overall distribution diagram

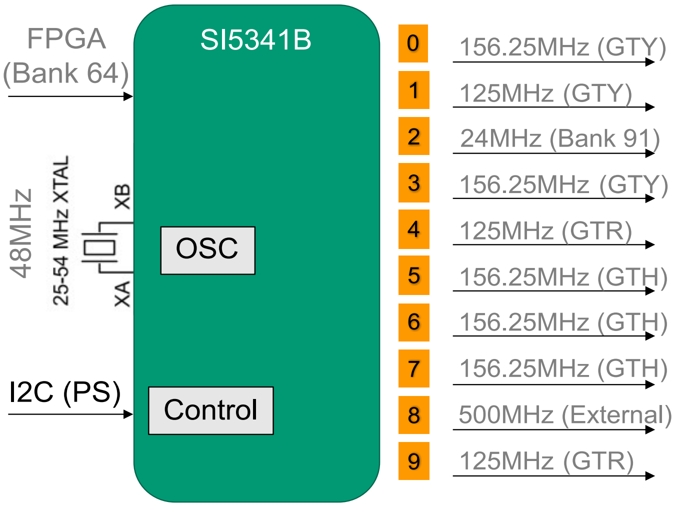


Figure 2.7: Clock chip SI5341 could generate up to 10 independent clocks.

For PCIe logic block, a PLL (ICS871S1022) jitter attenuator clock generator is employed to provide the reference clocks fully compliant to the PCIe hard-core Generation 4 Fig (2.6). The clocks are distributed in such a way to instantiate one single PCIe core generation 4 x8 lanes or two parallel PCIe cores that can support generation 4 x16 lanes. The mentioned clock scheme will be discussed in the next paragraph. For PL (Programmable Logic) part memory, a 300 MHz clock is provided to the Memory Interface Generator (MIG) IP Core. The MIG is an IP Core that combines the controller and physical layer (PHY) for interfacing the FPGA user designs to the DDR4 memory device. For PS (Processing System) part memory, the DDR4 connected with the PS part have an internal clock provided directly from the DDR4 controller integrated on the PS. The FireFly have two options: one from on-board SI5341 clock chip, and the second option is from external clocks that defined by users. This provides a flexible operation mode.

Because the PS part and PL part is working independently, once the PS is powered up, the PS I2C could be used to configure the programmable low-jitter PLL SI5341 in Fig (2.6). This allows the clocks to be ready before the PL starts to work. This low-jitter 10-output programmable clock generator enables a flexible clock source configuration with the output which could be programmed to 100 Hz to 1028 MHz. The reference clock of each GTH and GTY Quad could have both *on-board* clock source and external clock source. The user could configure the clock source according to the specific application through I2C directly connected to the processing system. The configuration of the clock distribution could be programmed by SILICON LABS ClockBuilder Pro software [47]. All the settings will be stored in a C header file. The user could import that file into Xilinx Software Development Kit (SDK) and program the PLL by the I2C interface managed by the ARM easily.

The Fig (1) in Appendix A shows the software structure in Xilinx SDK. Each channel can be configured to LVDS, LVPECL, LVCMOS, CML, and HCSL with different signal amplitude. Fig (2.7) shows the default clock configuration pre-programmed on the HighFlex2 card. The input reference clock of the PLL can be selected between a reference clock, programmable by the FPGA (Bank 64), or a reference clock provided by local quartz.

2.3 PCI Express circuit design

Fig (2.9) illustrated out the GTH transceivers and their connection to the PCIe giga-transfer serial lines. Four GTH quad are employed to connect the 16 full-duplex lines of the PCIe: GTH Quad 224, 225, 226 and 227. The GTH Quad 224 and 225 are connected to the lines from 8 to 15 of PCIe, which is connected to a PCIe hard-core with x8 lanes, generation 4. They use the 100 MHz clock source from ICS87S1022. The GTH Quad 226 and 227 together own the 0-7 RX/TX. The reference clock is shown in Fig (2.9).

The routed PCIe connection scheme, shown in Fig (2.9), allows higher flexibility on the PCIe implementation in the FPGA. Several solutions are possible in Fig (2.8).

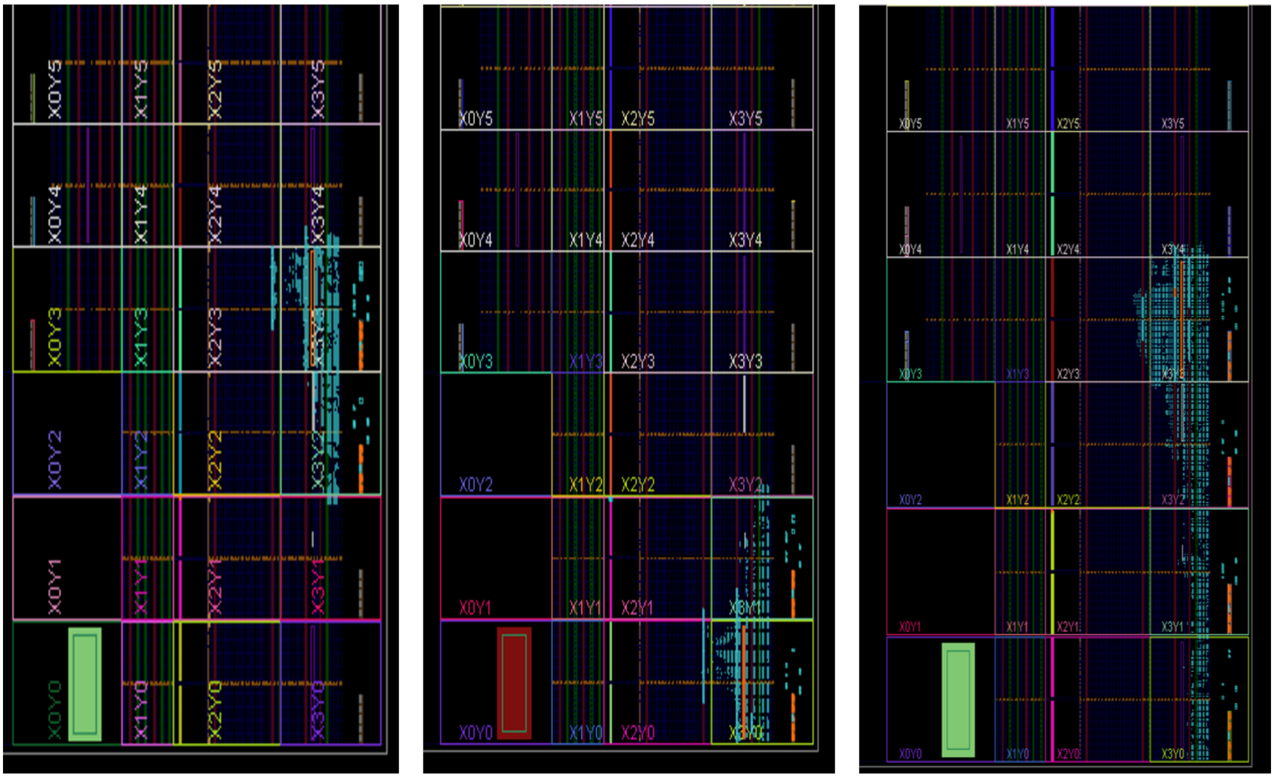


Figure 2.8: Three different PCIe configuration implementation. The left one is Gen 3/4 x8 lanes (TOP); The middle is the Gen 3/4 x8 lanes (BOTTOM); The right is generation 3 x 16 lanes.

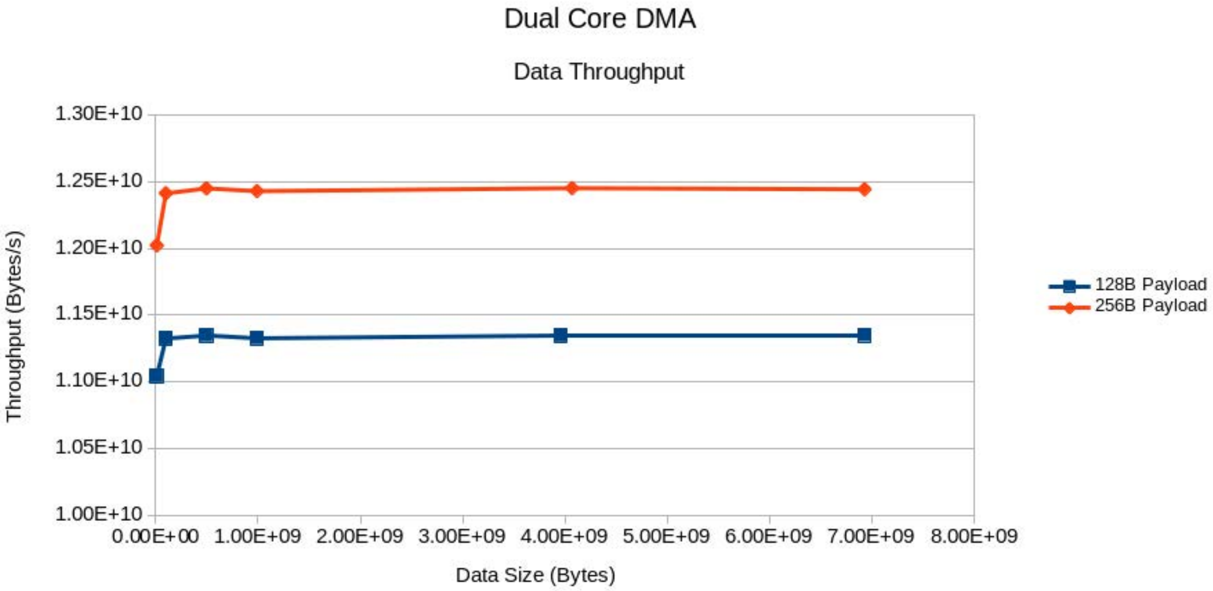


Figure 2.12: Performance of the dual-core KIT DMA engine for PCIe Gen 3 x16

A preliminary test has been done to evaluate the corrected functionality of both PCIe interfaces. The FPGA architecture deployed in the HighFlex2 is based on a double-core KIT-DMA [15] operating with x16 lanes, PCIe generation 3. The architecture of the DMA engine is shown in Fig (2.10). The architecture consists of 4 parts: the TX engine, RX engine, address

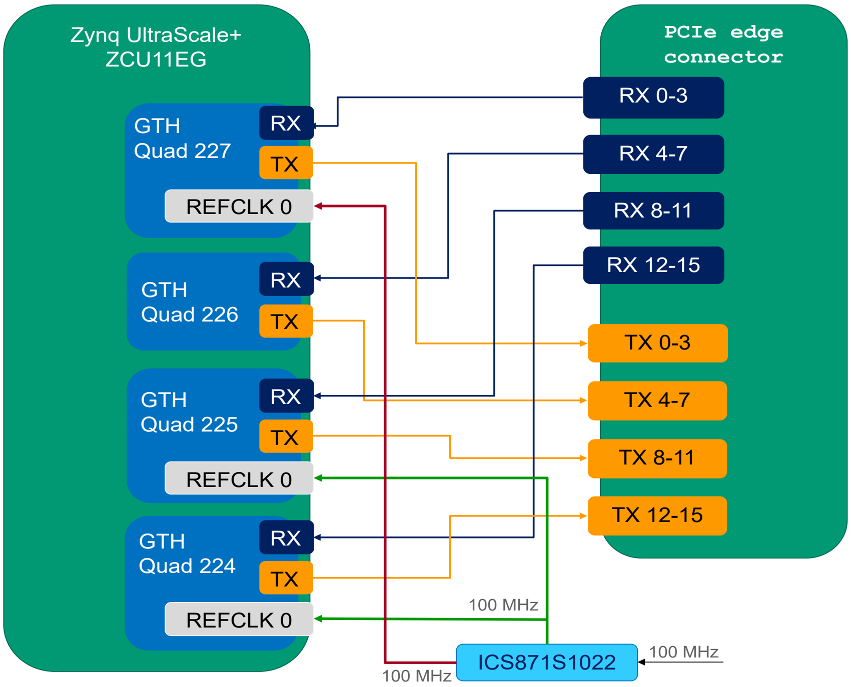


Figure 2.9: Four GTH Quad are assigned to the PCIe ×16 edge connector directly.

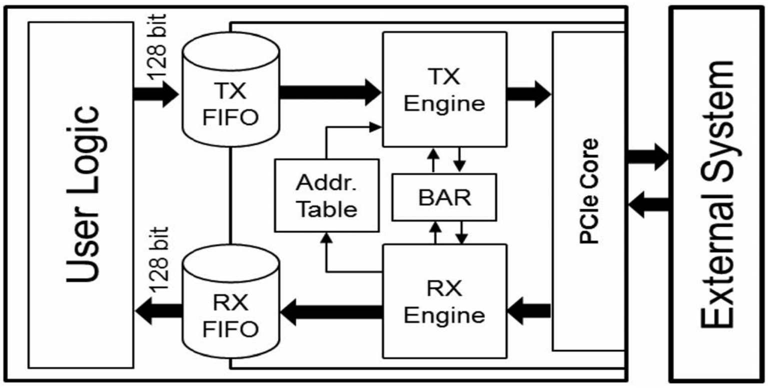


Figure 2.10: Architecture of the KIT DMA engine for PCIe Gen 3 x8 [15]

table, and PCIe BAR. The TX/RX engine is used for transmitting/receiving data to/from an external system (e.g. CPU/GPU based server). The address table stores the descriptors defined by the driver and written from the Linux driver. Linux driver allocates the pages in the kernel memory and it writes their physical addresses inside this address table. This write operation only happens at the initialization stage. After which the FPGA could start to transfer the data from FPGA to the corresponding address that allocated before. A detail working principle is explained in [15].

The double-core implementation is an extention of the single-core KIT-DMA shown in Fig (2.11). It consists in a Master-Slave architecture which is capable to work with two PCIe cores in parallel sharing the descriptor lists. The double-core architecture is shown in Fig (2.11).

To keep the architecture simple, a Master-Slave architecture was designed [15]. The principle is to have one DMA engine acting as a Master, and the second one as a Slave. Therefore, the Master initiates the DMA operation with the Linux driver. Once the DMA is started, the Master wakes the Slave up and both engines work together, shown in Fig (2.11). Both Mas-

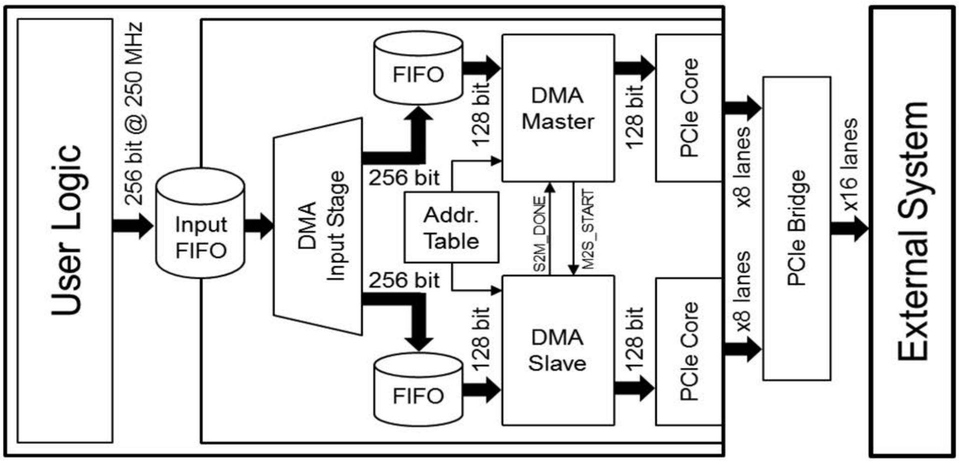


Figure 2.11: Architecture of the dual-core KIT DMA engine for PCIe Gen 3 x16 [15]

ter and Slave core are working in parallel. This could double the throughput compared with single-core mode.

The result obtained through dual-core is presented in Fig (2.12), where each point is the mean values measured for several tests. With $256B$ payloads, it could reach 12.4 GB/s throughput with PCIe generation 3 and is expected to reach a throughput over 24 GB/s by means of two PCIe cores, generation 4.

2.4 FireFly circuit design

FireFly Micro Flyover System is the first interconnect system that gives a designer the flexibility of using micro footprint with high-performance optical and low-cost copper interconnects interchangeably with the same connector system. Samtec FireFly copper and optical cable systems provide the flexibility to achieve higher data rates to 28 Gbps for each lane and/or greater distances, simplifying board design and enhancing performance. Two FireFly connectors are present on the HighFlex2, offering the possibility to integrate up to 12 full-duplex lanes with each one operating up to 28 Gb/s. Each pair is consists of a 1 UEC5 series FireFly edge card socket connector for data transmission and a 1 UCC8 series FireFly connector for the configuration, control, and power of the optical link. The slow control is connected with HD I/O Bank 90 as shown in Fig (2.5).

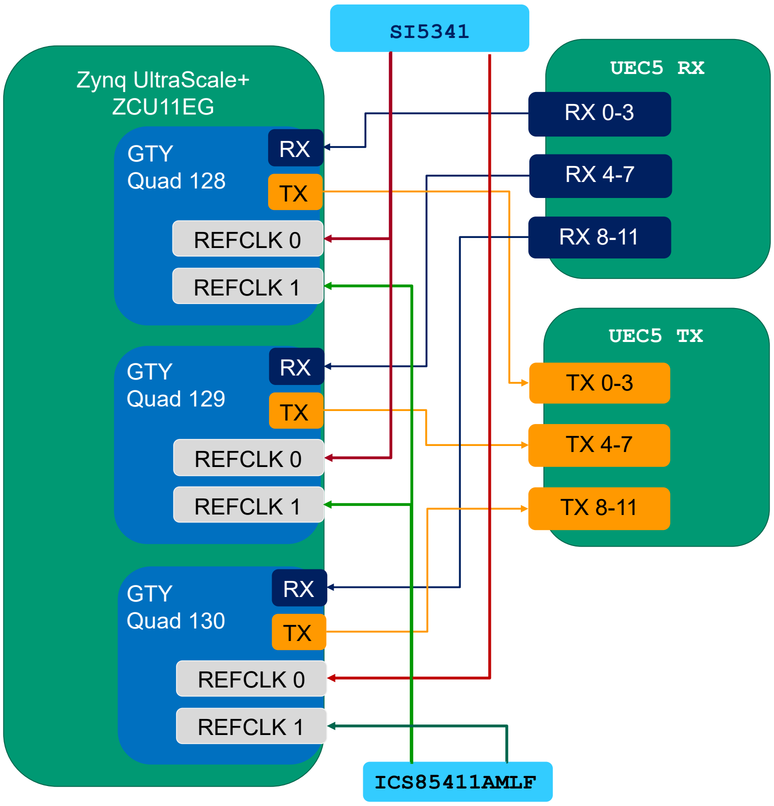


Figure 2.13: Two UCC8 connectors for slow control not shown here. *Two* UEC5 data 12 lanes are allocated to *three* GTY Quads.

The circuit routing on HighFlex2 is demonstrated at Fig (2.13). The on-board PLL SI5341 (red arrow) and 85411 clock buffer for external user clock input (green arrow) provide the reference clock of the GTY Quads. The blue arrows are the RX lanes routing, and the yellow arrows show the TX routing. Through this simple principle, the HighFlex2 achieves the 12 lanes full-duplex FireFly.

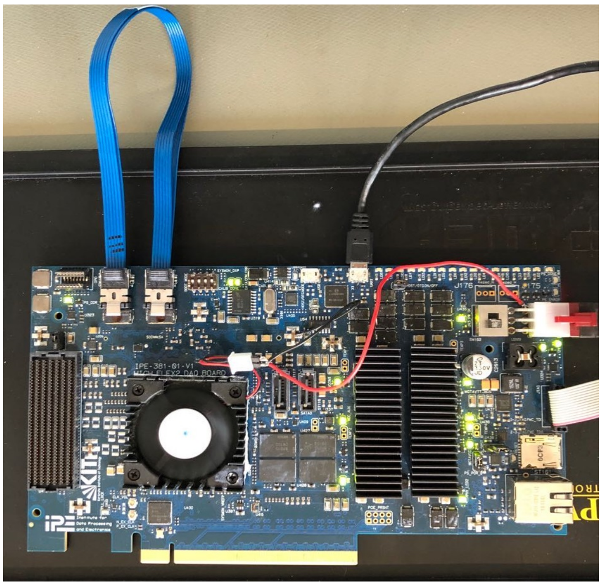


Figure 2.14: UEC5 TX to UEC5 RX is connected directly to enable the FireFly 12 lanes loop-back test on HighFlex2.

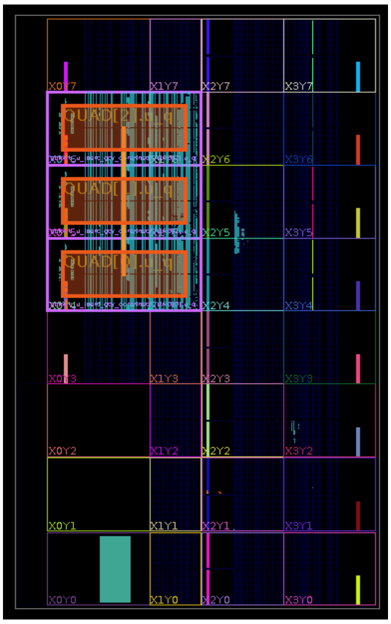


Figure 2.15: The figure demonstrate the IBERT IP implementation and three GTY Quads FPGA implementation on HighFlex2.

Two UEC5 serves as TX and RX for a different direction. To exploit the potentiality offered by the FireFly technology, the FireFly data lines have been connected to the GTY transceivers of the FPGA. Each GTY transceiver provided by Zynq MPSoC could offer a data-link with speed up to 32.75 Gb/s. The Quad 128, 129 and 130 are adjacent allowing an excellent clock distribution that can be shared among all quads. The clock sources are fully configurable and can be provided by two sources: the programmable clock chip SI5341, or the external user clock. This configuration could provide the flexibility of different clock source and configurations.

In order to test the performance of FireFly on HighFlex2, a FireFly copper cable is connected from UEC5 TX to UEC5 RX on HighFlex2 directly shown in Fig (2.14). This enables an on-board self loop-back test directly on HighFlex2 board. The Integrated Bit Error Ratio Tester (IBERT) is required. IBERT for UltraScale/UltrasScale+ GTY transceivers IP has been implemented and deployed in the FPGA. The FPGA placement of the FireFly GTY and the IBERT logic implementation location is shown in Fig (2.15). The IBERT test has been configured to test all 12 lanes in parallel at two different speeds, 20 and 25 GB/s. The test has been performed to test and verify the design of the fast differential transmission lines routed on the HighFlex2 board.

At IBERT IP configuration, two protocols are verified. One has a line rate of 20 Gbps, another is 25 Gbps. The highest speed of 25 Gbps is limited by the provided official IBERT core rather than the FireFly link itself. Both protocols are running with a reference clock of 156.25 MHz. The insertion loss at Nyquist is 20 dB. Data lines are AC coupled with a termination voltage VTT of 800 mV.

A 2D eye scan is provided in GTX/GTH IBERT IP of the Xilinx UltraScale+ series. Eye diagram is a series of bit periodic superposition on the horizontal axis to form an eye-like waveform. Eye height represents noise and eye width represents jitter. The influence of crosstalk and noise can be observed from the eye map, which reflects the overall characteristics of the digital signal.

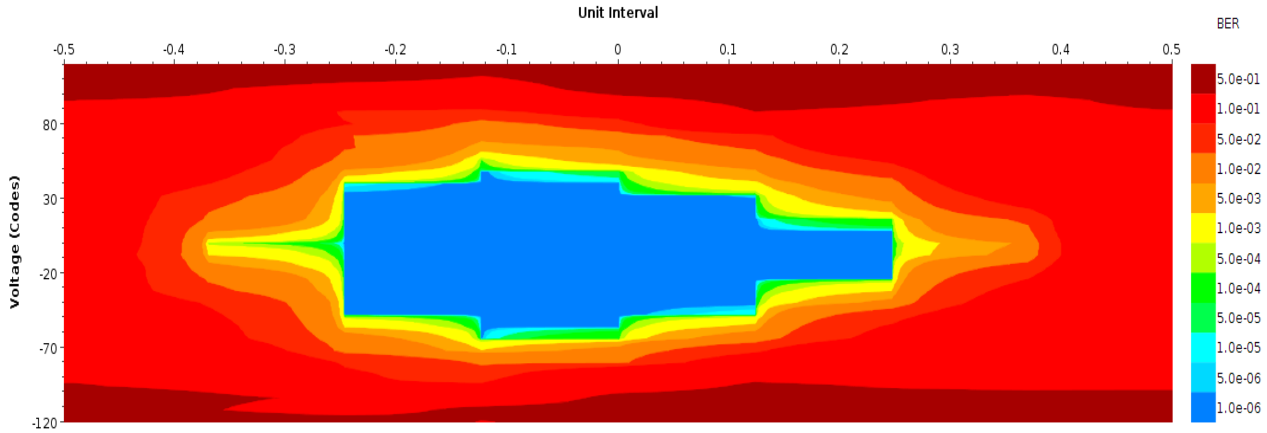
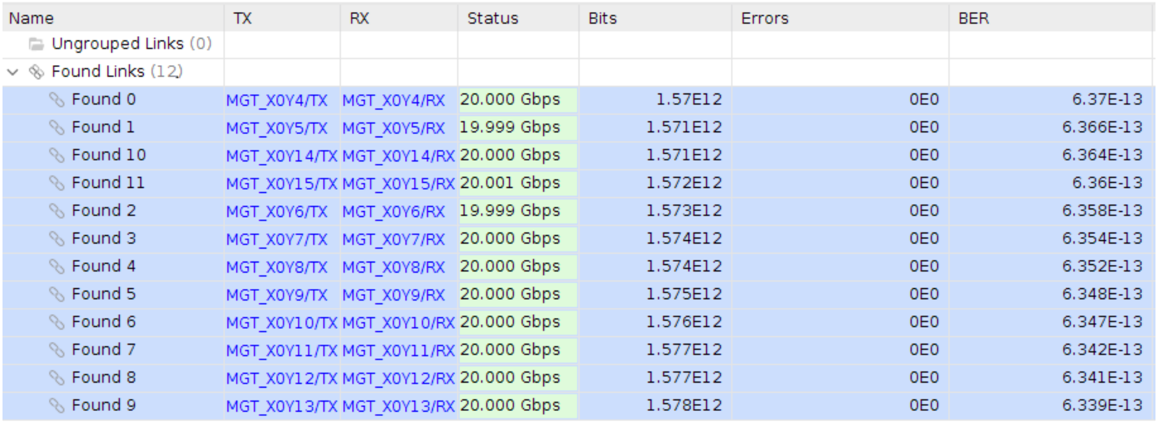


Figure 2.16: The eye diagram of GTY lane 0.



Name	TX	RX	Status	Bits	Errors	BER
Ungrouped Links (0)						
Found Links (12)						
Found 0	MGT_X0Y4/TX	MGT_X0Y4/RX	20.000 Gbps	1.57E12	0E0	6.37E-13
Found 1	MGT_X0Y5/TX	MGT_X0Y5/RX	19.999 Gbps	1.571E12	0E0	6.366E-13
Found 10	MGT_X0Y14/TX	MGT_X0Y14/RX	20.000 Gbps	1.571E12	0E0	6.364E-13
Found 11	MGT_X0Y15/TX	MGT_X0Y15/RX	20.001 Gbps	1.572E12	0E0	6.36E-13
Found 2	MGT_X0Y6/TX	MGT_X0Y6/RX	19.999 Gbps	1.573E12	0E0	6.358E-13
Found 3	MGT_X0Y7/TX	MGT_X0Y7/RX	20.000 Gbps	1.574E12	0E0	6.354E-13
Found 4	MGT_X0Y8/TX	MGT_X0Y8/RX	20.000 Gbps	1.574E12	0E0	6.352E-13
Found 5	MGT_X0Y9/TX	MGT_X0Y9/RX	20.000 Gbps	1.575E12	0E0	6.348E-13
Found 6	MGT_X0Y10/TX	MGT_X0Y10/RX	20.000 Gbps	1.576E12	0E0	6.347E-13
Found 7	MGT_X0Y11/TX	MGT_X0Y11/RX	20.000 Gbps	1.577E12	0E0	6.342E-13
Found 8	MGT_X0Y12/TX	MGT_X0Y12/RX	20.000 Gbps	1.577E12	0E0	6.341E-13
Found 9	MGT_X0Y13/TX	MGT_X0Y13/RX	20.000 Gbps	1.578E12	0E0	6.339E-13

Figure 2.17: Reults of the IBERT test on the 12 lanes FireFly at 20 Gb/s.

In Appendix A, Fig (4) shows 12 lanes eye diagram and all lanes works fine (successfully locked on the given data transfer speed). The Fig (2.16) is the detail of lane 0 eye diagram. Colour *blue* standards for a bit error rate (BER) of 10^{-6}. From colour *blue* to *dark red* represent a raise of error bits. The vertical coordinate is the voltage range from full vertical scale (-127 to +127) corresponds to 255 voltage offset. It shows the voltage offset from the baseline. The conversion for each code at y-coordinate is 1.6 mv per code (for GTY Quad). The horizontal unit is the unit interval (UI). At the 20 Gbps speed link, the unit is 50 ps.

At the optimal sampling time, the bit error rate is the lowest, while on both sides of the time axis, the bit error rate increases continuously, as shown in the Bathtub Curve figure in Appendix A Fig (5). In engineering, this curve for the error rate according to a signal sampling point, called *Bathtub Curve*. The TX diff-swing is 950 mV.

The horizontal range for Fig (2.16) and Appendix A Fig (5) is from -0.5 unit interval (UI) to 0.5 UI. They both declear an 55.56% open range percentage.

As shown in Fig (2.17), all 12 lanes are locked to the 20 Gbps. The total errors found after 1.3 minutes running is 0. The total bits transfered is about 1.57×10^{12}. The bit error rate (BER) is around 6.34×10^{-13}.

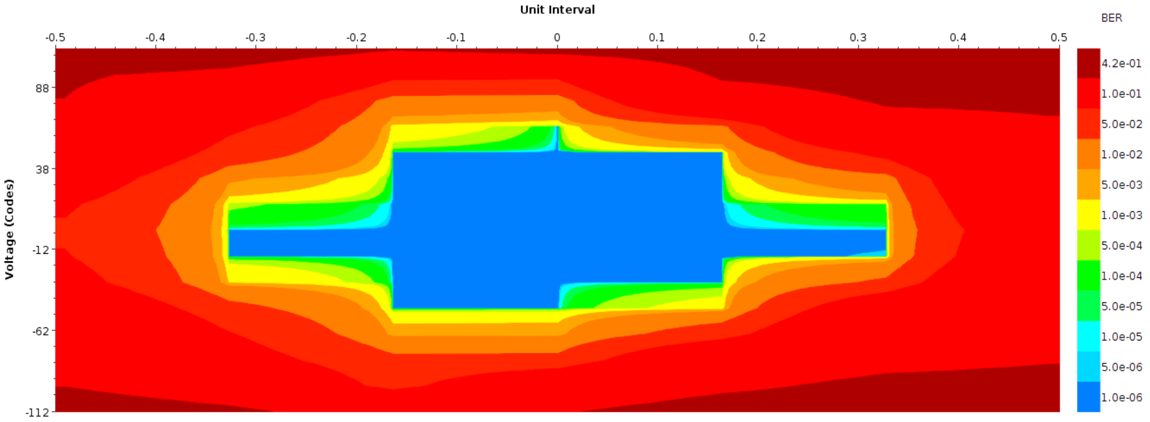


Figure 2.18: The 2-D eye scan of channel 1 FireFly running at 25 Gbps.

A 25 Gbps speed data link also is verified here. The Appendix A Fig (6) is 12 lanes's eye diagram. The Fig (2.18) is the eye diagram of channel 1. The unit of y-coordinate is 1.6mv per code that the same as the 20 Gpbs speed test. The UI at 25 Gbps speed link is 40 ps.

Different from 20 Gbps speed test, the TX diff-swing use 1040 mV. The colour *blue* is lower bit error rate with about 10^{-6}. The *dark red* colour represent a 4.1×10^{-1} bit error rate. The horizontal increment is 10, and vertical increment is 16. The open region of each lane is shown in Tab (2.1).

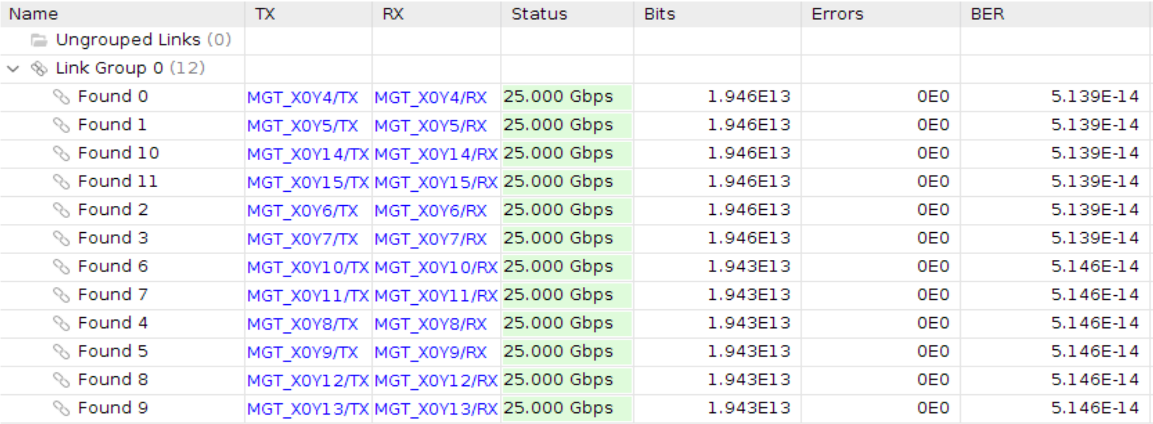
Table 2.1: Open percentage in horizontal and vertical direction

Lane	Horizontal Percentage	Vertical Percentage
0	57.14	20
1	71.43	46.67
2	57.14	33.33
3	42.86	26.67
4	57.14	40.00
5	71.43	40
6	57.14	40
7	57.14	46.67
8	71.43	46.67
9	57.14	33.33
10	57.14	46.67
11	57.14	40.00

The Tab (2.1) shows that all data link have a relatively large open area in both x-coordinate and y-coordinate running at 25 Gbps. This denotes that all the 12 lanes are working properly and proves the PCB routing for FireFly is of high quality.

Bathtub cureve in Appendix A Fig (7) explain the channel 1 in another aspect. The same as Fig (2.18), declearing an 71.43% open range percentage.

As shown in Fig (2.19), all 12 lanes are locked to the 25 Gbps. The total errors found after about 12 minutes running is 0. The total bits transfered is about 1.946×10^{13} bits. The bit error rate (BER) is around 5.139×10^{-14} to 5.146×10^{-14}.



Name	TX	RX	Status	Bits	Errors	BER
Ungrouped Links (0)						
Link Group 0 (12)						
Found 0	MGT_X0Y4/TX	MGT_X0Y4/RX	25.000 Gbps	1.946E13	0E0	5.139E-14
Found 1	MGT_X0Y5/TX	MGT_X0Y5/RX	25.000 Gbps	1.946E13	0E0	5.139E-14
Found 10	MGT_X0Y14/TX	MGT_X0Y14/RX	25.000 Gbps	1.946E13	0E0	5.139E-14
Found 11	MGT_X0Y15/TX	MGT_X0Y15/RX	25.000 Gbps	1.946E13	0E0	5.139E-14
Found 2	MGT_X0Y6/TX	MGT_X0Y6/RX	25.000 Gbps	1.946E13	0E0	5.139E-14
Found 3	MGT_X0Y7/TX	MGT_X0Y7/RX	25.000 Gbps	1.946E13	0E0	5.139E-14
Found 6	MGT_X0Y10/TX	MGT_X0Y10/RX	25.000 Gbps	1.943E13	0E0	5.146E-14
Found 7	MGT_X0Y11/TX	MGT_X0Y11/RX	25.000 Gbps	1.943E13	0E0	5.146E-14
Found 4	MGT_X0Y8/TX	MGT_X0Y8/RX	25.000 Gbps	1.943E13	0E0	5.146E-14
Found 5	MGT_X0Y9/TX	MGT_X0Y9/RX	25.000 Gbps	1.943E13	0E0	5.146E-14
Found 8	MGT_X0Y12/TX	MGT_X0Y12/RX	25.000 Gbps	1.943E13	0E0	5.146E-14
Found 9	MGT_X0Y13/TX	MGT_X0Y13/RX	25.000 Gbps	1.943E13	0E0	5.146E-14

Figure 2.19: The transmission lane information.

2.5 DDR4 for programmable logic

The external memory for FPGA logic is one of the important auxiliary equipment for FPGA DAQ. It is used to temporarily store data from detectors as well as during the FPGA data processing.

HighFlex2 is equipped with four MT40A256M16GE-083E DDR4 chips. Each MT40A256M has a memory size of 4 Gb. Therefore in total, the PL memory is 2 GB in size. The PL DDR4 connection scheme between the four DDR4 devices and the FPGA is shown in Fig (2.20).

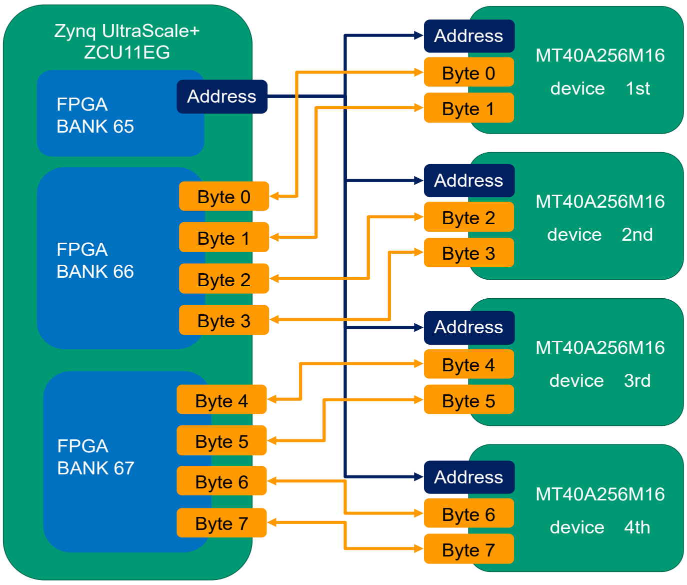


Figure 2.20: Four PL DDR4 chips topoloty

All four chips share the address bus, which is connected with FPGA bank 65. The byte 0 to byte 3 is assigned to bank 66, and the byte 4 to byte 7 is assigned to bank 67. There are many options to assign the ddr4 signals to FPGA bank. These banks have been selected is because the bank 65, 66 and 67 are located in the same side of ZCU11EG FPGA package as shown in Fig 2.21, allowing an optimal orientation for the routing of all fast data, address and control lines on the board.

The MIG core provided by Xilinx integrates a test feature to check the signal integrity during both the write and read operations. After downloading the bitstream containing MIG

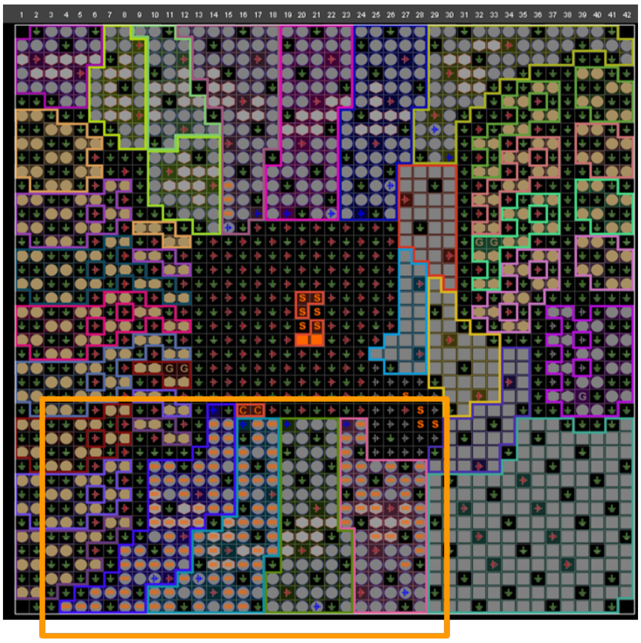
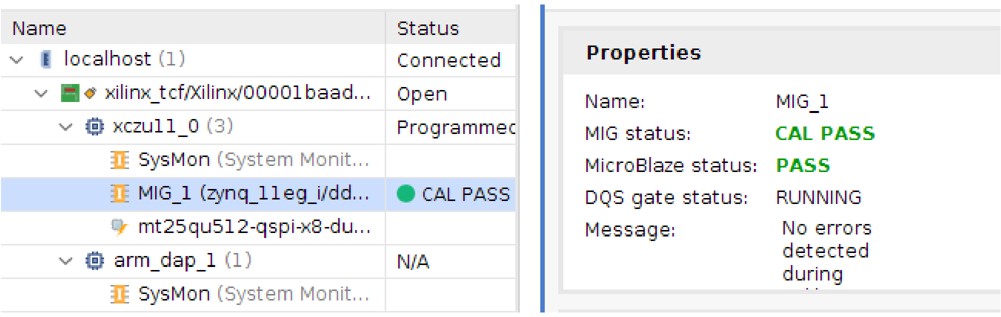


Figure 2.21: Zynq XCZU11EG-1FFVC1760 package pin, pin region coloured with orange where the four DDR4 devices are connected.



Name	Status
localhost (1)	Connected
xilinx_tcf/Xilinx/00001baad...	Open
xczu11_0 (3)	Programmec
SysMon (System Monit...	
MIG_1 (zynq_11eg_i/dd...	● CAL PASS
mt25qu512-qspi-x8-du...	
arm_dap_1 (1)	N/A
SysMon (System Monit...	

Properties

Name:	MIG_1
MIG status:	CAL PASS
MicroBlaze status:	PASS
DQS gate status:	RUNNING
Message:	No errors detected during

Figure 2.22: The PL DDR4 calibration passed

core into HighFlex2, the Vivado hardware manager will automatically loads the benchmarking core and it execute the test of the status of DDR4 after the memory calibration.

The first message is there are no errors detected during calibration. And during the calibration stage, there are about 25 tests, in which, 17 tests are passed shown in Appendix A Fig (8). This means a succssful communication with DDR4 is estabilised.

2.6 PS peripherals design

A complete Linux embedded hardware system requires the support of complete peripherals. As aforementioned, there are clock chip required to connected with PS directly, and also the Ethernet. Besides this, the SD card and Flash chip are required to boot the system. The PS Multiplexed I/Os (MIO) is used to route with different peripherals as shown in Fig 9 at Appendix A.

The Appendix A Fig 9 illustrate that the PS part MIO could be routed with different peripherals controllers: USB , NAND, SD2.0/3.0, QSPI, SPI, CAN, I2C, UART, GPIO, etc.

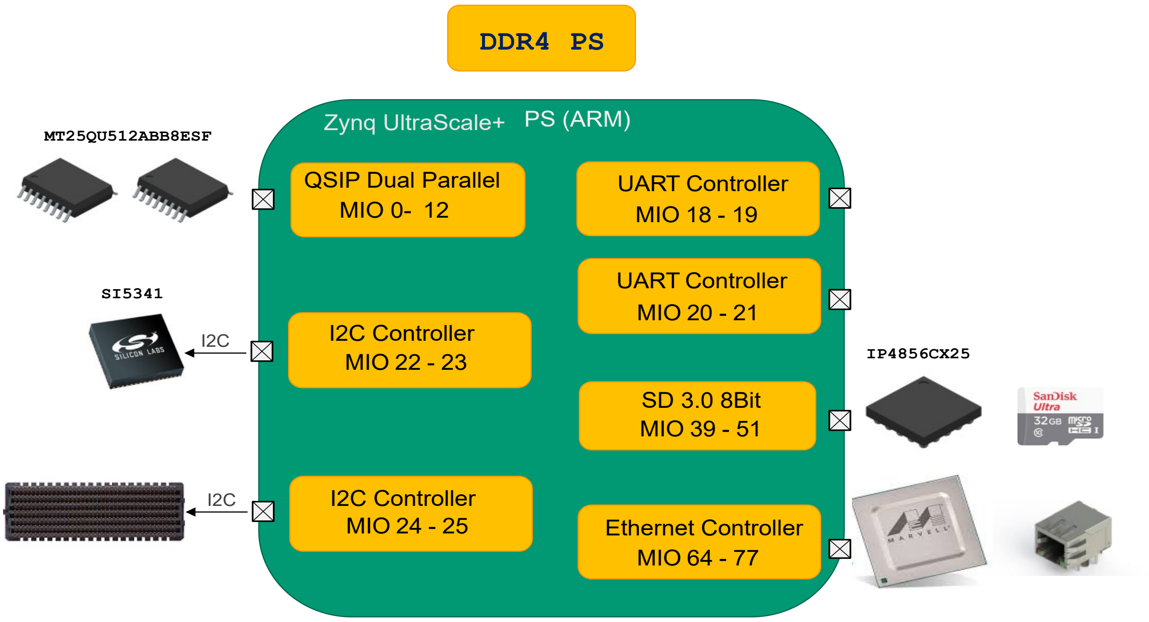

DDR4 PS
Zynq UltraScale+ PS (ARM)
MT25QU512ABB8ESF
QSIP Dual Parallel MIO 0- 12
UART Controller MIO 18 - 19
UART Controller MIO 20 - 21
SI5341
I2C
I2C Controller MIO 22 - 23
IP4856CX25
SD 3.0 8Bit MIO 39 - 51
I2C
I2C Controller MIO 24 - 25
Ethernet Controller MIO 64 - 77

Figure 2.23: HighFlex2 MIO resources allocation

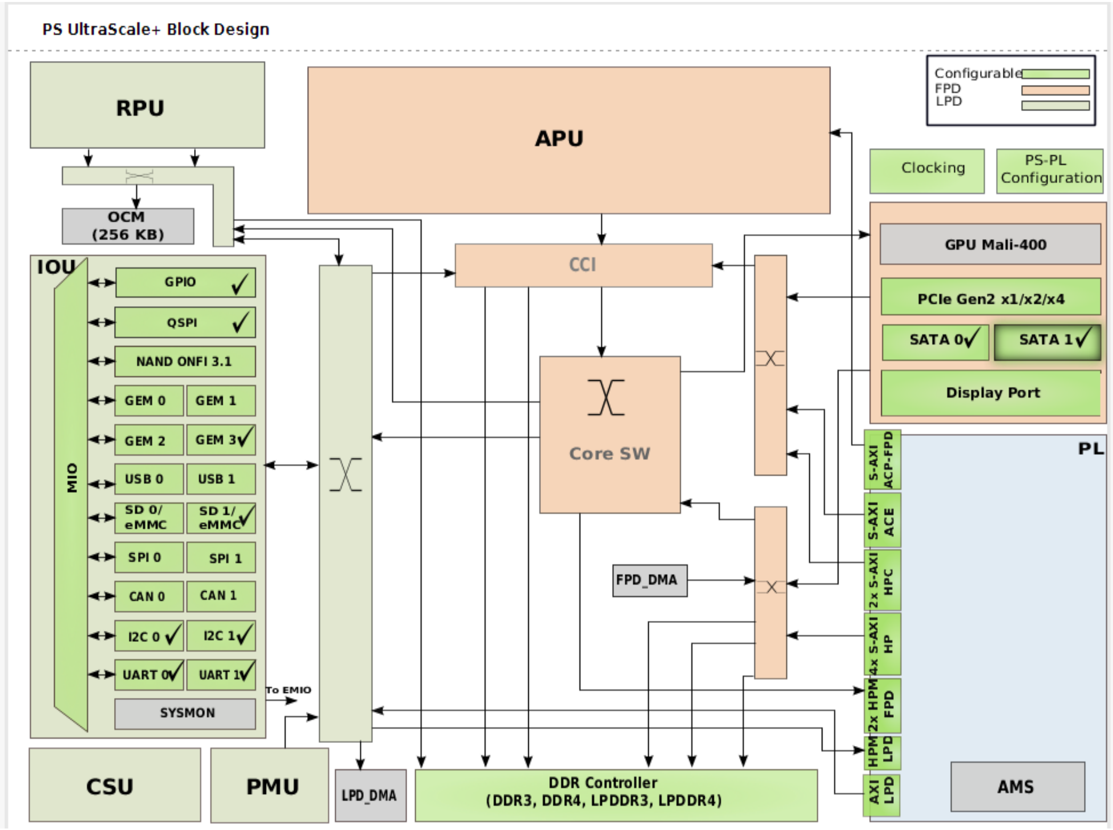


Figure 2.24: HighFlex2 MIO resources allocation in the Zynq UltraScale+ PS IP configuration, all the pheripharals selected have a check mark on the IP.

The Fig (2.23) illustrates the detail MIO allocation for HighFlex2. The Flash located at MIO 0 to 12. There are 2 MT25QU512ABB8ESF flash chips connected with HighFlex2, enabling a 1 Gb size of external memory. The MIO 18 and 19 is the first UART RX and TX. The MIO 20 and 21 is the second UART. The I2C controller located at MIO 22 and 23 is used for configuring the programmable clock device. The I2C at MIO 24 and 25 is connected to the FMC+ I2C position, and this could provide the PS part with a possibility to directly talk with FMC without going through FPGA (PL part). The SD card is located at MIO 39 to 51, and a flexible size micro SD card could be inserted into the board with embedded Linux image and device tree. The Marvel Alaska 88E1111 physical layer device connected with MIO from 64 to 77. This could provide PS with Ethernet capability. All the peripherals are tested and verified. The Fig (2.24) shows this MIO allocation in Zynq UltraScale+ PS IP configuration.

2.7 Power supply design

As the Zynq device is a Multiprocessor systems-on-chip, there are different power domains for the PL part and PS part. The power supply solution for Zynq MPSoC is complicated. The Zynq UltraScale+ device has four different power domains: low-power domain (LPD), full-power domain (FPD), PL power domain (PLPD), and battery power domain (BPD). The high level use cases for power consolidation solution have two: **Always on** and **Full power management flexibility** [48]. The full power management means for each power rail needs independent software control to reduce the power usage in unused domains. **Full power management flexibility** have 14 different power rails. This method will use a large number of regulators which will cause a huge PCB layout that goes against the goal of a small board. This brings the solution to **Always on**.

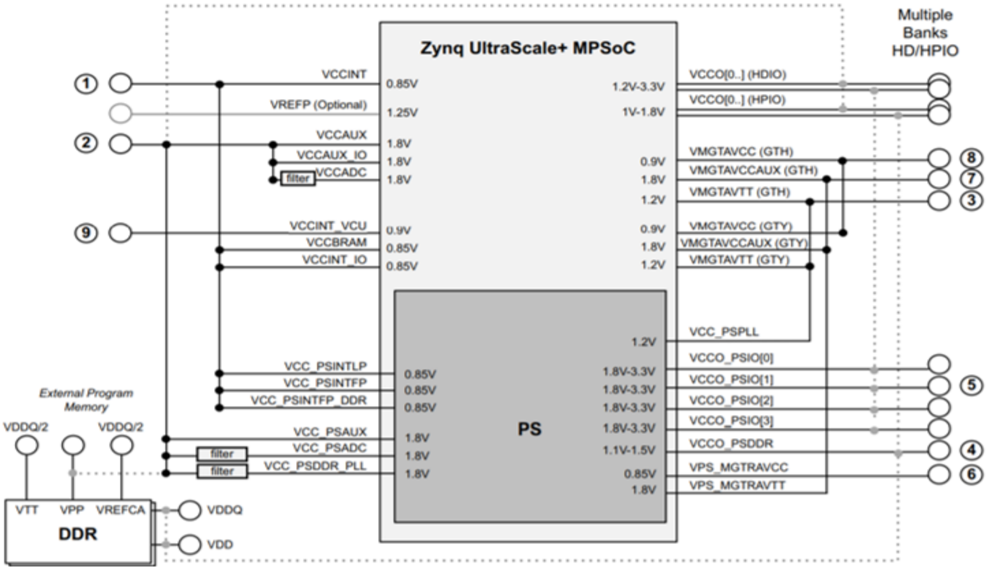


Figure 2.25: MPSoC Power Consolidation Solution [48]

For the speed grade −1 chip selected, the Always on should use *Optimized for Cost*. As shown in Fig (2.25), there are 9 power rail needs to supply except the DDR. As our device is EG series, not EV, the Video Codec encoder does not existed on the chip, therefore the number 9 power rail `VCCINT_VCU` could be ignored.

An 8 power supply rail configuration needs at least 8 regulators. This will increase the difficulty of the layout arrangement. The voltage regulator LTM4650[49] provides two outputs each of them could be configured independently and range from 0.6 V to 1.8 V. For each output channel, it has one internal 0.12 µH inductor and saves a lot of space where needs to place a relatively big inductor peripherally.

The Zynq UltraScale+ device has strict demand for power-on/off sequencing [50]. For PS part, the LPD and FPD they have their own recommended power-on sequence. The power-off sequence is the reversed sequence of power-on. For LPD, first the VCC_PSINTLP, then secondly is VCC_PSAUX, VCC_PSADC, and VCC_PSPLL. Last is VCCO_PSIO for PS I/O supply. The VCC_PSINTFP and VCC_PSINTFP_DDR firstly be driven for FPD. Secondly VPS_MGTRAVCC and VCC_PSDDR_PLL is needed. Last sequence is the VPS_MGTRAVTT and VCCO_PSDDR. The Fig (2.26) demonstrate our solution to powering in sequence.

The PL power sequence is firstly VCC_INT, secondly the VCCINT_IO / VCC_BRAM /VCCINT_VCU, thirdly the VCC_AUX / VCC_AUX_IO, and lastly VCCO. The PS and PL are isolated in devices and independent, so consolidation of the PS and PL power rail is possible if they have the same voltage level.

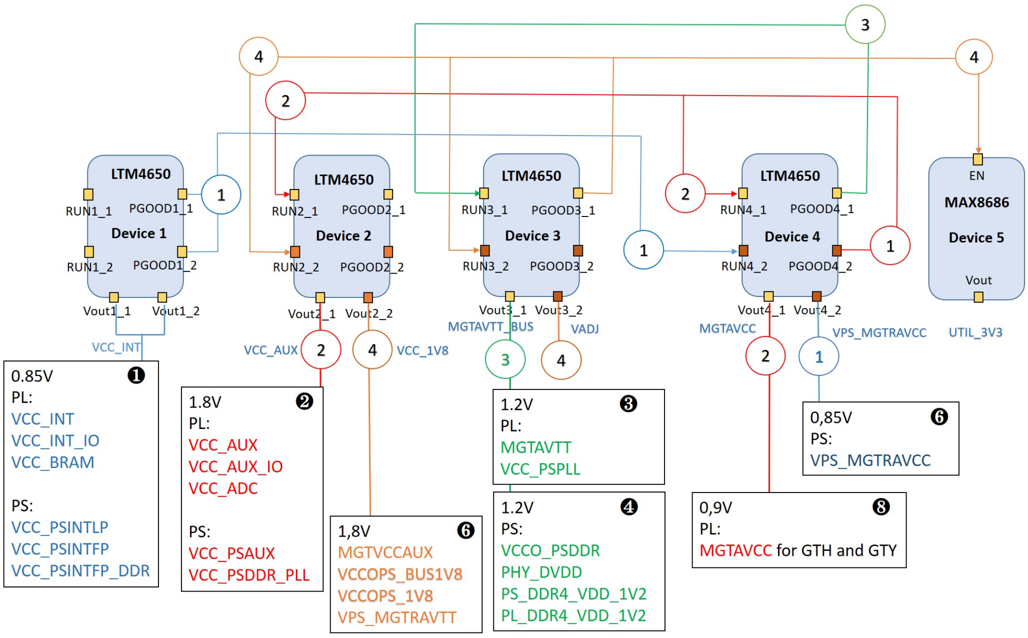


Figure 2.26: MPSoC Power Sequence Solution

The solution is combined with 5 individual voltage regulator. Four are LTM4650 to cover the voltage level from 0.6 V to 1.8 V. Another MAX8686 is used for 3.3 V. Each device have the run control PIN to turn on according to the channel, and each channel has open-drain logic for power good indicator. The first device is used for VCC_INT, where is the main power for PL Internal supply voltage. A combination of two output channel could guarantee 50A current. This triggers the $2nd$ channel of the fourth LTM4650 device, where supply the VPS_MGTRAVCC. The power good signal triggers the $2nd$ stage of channel 1 of device 2 and channel 1 of device 4 that marked by red lines. Then stage 2 starts the stage 3 that marked by green lines on $1st$ channel of $3rd$ device, finally, the channel 2 of device 2 and device 3 first channel is enabled. In such a way also the adjustable voltage on channel 2 of device 3 is provided for I/Os. Under each power, rail is related to one or two blocks that contain the names of power for Zynq. The number in the black circle on the top-right of each block is related to the power consolidation number in Fig (2.25).

2.8 Conclusion

This chapter focuses on the design and hardware validation of the novel PCIe readout card based on Zynq UltraScale+ MPSoC. The x16 lanes of PCIe generation 4 could provide up to 240 Gb/s data throughput with low-latency. The high data throughput combined with the low latency opens a novel dimension of heterogeneous systems where FPGAs, CPUs, and GPUs work together in a distributed architecture. This provides a flexible framework for real-time data processing for several applications, for example, high-energy physics, hadron physics, photon science and beam diagnostics. Some of the large AI models will also benefit from the aforementioned heterogeneous architecture because the computation tasks could be distributed and optimized between FPGAs and CPU/GPU.

Additionally, the full-duplex optical/copper FireFly data link enables a bi-direction data throughput up to 330 Gb/s between FPGA and modern architecture platforms such as Advanced Telecommunications Computing Architecture (ATCA), Advanced Mezzanine Cards (AMC), and uTCA. Such high data throughputs have been designed to satisfy the large-volume data readout produced by modern beam diagnostics systems like KAPTURE and KALYPSO, large detectors like PANDA, and sophisticated cryogenic detectors (ECHo). Novel architecture based on modern remote direct memory access (RDMA) technology i.e. RoCE (RDMA over Converged Ethernet) and iWARP could enable novel concepts of DirectGPU technology where FPGA with GPU/CPU are connected by commercial high-performance ethernet data link.

Chapter 3

System-on-Chip on HighFlex2

The term *Internet of things(IoT)* is one of the significant components of new-generation information technology. Firstly, the underlying key point and infrastructure is internet technology. Secondly, the server and client extends to *things*, *things* are connected together and exchange information with each other. This functionality is achieved through the embedded system using Zynq UltraScale plus device.

The Experimental Physics and Industrial Control System (EPICS) [51] is a set of Open Source software tools, libraries and applications developed collaboratively and used worldwide to create distributed soft real-time control systems for scientific instruments such as a particle accelerator, telescopes and other large scientific experiments. Bringing EPICS directly inside the data acquisition system in the field other than host computer has one benefit that users and scientists could monitor and control the hardware parameters in the same way as controlling other devices and facilities under EPICS application.

The embedded Linux system could help to build EPICS and Webserver on the hardware. Firstly, it is open source and has abundant software resources. Secondly, the Linux kernel is powerful, stable, compact, tailorable, and flexible. The kernel takes charge of process management, memory management, file systems, device control and networks. Thirdly in the field of network, embedded Linux supports all the standard Internet protocols. It has a lot of network management and network services, the user could build routers, firewall or servers.

3.1 Next generation DAQ system integration

With HighFlex2 board, the data transfer topology for large scale experiment is shown in Fig (3.1). When the data coming from detectors go to HighFlex2, there are two principles to process the data in real-time. One is using HighFlex2 as a data transfer centre that bridge the data from detectors to post-processing unit through FireFly. Another principle is to use HighFlex2 as PCIe endpoint under FPGA-GPU heterogeneous system that enables FPGA to transfer data directly to GPU memory [14, 35]. For control and monitoring purpose, as each HighFlex2 card embedded with EPICS and Webserver monitor, that user could use ethernet to remotely coordinates and check the current status of *PL* of the HighFlex2.

To guarantee the functions aforementioned Fig (3.1), the whole system integration in HighFlex2 is demonstrated as Fig (3.2). This could first enable multiple application working together. Firstly, it has the basic high-speed data transfer at PL part, to pass the incoming data to FireFly or PCIe. The machine learning approach could be added in the PL part for data processing like classification or control. Secondly, enables some of the data transfer between PS and PL through AXI DMA. This will be used for the data monitor. Thirdly, the AIX lite mapped regis-

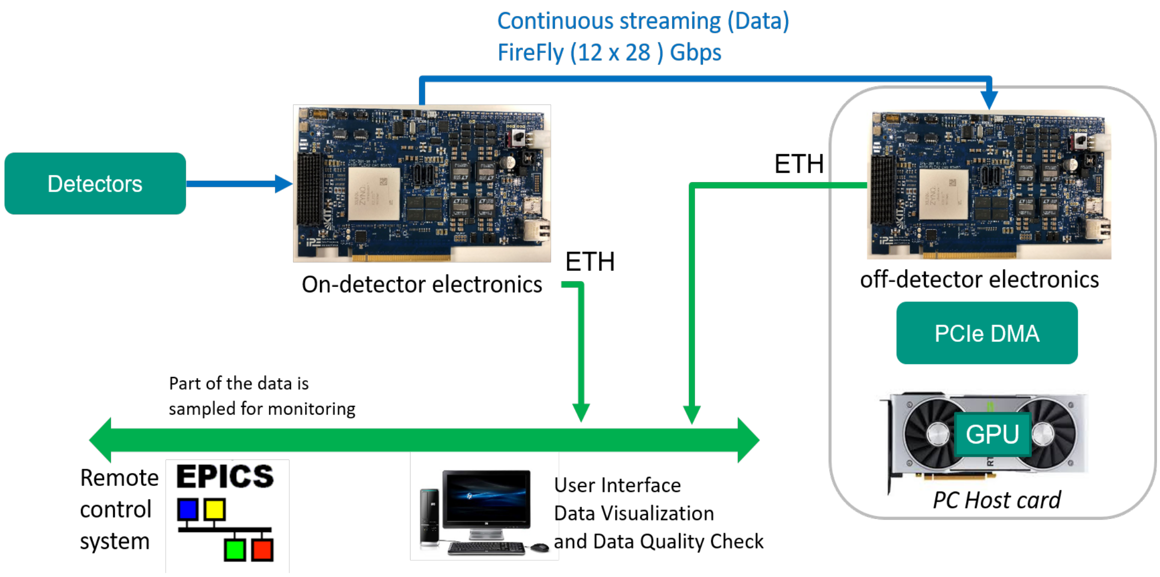


Figure 3.1: the HighFlex2 card in high-performance data transfer for large scale experiment.

ter (blue block) could receive the commands from PS part, and also through this register, could denote some critical signals at PL part, and reflect the status directly to the PS part without going through AXI DMA. At PS part, the Webserver and EPICS also use the AXI Lite interface to control these registers also.

3.2 Hardware design of SoC

The term *hardware* design in Zynq MPSoC means the programmable logic design. This is the footstone of the embedded Linux system. Theoretically, the PL part could leave empty because PS have four gigabit Ethernet controller (GEM), and it is hardly connected on the MIO 64 to 75. But to prove a hardware control in the FPGA side, a data flow is built for a software-hardware co-design as shown in Fig (3.3). The dummy data generator generates pattern data and sends the data to FIFO. The FIFO is considered as a relatively small data buffer between different logic blocks. The high data rate coming from the physics experiment normally exceeds several Gigabit per second, and this proposes the requirement of high capacity buffer for high-speed data. The memory size of FIFO initialized inside FPGA is limited. Therefore, a large storage buffer like external DDR3 or DDR4 is needed. In our case, the HighFlex2 have PL side DDR4 (2GB), which could be used as a regular external FIFO. The PL DDR4 connected with the memory interface generator (MIG) in block design. The data continue to pass through a second FIFO to cross the DDR4 clock to PL clock. Then a small block called tLAST is used to transfer the FIFO interface to AXI4-Stream interface. Then the data goes into AXI Direct Memory Access (DMA) that transfer the AXI4-Stream data into memory-mapped data located at PS side memory. Through this architecture, the PS side has the information at FPGA side and a webserver could be built as an application in the embedded Linux to monitor the data coming from PL easily. There is also some control register in the PL part, which could be accessed by AXI4-Lite interface. All the blocks in the PL part that have AXI4-Lite interface could be integrated into the embedded Linux system, and control or accessed by ARM.

The hardware block is built in Xilinx tool Vivado. Fig (3.3) demonstrates the full data chain from a dummy data generator to PL DDR4 and finally reach to PS DDR4. The left-

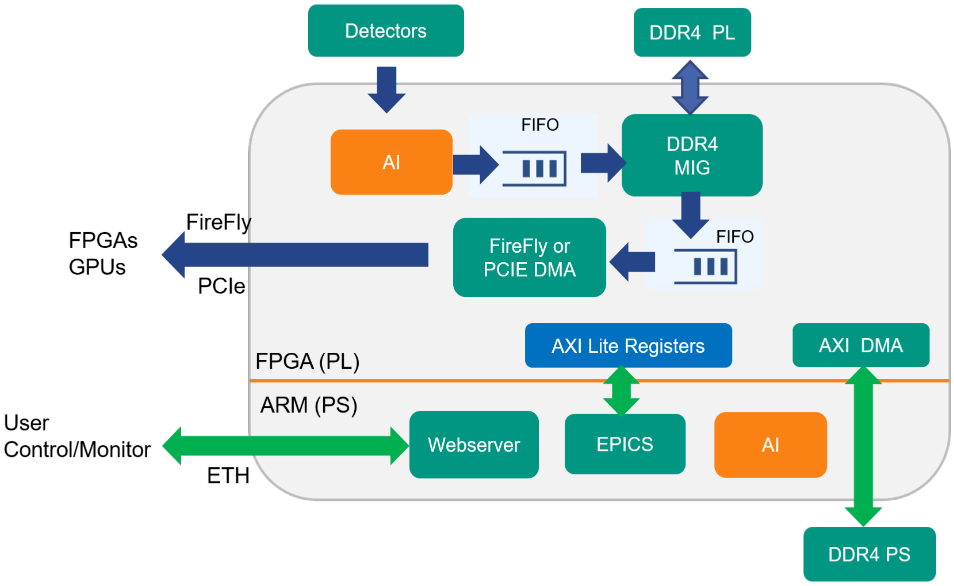


Figure 3.2: The system integration combines all the PS and PL components together, thus enable multiple functionalities at the same time.

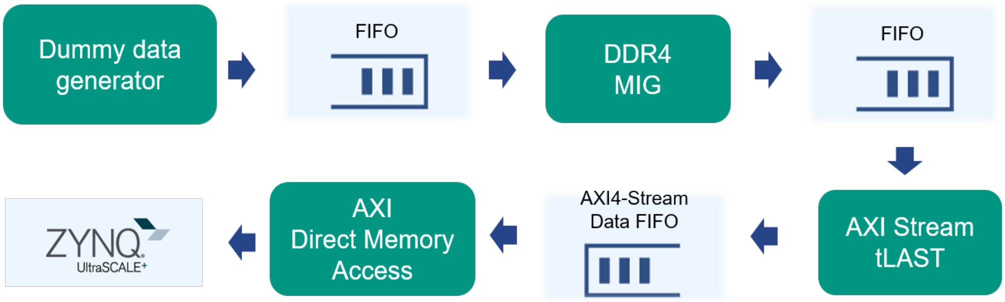


Figure 3.3: The FPGA hardware block design for system on chip. This infrastructure will test two important PL components: PL DDR4, and AXI DMA. Then the embedded Linux will be built based on this hardware design. The webserver will be used to take the data from dummy data generator and shows to the user. The EPICS is used to control the stage like start/stop dummy data transfer, start/stop DDR4 transfer, and start/stop DMA transfer.

bottom block Zynq UltraScale IP in the Fig (3.3) generally represents the PS part of Zynq [52]. It has the software interface like AXI I/O groups around the Zynq UltraScale+ Processing System. These interfaces could be divided into two parts: the outside interfaces and internal interfaces. For outside interface, for example, the PS peripherals (UART, USB, etc) and PS DDR4 is included in the configuration of this IP. And for the internal interface, like the AXI_HP interface, is the bridge between PS and PL. The AXI DMA shown in this Fig (3.3), is connected with this AXI_HP.

After the FPGA bitstream is generated (contains all the PS interfaces, PL, and PS-PL connections), the Hardware Definiton File) (.hdf) could be exported for the embedded Linux. The HDF is a container file that contains all the information needed to build a platform for a users target devices, such as the CPU (or CPUs, here is actually ARM cores), Buses (different kind of AXI4 buses), IP and the ports and pins used in the system such as interrupts.

3.3 Yocto Project

The ecosystem to develop custom embedded Linux distribution for Zynq device is Yocto Project (YP). The Yocto Project [53] is an open source project that reduce developing time/money. It is hosted by Linux Foundation and have a plenty of industrial participation such as Xilinx, AMD, Intel, TI, DELL and so on. The first step is to define the hardware. The bblayers.conf and local.conf files `/build/conf` could configure the build. In the local.conf, the user needs to point out the variable "HDF_PATH" that point to the HDF above.

One of the typical character for Yocto Project is that it has Layer Model. A collection of related recipes. The user could use multiple layers to partition their embedded Linux functionality. It is relatively easy to generate several applications use command:

```
bitbake-layers create-layer [name of layer]
```

Layers are repositories that owns sets of instructions to tell how the embedded system should do. In this chapter, there are two layers: EPICS and Webserver. Each layer contains the necessary drivers and operating mode.

3.4 EPICS

EPICS is a distributed client and server architecture. The client normally means the user application and scientists who operate the machine. EPICS software mainly consists of Input/Output Controller (IOC) and Operator Interface (OPI). For server, the IOC needs to implement, and for client, it needs OPI. For a complete IOC application, it includes Channel Access (CA) server interface, the Database access interface, dynamic database, device support module, device driver module, and so on. The OPI level is the EPICS tools running on Linux operating system.

3.4.1 EPICS on hardware

To integrated the EPICS into hardware, first task is to put the EPICS Base [54] into one Yocto layer as an embedded Linux application. The version of EPICS is documented in [55]. The whole software structure is shown in Fig (10) at Appendix B.

The client can probe the entire network or a single server address, while the server can make itself visible to the entire network or a single client address. With this, the client can access several servers at one time, for instance to control a common parameter present on multiple servers/hardware. The parameters are represented in the form of database entries. The database consists in a list of "records". A record has a number of variables called Process Variables (PV) which can be inputs or outputs or inouts. These PV are the variables accessible by the client, and also is the key channal could access the hardware.

A number of pre-existing records are distributed with the EPICS base. A "sub" type record is employed to call a subroutine written in C language. This record is created in the epics-dmareg.db saved in */myiocAPP/Db*.

Record Creatation

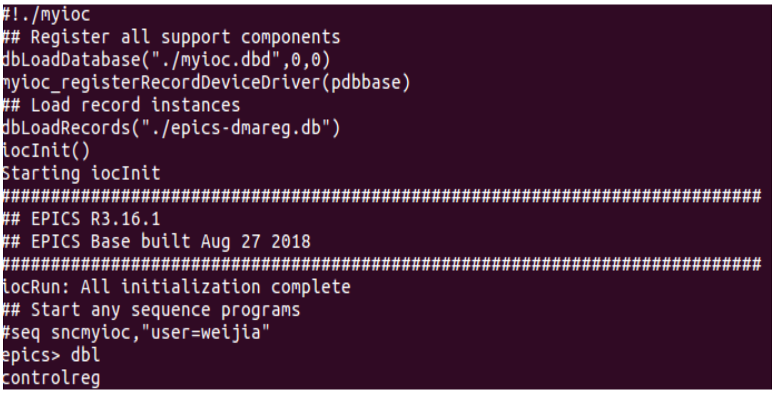
/app/myiocApp/Db/epics-dmareg.db:

record(sub, controlreg) $\{field(SNAM, ''epics\_dmaset'')\}$

At each time the PV in the record are accessed by the client, the C application that have the memory mapping similar with last section will run. It brings the link between EPICS client request to the hardware. This link between both is established by the "SNAM" field. In this case, the "epics_dmaset" function will be called at each time when the PV in the record named "controlreg" are accessed by user. Moreover, this record is directly mapped to the hardware control register, which is a 32 bits width of register. Some of the bit of this register is linked with control signal in PL part for example, the PL reset, dummy data start, DDR4 read enable, etc.

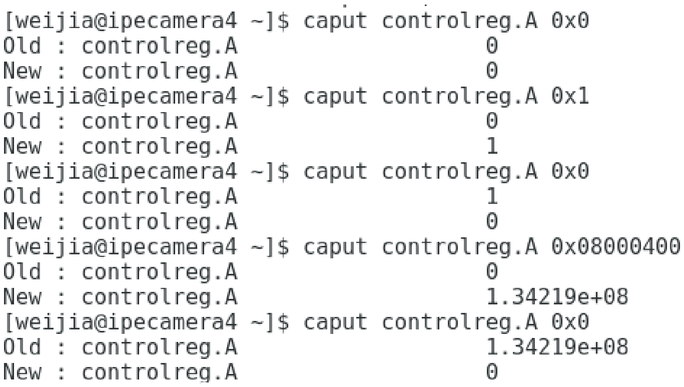
3.4.2 EPICS control on HighFlex2

Once the EPICS server is running on the HighFlex2 board, in terminal one could have the following information shown in Fig (3.4). On the HighFlex2 side, firstly loads the database and opens a command prompt. When type command "dbpr controlreg", will print out the PV of the record and its own current value. At client side, the user could change the value on the PV and the records as shown in Fig (3.5)



```
#!./myioc
## Register all support components
dbLoadDatabase("./myioc.dbd",0,0)
myioc_registerRecordDeviceDriver(pdbbase)
## Load record instances
dbLoadRecords("./epics-dmareg.db")
iocInit()
Starting iocInit
############################################################################
## EPICS R3.16.1
## EPICS Base built Aug 27 2018
############################################################################
iocRun: All initialization complete
## Start any sequence programs
#seq sncmyioc,"user=weijia"
epics> dbl
controlreg
```

Figure 3.4: EPICS embedded server running on hardware



```
[weijia@ipecamera4 ~]$ caput controlreg.A 0x0
Old : controlreg.A                 0
New : controlreg.A                 0
[weijia@ipecamera4 ~]$ caput controlreg.A 0x1
Old : controlreg.A                 0
New : controlreg.A                 1
[weijia@ipecamera4 ~]$ caput controlreg.A 0x0
Old : controlreg.A                 1
New : controlreg.A                 0
[weijia@ipecamera4 ~]$ caput controlreg.A 0x08000400
Old : controlreg.A                 0
New : controlreg.A                 1.34219e+08
[weijia@ipecamera4 ~]$ caput controlreg.A 0x0
Old : controlreg.A                 1.34219e+08
New : controlreg.A                 0
```

Figure 3.5: EPICS client running on host PC

The client can be any computer where the base distribution is installed. It needs to be able to access the server through its IP address. In the generic configuration, the server signals its presence at regular intervals to all the IP addresses encompassed by its broadcast address. The client sends its commands the same way. Therefore, this configuration works if both the client and the server have the same broadcast address. In our case the host computer and HighFlex2 are connected within the same router. Then at client side, the PV is recognized as shown in Fig (3.5).

If user change the control register from 0 to 1 and back to 0, then that means a reset. When it became 1, the LEDs have the following status:

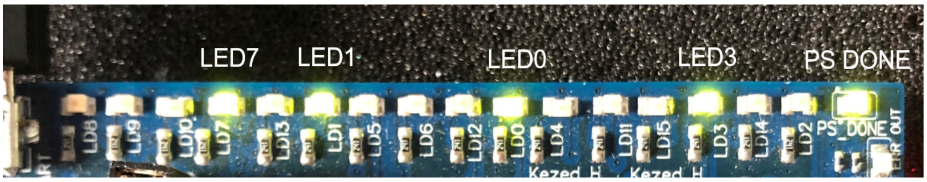
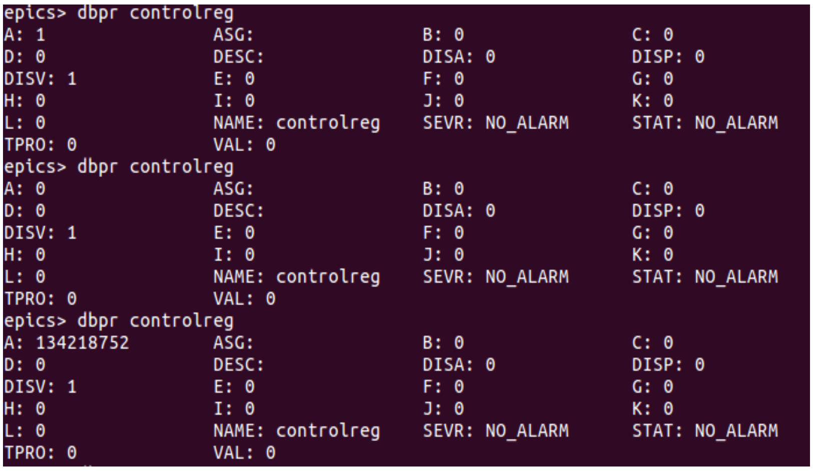


Figure 3.6: EPICS client control in reset mode

The $LED7$ is directly connected with control register bit[0]. This bit is the reset bit, and active high. When the user set control register to $0x00000001$, then this bit is one, the FPGA firmware became reset mode. The $LED1$ is connected with the global reset, which is the AND gate between PL DDR4 calibration done signal and control register reset bit. The $LED0$ and $LED3$ is AXI DMA FIFO programmable full and programmable empty respectively. When it in reset mode, both signal are pulled to high.

When user set the controlreg record field A back to $0x0$ as shown in Fig (3.5), the reset mode is de-asserted. Fig (3.8) is the LEDs result. Only the FIFO empty signal is left. The server side could also see the changed value at hardware side Fig (3.7).



```
epics> dbpr controlreg
A: 1                ASG:                B: 0                C: 0
D: 0                DESC:               DISA: 0             DISP: 0
DISV: 1             E: 0                F: 0                G: 0
H: 0                I: 0                J: 0                K: 0
L: 0                NAME: controlreg    SEVR: NO_ALARM      STAT: NO_ALARM
TPRO: 0             VAL: 0
epics> dbpr controlreg
A: 0                ASG:                B: 0                C: 0
D: 0                DESC:               DISA: 0             DISP: 0
DISV: 1             E: 0                F: 0                G: 0
H: 0                I: 0                J: 0                K: 0
L: 0                NAME: controlreg    SEVR: NO_ALARM      STAT: NO_ALARM
TPRO: 0             VAL: 0
epics> dbpr controlreg
A: 134218752        ASG:                B: 0                C: 0
D: 0                DESC:               DISA: 0             DISP: 0
DISV: 1             E: 0                F: 0                G: 0
H: 0                I: 0                J: 0                K: 0
L: 0                NAME: controlreg    SEVR: NO_ALARM      STAT: NO_ALARM
TPRO: 0             VAL: 0
```

Figure 3.7: EPICS embedded server reaction

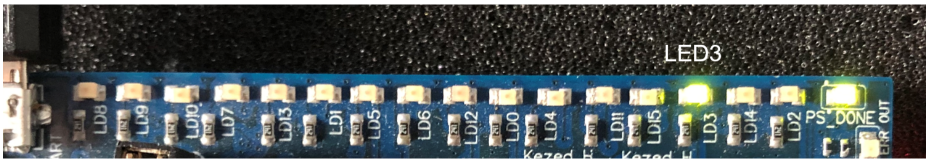


Figure 3.8: EPICS client control de-assert reset mode

When user starts the data transfer from dummy data generator, first the data transfer enable $LED6$ is on. The AXI DMA FIFO is full because the PC driver side don't ask data now. And also the blinking $LED2$ represent that the PL DDR4 writing control signal is working. By this the EPICS could control the PL side blocks that shown in Fig (3.3)

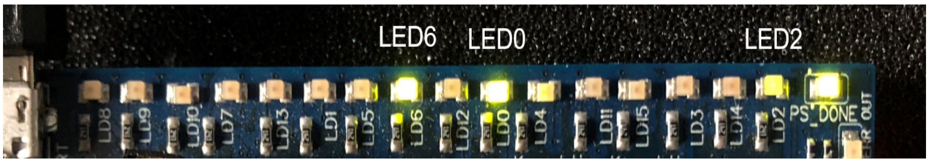


Figure 3.9: EPICS client starts data transfer

3.5 Performance comparison

To measure the performance of the Zynq EPICS server and CPU EPICS server, thesis compile the same ’sub’ type of record with the same PV name ’controlreg’ on the CPU. The testbench is a 100 times loop running to continue write the same value on the "controlreg.A" PV of CPU and Zynq server. EPICS camonitor is estabilied to monitor the access of PV. Every time a write access will cause the callback function ($HighFlex2$) that generate one timestamp to record this interaction. The difference of each of two adjacent timestamp is measured, which will be the latency of server reaction time.

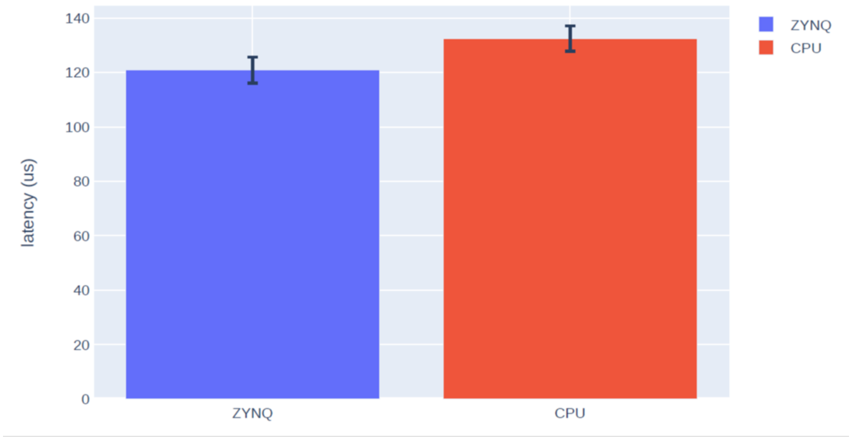


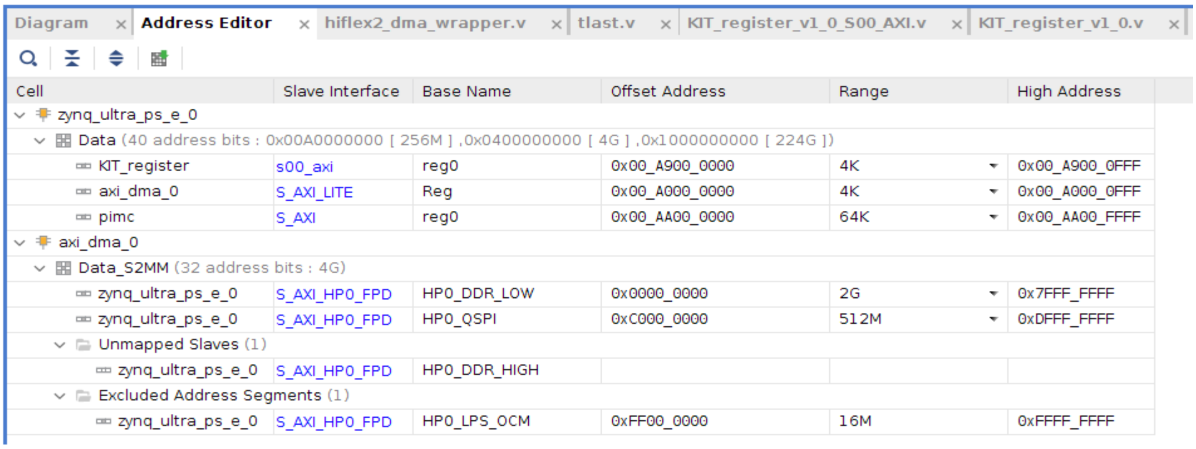
Figure 3.10: EPICS Server run trip latency comparision

Fig (3.10) illustrates the EPICS server reaction time comparision of Zynq and CPU. The Zynq have about 120 µs average time and CPU own around 132 µs average time. Zynq have standard error about 4.8 µs and CPU own around 4.67 µs standard error.

3.6 Webserver application

For Webserver application, our purpose is to start the dummy data transfer, then let the data pass through all necessary PL blocks, goes into the PS memory, and monitored by Webserver.

All these operations needs access to global control register, AXI DMA control register, and PS memory. The memory map method is one efficient way to access all of these addresses. After boot up the Linux system on Zynq, there is one file called: `/dev/mem`. This character device represents the whole physical memory of our hardware definition. The offset for each device or peripherals could be edited in Vivado Memory Editor, shown in Fig (3.11). The offset could be customized and user application in embedded Linux will reference this address to access the corresponding devices. For example, the global control register KIT_register, the *reg0* has offset address 0x00A9000000 and AXI DMA control register has offset address 0x00A0000000.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
zynq_ultra_ps_e_0					
Data (40 address bits : 0x00A0000000 [256M] ,0x0400000000 [4G] ,0x1000000000 [224G])					
KIT_register	s00_axi	reg0	0x00_A900_0000	4K	0x00_A900_0FFF
axi_dma_0	S_AXI_LITE	Reg	0x00_A000_0000	4K	0x00_A000_0FFF
pimc	S_AXI	reg0	0x00_AA00_0000	64K	0x00_AA00_FFFF
axi_dma_0					
Data_S2MM (32 address bits : 4G)					
zynq_ultra_ps_e_0	S_AXI_HP0_FPD	HP0_DDR_LOW	0x0000_0000	2G	0x7FFF_FFFF
zynq_ultra_ps_e_0	S_AXI_HP0_FPD	HP0_QSPI	0xC000_0000	512M	0xDFFF_FFFF
Unmapped Slaves (1)					
zynq_ultra_ps_e_0	S_AXI_HP0_FPD	HP0_DDR_HIGH			
Excluded Address Segments (1)					
zynq_ultra_ps_e_0	S_AXI_HP0_FPD	HP0_LPS_OCM	0xFF00_0000	16M	0xFFFF_FFFF

Figure 3.11: Vivado Address Editor denotes the offset address that used in both bare-mental application and embeded Linux application.

The function *mmap* could let user application (user space) directly access the device memory (kernel space). Firstly, the function $open()$ could use to open the device file `/dev/mem`, then use the offset address to map the file to the virtual address. The $mmap()$ function will return one address as a pointer that the user could handle this device as a normal pointer in C/C++ coding style.

Through webserver the user could both control or monitor the device. There is a relatively long way from Web to hardware. Fig (3.12) shows how to use Common Gateway Interface (CGI) defines how the information transfer between Webserver and client-side scripts.

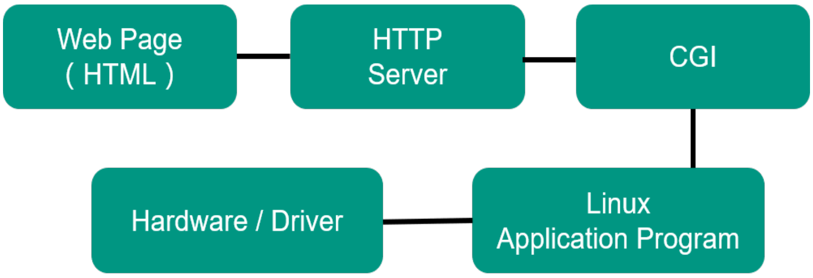


Figure 3.12: Webserver CGI Control. Through CGI, the Web Page is linked together with user application that the user could map the necessary data on the Web. In this case, the PL data is transferred to the PS DDR4, then it is mapped directly into the Web page.

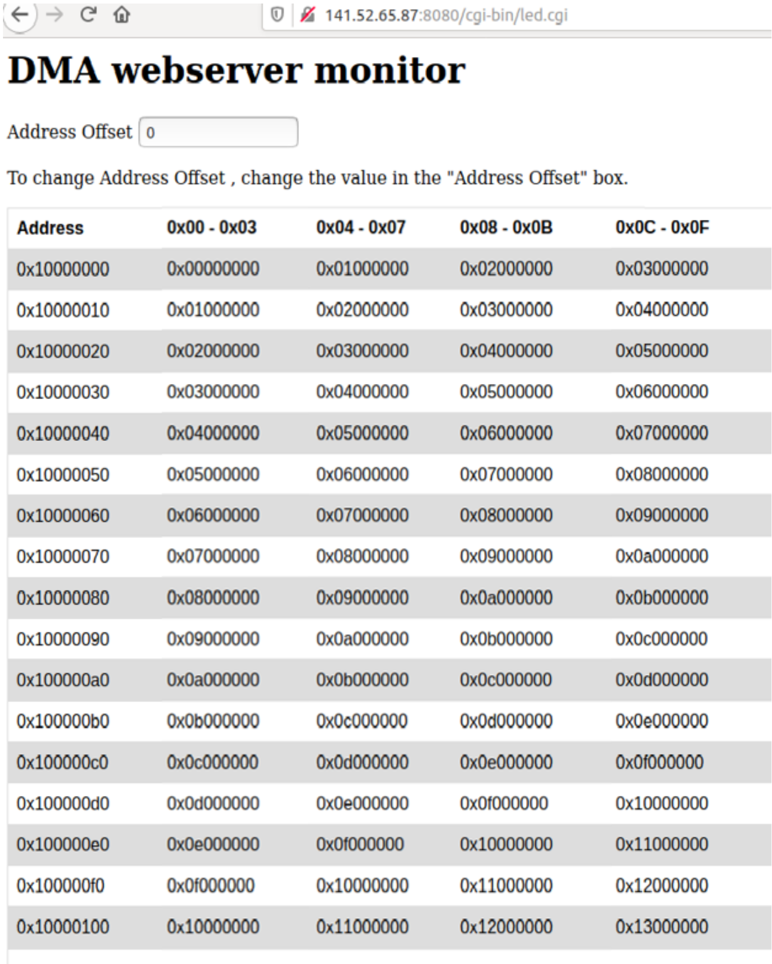
Fig (11) in Appendix B shows the file structure of webserver Yocto Layer named with *meta-ww*. The webserver interface is defined in *.htm* files in Fig (11) (Appendix B). The CGI source files inside filder *led.cgi* is the CGI implemented by C, and this connect the content on web with Linux application. In CGI also map all the hardware or driver into system memory where the CGI Linux application could access.

Because the CGI application is installed in `/home/httpd` at HighFlex2 embedded Linux system, then a *httpd* command could be used to start one HTTP server on HighFlex2:

```
httpd -p 8080 -h /home/httpd/
```

The command -h point to where the web file *.htm* is located. The CGI transfer the data from the user Linux application to the web. Through open the address in a web browser:

```
<HighFlex2 IP Address>:8080/cgi-bin/led.cgi?page_offset=0
```



141.52.65.87:8080/cgi-bin/led.cgi

DMA webserver monitor

Address Offset 0

To change Address Offset , change the value in the "Address Offset" box.

Address	0x00 - 0x03	0x04 - 0x07	0x08 - 0x0B	0x0C - 0x0F
0x10000000	0x00000000	0x01000000	0x02000000	0x03000000
0x10000010	0x01000000	0x02000000	0x03000000	0x04000000
0x10000020	0x02000000	0x03000000	0x04000000	0x05000000
0x10000030	0x03000000	0x04000000	0x05000000	0x06000000
0x10000040	0x04000000	0x05000000	0x06000000	0x07000000
0x10000050	0x05000000	0x06000000	0x07000000	0x08000000
0x10000060	0x06000000	0x07000000	0x08000000	0x09000000
0x10000070	0x07000000	0x08000000	0x09000000	0x0a000000
0x10000080	0x08000000	0x09000000	0x0a000000	0x0b000000
0x10000090	0x09000000	0x0a000000	0x0b000000	0x0c000000
0x100000a0	0x0a000000	0x0b000000	0x0c000000	0x0d000000
0x100000b0	0x0b000000	0x0c000000	0x0d000000	0x0e000000
0x100000c0	0x0c000000	0x0d000000	0x0e000000	0x0f000000
0x100000d0	0x0d000000	0x0e000000	0x0f000000	0x10000000
0x100000e0	0x0e000000	0x0f000000	0x10000000	0x11000000
0x100000f0	0x0f000000	0x10000000	0x11000000	0x12000000
0x10000100	0x10000000	0x11000000	0x12000000	0x13000000

Figure 3.13: Result of DMA webserver. From the webserver, the user could monitor the data coming from PL side. Also, the address offset could change to other parts to verify the data transfer is consistent or not.

The Fig (3.13) displays the contents of the part of the memory allocated to the DMA operation. Therefore, the first address in the "Address" column on the left is 0x0000000000, it is actually interpreted as an offset from the DMA's memory within the global memory shown in Fig (3.11). The data input in this demo is the dummy data generator, which is actually a counter that was implemented in the hardware design part of PL. The data is represented in Little Endian format, so the increment is visible on the leftmost byte of the four bytes words. The counter must be followed line by line, within the first column. The data in the three other columns also come from the data generator module but are shifted each by one, compared to their preceding column. The Address Offset entry can be employed to modify the offset, and thus the part of memory that is displayed. On a page, the first 256 bytes are printed, corresponding the addresses of sixteen 32-bytes lines. The Address Offset value shifts the addresses by one full web server page: an offset of 0 prints the addresses 0x00000000 to 0x000000F0, an offset of 1 print the addresses 0x00000100 to 0x000001F0 and so on.

3.7 Conclusion

This chapter introduces a unified system with HighFlex2 Zynq device, where the FPGA firmware and embedded Linux device driver is integrated. The development of embedded Linux under Yocto Project (YP) requires a hardware description. This chapter provides Linux with such description including Programmable Logic 2GB DDR4 memory, FIFOs, and AXM DMA. Within this hardware description, some of the critical blocks are packaged as AXI4-Lite IP so that PS could access through AXI interconnector for controlling and monitoring purpose. The userspace memory implemented on the FPGA is mapped in the Linux application, i.e. EPICs, TANGO, etc. Webserver and EPICS are developed as YP layers, where Web server could be employed to send the status and flags on the quality of the data collected by the detector and EPICS is employed for the integration of the whole detector system within the experiment control system in a distributed client and server architecture.

System integration dramatically improves the scalability and flexibility of the DAQ system. ARM and FPGA could share information and tasks. This opens multiple possibilities for developers to allocate the different part of the particular application into different computing resources on heterogeneous MPSoCs.

A higher level of a heterogeneous system is introduced. In the system, GPUs and FPGAs provide a high-performance data transfer architecture for data-intensive experiments. Within such a heterogeneous system, HighFlex2 plays an important role in data transferring.

Part II

Artificial Intelligence on Hardware

Chapter 4

Machine Learning preliminaries

4.1 Introduction on Supervised Learning

In [56], neurophysiologist Warren McCulloch and mathematician Walter Pitts proposed the mathematical description and structure of neurons and demonstrated that any computational function could be simulated with enough simple neurons connected and running in synchronized operation. The word *Deep* corresponds to the word *Shallow*, which means the neural network has at least one hidden layer and also means the neural network has at least 2 layers in total (the number of hidden layers plus the output layer, the input layer is not taken into account). From 1990 to 2012, the theory and method of machine learning have been improved and enriched, there comes Support Vector Machine (SVM) [57], AdaBoost [58], Manifold Learning [59], LSTM [60] and Random Forest [61]. SVM represents the victory of nuclear technology, which is the idea that by implicitly mapping input vectors into higher-dimensional space, nonlinear problems can be handled well. AdaBoost, on the other hand, represents a triumph for the integrated learning algorithm, which achieves amazing accuracy by integrating a few simple weak classifiers.

Although the backpropagation was already proposed [62] in 1986, it was limited by the algorithms itself in several different aspects: the vanishing gradient problem, the limitation of lack of training data and computing power. In the competition with SVM, the neural network was inferior to SVM for a long time. In 2006, Hinton et al. proposed a method [63] to train the deep neural network by using Restricted Boltzmann Machine to train each layer of the multilayer neural network, obtain the initial weight, and then continue to train the whole neural network. This solves the gradient vanishing issue during training deep neural network.

In the year of 1980, Fukushima [64] proposed the neurocognitive machine and convolutional neural layer sharing the weights, which is regarded as the prototype of the convolutional neural network. In 2012, AlexNet [65], a deep convolutional neural network (CNN) invented by Hinton group, first succeeded in image classification, and then it was applied to various problems of machine vision, including general target detection, face detection, pedestrian detection, face recognition, image segmentation, image edge detection, etc. In these problems, the convolutional neural network has achieved the best performance. In the later years from 2013 to 2016, during ImageNet image recognition competition [66], comes the ZFNet [67], VGGNet [68] and ResNet [69]. In the same periods, deep learning network structure, training methods and GPU hardware have continuous progress, it is then constantly conquering the battlefield in fields of computer vision problems.

In another class of problems called time series analysis, Rerurrent Neural Networks (RNN) [60] have achieve success. Typical examples are speech recognition, natural language processing, and the use of deep RNN can significantly improve the accuracy of speech recognition until it meets the requirements of practical application.

4.2 Development of Reinforcement Learning

From March $8th$ to March $15th$, 2016, in Seoul, South Korea, there was a five-game Go match between AlphaGo [19] and Lee Sedol, who is ranked second in international titles of Go player. AlphaGo won four over five matches. Where in AlphaGo, Monte-Carlo tree search and deep reinforcement learning have been played important role. This success in AI highlighted another major branch of machine learning, Reinforcement Learning (RL).

The development of reinforcement learning has a long history. Minsky first proposed the concepts and terms of “reinforcement" and “reinforcement learning" in 1954. The term “optimal control" came into use in the late $1950s$ to describe the problem of designing controllers to minimize or maximize the behaviour of dynamic systems over time. One of the method that could solve such problem is by extending the theory of Hamilton [70] and Jacobi[71]. This method is called Dynamic Programming (DP) [72], using the concept of state and value functions of dynamic systems or *optimal return functions* to define function equations, now commonly referred to as Bellman equations. Bellman [73] also introduced the discrete stochastic version of the optimal control problem known as Markov Decision Processes (MDPs), and Ron Howard [74] designed the policy iteration method for MDPs. All of which is the fundamental components in reinforcement learning. These will be introduced in Chapter 6. Ian Witten first proposed the temporal-diference learning rule [75, 76]. Thus, Witten’s 1977 paper conbined both major threads of reinforcement learning, research, the “trial-and-error" learning and the optimal control. This combination is enhanced by Watkins in 1989 [77], where the *Q-learning* is proposed.

In 2013, DeepMind using the neural network in Atari game finished multiple games beyond the level of human players [78]. The strategy is to use deep neural network as function approximator of value function and policy, this avoids the problem of table storage space is large, and slow in query the table. This combination of deep learning and reinforcement learning is called Deep Q-Learning. Based on that, and the Actor-Critic structure [79], there are several prevalent reinforcement learning algorithm is developed, for example, the Proximal Policy Optimization (PPO) [80], the Asynchronous Actor-Critic Agents (A3C) [81], and the Deep Deterministic Policy Gradient (DDPG) [82].

4.3 Basic structure in Deep Learning

In the following part will mainly introduce basic components used in supervised learning and reinforcement learning: neural networks. The fully connected neural network and convolutional neural network will be delivered.

4.3.1 Introduction of fully connected neural network

In this section will introduce the structure of neural network and convolutional neural network. An simplified artificial neuron is called a *perceptron* [56]. The working principle of a perceptron is taking several of inputs, and produce a single binary output, as shown in the Fig (4.1).

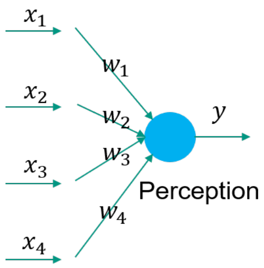


Figure 4.1: A *perceptron* takes several inputs, applys the weight on the conbinations between input and perceptron, doing linear operation, then produce one output

The perceptron's output is either 1 or 0 determined by the weighted sum $\sum_i w_i x_i$ is going over the threshold or not.

Based on such simple structure, one could adds the bias and activation function and get one complete neuron. The activation normally is non-linear functions. If the nonlinear function is used, the activation function introduces nonlinear factors to the neurons, so that the neural network can approximate any nonlinear function arbitrarily, and then the neural network can be applied to many nonlinear models. Fig (4.2) shows the working principle of one neuron, the output is not 0 or 1, instead, it is determined by the according activation function. Here the example show the sigmoid function, then the output is $\sigma(\sum_i w_i x_i + b)$.

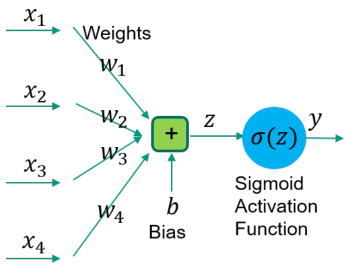


Figure 4.2: A *neuron* takes several inputs, applys the weight on the conbinations between input and perceptron, doing linear operation with bias, then pass though the activation function to produce one output

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{4.1}$$

Equ (4.1) is the sigmoid activation function. There are multiple different activation functions discussed in [83]. The most commonly used are ReLU, Sigmoid and Tanh function. If these neurons are combined layer by layer, will form Fully-Connected (FC) Feedforward Neural Network as shown in Fig (4.3). Such FC NN could be extened in any form and structure in this way to fit for different input data and more deep layers.

4.3.2 Introduction on convolutional neural network

The name "Convolutional" comes from one of the most important operations in the CNN: convolutional layer [84]. A basic CNN structure is conbined with multiple convolutional layers, activation layers, and maxpooling layers.

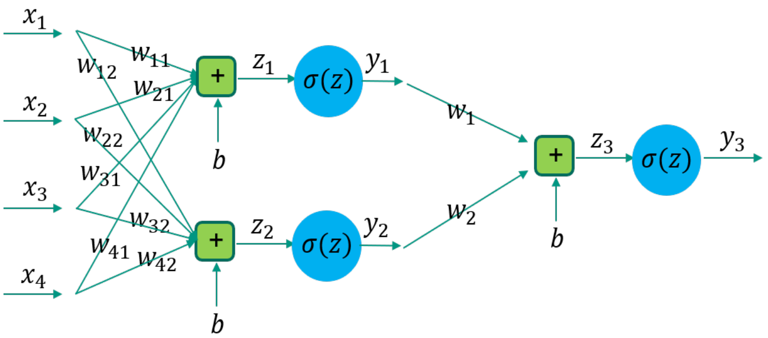


Figure 4.3: Two *neurons* form the first layer and its two outputs connected with the input of next layer, which have only one neuron

Convolutional layer

The word *convolutional* is different than the same word in signal processing [85]. As shown in Fig (4.4), the input data is 7 times 7 matrix, and *convolution kernel*(also named *filter*) is 3 times 3 matrix.

In convolution neural network, the convolution kernels of layer (or filter) is similar to a sliding window, in the input image sliding back and forth in a specific step on the whole training process of the network, the convolution kernel containing weights will also be updated until the training is completed. After each convolution operation, one of the local characteristics of the input image map is obtained, and form the output image.

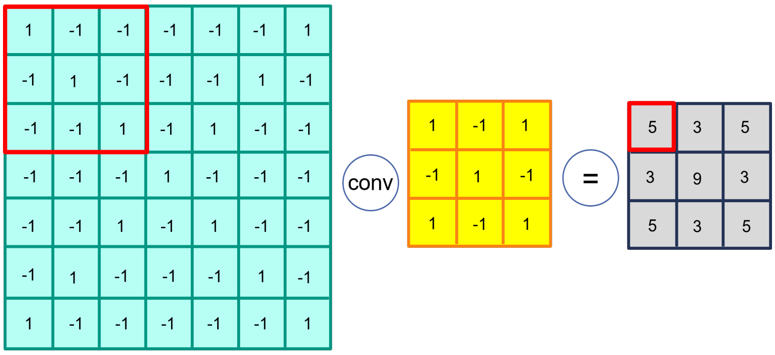
In the whole training process of the network, the convolution kernel containing weights will also be updated until the training is completed.

Under this case, the 3 by 3 convolution kernel will moving from the top left corner to the right bottom corner, and the moving stride here is 2, means the filter moves over the image 2 pixel per time. As denoted by the red frame in figure, the 3 by 3 filter will cover a 3 by 3 region of image, and the filter coefficient will multiply the image pixel accordingly, this will get $1 \times 1 + (-1) \times (-1) + (-1) \times (1) + (-1) \times (-1) + 1 \times 1 + (-1) \times (-1) + (-1) \times 1 + (-1) \times (-1) + 1 \times 1 = 5$

The formulation expression under two-dimensional convolution is [86]:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m,j-n) \tag{4.2}$$

The two dimension I is the input, the 2-D filter(kernel) is K.



1	-1	-1	-1	-1	-1	1
-1	1	-1	-1	-1	1	-1
-1	-1	1	-1	1	-1	-1
-1	-1	-1	1	-1	-1	-1
-1	-1	1	-1	1	-1	-1
-1	1	-1	-1	-1	1	-1
1	-1	-1	-1	-1	-1	1

1	-1	1
-1	1	-1
1	-1	1

5	3	5
3	9	3
5	3	5

Figure 4.4: the convolutional layer operation

Normally at each convolutional layer, the filter is not only one, but several. For example in ResNet [69], the *conv1* layer have 64 seven by seven filters with stride step 2. Even though, this reduce the parameters dramatically compared with fully connected layer. Because all the pixels in the image are sharing the same filters. This brings the possibility of parallelized computing on FPGA.

The output image size could be calculated:

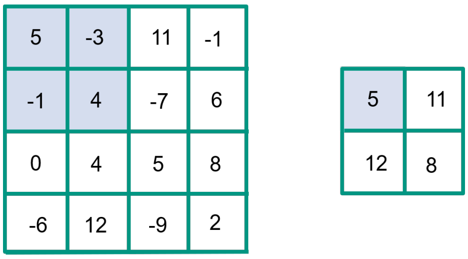
$$W_o = \frac{W_i - K_w + 2P}{S} + 1 \tag{4.3}$$

$$H_o = \frac{H_i - K_h + 2P}{S} + 1 \tag{4.4}$$

In Equ (4.3, 4.4), the W_o and H_o is the output image width and height, the W_i and H_i is the input image width and height. the K_w and K_h is the width and height of kernel, the P is padding size, and the S is stride step size. The padding operation is used to adjust the final output image size.

Pooling Layer

There are two widely used pooling operations: average pooling and Max pooling, of which the maximum pooling is the most commonly used operation and its effect is generally better than average pooling. Pooling layer is used to reduce the dimension of feature space on the convolutional neural network, but not the depth. The maximum value of input areas is choosed when the maximum pooling layer is used, and the average of the input areas is used when the average pooling is used.



5	-3	11	-1
-1	4	-7	6
0	4	5	8
-6	12	-9	2

5	11
12	8

Figure 4.5: Max Pooling Operation

As shown in the Fig (4.5), the left side is the output image from the convolutional layer, also called feature map. The right side is the output of max pooling layer with a pooling region width of two by two pixels. The pooling region is different depends on different application. After the pooling layer is the activation layer, the same with fully connected layer aforementioned in fully connected layer.

4.4 Implementation consideration of Supervised Learning

The supervised learning approach belongs to *statistics* problem. Different from *probability* problem, which use known models and parameters to predict the data, the *statistics* problem is to use the collected/sampled data to reconstruct the models and its parameters.

Generally speaking, collecting training dataset is the starting point to build one supervised learning model. The training dataset is a collection of samples. Each sample has independent features (the input variable of the model) and the class label (the output of the model). The task of the training process is to decrease the difference between prediction value and label value and is happening offline on CPU/GPU in general. For training large model, some of the typical GPU based machine learning acceleration platform is introduced in [87, 88].

After the supervised learning model being trained and verified, the project needs deployment. The same as privious training process, CPU/GPU could be used for accelerating the inference part of the model, and this is because, in principle, the calculation required in forward pass (inference) and backpropagation (training) are similar: both are multiplication and addition of matrices. There are plenty of tools that help machine learning experts to transfer their knowledge into hardware projects, for instance, the Tapir XLa [89] and TVM [90], etc. These machine learning compiler could generate the source code of CUDA device or exacuatable files from the machine learning model from Caffe, Flux, MXNet or Tensorflow.

FPGAs are considered as superior energy efficiency devices, providing high data bandwidth and low latency. [91, 92] surveyed several hardware deployment methods for example, data transfer and on-chip memory design, pipelining and quantization. The [92], [93] and [94] are three typical instances of these solution. All of these are the accelerator for Convolutional Neural Network (CNN). Although CNN are successful on image processing and computer vision, this didn't means CNN cound cover the whole supervised learning algorithm set. Different approaches are suitable for different application scenarios.

4.4.1 Implementation consideration of Reinforcement Learning

Because of the difference in mathematical theory foundation, reinforcement learning application implementation and supervised learning implementation have a different viewpoint of consideration. This difference is especially evident during the training process of each method. As mentioned above, the supervised learning is learning offline, and in princile, it have no time constraints on training process.

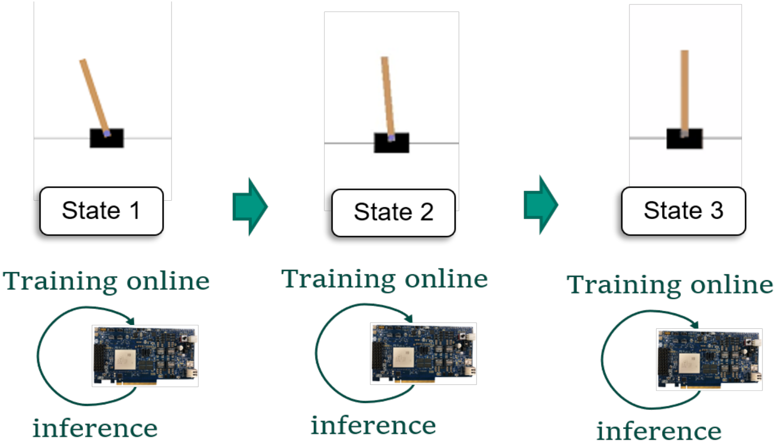


Figure 4.6: The reinforcement learning training process happens under MDP, and interacting with environment on-line, the time-constraint depends on the step intervals

The reinforcement learning (RL) have a training process that agent (RL) needs to interact with environment. The whole form of interaction is under finite Markov Decision Process (MDP). Every step, the agent needs to decide which action or movement should be taken. And this *consideration time* for agent depends on the environment. Consider a OpenAI Gym [95] testbench controlling system: CartPole system [96]. The agent is the cart as shown in Fig (4.6), and it may move left or right to keep the pole to be balanced as long as possible. Firstly, this movement needs to decide by agent in this time slot before the environment change to next state. And this decision is inference of policy neural network. Secondly, according to some structure of algorithm like Deep Deterministic Policy Gradient (DDPG) [82], the *one-step* training needs to finish also in this time slot. This brings the strict time constraint when this cart-pole problem is not simulation on CPU but in real experiment, which have a clear view in Fig (4.6), where the hardware needs to finish this two task in one-step time slot. Chapter 7, 8 and 9 will give the hardware solution that could guarantee the real-time requirements.

4.4.2 CPU, GPU and FPGA acceleration

CPU (Central Processing Unit) is one of the main components of the computer. The main functionality is to translate computer instruction and execute a complex computer program. Trough the increasing clock frequency, the performance have had an ever-increasing trend until the year 2005 predicted by Moore's law[97]. This trend comes to the end because of the major factor of the power wall. The reduction of the physical size of the transistor for the same logic or function of the processor will cause an increased power density, resulting in the electrical leakage and heat dissipation problems.

The Graphic Processing Unit (GPU) is an important component of graphics system structure and a link between computer and display terminal for graphics rendering, shader and processing in the early days. The computing architecture Compute Unified Device Architecture (CUDA) [98] that enables GPU to be general computing unit tasks that require high computational power. The benchmarks like H.264, Finite-Difference Time-Domain (FDTD), MRI-Q and many others mentioned in [99] provides the use cases of CUDA.

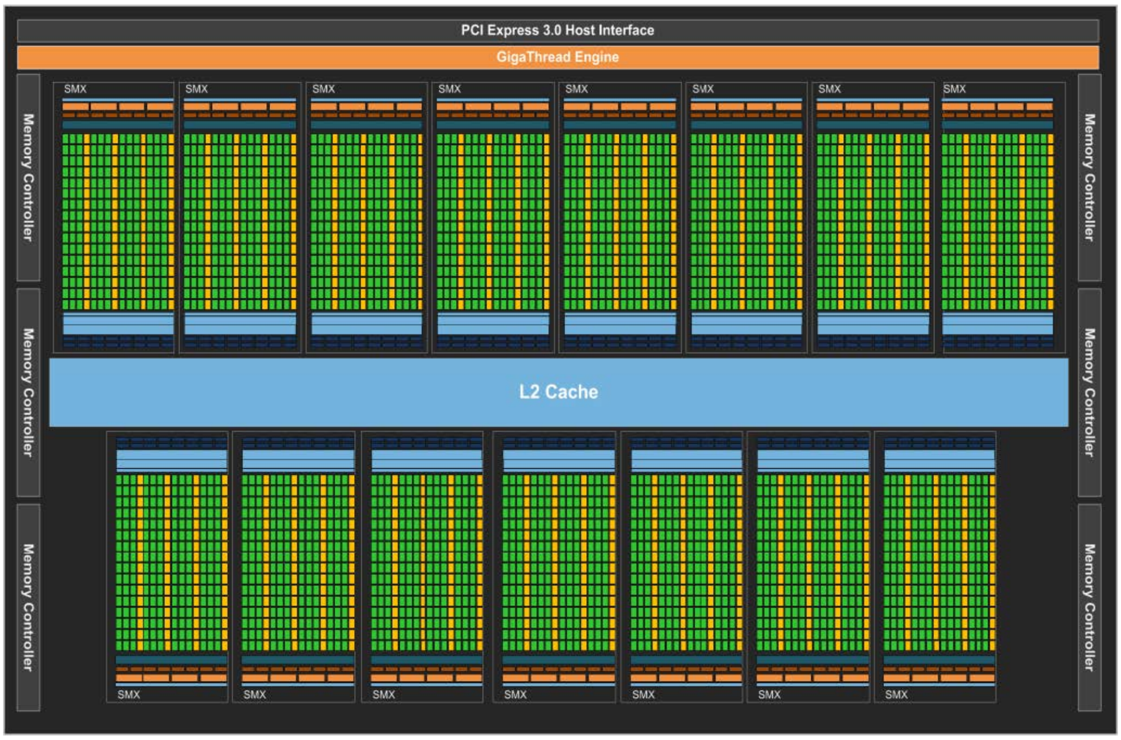


Figure 4.7: NVIDIA K40c GPU GK110 Architecture

As shown in Fig (12) in Appendix B is the Nvidia Tesla K40C GPU. K40 belongs to Kepler GK110/210 architecture [100]. Demonstrated in Fig (4.7), it has 15 Multiprocessors, or called

SMX (Streaming Multiprocessor) in [101]. Each SMX has 192 single-precision CUDA cores per SMX, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST). Under the CUDA programming model, the threads are managed through Grid, Block, and Thread in software level. Normally the host computer (CPU) launches one kernel and appoints the size of the grid and block through configuration <*grid, block*>. Both the grid and block could be a 1-dim, 2-dim or 3-dim structure. Each block could have up to 1024 threads in total. From the hardware point of view, the threads are packaged as warps, each warp has 32 threads on a single SMX and each SMX uses SIMT (Single-Instruction, Multiple-Thread) method for parallel threads/tasks.

Both of the training and inference of Neural Network, Convolutional Neural Network and Graphic Neural Network mentioned above, are highly parallel in nature. Thus they are naturally suited for GPU acceleration. That is similar for FPGA acceleration.

The FPGA can connect directly to the full data port (parallel input), and it could initialize multiple computing cores for acceleration, which is similar to GPU. The FPGA can also carry out parallel pipeline processing. FPGA don't need to wait for all data to come in. With pipeline, FPGA could start to handle one packet of data at each pipeline stage. This saves time in data transmission and further saves time by starting processing earlier than when the data is first received, rather than waiting until the data is collected. If the algorithm is determined and the amount of data processed is basically determined in this pipelined way, then the delay of FPGA processing is also fixed and can be accurate to an exact number of clock cycles. This will be demonstrated in Chapter 9 for a forward pass neural network.

Chapter 5

Supervised Learning on FPGA

5.1 Optimazation methods targetting to FPGA

Optimizing the fully connected structure and Convolutional Neural Network is in the domain of supervised learning. The accelerating part only focus on the inference. To cover the CNN on HighFlex2, the Xilinx Deep Learning Processing Unit (DPU) [94] is the appropriate candidate, which is optimized for the convolutional neural network. It also beed deployed for VGG, ResNet, GoogLeNet and many others. The Xilinx DPU is a standard IP core that could be deployed in the PL part.

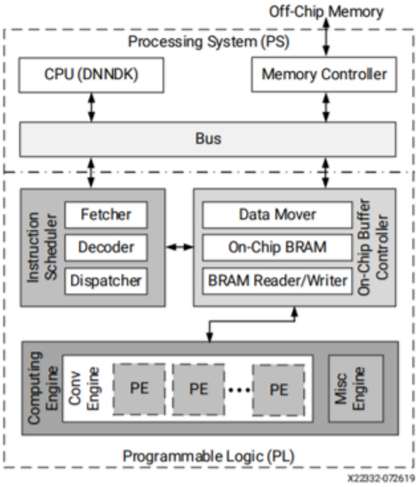


Figure 5.1: Xilinx DPU architecture [94]

As shown in the Fig (5.1), the Xilinx DPU is consists of three parts, the instruction scheduler, the on-chip buffer controller and the computing engine. Through the data mover in the on-chip buffer, the PS part load and read the result from the PL part.

The DPU owns configurable AXI master interface with 64 or 128 bits for accessing data so that DPU can access the PS DDR memory space. These master ports are connected with PS slave ports (S_AXI_HP*_FPD) in Fig (5.2).

The Fig (5.3) shows the detail block DPU. The part with yellow underpainting is the DPU IP(version 3.0) and its corresponding peripherals (PLLs and interruputs). The left connection is the control signal form PS side. The right part signals named after *H_AXI* is connected with high performance interface of PS part for high throughput required from data mover in DPU IP.

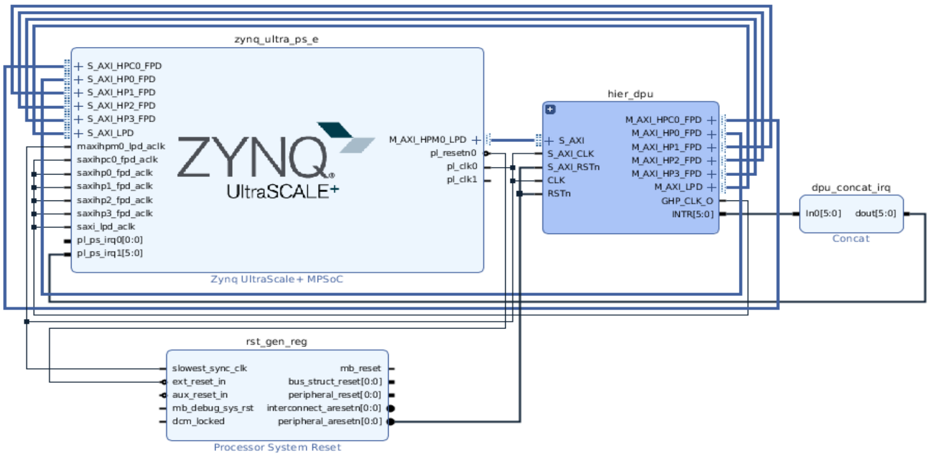


Figure 5.2: Xilinx DPU Block Design on HighFlex2

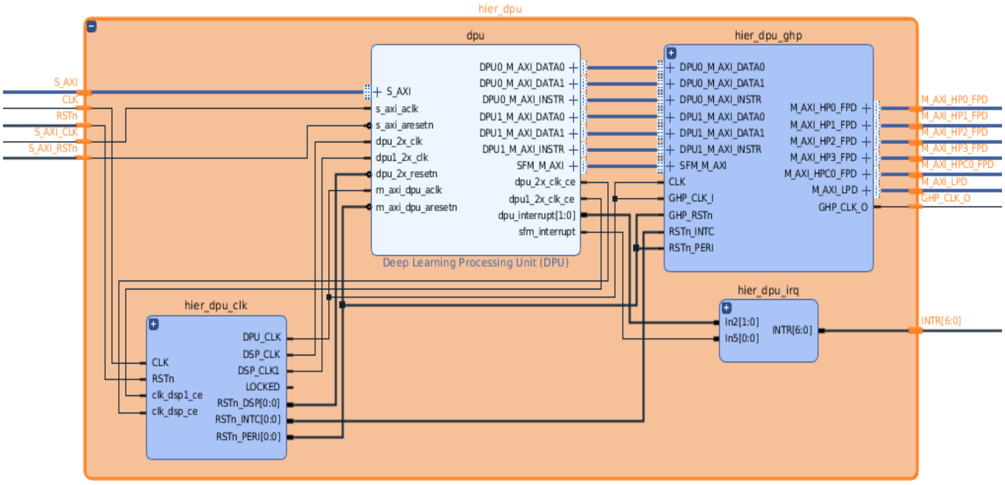


Figure 5.3: Block Diagram for Xilinx DPU [94]

5.1.1 Xilinx DPU configuration options

The DPU IP could be configured for different performance and hardware resources utilization. Firstly the number DPU cores, from 1 to 3 could be choosed, 3 could brings most performance but a lot resource cost. Here the selection is 2 DPU cores. The convolution architecture choose B4096, which means the peak operations is 4096 per clock cycle. The reason for choose B4096 is because the reference design for Xilinx ZCU102 [102] board is use this configuration also. HighFlex2 exceeds ZCU102 in the resources like numbers of Block RAM, Flip-Flops, LUTs, Distributed RAM, DSPs [103].

The greened highlighted part is the DPU IP implemented design shown in Fig (5.4).

The principal part is the BRAM where 86 percent of the BRAM is costed because of the weights and bias, and also other intermediate variables. DSP slices are used for the convolution operations and non-linear operations.

The next step is to integrate the DPU driver and dependent libraries in to the PetaLinux. PetaLinux is an embedded Linux Software Development Kit(SDK) targeting FPGA-based System-on-Chip design [104]. The major part that PetaLinux contains is Yocto Extensible SDk [105](e-SDK). This e-SDK contains all the necessary layers from Xilinx for Zynq UltraScale plus devices, and also the sstate-cache that enables incremental builds. The e-SDK also

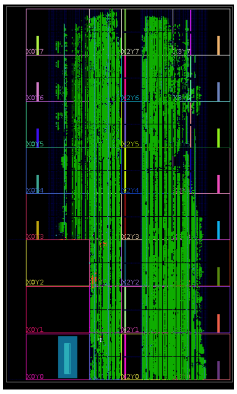
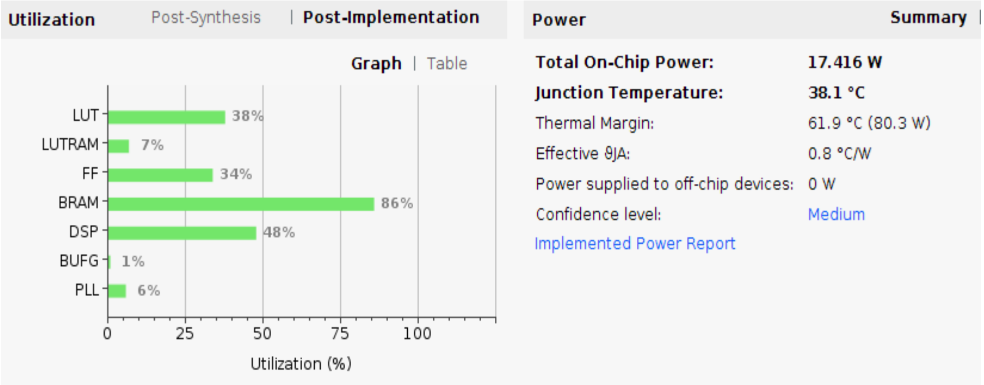


Figure 5.4: Xilinx DPU Implemented Design on HighFlex2



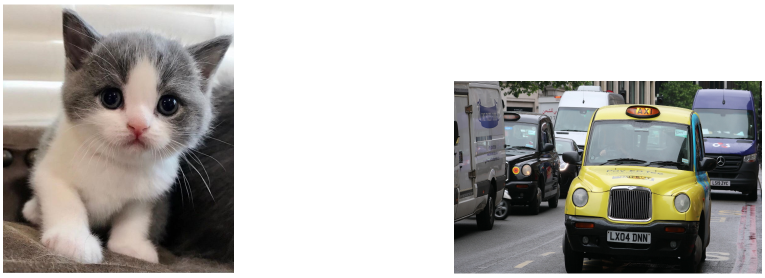
Total On-Chip Power: 17.416 W
Junction Temperature: 38.1 °C
Thermal Margin: 61.9 °C (80.3 W)
Effective ϑJA: 0.8 °C/W
Power supplied to off-chip devices: 0 W
Confidence level: Medium
Implemented Power Report

Figure 5.5: Xilinx DPU Resources Utilization on HighFlex2

contains the sysroots that stores the directory of the Linux system.

Firstly needs to build a PetaLinux project and configure the hardware with the hardware description file that generated from the hardware implementation Vivado project. Then add the drivers given by Xilinx into the PetaLinux projects, built the project as a Image and start HighFlex2 with SD card mode.

5.2 Demostration result



(a) cat

(b) cab

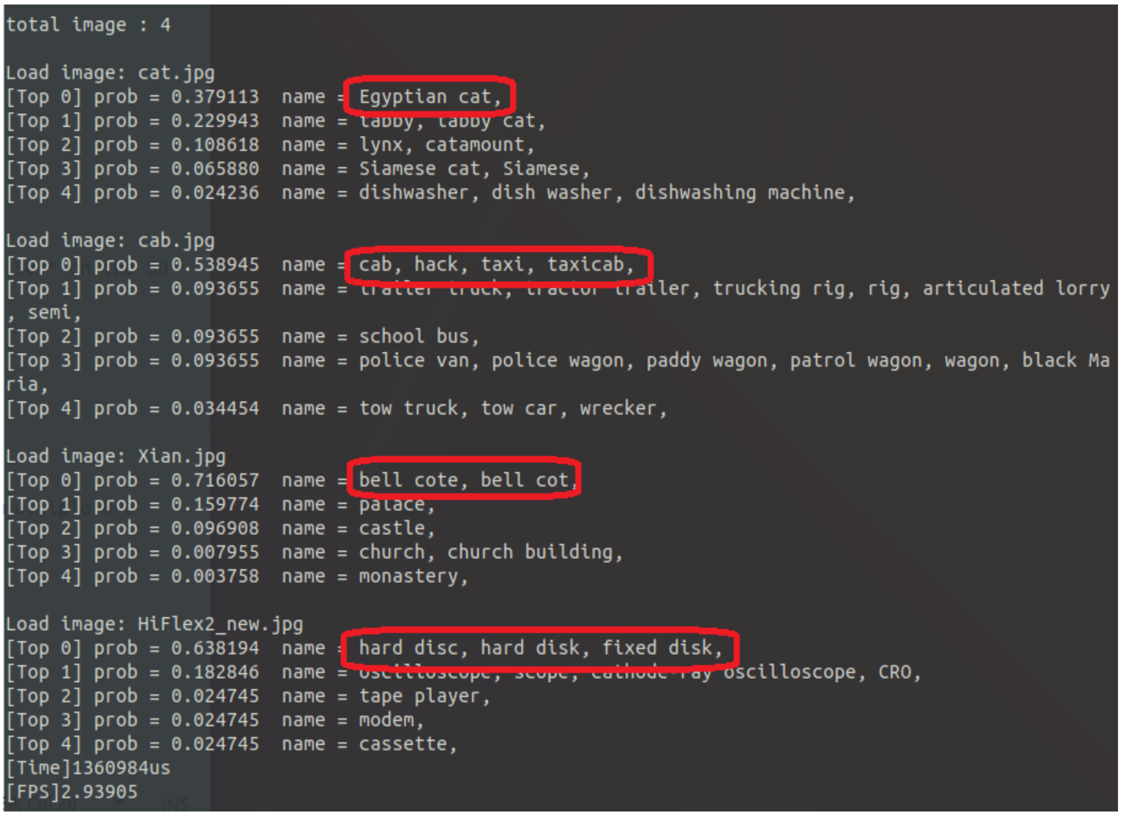
Figure 5.6: First part of photos to demo resnet50 on HighFlex2.

As shown in Fig (5.8), the cat is recognized with 60.9056 percent $(37.9113 + 22.9943)$. The cab picture is recognized with 53.8945 percent. The picture Xi'an is the historical landscape

(a) Xi'an (b) HighFlex2

Figure 5.7: Second part of photos to demonstrate resnet50 on HighFlex2

Bell Tower in Tang Dynasty of the ancient China. The name comes from several large bronze-cast bells on the tower. The resnet50 successfully recognized it as bell cote with 71.6057 percent confident. Also interestingly the resnet50 recognize the HighFlex2 board as hard disk with 63.8194 percent. The resson is that in the ImageNet [66] there is no "HighFlex2" category and because the PCB board is looks like a hard disk. The total latency for this 4 image is 1360984 µs, and the frame rate is 2.93905 FPS.



```
total image : 4

Load image: cat.jpg
[Top 0] prob = 0.379113  name = Egyptian cat,
[Top 1] prob = 0.229943  name = tabby, tabby cat,
[Top 2] prob = 0.108618  name = lynx, catamount,
[Top 3] prob = 0.065880  name = Siamese cat, Siamese,
[Top 4] prob = 0.024236  name = dishwasher, dish washer, dishwashing machine,

Load image: cab.jpg
[Top 0] prob = 0.538945  name = cab, hack, taxi, taxicab,
[Top 1] prob = 0.093655  name = trailer truck, tractor trailer, trucking rig, rig, articulated lorry
, semi,
[Top 2] prob = 0.093655  name = school bus,
[Top 3] prob = 0.093655  name = police van, police wagon, paddy wagon, patrol wagon, wagon, black Ma
ria,
[Top 4] prob = 0.034454  name = tow truck, tow car, wrecker,

Load image: Xian.jpg
[Top 0] prob = 0.716057  name = bell cote, bell cot,
[Top 1] prob = 0.159774  name = palace,
[Top 2] prob = 0.096908  name = castle,
[Top 3] prob = 0.007955  name = church, church building,
[Top 4] prob = 0.003758  name = monastery,

Load image: HiFlex2_new.jpg
[Top 0] prob = 0.638194  name = hard disc, hard disk, fixed disk,
[Top 1] prob = 0.182846  name = oscilloscope, scope, cathode-ray oscilloscope, CRO,
[Top 2] prob = 0.024745  name = tape player,
[Top 3] prob = 0.024745  name = modem,
[Top 4] prob = 0.024745  name = cassette,
[Time]1360984us
[FPS]2.93905
```

Figure 5.8: HighFlex2 Image Classification Results

5.3 Conclusion

This chapter deals with a practical implementation of one supervised learning implementation method on HighFlex2. This solution is based on Xilinx deep learning processor units (DPU) and its embedded Linux drivers have been developed and implemented on the Zynq device.

The DPU is meant to accelerate the convolutional neural network (CNN) inference part. This allows HighFlex2 to implement several different types of CNN models on the programmable logic part. The convolutional neural network implemented and tested is GoogLeNet. Other models, for example, VGG, the YOLO-v3 for ADAS (Advanced Driver Assistance Systems), and the FPN, also have the related example design and easy to implement. This opens a novel possibility of real-time image processing based on the advanced UFO camera developed at KIT for the ultrafast computer tomography. Furthermore, the aforementioned technology will provide relatively fast deployment of the standard CPU/GPU based TensorFlow PyTorch and Caffe models on the FPGA.

Chapter 6

Reinforcement Learning Introduction

In the recent years, with the increasing computing power capacity of High Performace Computing (HPC), the ubiquity of big data and rapid development in deep learning technology, reinforcement learning (RL) made great progress. RL has been considered and used in many fields, for example, AlphaGo [19, 106], the CARLA driving simulator [107] in automatic driving field, the MILABOT model [108] and Facebook artifitial dialogue [109] in the field of Natural Language processing (NPL), and many other applications in [110].

Because the thesis is intend to accomplish the full reinforcement learning platform, it is necessary to review the basic knowledgement in reinforcement learning and bring out the technical terms clearly before the algorithm implementation and application.

6.1 Markov property

Following the notation in [111], the field of reinforcement learning is briefly introduced below. For a more detailed description of the subject, it will be delivered in the next sections within this chapter.

Reinforcement learning is the computational approach to goal-directed learning from interaction. In contrast to other sub-fields of machine learning, its learning paradigm does not require a pre-existing data set, but merely an *environment* to interact with. The learner and decision maker, usually called the *agent*, continuously interacts with the environment learning from past experience and thereby improving its behaviour. At every time step, the agent perceives the current *state* S_t of the environment and carries out an *action* A_t. Based on the chosen action, the environment transitions to a new state S_{t+1} and generate a scalar *reward* R_{t+1} back to the agent as shown in Fig (6.1).

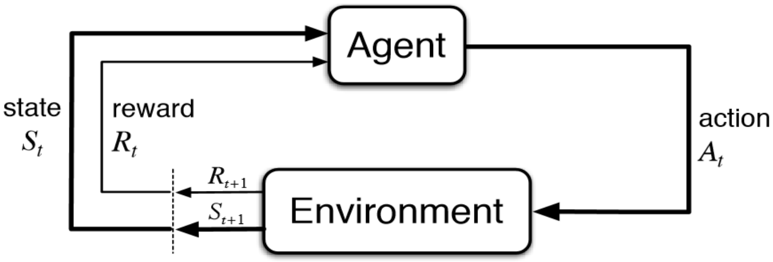


Figure 6.1: The agent-environment interaction in a Markov decision process.

The reinforcement learning problem is formally described as a Markov Decision Process

(MDP), demanding that the sequence of states fulfils the Markov property

$$p(S_{t+1}|S_t) = p(S_{t+1}|S_1, \ldots, S_t) \; , \tag{6.1}$$

where $p(S_{t+1}|S_t)$ denotes the conditional probability of transitioning from state S_t to state S_{t+1}. At every time step t, the state S_t is thereby required to provide all relevant historical information about the transition dynamics of the environment. While many problems can be modelled in this form and the Markov property allows precise theoretical statements, it can be difficult to fulfil this requirement in its most rigorous formulation for some practical applications. The following definitions give formal expression in mathematic.

The Markov property means the next state s_{t+1} only have the relationship with the previous state s_t, and unrelated to the state before the s_t. As also mentioned in the Equ (6.1).

Definition 6.1.1. A state S_t is *Markov* if and only if [111]:

$$\mathcal{P}[S_{t+1} \,|\, S_t] = \mathbb{P}[S_{t+1} \,|\, S_1, \ldots, S_t]$$

In Def (6.1.1), the current state covers all the information from the history, and once the agent have the information of the current state, it gets all the historical information, and what happen in the next will only depends on the current state.

Definition 6.1.2. A *Markov Process* (or *Markov Chain*) is a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$ [111]

The Def (6.1.2) is the Markov Process. Where the $\mathcal{S}$ is a (finite) set of states, $\mathcal{P}$ is a state transition probability matrix [111],

$$\mathcal{P}[S_{t+1} \,|\, S_t] = \mathbb{P}[S_{t+1} \,|\, S_1, \ldots, S_t]]$$

Definition 6.1.3. A *Markov Reward Process* (MRP) is a tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ [111]

Where the $\mathcal{S}$ is a finite set of states, P is a state transition probability matrix, denotes as $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \,|\, S_t = s]$, the $\mathcal{R}$ is the reward function, and represent as $\mathcal{R}_s = \mathbb{E}[R_{t+1} \,|\, S_t = s]$. And finally the γ is the discount factor, $\gamma \in [0, 1]$. This parameter is used for calculate the accumulated return. Here gives the definition of *Return*:

6.1.1 Return

The return is denotes how much value the agent get from environment for a long time. It is the sum of rewards.

Definition 6.1.4. The *Return* is the total discounted reward from time-step *t* [111]

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-2} R_T = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

If γ is close to 0, then it is a "myopic" evaluation, that the agent don't care the far return, and if the γ is close to 1, then the agent think that the future return is important. The R_{t+1} is the current time step t immediate reward that return from the environment, and the R_{t+2} is the immediate reward received after the current step, and the R_T is the immediate reward that get from the *teminate state* at the end of this *episode*. The episode could be regarded as one complete chain of the interaction between agent and environment, and the end of the episode is the break point in a continuous Markov process. Here the G_t is only one sample in the whole experiment, the goal of the RL is to maximize the average of the G_t in MRP.

6.1.2 Value Function

Definition 6.1.5. The *state-value function v(s)* of a MRP is the expected return starting from state *s* [111]

$$v(s) = \mathbb{E}[G_t \,|\, S_t = s]$$

The value function is used to describe the long-term value of state s.

Bellman Equation for MRPs

The $v(s)$ could be rewrited in following [111]:

$$\begin{aligned}\mathcal{R}_s &= \mathbb{E}[R_{t+1} \,|\, S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \ldots \,|\, S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \ldots) \,|\, S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_t \,|\, S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \,|\, S_t = s]\end{aligned} \tag{6.3}$$

The Bellman equation in Equ (6.3) shows the value in one state could be calculated by the expected value of sum of immediate reward R_{t+1} and the discounted value of successor state $\gamma v(S_{t+1})$, shown in the Fig (6.2) below, where the $P_{ss'}$ is the transition probability from state s to next state s' in Equ (6.1).

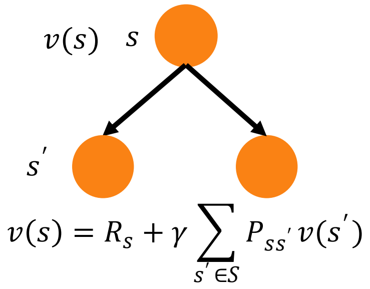


Figure 6.2: the value function in one state could calculate through the value in successor state

6.1.3 Markov Decision Process

A Markov Decision Process (MDP) is a Markov Reward Process with decisions. It is an *environment* in which all states are Markov [111].

Definition 6.1.6. A *Markov Decision Process* is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

Where there are three different points than Markov Reward Process (MRP). First it have the finite set of action A, and the P is changed to:

$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' \,|\, S_t = s, A_t = a] \tag{6.4}$$

the reward $\mathcal{R}$ is changed to:

$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1} \,|\, S_t = s, A_t = a] \tag{6.5}$$

6.1.4 Policies

Definition 6.1.7. A *policy* π is a distribution over actions given states

$$\pi(a\,|\,s) = \mathbb{P}[A_t = a\,|\,S_t = s]$$

The agent policy maps the current state to the action. The policies depend on the current state, this means the possibility that the agent takes action a at state s. The policy could be divided into two categories: one is stochastic policy, which is Def. (6.1.7); another is deterministic policy, which means at each state, the agent will generate a fixed action and could be simply denoted as $a = \pi(s)$, rather than a probability distribution. This is used in the Deep Deterministic Policy Gradient, and the component that generates such fixed action always is called *actor network*, which will discuss in the later chapters.

After the Def. (6.1.7), the MDP could interpreted as, given an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π, the state sequence $S_1, S_2, S_3, \ldots$ is a Markov Process, and the state and reward sequence $S_1, R_2, S_2, R_3, \ldots$ is a MRP $\langle \mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma \rangle$, and most important, in which the [111]:

$$\mathcal{P}^\pi_{s,s'} = \sum_{a \in \mathcal{A}} \pi(a\,|\,s)\mathcal{P}^a_{ss'} \tag{6.6}$$

$$\mathcal{R}^\pi_s = \sum_{a \in \mathcal{A}} \pi(a\,|\,s)\mathcal{R}^a_s \tag{6.7}$$

The Equ (6.6) illustrates that under a policy π, if the agent is on state s, it have a possibility $\pi(a_1\,|\,s)$ to choose action a_1, then based on this action a_1, the agent have a transition possibility $\mathcal{P}^{a_1}_{ss'}$ transfer from s to s'. So the final possibility of transition is a conclusion among action space $\mathcal{A}$.

6.2 State-Value Function

After have the definition of policy, the value function here is the extension over Def. (6.1.5):

Definition 6.2.1. A *state-value fuction* $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π

$$v_\pi(s) = \mathbb{E}_\pi[G_t\,|\,S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k \mathcal{R}_{t+k+1}\,|\,S_t = s], \text{for all } s \in \mathcal{S}$$

The state-value function evaluates how much expected reward the agent will get at this state s afterwards following the policy π.

6.3 Action-Value Function

There also the definition to evaluate how much reward one agent could receive after take a specific action [111]:

Definition 6.3.1. A *action-value fuction* $q_\pi(s, a)$ is the expected return starting from state s, taking action a, and then following policy π

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t\,|\,S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k \mathcal{R}_{t+k+1}\,|\,S_t = s, A_t = a]$$

This definition gives the method to evaluate *how good* if the agent takes action a when the agent is in state s under a given policy π through the expected return. This $v_\pi(s)$ and $q_\pi(s,a)$ could be measured by random samples, and this method is *Monte-Carlo methods*. If the number of states is too much and it is not possible to do such much experiment to get the estimation of $v_\pi(s)$ and $q_\pi(s,a)$, then parameterized function could be used to predict the value of these two functions. This is used in Deep Q-Network [78] and Deep Deterministic Policy Gradient [82], which will be illustrated later.

6.4 Temporal-Difference Prediction

Temporal-Difference (TD) combines the ideas of Dynamic Programming and Monte-Carlo method [111]. The TD learning could learn from two continuous state without waiting the finish of one complete episode. It is updated every single step.

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{6.8}$$

The Equ (6.8) shows one of the TD based method which is ***TD(0)***, or called *one-step TD*. The $R_{t+1} + \gamma V(S_{t+1})$ is called *TD-target*, and the $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called *TD-error*. This error is the diviation between old estimate $V(S_t)$ and a current better estimate $R_{t+1} + \gamma V(S_{t+1})$.

Algorithm 1 Tabular TD(0) for estimating v_π [111]

```
Input: the policy $\pi$ to be evaluated
Algorithm parameter: step size $\alpha \in (0,1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$
loop for each episode:
    Initialize $S$
    loop for each step of episode:
        $A \leftarrow$ action given by $\pi$ for $S$
        Take action $A$, observe $R$, $S'$
        $V(S) \leftarrow V(S) + \alpha(R + \gamma V(S') - V(S))$
        $S \leftarrow S'$
    end loop
    until $S$ is terminal
end loop
```

The same as Dynamic Programming, the $V(S)$ is updated based on itself. Therefore, it is called *bootstrapping* method. When the γ is sufficiently small, the TD(0) method could garantee a convergence.

6.5 Model-Free Control

The last subsection introduce TD-prediction methods that could predict the value function. Such prediction could be used to generate the policy.

6.5.1 ϵ-Greedy Exploration

The ϵ-Greedy method could help all the state-action pairs are explored infinitely many times so that the agent will not be stuck at only one policy that maybe not the optimal one. The mathematical way is intuitionally to give the π with a non-zero value to try all the actions as shown in Equ (6.9). And this is proved to guarantee the policy is continuously being improved.

$$\pi(a\,|\,s) = \begin{cases} \epsilon/m + 1 - \epsilon, & \text{if } a^* = \arg\max_{a \in \mathcal{A}} Q(s,a) \\ \epsilon/m, & \text{otherwise.} \end{cases} \tag{6.9}$$

6.5.2 Q-learning: Off-Policy Temporal-Difference Control

Q-learning is an *off-policy* method. The agent choose the next action by behaviour policy $A_{t+1} \sim \mu(\cdot\,|\,S_t)$, and learns from the target policy $A' \sim \pi(\cdot\,|\,S_t)$.

Algorithm 2 Q-learning (Off-policy TD Control) for estimating $\pi \approx \pi_*$ [112]

```
 1: Algorithm parameters: step size $\alpha \in (0,1]$, small $\epsilon > 0$
 2: Initialize $Q(s,a)$,for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}$, arbitrarily except that $Q(terminal, \cdot)= 0$
 3: loop for each episode:
 4:     Initialize $S$
 5:     loop for each step of episode:
 6:         Choose $A$ from S using policy derived from $Q$ (e.g., $\epsilon$-greedy)
 7:         Take action $A$, obseve $R$, $S'$
 8:         $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma max_a Q(S',a) - Q(S,A)]$
 9:         $S \leftarrow S'$
10:     end loop
11:     until $S$ is terminal
12: end loop
```

In practice, the behaviour choose the ϵ-greedy, which allows some probability for exploration, and the target policy is greedy with respect to action-value, as shown in Equ (6.9), the $\arg\max_a Q(S,a)$.

In the Q-learning, the target will be $R_{t+1} + \gamma Q(S_{t+1}, A')$, the A' is the aciton following the greedy policy above. The algorithm first take action A, then observe the reward R and the next step state S', the error here is the deviation between $R + \gamma \max_a Q(S', a)$ and $Q(S, A)$. The front one is $Q(s_t, a_{t+1})$, which is target, the later one is the $Q(s_t, a_{t+1})$, which is estimate. α is the learning rate. The γ is the discounted factor that assign different weights to different Q-value.

6.6 Function Approximation

Function Approximation approach is a booster to help RL method to solve practical engineering problems. For some large scale of RL problem, the state space is always very large, for example, the automatic driving [30], the state for the car is the external image and the surrounding object observed by the car, and this observation is continuously changing. This means the state number in step 7 in Alg (2) is infinity. A tabular method for Alg (2) is not practical. This tabular based approach is exactly same as LookUp Table(LUT) in FPGA, a large Q-table should be implemented to store the action-value $Q(s,a)$ for each action and state pair. Normally if the problem set is relatively large, a "case" function in the code that will cost a lot of LUTs in FPGA. Implementing this table on CPU/GPU also requires a large memory resources or computational resource.

The $\boldsymbol{\theta} \in \mathcal{R}^{d'}$ could be used as parameter vector. The basic idea of function approximation is to build one approximator like $\hat{v}(S, \boldsymbol{\theta})$ for value function or $\hat{q}(S, A, \boldsymbol{\theta})$ for action-value function. This will help map a state or state-action pair in a continuous space to one value. The $\boldsymbol{\theta}$ is the parameters in approximator, and it depends on this function approximator is a linear function, neural network, decision tree, nearest neighbour, fourier or wavelet bases approximator and many others.

Establish the deviation between the approximated action-value function and the true action-value function:

$$\mathcal{J}(\boldsymbol{\theta}) = \mathbb{E}_\pi[(q_\pi(S, A) - \hat{q}(S, A, \boldsymbol{\theta}))^2] \tag{6.10}$$

The basic machine learing approach stochastic gradient descent can easily solve this optimization problem:

$$\Delta\boldsymbol{\theta} = \alpha(q_\pi(S, A) - \hat{q}(S, A, \boldsymbol{\theta}))\nabla_{\boldsymbol{\theta}}\hat{q}(S, A, \boldsymbol{\theta}) \tag{6.11}$$

And for TD(0) method have this update formula:

$$\Delta\boldsymbol{\theta} = \alpha(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{\theta}) - \hat{q}(S_t, A_t, \boldsymbol{\theta}))\nabla_{\boldsymbol{\theta}}\hat{q}(S_t, A_t, \boldsymbol{\theta}) \tag{6.12}$$

In the Equ (6.12), $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{\theta})$ is the TD target and $\hat{q}(S_t, A_t, \boldsymbol{\theta})$ is the function prediction value. The deviation of two is the TD error. In the following chapters will introduce how standard deep learning techniques will be used for function approximation in Policy Gradient and Deep Deterministic Policy Gradient.

6.7 Deep Q-Learning

A function approximator $Q(s, a; \boldsymbol{\theta})$ in last subsection is used for a practical estimation of $Q^*(s, a)$. Then a loss function comes from TD-error could help for training the Q-network $Q(s, a; \boldsymbol{\theta})$:

$$L_i(\boldsymbol{\theta}_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}[(y_i - Q(s, a; \boldsymbol{\theta}_i))^2] \tag{6.13}$$

The target y_i, similar with the concept of labeled data in supervised learning:

$$Q^*(s, a) = \mathbb{E}_{s'\sim\varepsilon}[r + \gamma max_{a'}Q^*(s', a'; \boldsymbol{\theta}_{i-1}) \,|\, s, a] \tag{6.14}$$

The subcript i here is the number of iteration. The $\rho(s, a)$ is a probability distribution, also called *behaviour distribution*. Then the same as supervised learning training process [113], calculate the derivative with respect to the parameter $\boldsymbol{\theta}_i$ as also shown in Equ (6.11). But if one directly use a non-linear function for a TD(0) control, it will not guarantee for convergence, it is true for both on-policy and off-policy. But in practice, although there is no garantee, the Q function still works and moves stablely towards improvement.

The Deep Q-learning or sometimes called Deep Q-network (DQN), is the method that combines the function approximation and *experience replay* [114]. In the Alg (3) From the step 9 to step 13 clearly shows how the experience replay performs. Because in [78] the input state is image of Atiri Game, The ϕ_t and ϕ_{t+1} here is the preprocessing to fit the input of neural network, here it could be understood as s_t and s_{t+1}. Every time the agent finish one step in step 9, the transition will be stored in one memory. Then the same as supervised learning, the training data

Algorithm 3 Deep Q-learning with Experience Replay [78]

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for for episode = 1, M do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for t=1, T do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = max_a Q^*(\phi(s_t), a; \boldsymbol{\theta})$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from $\mathcal{D}$
        Set $y_i = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \boldsymbol{\theta}) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_i - Q(\phi_j, a_j; \boldsymbol{\theta}))^2$
    end for
end for
```

set will be extracted from this memory and, performs the gradient descent, or other optimization method to minimize the loss function. The motivation for experience replay is, firstly, the training data should be Independent Identical Distribution (IID). Secondly, the random sample from the memory could break the dependency between two transitions $(\phi_t, a_t, r_t, \phi_{t+1})$.

6.8 Policy Gradient

Like the Q-learing and DQN, they indicate the Q-Value based method approximating the value under current state, but the *policy gradient* method introduced in this subsection will parametrize the policy directly:

$$\pi(a \,|\, s, \boldsymbol{\theta}) = Pr\{A_t = a \,|\, S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\} \tag{6.15}$$

The π is the policy with parameters $\boldsymbol{\theta}$ used in this episode or step. In practice, this parameterized function will be a neural network where it is plugged into the current state and the last layer will give out the possibility that how to take each action. The quality of a policy is measured with a policy score function $J(\boldsymbol{\theta})$ that collecting all the expected reward. Maximizing the score function means looking for the optimal policy. Then the goal is to find the best $\boldsymbol{\theta}$ that maximizes the objective function $J(\boldsymbol{\theta})$. This is again the basic optimization method that is similar to supervised learning. To maximize or minimize a function, if one knows the gradient of the function, then an ascent or descent direction could be used for searching the optimal point of the function. The policy gradient theorem [115] gives such a direction to update the parameters towards the gradient ascent on policy function:

There are three different definition types of policy objective function:

Definition 6.8.1. In episodic environment one can use the *start value*, the policy objective function is:

$$J_1(\boldsymbol{\theta}) = V^{\pi_{\boldsymbol{\theta}}}(s_1) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[v_1]$$

Here in one episode, the start value defines how much reward will get if the agent starts from the state s_1 onwards.

Definition 6.8.2. In continuing environment one can use the *average value*, the policy objective function is:

$$J_{avV}(\boldsymbol{\theta}) = \sum_s d^{\pi_{\boldsymbol{\theta}}}(s) V^{\pi_{\boldsymbol{\theta}}}(s)$$

Where $d^{\pi_{\boldsymbol{\theta}}}$ is the stationary distribution of Markov chain for $\pi_{\boldsymbol{\theta}}$. Practically $d^{\pi_{\boldsymbol{\theta}}} = \lim_{t\to\infty} P(s_t = s \mid s_0, \pi_{\boldsymbol{\theta}})$, that shows the probability transfer from state s_0 to s_t following policy $\pi_{\boldsymbol{\theta}}$. The Def (6.8.2) use this state distribution to average reward simply average all the possible states following the $\pi_{\boldsymbol{\theta}}$.

Definition 6.8.3. In continuous environment one can use the *average reward per time-step*, the policy objective function is:

$$J_{avR}(\boldsymbol{\theta}) = \sum_s d^{\pi_{\boldsymbol{\theta}}}(s) \sum_a \pi_\theta(s,a) V^{\pi_{\boldsymbol{\theta}}}(s)$$

In each step, the Def (6.8.3) average all the state firstly and average all the possible policy times immediate reward get at current state-action pair.

The key point is, one could not know the exact mathematical expression of those objective function $J(\boldsymbol{\theta})$, but once the difference of objective function $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ with respect to parameter $\boldsymbol{\theta}$ is given, then the parameters could be updated and policy could be improved accordingly. Before to understand the policy gradient theorem, let's consider only one-step MDPs and the average reward per time-step type of objective function. If starting from state s, get immediate reward $r = \mathcal{R}_{s,a}$, then the episode is finished. Then one will have expectation expression of objective function:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[r] \\ &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_{\boldsymbol{\theta}}(s,a)\mathcal{R}_{s,a} \end{aligned} \tag{6.16}$$

In Equ (6.20), the first sum is to calculate the expectation over starting state and the second sum is the expectation over all the possible action with respect to the π. Then according to this, the policy gradient will be:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_{\boldsymbol{\theta}}(s,a) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s,a)\mathcal{R}_{s,a} \\ &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} log \pi_{\boldsymbol{\theta}}(s,a) r] \end{aligned} \tag{6.17}$$

In machine learning, the target for *Maximum Likelihood Estimation* is to find one set of pramameters $\boldsymbol{\theta}$ that try to fit the model with the observation data or training data with the largest probability, which has the samilar concept with *Cross Entropy*, it normally have the format as below:

$$H(p,q) = -\sum_x p(x) \log q(x) \tag{6.18}$$

The purpose of a standard supervised machine learning problem is to minimize the cross-entropy, by doing such will minimize the difference between labeled data and the prediction data. In Equ (6.18) where the H denotes the Entropy, that shows the difference between two distribution. The p is the distribution of the reference data(or labeled data), the q is the distribution of the prediction data. Then compare the Equ (6.18) with the Equ (6.17) there is no sum operation $\sum$, only the $\log \pi_{\boldsymbol{\theta}}(s,a)$, that means, the output prediction data by model is $\pi_{\boldsymbol{\theta}}$, and the true label is 1 according to this prediction, others are 0. Because in practical implementation, the expectation will be estimated by sampling the data from experiments. The Equ (6.17) can write down as:

$$H = -\sum_i \hat{y}_i \log y_i \tag{6.19}$$

Where the $\hat{y}_i$ is always in a format of $[\ldots, 0, 0, 1, 0, \ldots]$, for a discrete action space problem, the size of this array is depends on the dimension of the action space. The location of the 1 in the labeled data is related to which action the agent takes in this experience, other labels are all 0. Then the Equ (6.17) becomes only $log\pi_{\boldsymbol{\theta}}(s,a)r$. Such true data distribution try to increase or decrease the probability of taking this action a when see this state s, and which direction is depends on the sign of the reward r, if the reward r is negative, the it discourage the agent when see state s to carry out action a, when the r is possitive, then agent will be encouraged.

Without proof, gives the policy gradient theorem here, it generalizes the likelihood ratial approach from one-step MDPs to multi-step MDPs, where replace the immidiate reward r with the action value Q.

Theorem 6.8.1. *For any differentiable policy $\pi_{\boldsymbol{\theta}}(s,a)$, for any of the policy objective functions $J = J_1, J_{avR}$, or $\frac{1}{1-\gamma}J_{avV}$, the policy gradient is:*

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= E_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s,a) Q^{\pi_{\boldsymbol{\theta}}}(s,a)] \\ &= E_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(s,a) Q^{\pi_{\boldsymbol{\theta}}}(s,a)]\end{aligned} \tag{6.20}$$

6.9 Monte-Carlo Policy Gradient (REINFORCE)

Algorithm 4 Function REINFORCE

```
initialize $\boldsymbol{\theta}$ arbitrarily
for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_{\boldsymbol{\theta}}$ do
    for $t = 1$ to $T - 1$ do
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} log\pi_{\boldsymbol{\theta}}(s_t, a_t)v_t$.
    end for
end for
return $\boldsymbol{\theta}$;
```

The policy gradient theorem inform a very important message that the parameter gradient to the policy objective function is propotional to $\nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(s,a)Q^{\pi_{\boldsymbol{\theta}}}(s,a)$, one could dirrectly use such information to update the policy network. Here introduce *REINFORCE* update solution for parameters in the policy $\pi_{\boldsymbol{\theta}}$:

$$\Delta \boldsymbol{\theta}_t = \alpha \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s,a) v_t \tag{6.21}$$

Where in Equ (6.21) use the experiment sampling return v_t to replace the $Q^{\pi_{\boldsymbol{\theta}}}(s,a)$. This update method will be the main step in policy gradient as shown in Alg (4).

6.10 Conclusion

In this chapter, some of the important concepts in reinforcement learning are introduced, a clear definition for state, reward, and action are given. The state-value function and action-value function also is proposed. TD-based methods are introduced which are the base for Deep Deterministic Policy Gradient used in Chapter 8 and 9. The Monte-Carlo Policy Gradient method is shortly introduced; this is the algorithm used in Chapter 7.

Chapter 7

Policy Gradient on embedded processor

This chapter will gives the specific CartPole problem solved by Policy Gradient and its performance on hardware. The CartPole problem (environment) comes from [116], and a same version of environment is implemented in OpenAI environment with name CartPole-v1 [117]. This two blocks (agent and environment) are communicated with each other and both are running on MPSoC (PS part). This environment simulates the CartPole in a horizontal axis. This fulfils the control equations, and receives the actions from the agent, then after which, the pole moves. The goal of the control is to keep the CartPole balanced (the pole stays in a narrow angle rang) as long as possible, as shown in the Fig (7.1), the agent should learn to keep the absolute value θ as small as possible.

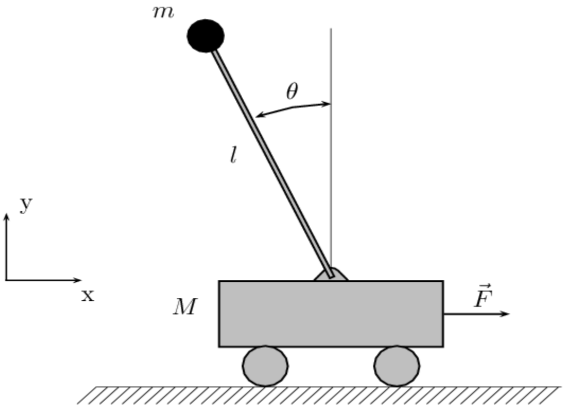


Figure 7.1: Cartpole problem

The Alg (5) illustrate the basic procedure of interaction between the agent and the CartPole environment. It also demonstrates how the agent learns through policy gradient (PG) method. At each step (the inner *repeat* loop in Alg (5)), the interaction procedure between agent and environment is the same as the basic interaction scheme of reinforcement learning as shown in Fig (6.1). Beginning from step 8 of Alg (5), the agent chooses one action A_t according to the current state (observation) S_t and policy π. In step 9, the environment will apply this action, and this transition will lead the agent to the next state (S_{t+1}). At the same time, agent also collects a scalar reward from the environment. At step 11, the agent needs to store the current step information into memory, which includes the current state, the chosen action, the received reward, and the next state. After that, the agent checks whether the episode is finished or not. In CartPole case, a fallen pole means that the cart could not keep the pole back to balanced

Algorithm 5 Policy Gradient Implementation

```
 1: differential policy parameterization $\pi_{\boldsymbol{\theta}}(a|s)$ (fully connected neural network)
 2: initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^d$ and learning rate $\alpha > 0$
 3: allocate memory to store information about the interaction with the environment
 4: repeat
 5:     reset the $S_0$ to a random staring state
 6:     $t \leftarrow 0$
 7:     repeat
 8:         choose action $A_t$ according to $\pi_{\boldsymbol{\theta}}(\cdot|S_t)$
 9:         $S_t, R_t, S_{t+1} \leftarrow$ env.step($A_t$)
10:         $t \leftarrow t + 1$
11:         store the $S_t, A_t, R_t, S_{t+1}$ in memory
12:     until $S_{t+1}$ is terminal ($t = T$)
13:     calculate the cumulative discounted reward $G_t$
14:     loop over steps in this episode $t = 0, 1, \ldots, T - 1$
15:         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha G_t \nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(A_t|S_t)$
16:     end loop
17: until episode has reached a threshold number of steps
```

condition and the current episode is finished (reach to the terminal state). At the terminal point of one episode, the RL agent collects all information related to this episode and calculates the discounted cumulative reward G_t in step 13. This information is then used to update the parameters of the policy in step 15. In step 17, if one episode, also means one effort of balance, keep the pole for more than the stated steps (the author customized value, in later chapter the author uses 500, 1000, 2000, 5000 number of steps. This value is required for the performance analysis). It means the agent have learned well how to balance the pole, then the whole game stops. Also one could keeping the game continue and let the agent learning forever. The next chapter will deliver the backpropagation formula derivation and its implementation on Zynq.

In this subsection, a concrete example will be given to deliver the technique detail of the backpropagation, which is the most important part and difficult part in all reinforcement learning algorithm implementation, as shown from step 13 to step 16 in Alg (5). One specific episode taken by agent will be demonstrated later and the parameters update procedure in hardware will be on display. Finally, the training procedure will be summarized.

7.1 Policy network

The policy approximation here use a 2-layer neural network for step 1 in Alg (5). As shown in the picture 7.2, the policy takes the obvervation of the environment, gives out the probability of the action. In the CartPole problem, the action space is simple : $\{left, right\}$. The input layer takes 4 elements as the input observation space:

x position of the cart on the x-axis

θ the angle of the pole according to y-axis

$\dot{x}$ the derivative of position

$\dot{\theta}$ the derivative of angle

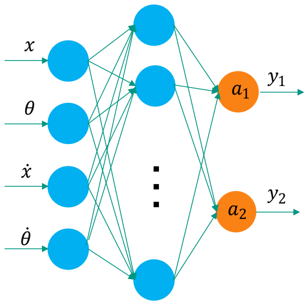


Figure 7.2: The policy network use a 2-layer fully connected neural network, the input layer has 4 units, hidden layer has 30 units, and the output layer has two unit related to the two-dimension of action space. This two dimension is then implemented through a softmax layer so that a probability could be generated in y_1 and y_2

The method of counting number of layers of neural network comes from [118], where the input layer is not counted into the total number. The whole dissertation will follow this convention.

7.2 One specific episode

During the experiment, taking the first episode as the example. The agent interact with the environment, the pole fall down at 13th step, shown in figure 7.3.

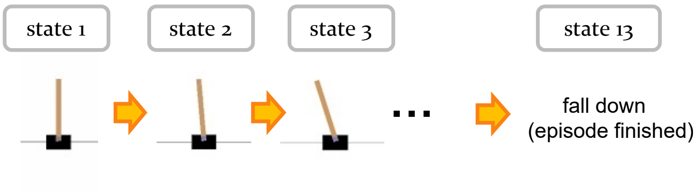


Figure 7.3: 13-step movement

Table 7.1: Output of NN and Actions

Step	Left Prob	Right Prob	Left	Right
1st	0.5345	0.4653	**1**	0
2nd	0.5236	0.4763	0	**1**
3rd	0.5343	0.4656	**1**	0
4th	0.5233	0.4766	**1**	0
5th	0.5117	0.4882	**1**	0
6th	0.4995	0.5005	**1**	0
7th	0.4871	0.5128	**1**	0
8th	0.4745	0.5255	**1**	0
9th	0.4619	0.5254	**1**	0
10th	0.4495	0.5504	0	**1**
11th	0.4546	0.5453	**1**	0
12th	0.4421	0.5578	0	**1**
13th	0.4467	0.5533	**1**	0

The policy output(action probability), and the actual actions which are picked up in history is shown in table 7.1. The $2nd$ and $3rd$ column is the output of the neural network, that gives the probability to take each action. The "suggestion" given by the policy is similar between going left or right and that is because it is the first episode, there is no backpropagation are taken so far and the policy network did not learn anything. This explain that the network randomly gives the probability about 50 percent to go *left* or *right*. The right two columns in Tab (7.1) is the real action that was taken by the agent. The bold number 1 in Tab (**??**) denotes the actual action taken by the agent. The action selection is not according to the largest probability in the policy. It chooses an action based on probability.

7.2.1 Loss function of Policy Gradient

The basic loss function used in Policy Gradient is:

$$Loss = -\frac{1}{N}\sum_{i=1}^{N} v_{ti} \log y_{ia} \tag{7.1}$$

The N is the number of steps the agent experienced in one episode. The y_{i_a} here is the column 2 and 3 in Tab (7.1), where the subscript i represents which step it is. The subscript a denotes the related action. For example the y_{1_2} is in step 1, the propobility to take action 2 (going right) is 0.4653 in Tab (7.1). During training, the loss function will be divided into small pieces, each piece represent one step.

7.2.2 Reward function of Policy Gradient

Reward function is one of the most important signal in RL. One need to clarify the meaning of immediate reward r and the meaning of discounted reward v_t in Equ (7.1). In CartPole problem, the immediate reward r feedback by environment is always 1 when the agent successfully balance the pole for *one more* single step. But when the pole is fall down, the reward coming from environment is 0 at last step in one entire episode as shown in Fig (7.4).

The v_{ti} is the discounted reward where the subcript i denotes the step index in one episode. The Alg (6) described how to calculate the normalized discouted reward through immediate reward r.

Under such definition of reward signal, the agent learns to acquire more reward from environment, and through which could keeping the pole for longer time. But this reward can not be used directly in Equ (7.1) because in different steps, the v_{ti} have different meanings. At the begining of the game, the agent keeping the pole very well, so the corresponding discounted reward should be a large positive number, and then the steps close to the terminal point, the agent show worse performance, the related v_{ti} should be large negative number. To transfer the each step reward $(+1)$ into a discounted and normalized v_{ti} needed in the Equ (7.1), one needs the algorithm (6).

The Alg (6) shows the standard way to map the immediate reward to the discounted return. Alg (6) firstly clean up the reward array, the $ep\_rs[step]$ is always 1 in CartPole problem shown in Fig (7.4). Then calculate the accumulated return and the discounted return in each step at Alg (6) step 9 and 10. The second for loop calculate the mean of the discounted return and minus this mean to normalize the discounted return array. After which, the discounted return value in this example is shown in Tab (7.2).

After these effort, the value in the last column of table 7.2 is related to each seperate step that could be applied into the Equ (7.1). It is applied in the step 14 of Algorithm 5. For example,

Algorithm 6 Function Normalized Discounted Return

```
receive the reward array $ep\_rs[step]$ from the environment every step
define gamma $\gamma$ as the discounted factor to 0.99
define $accu$ as the current accumulated return
define $dis$ as the discounted return array
$accu \leftarrow 0$
for $i = 1$ to $N$ do
    $accu \leftarrow accu * \gamma + ep\_rs[i]$
    $dis[i] = accu$
end for
calculate the mean of the discounted return array $mean\_dis$
calculate the standard deviation of the discounted return array $std\_dis$
for $i = 1$ to $N$ do
    $dis[i] \leftarrow (dis[i] - mean\_dis)/std\_dis$
end for
return $dis$;
```

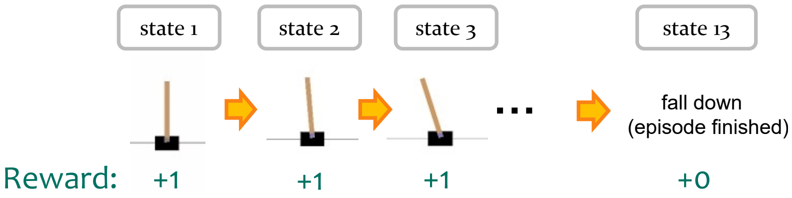


Figure 7.4: thirteen-step-rewards

Table 7.2: Output of NN, Taken Action and Return

Step	Left Prob	Right Prob	Left	Right	Dis Return
1st	0.5345	0.4653	**1**	0	5.5203
2nd	0.5236	0.4763	0	**1**	4.6339
3rd	0.5343	0.4656	**1**	0	3.7386
4th	0.5233	0.4766	**1**	0	2.8342
5th	0.5117	0.4882	**1**	0	1.9207
6th	0.4995	0.5005	**1**	0	0.9979
7th	0.4871	0.5128	**1**	0	0.0659
8th	0.4745	0.5255	**1**	0	-0.8755
9th	0.4619	0.5254	**1**	0	-1.8265
10th	0.4495	0.5504	0	**1**	-2.7871
11th	0.4546	0.5453	**1**	0	-3.7574
12th	0.4421	0.5578	0	**1**	-4.7375
13th	0.4467	0.5533	**1**	0	-5.7275

the accumulated return -5.7275 is v_{t13} and it is related to the last action “going Left", the y_{131}, that the first subscript 13 denotes the $13th$ step, the second 1 denotes the action index “going Left". A big negative value also means that the agent did a wrong step so the environment gives a big punishment on the agent on the $13th$ step. In reverse, in the first step, the discounted return v_{t1} is 5.5203, a positive number, which means that the action taken “going Left" is good.

Furthermore, according to the Equ (7.1), the final loss function in Equ (7.1) for this episode could be discribed more precisely through the value coming from Tab (7.2):

$$\begin{aligned} Loss &= -\frac{1}{N}(v_{t1} * \log(y_{11}) + v_{t2} * \log(y_{22}) + ... \\ &+ v_{t13} * \log(y_{131})) \\ \\ &= -\frac{1}{13}(5.5203 * \log(0.5345) + 4.6339 * \log(0.4763) \\ &+ ... + -5.7275 * \log(0.4467)) \end{aligned} \tag{7.2}$$

7.2.3 Backpropagation for Policy Gradient

A common way for ML development is first to build the computation diagram, secondly define the loss function, thirdly in order to minimize the loss function, needs to calculate the derivative of parameter w_i with respect to loss function $Loss(w_1, w_2, ..., w_n)$, and finally update the parameters by SGD or other optimization method. The Equ (7.3) shows how to use the gradient information to update one parameter, where the α is known as *learning rate*.

$$w_1 = w_1 - \alpha * \frac{\partial L}{\partial w_1} \tag{7.3}$$

The most difficult step is step 3, the caculation of gradient. For example in PyTorch, there is Autograd [119]. The method to calculate the gradient is called backpropagation [86].

Here take one easy machine learning example to explain the concept of forward propagation and backpropagation. A detail implementation method also mentioned in [84].

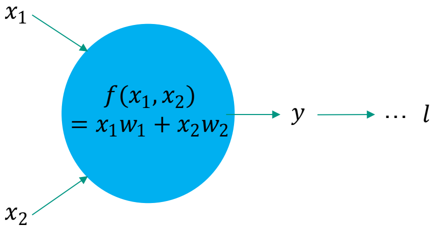


Figure 7.5: forward pass calculation

As shown in the Fig (7.5) is the forward pass, always be the first step when build up one network. The green arrows shows the forward pass, the input of function f is x_1 and x_2, pass through function $x_1 * w_1 + x_2 * w_2$, get result y, then after several step get the final loss function l.

The red arrows in Fig (7.6) shows the backward pass, on the right side, the input of node is the differential value $\partial l/\partial y$ coming from the loss value. One could first assume that every time will get this input value. Secondly, *local gradient* could be deduced by $\partial y/\partial x_1 = w_1$, then $\partial l/\partial x_1$ is $\partial l/\partial y \cdot w_1$, this is the output of backpropagation, afterwards this value will be the input of next compute node, until calculate to the input of the whole forward pass.

Then one could use backpropagation at step 15 in Alg (5). We now have the v_{ti} and the well-defined loss function in last few subsections, we could calculate the derivative of the parameter with respect to the loss function through backpropagation.

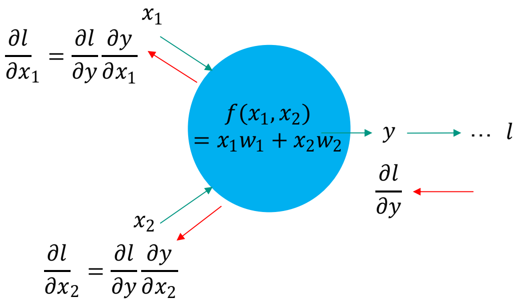

x_1
$\frac{\partial l}{\partial x_1} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial x_1}$
$f(x_1, x_2)$
$= x_1 w_1 + x_2 w_2$
y
$\dots\ l$
$\frac{\partial l}{\partial y}$
x_2
$\frac{\partial l}{\partial x_2} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial x_2}$

Figure 7.6: backpropagation calculation

We first review the policy network through Fig (7.2), and start from the last layer, the softmax layer. The Equ (7.4) is the forward pass of softmax layer.

$$y_i = softmax(a_i) = \frac{e^{a_i}}{\sum_{i=1}^{N} e^{a_i}} \tag{7.4}$$

The purpose of softmax layer is to map the final dense layer output to the action space, and meanwhile it normalize the large or small value into numerical value between 0 and 1. The subscript i in Equ (7.4) denotes which action, only 1 or 2 for CartPole case here. In Equ (7.4), the a_i is the input of the softmax activation layer, the y_i is the output of the softmax. This helps to transfer the array (a_i) into a probability distribution in Fig (7.7).

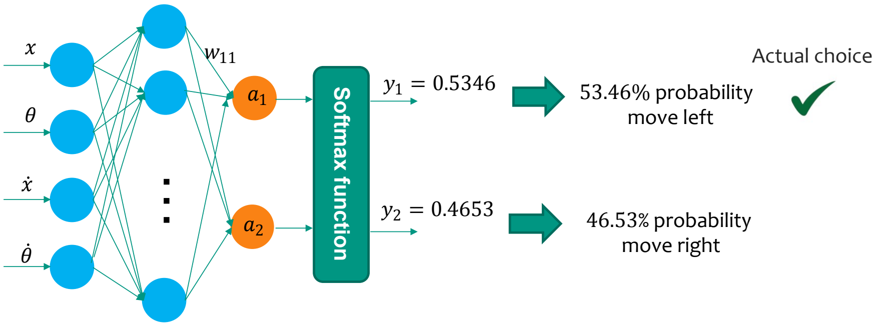


Figure 7.7: The softmax layer gives the probability of taking each action. The policy *selecting the action according to the probability*. As a result the policy choose the left, “left” is the action that agent took in real life.

In backpropagation of softmax, it is important to note that every action a_i has relationship with all y_i. To be specific, the a_1 have relationship with both y_1 and y_2, and it is the same for a_2. This is the most important part in the implementation of backpropagation for softmax layer. Taking one parameter update solution in the neural network for an example, shown in Fig (7.8).

$$\frac{\partial L}{\partial w_{11}} = \frac{1}{N}\left(\sum_{i=1}^{N} \frac{\partial L_i}{\partial y_{i1}} \frac{\partial y_{i1}}{\partial a_1} \frac{\partial a_1}{\partial w_{11}} + \frac{\partial L_i}{\partial y_{i2}} \frac{\partial y_{i2}}{\partial a_1} \frac{\partial a_1}{\partial w_{11}}\right) \tag{7.5}$$

Now we can calculate the derivative of w_{11} with respect to the loss function, as the w_{11} only have relationship with a_1, then the caculation will only sum over the a_1 as shown in the Equ (7.5). The major three part needs to solve in Equ (7.5) is $\partial L_i/\partial y_{i1}$, $\partial y_{i1}/\partial a_1$ and $\partial a_1/\partial w_{11}$. First of all, the Loss function could be regarded as the sum of $L_i = v_{ti} * log(y_{ia})$, and this also means each single step create a *small loss* in the global loss funcion. This *small* *loss* L_i is the partial loss got in current single step i. It is very easy to deduce the first part:

$$\frac{\partial L}{\partial y_{ia}} = v_{ti} * \frac{1}{y_{ia}} \tag{7.6}$$

We focus on only the first step, so the step subscript i here will be ommited later. Later, other step for example L_2, L_3 could use the same method to calculate. By doing this, only leaves the subscript a to denotes which action.

$$\frac{\partial L_1}{\partial y_a} = v_t * \frac{1}{y_a} \tag{7.7}$$

The Equ (7.7) shows the *small loss* L_1 which belongs to the first step. For a proper caculation for backpropagation, the $v_t * 1/y_a$ is devided into two part, the v_t and $1/y_a$. In the actual implementation, this part is $onehot[\,] * v_t * 1/y_a$. As shown in Fig (7.8), the $onehot[\,]$ denotes which action the agent takes in this step as shown in the column 4th and 5th in Table (7.2). If action a is taken, then $onehot[a]$ is 1, others are 0. The v_t here is the discounted return in the last column of Tab (7.2), and the product of $onehot[\,]$ and v_{ti} is the input of the backpropagation operation.

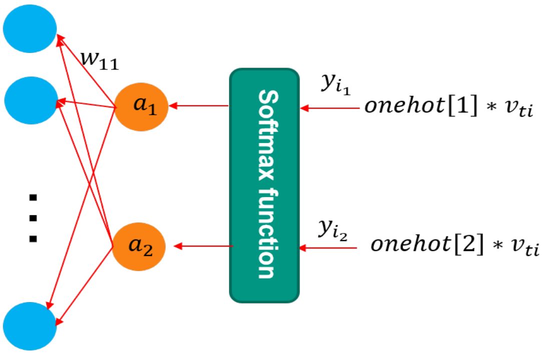


Figure 7.8: The beginning of the backpropagation is determined by the action taken in reality and the discounted return, From the last layer's softmax activation layer, the derivative to the loss function will backward propogate from this layer, and pass through all layer until reach to the input layer

The Equ (7.2) also demostrates this property. In Equ (7.2), the first partial loss (related to first step) L_1, $v_t * log(y_1)$ is equals to $5.5203 * log(0.5345)$. Here the action taken showing that it is not related to the action “going right", only related to the "going left". So in backpropagation, the "going right" part could be ignored. This is ignored by the $onehot[2]$, to select only the action that occured ($onehot[1]$), the reason is that the $onehot[2]$ is 0, then there is no backpropagation value pass through the neural network.

Then the second part is $\partial y_1/\partial a_1$, this is related to the softmax layer property. Without decude, here directly gives the result. When the subscript of the y_a, a, is equals to the subscript of the denominator a, for example here $1 = 1$, then the $\partial y_1/\partial a_1$ is:

$$\frac{\partial y_{i1}}{\partial a_1} = y_{i1} - {y_{i1}}^2 \tag{7.8}$$

If the subcript are different, then:

$$\frac{\partial y_{i2}}{\partial a_1} = -y_{i1} * y_{i2} \tag{7.9}$$

For the third part $\partial a_1/\partial w_{11}$, because the a_1 it is calculated by the fully connected neural network where:

$$a_1 = h_1 * w_{11} + h_2 * w_{12} + \cdots + h_{30} * w_{130} + b_1 \tag{7.10}$$

Here the h_1 is the output value of first unit in hidden layer, the w_{11} is the weight between first unit in hidden layer and the layer layers' first unit. The b_1 is the biases of the first unit of hidden layer.

So the $\partial a_1/\partial w_{11}$ could be easily deduced:

$$\frac{\partial a_1}{\partial w_{11}} = h_1 \tag{7.11}$$

After have all the information, the parameter of delta w_{11} will get. Then it could be updated by:

$$\begin{aligned}\frac{\partial L_1}{\partial w_{11}} &= \frac{\partial L_1}{\partial y_1}\frac{\partial y_1}{\partial a_1}\frac{\partial a_1}{\partial w_{11}} + \frac{\partial L_1}{\partial y_2}\frac{\partial y_2}{\partial a_1}\frac{\partial a_1}{\partial w_{11}} \\ &= v_t * \frac{1}{y_1} * 1 * (y_1 - {y_1}^2) * h_1 + v_t * \frac{1}{y_2} * 0 * (-y_1 y_2) * h_1 \\ &= v_t * (1 - y_1) * h_1\end{aligned} \tag{7.12}$$

$$w_{11} = w_{11} - \frac{\partial L}{\partial w_{11}} * 0.1 \tag{7.13}$$

The C code segment in Appendix C has the full implementation detail cover the above part from loss function to the partial difference to all the parameters. The only left part is how to use this information to update the weights and biases. One possible solution is shown in Equ (7.13) is a standard gradient descent method.

In Equ (7.13), 0.1 is the user learning rate that is the hyper-parameter for searching the gradient descent direction of the loss function. It is the same for updating bias of b. The user could also adjust to other value. It is the same as other weights and biases. The derivative of weights and the biases of the input layer and hidden layer is relatively easy, it can use the backward derivative pass from the hidden layer to calculate.

7.3 Policy Gradient training hardware design

The flow chart is widely used in technical design, working procedure and presentation. Fig (7.9) is the flow chart from step 13 to 16 in Alg (5). The training is *episodic*. In the CartPole case means the agent only learns at the end of each episode (when the pole fell down). After having the normalized discounted return, it will loop over each step. The V_{ti} will attach to the training data for step i. Here the training data is the history of how the agent interacted with the environment at step i. Then a backward propagation will pass the partial loss function l_i though the whole policy network to get the derivative of the parameter with respect to this partial loss function. An optimization method could use to update the parameter in the last step. When it finishes, the next partial loss function will be calculated for backward propagation and similar parameter updating happens until all the steps are covered in this episode.

At Fig (7.10), there are 6 files given to struct the Policy Gradient testbench on HighFlex2 PS side. It is written in C++. The *main.cc* is the top file that contains the basic reinforcement learning interaction steps. The *nnet_common.h* defines the basic data types of weights abd

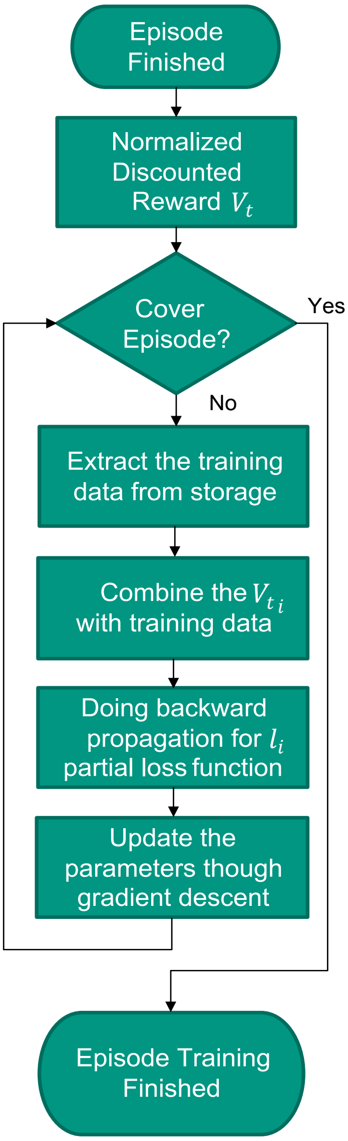
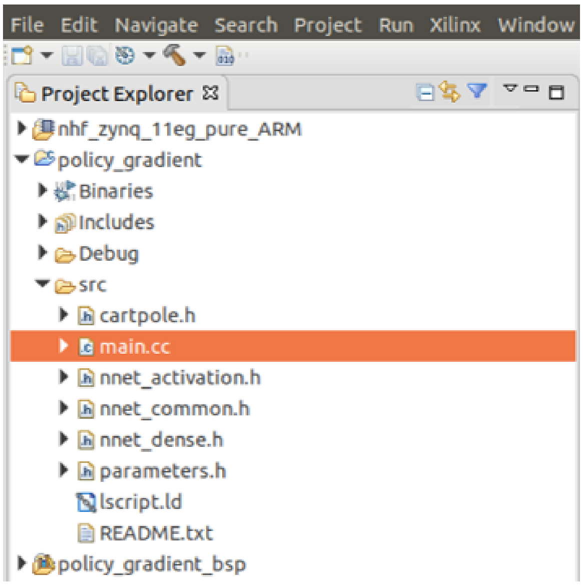

Episode Finished
Normalized Discounted Reward V_t
Cover Episode?
Yes
No
Extract the training data from storage
Combine the $V_{t\,i}$ with training data
Doing backward propagation for l_i partial loss function
Update the parameters though gradient descent
Episode Training Finished

Figure 7.9: episodic training flow chart

biases. The *parameters.h* defines the dense layer structures and activation layer structures. The *nnet_activation.h* The *nnet_dense.h* defines the C++ functions of forward pass. And finally the *cartpole.h* simulates the CartPole environment. All the backpropagation is directly written in C++ at *main.cc*. The Appendix C contains the whole part of backpropagation from softmax layer back to first layer. Which is also one of the block in Fig (7.9). After the backpropagation, the partial loss function is got. Then could use this information to update the weights.

7.4 Policy Gradient training curve

This section will exhibit the training curve on Zynq device, to proof the function validity on hardware. The unit under test will be the training procedure demonstrated in flow chart in Fig (7.9). The experiment is done on an Linux based PC and HighFlex2 DAQ board. The standard PC use Ubuntu 16.04.6 LTS operation system, on 7 Intel(R) Core(TM) i7-4710HQ @ 2.50GHz processors (cache size 6144KB). The result on CPU is shown in next four figures. Because the CPU version and GPU version have similar result, and also the purpose of this



- nhf_zynq_11eg_pure_ARM
- policy_gradient
  - Binaries
  - Includes
  - Debug
  - src
    - cartpole.h
    - main.cc
    - nnet_activation.h
    - nnet_common.h
    - nnet_dense.h
    - parameters.h
  - lscript.ld
  - README.txt
- policy_gradient_bsp

Figure 7.10: The policy gradient software structure at Xilinx SDK

section is verification, then here only demostrate the CPU version. According to the step 17 of Alg (5), a fixed threshold is set for a training trend comparison. The threshold value is set to 500, 1000, 2000, and 5000 steps in one episode.

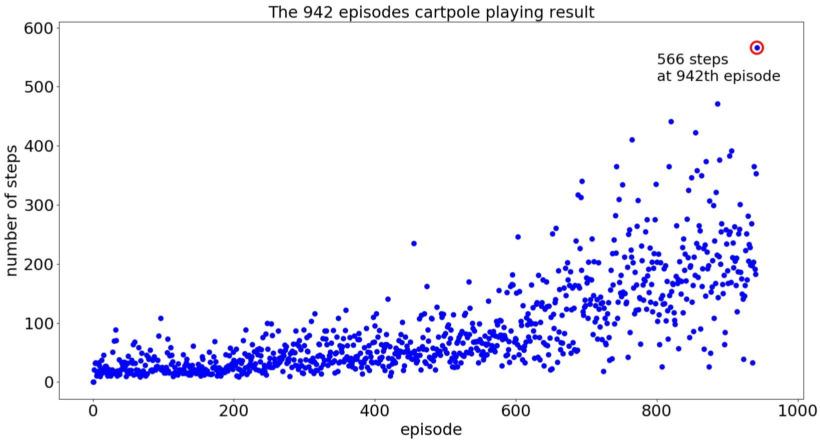


Figure 7.11: 942 episode on CPU

In figure 7.11, each blue point means one episode test. The y coordinate axis means how many steps in this episode. That means if in one episode, the cart could keep the pole for more than 500 steps, then the whole game stop. First of all, one could clear see the trend that the agent try to learn keeping the CartPole balanced for more number of steps in each episode. In the top right corner of the figure, the blue point circled by the red circle means that in the 942th episode, the pole is kept balanced for 566 steps on CPU.

In figure 7.12, the HighFlex2 Zynq test shows the pole is kept balanced for 515 steps at 936th episode as shown in the red cycle at the top right of the picture. The game finished at 936th episode. Compared with Fig (7.11) on CPU, the HighFlex2 Zynq have the similar behaviour.

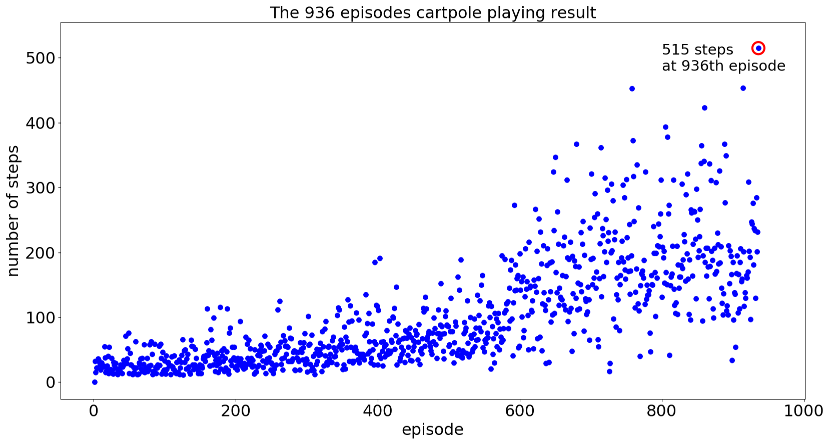


Figure 7.12: 936 episode on HighFlex2 Zynq

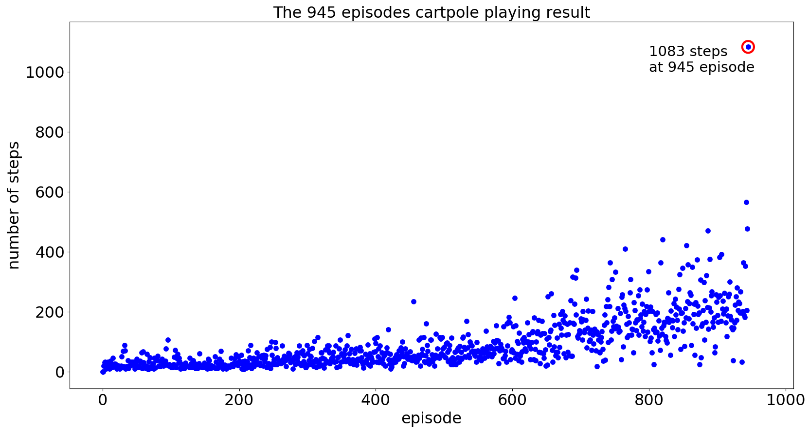


Figure 7.13: 945 episodes on CPU

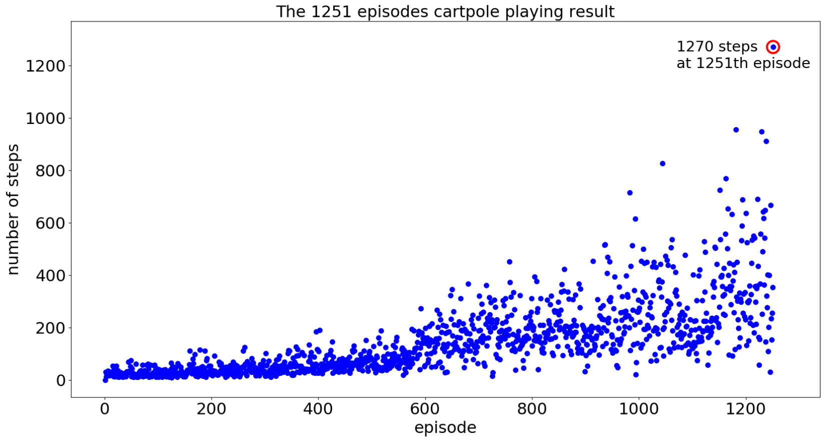


Figure 7.14: 1251 episodes on HighFlex2 Zynq

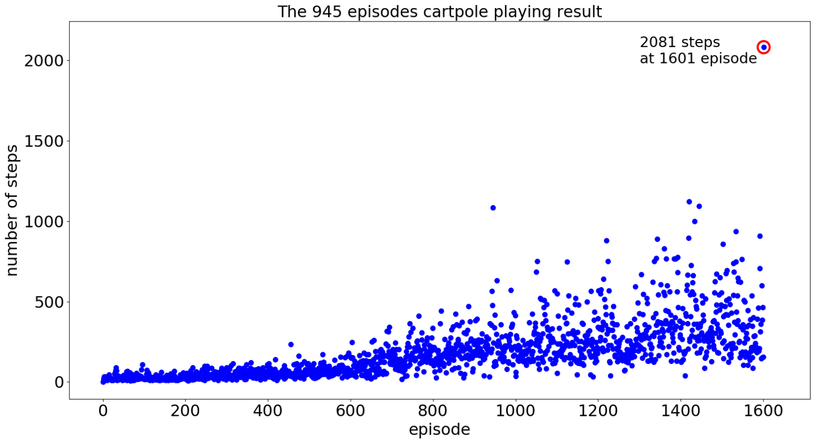


Figure 7.15: 1601 episodes on CPU

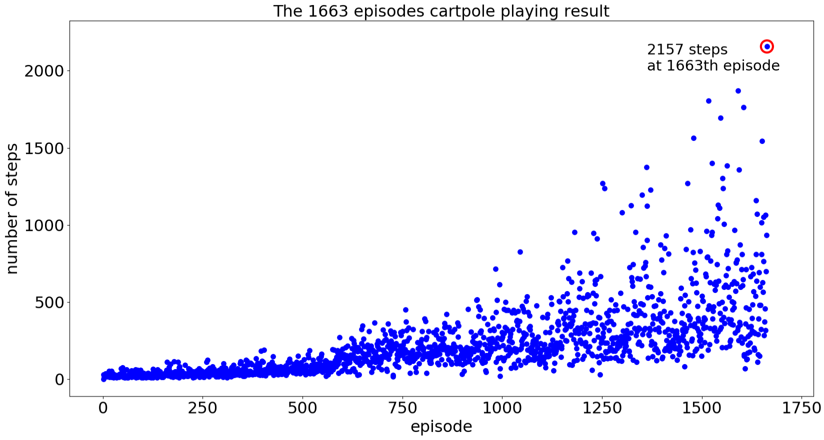


Figure 7.16: 1663 episodes on HighFlex2 Zynq

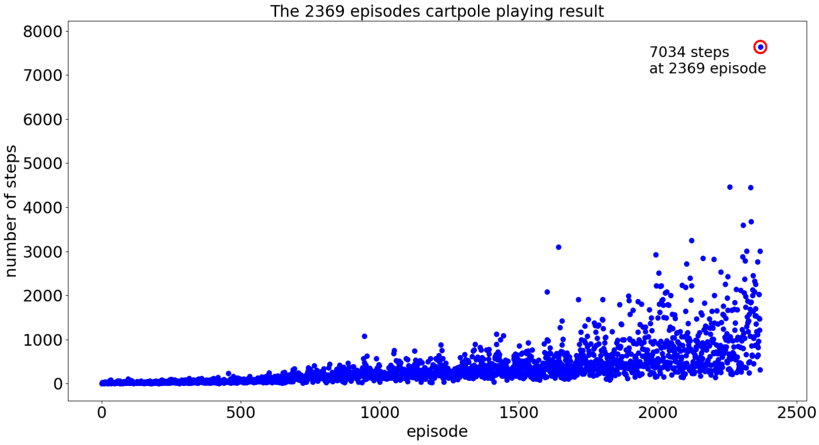


Figure 7.17: 2369 episodes on CPU

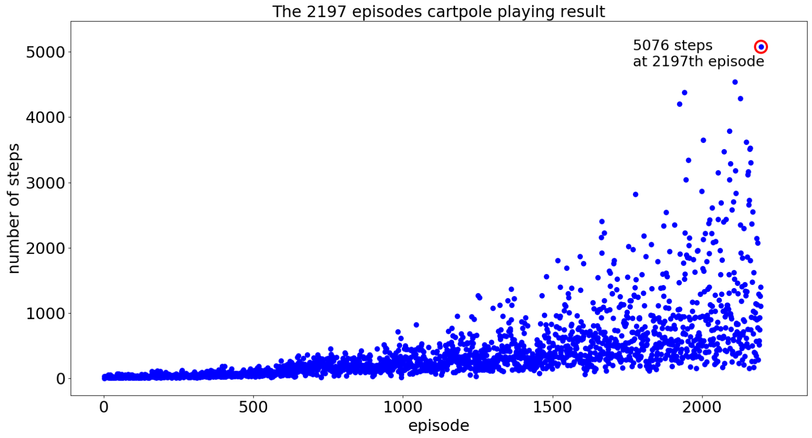


Figure 7.18: 2197 episodes on on HighFlex2 Zynq

In figure 7.13 is the testbench for searching episode that goes over 1000 steps. It shows the pole is kept balanced for 1083 steps at 945th episode on CPU. In figure 7.14, HighFlex2 test shows the pole is kept balanced for 1270 steps at 1251th episode as shown in the red cycle at the top right of the picture. The total training trend is same between CPU Fig (7.13) and HighFlex2 Fig (7.14).

The Fig (7.15) experiments that the CPU keep the pole balanced more than 2000 steps at 1601th episode. Fig (7.16) shows the pole is kept balanced for 2157 steps at 1663th episode as shown in the red cycle at the top right of the picture.

In Fig (7.18), HighFlex2 test shows the pole is kept balanced for 5076 steps at 2197th episode as shown in the red cycle at the top right of the picture. It is the samiliar behaviour shown in Fig (7.17) that the CPU version used 2369 episodes to reach a good episode that owns over 5 thousands steps.

7.5 Policy Gradient training performance

As the major purpose of this chapter is to focus on training latency of reinforcement learning. In terms of policy gradient, the step 13 to step 16 in Alg (5) is the research emphasis. The number of steps in one episode leads to different training time. Deduce from the Alg (5) itself, the training process is a loop that loop over each step in one episode, thus the training latency is roughly proportional to the step number if running purely on hardware without any overhead. But in reality, it is hard to precisely control the movement of agent manually in a fixed number of steps because the number of steps per episode depends on random action space searching seed, the state of the environment, the level of agent's learning (how well the policy network is trained so far), etc. Thus, to realize a ponderable and reasonable comparison between CPU, GPU, and Zynq, the step number in the episode should be the same. In this section, the *mandatory parameter set* is used for step number. This is set in step 4 of Alg (7).

The Alg (7) is the testbench for policy gradient method training process performance test. Only the training part in the algorithm is the unit under test, which is the step 12 to step 17. This means only from step 13 to step 16 is under test, and each test is 100 times, in order to get information about training stability. Step number for each episode will be forced as 1 steps, 100 steps, 5000 steps per episode. The number above 10000 steps is not reasonable because according to benchmarking [120], some of the cutting-edge reinforcement learning algorithms like REINFORCE, Trust Region Policy Optimization (TRPO), and Deep Deterministic Pol-

Algorithm 7 Policy Gradient Fixed Steps Training Testbench

```
 1: differential policy parameterization $\pi_{\boldsymbol{\theta}}(a|s)$ (fully connected neural network)
 2: initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^d$ and learning rate $\alpha > 0$
 3: allocate memory to store information about the interaction with the environment
 4: prepare a fixed number, that guarantee a episode only have that amount of steps
 5: repeat
 6:     choose action $A_t$ according to $\pi_{\boldsymbol{\theta}}(\cdot|S_t)$
 7:     $S_t, R_t, S_{t+1} \leftarrow$ env.step($A_t$)
 8:     $t \leftarrow t + 1$
 9:     store the $S_t, A_t, R_t, S_{t+1}$ in memory
10: until $t$ equals to fixed number of steps
11: loop over $100$ times of training test
12:     extract the episode information from memory
13:     calculate the cumulative discounted reward $G_t$ for this episode
14:     loop over steps in this episode $t = 0, 1, \ldots$
15:         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha G_t \nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(A_t|S_t)$
16:     end loop
17: end loop
```

icy Gradient (DDPG), the deployed algorithms is 4693.7 steps, 4869.8 steps and 4634.4 steps respectively in terms of average return. So in principle, a fixed training set is fed into the backward function of CPU, GPU, and ARM implementation. In order to acquire a thorough analysis of test data and different hardware, the experiment is tested 100 times for each *different size* of training data set.

The GPU agent and environment is implemented by TensorFlow and OpenAI, on NVIDIA Tesla K40c device. A detail parameters is exhibit in Fig (12) (Appendix B). The CPU running environment is same with GPU version, disabling the GPU. For CPU performance test, the code difference between GPU is only to disable the CUDA visbible device. The CartPole environment is fully implmented on Zynq by C.

The training process of policy gradient is running on the processor system part of Zynq. The unit under test is the same with GPU/CPU version in algorithm level, which is step 13 to 16 in Alg (5). In the ARM GNU compiler provides different optimization level for Cortex-A53 processor. The n in $-On$ denotes the optimization level. $-O0$ is **no optimization**, $-O1$ is **optimize medium**, $-O2$ is **optimize more** and $-O3$ is **optimize most**. The $-O0$ is normally for debugging, and others for release. In the following the training will use $-O3$ optimization level as default method unless otherwise specifically statement.

A thorough comparison of training latency under same condition is made by error-bar. If the x_i is the sampled test value, the $\overline{x}$ is the mean of x_i, then the standard deviation (SD) is Equ (7.14), and Equ (7.15) is standard error (SE), which denotes the individual sample and populations differences. The average time of training latency and standard error (SE) is used in error-bar. In this case, a low SE means a stable training latency.

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{N}(x_i - \overline{x})^2} \tag{7.14}$$

$$\sigma_{\overline{x}} = \frac{\sigma}{\sqrt{n}} \tag{7.15}$$

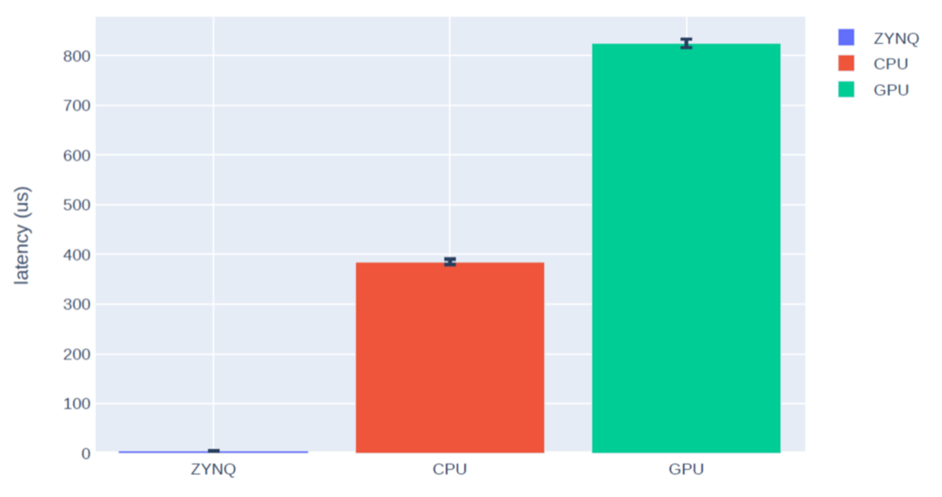


Figure 7.19: one-step episode error bar compared with ARM, CPU and GPU

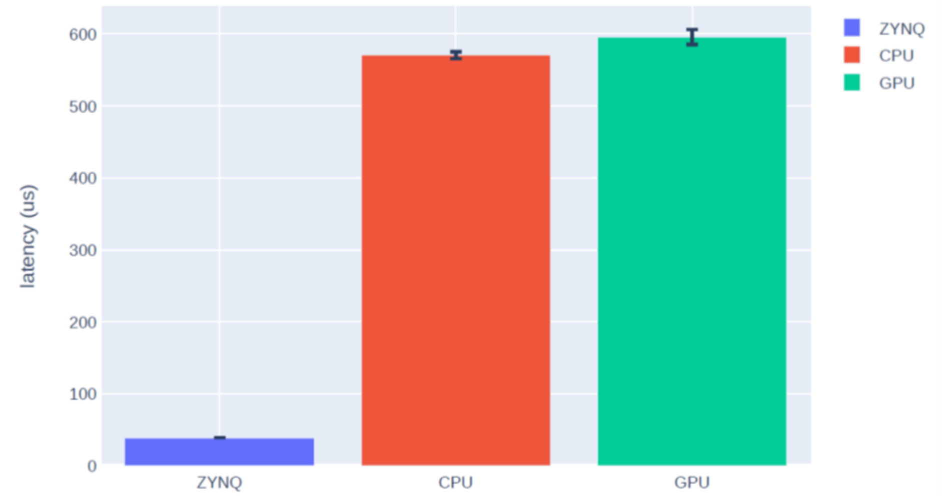


Figure 7.20: one-hundred-step episode error bar compared with ARM, CPU and GPU

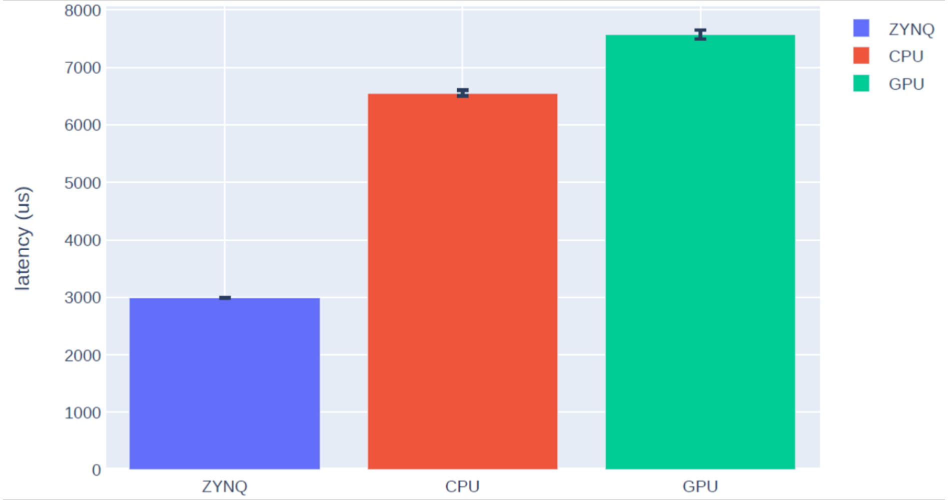


Figure 7.21: five-thousands-step episode error bar compared with ARM, CPU and GPU

In Fig (7.19), each bar stands for 100 times of one-step long episode training test. The level-3 optimization of Zynq is taken for comparison because of low latency. The height of the blue bar is the mean of training latency. The Zynq left have the 5.104 µs lowest training latency. The I-shaped on the bar represents the standard error of the training latency samples. Zynq have a rough 0.0145 µs SE. The CPU and GPU have around 385 µs and 824 µs respectively. The SE for CPU and GPU is about 5.47 µs and 8.57 µs, which is much higher than the Zynq version.

The Fig (7.19) is the error bar that combine the information of *100* steps episode test for CPU, GPU and Zynq. This means the training data size is 100. Each bar stands for the average training time for 100 times of *one-hundred-step* long episode training test. The height of the blue bar is the mean of training latency. The Zynq -O3 optimization left have the 38.96 µs lowest average training latency. The CPU and GPU have around 385 µs and 824 µs respectively. Zynq have a rough 0.0145 µs SE. The SE for CPU and GPU is about 5.47 µs and 8.57 µs, which is much higher than Zynq version.

The height of the blue bar in Fig (7.21) is the mean of training latency on Zynq for 5000-step long episode. The Zynq -O3 optimization left have the 2990 µs lowest training latency. The I-shaped on the bar represent the standard error of the training latency samples. Zynq have a rough 2.0577 µs SE. The CPU and GPU have around 6554 µs and 7571 µs respectively. The SE for CPU and GPU is about 49.9 µs and 79.12 µs, which is much higher than Zynq version.

7.6 Conclusion

This chapter gives the full implementation method for Policy Gradient. It contains several main contributions. First, it gives a detailed reference from the mathematic deduce of the training process for CartPole. This is the implementation theory reference for all hardware platforms. Second, the training curve proves the convergence of training on HighFlex2 and also proves that the traning latency has a much lower latency than other platforms. This full implementation method totally circumvents a CPU/GPU Keras or TensorFlow based training platform. It also provides an important starting point for the next chapters.

Chapter 8

DDPG on embedded processor

In the previous Policy Gradient (PG) chapter, the policy network could produce continuous action space. This method has some drawbacks. 1, this method have the problem of *high variance*; 2, it is hard to judge an action with only the reward, because the final accumulated reward is depending on many steps and actions, then a proper action criterion is required; 3, PG has a relatively low speed of convergence; 4, PG could only use in an episodic environment. *Actor-Critic* could solve such problems.

From the Equ (6.20), the Actor-Critic method is almost there. Actor-Critic Methods combines the Policy Gradient, the Policy Iteration and Value Iteration method. As mentioned in the Policy Gradient section, the Q value is used to behave as a critic, adjusting the probability of the appearance of the current trajectory. The critic part could use another neural network, known as the *evaluation network*, also called *critic network*.

To be specific, the $Q^{\pi_\theta}(s, a)$ in Equ (6.20) will be replaced with *critic network*: $Q(s{,}a\,|\,\theta^Q)$. The updating method of this network is the same as DQN, using TD-error as the loss function. The Deep Deterministic Policy Gradient (DDPG) algorithm is a typical Actor-Critic based method. Different from the basic version of Actor-Critic, DDPG has two additional networks than the *policy network* (Actor) and *evaluation network* (Critic): the *critic target network* and the *actor target network*. Each of the target networks is a copy from its counterpart at the beginning but update slowly towards its corresponding evaluation network. As the major work of the dissertation is to build the algorithm on the hardware, so the thesis here don't put the principle of the algorithm here, but only deliver the implementation details. The following directly give the solution to use the Deep Deterministic Policy Gradient algorithm (Alg (8)) to solve the pendulum problem in Fig (8.1).

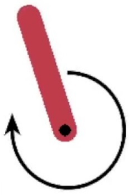


Figure 8.1: Pendulum Problem environment have three state, $\cos(\theta)$, $\sin(\theta)$, and $\dot{\theta}$. The theta is the angle between the pole and the vertical direction.

Algorithm 8 Deep Deterministic Policy Gradient [82]

```
Randomly initialize *critic network* $Q(s,a \mid \theta^Q)$ and *actor network* $\mu(s \mid \theta^\mu)$ with weights $\theta^Q$
and $\theta^\mu$
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay bufer $R$ (memory buffer)
**for** episode=1, M **do**
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    **for** $t = 1, T$ **do**
        Select action $a_t = \mu(s_t \mid \theta^\mu) + \mathcal{N}_t$ according
         to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1} \mid \theta^{\mu'})\theta^{Q'})$
        Update critic by minimizing the loss
         $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i \mid \theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
            $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a \mid \theta^Q) \mid_{s=s_i, a=\mu(s_i)} \nabla\theta^\mu \mu(s \mid \theta^\mu) \mid_{s_i}$
        Update the target networks:
                $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
                $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
    **end for**
**end for**
**return** $\theta$;
```

8.1 Pendulum control

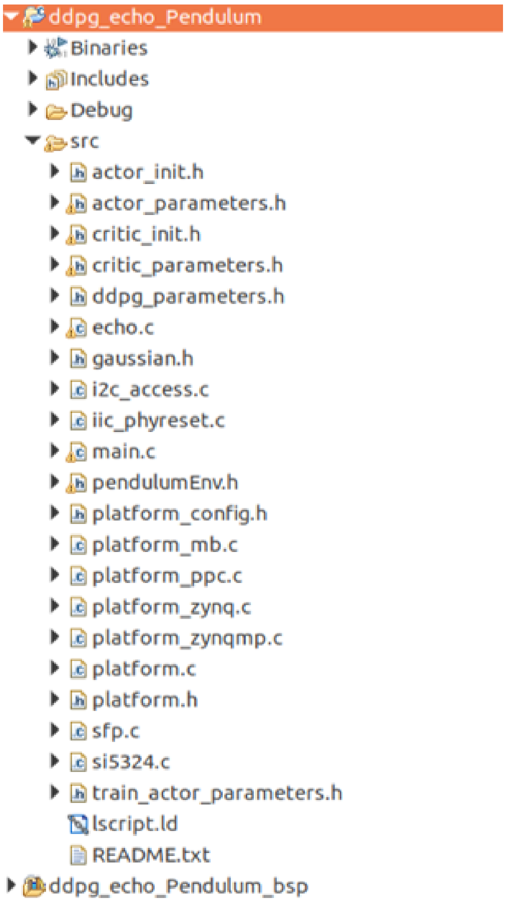
This section will introduce and solve one problem on HighFlex2: Pendulum problem, which is a standard reinforcement learning test bench mentioned in [82]. The environment version used is from OpenAI gym, Pendulum-v0 [121]. The pendulum problem is a classic control problem, also called pendulum swing-up problem, literally meaning use force to bring the pendulum to the upright position and try to keep balanced. The pendulum is a one-link pole with a 1-meter length, 1 kg weight and friction coefficient of 1 Nsm/rad. The balance purpose of pendulum control is to keep the pendulum standing up as shown in Fig (8.1). This means the targeting angle of the pole is 180 degree or -180 degree. The Alg (8) DDPG is used to solve the problem.

8.2 Build DDPG networks

This section will introduce the software structures and how the DDPG networks are established through C code. As shown in Fig (8.2) is the Xilinx SDK DDPG implementation. The DDPG networks definition are located in the files of *actor_parameters.h* and *critic_parameters.h.*

8.2.1 Critic network

A complete DDPG hardware implementation require the realization from first step fo last step of Alg (8). First is *critic network* and *actor network.*



```
ddpg_echo_Pendulum
  Binaries
  Includes
  Debug
  src
    actor_init.h
    actor_parameters.h
    critic_init.h
    critic_parameters.h
    ddpg_parameters.h
    echo.c
    gaussian.h
    i2c_access.c
    iic_phyreset.c
    main.c
    pendulumEnv.h
    platform_config.h
    platform_mb.c
    platform_ppc.c
    platform_zynq.c
    platform_zynqmp.c
    platform.c
    platform.h
    sfp.c
    si5324.c
    train_actor_parameters.h
    lscript.ld
    README.txt
ddpg_echo_Pendulum_bsp
```

Figure 8.2: The Xilinx SDK software version for DDPG that solving Pendulum

The most basic three signals in RL interaction are state, reward, and action as shown in Fig (6.1). The state that represent the status of pole is the position information $\cos(\theta)$, $\sin(\theta)$, and the speed information $\dot{\theta}$.

In Fig (8.3), the task of the *critic network* is to predict the $Q(s, a)$ value. The input of the *critic network* is action a and state s. In the Pendulum-V0 environment, the state is $\cos(\theta)$, $\sin(\theta)$, and $\dot{\theta}$, and the action is the joint effort, range from -2.0 N to 2.0 N, this denotes how much effort the agent apply on the joint. Then the total number of input unit is four. The output of the *critic network* is to give out the Q value, a scalar value. Therefore the output layer must be one unit to generate such scalar value. The hidden layer could be any form. Here we choose *four* fully-connected hidden layer as the major form, the activation layer after each hidden layer is ReLU [122, 123, 124]. By conclusion, the structure of *critic network* is: 4 input units (state signal and action signal), then *four* fully connected layers: 32 units hidden units (ReLU activation), and one output unit (linear activation). The θ^Q is the weights and biases within these 5 layers. One top function of the *critic network* implementation is given in Appendix D. All the parameters are stored in the *critic_parameters.h* as C struct.

```
struct critic_layer2_dense{
float weights[CRITIC_SECOND_LAYER_UNI]
[CRITIC_OUTPUT_LAYER_UNI];
float biases[CRITIC_OUTPUT_LAYER_UNI];
float acc[CRITIC_OUTPUT_LAYER_UNI];

// for current forward propagation
```

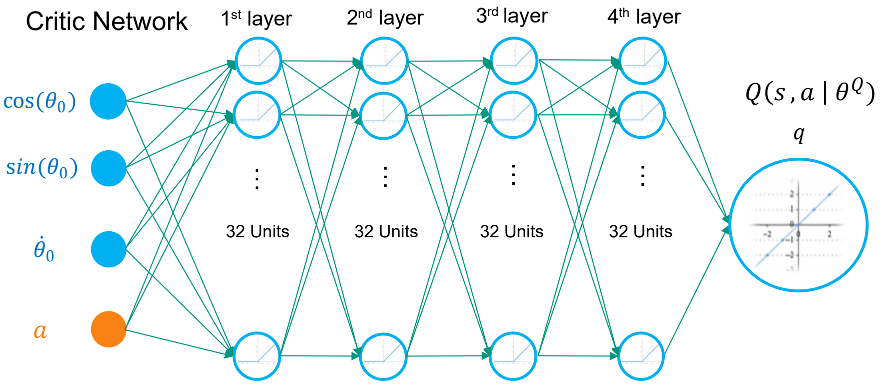


Figure 8.3: DDPG *critic network*

```
float datainput[CRITIC_SECOND_LAYER_UNI];
float result[CRITIC_OUTPUT_LAYER_UNI];

// partial derivative of loss function
float diff_weights [CRITIC_SECOND_LAYER_UNI]
[CRITIC_OUTPUT_LAYER_UNI];
float diff_biases [CRITIC_OUTPUT_LAYER_UNI];

float diff_in[CRITIC_OUTPUT_LAYER_UNI];
float diff_out[CRITIC_SECOND_LAYER_UNI];

float mt_weights [CRITIC_SECOND_LAYER_UNI][CRITIC_OUTPUT_LAYER_UNI];
float mt_biases [CRITIC_OUTPUT_LAYER_UNI];
float vt_weights [CRITIC_SECOND_LAYER_UNI][CRITIC_OUTPUT_LAYER_UNI];
float vt_biases [CRITIC_OUTPUT_LAYER_UNI];
}critic_layer2_dense;
```

Here is the example for layer2 of *critic network*. This C struct stores the important parameters that used in forward pass (datainput, result), backpropgation (diff, diff_in, diff_out), and Adam updating (mt and vt). And also the weights and biases of this layer. It is the same definition for *critic target network*, *actor network*, and *actor target network* layers.

8.2.2 Actor network

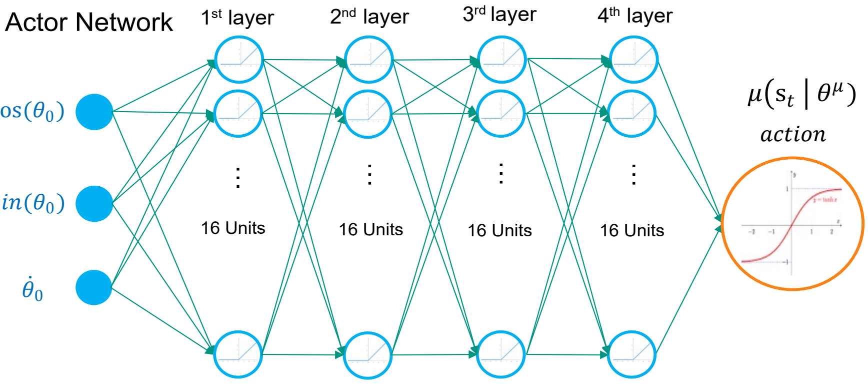


Figure 8.4: DDPG *actor network*

As shown in Fig (8.4), the task of the *actor network* is to predict appropriate action $a = \mu(s \mid \theta^{\mu})$ using the information state, then the input of the network is state, the output is action. For *critic network*, the input is state and action signal. State is $\cos(\theta)$, $\sin(\theta)$, and $\dot{\theta}$. And the action is the joint effort. The number of total input unit is three. The output of the *actor network* one. The hidden layer could be any form. Here we choose four hidden layers as the major form, the activation layer after each hidden layer is ReLU. By conclusion, the structure of *actor network* is: 3 input units (state signal), then*four* fully connected layers with 16 units hidden units (ReLU activation), and one output unit (Tanh activation). Tanh activation layer is used to generate the output from -1 to 1. To fulfil the -2 to 2 requirement from the environment only needs one linear multiplier afterwards. The θ^{μ} is the weights and biases in the 5 layers. One top function of the *critic network* implementation is given in Appendix E.

The $2nd$ step in Alg (8) is to build the corresponding target network. The target *critic network* and target *actor network* architecture is the same with *critic network* and *actor network*. In the beginning, the value of weights and biases are copied from its counterpart. Until now, we have four neural networks: *critic network*, *actor network*, and its reference: *target critic network*, *target actor network*. These four neural networks transmit information between each other during the training process. Because of this indistinguishable property, the following will use *italic format* to denotes these 4 neural networks.

8.2.3 Memory buffer

Following the Alg (8), the 3rd step is to prepare the replay buffer R, also called memory buffer. This could be implemented by hardware and software. To have a better comparison between software version, the test keeps as much as possible components same as Keras-RL [125], only the most important part, the agents (four neural networks) on the hardware. In this case, the memory buffer and the training dataset is provided by Keras-RL, only the *forward()* and *backward()* function in Keras-RL is replaced by HighFlex2 RL implementation. Although the buffer is implemented by Keras-RL, here the hardware implementation method will be given for a better explanation of DDPG and how memory buffer works inside.

Now comes down to the $1st$ loop (start from step 4) and $2nd$ loop (start from step 7) in Alg (8). The exterior loop ($1st$ loop) is episodic, looping over a fixed number of episodes during the whole interaction. This could be hundreds to tens of thousands, depends on the AI designer. In the inner loop, and also the most important loop, from step 8 to 10, is how the agent interacts with the environment in each step. This inner loop exactly reflects how the agent interacts with the environment following MDP. The same with Fig (6.1), a detail version of Fig (8.5) explain a one transition in the Alg (8) step 8 to 9. The current s_t is at θ_0, and policy network (*actor network*) gives out the action a_t, for example 1.6 newton (N) joint effort. After applying this action to the environment, the Pendulum moves from state0 (θ_0) to state1(θ_1). At the same time, the agent receives the immediate scalar reward r_t 1.3. Practically, the term *agent* receive all the information that happened at this interaction and stores them into the memory buffer.

In step 10, the agent needs to store the transition $\{s_t, a_t, r_t, s_{t+1}\}$ into the memory buffer, as shown in Fig (8.6). For example, if the agent already moves 323 steps from the beginning, then the memory pointer moves to the memory buffer position 323.

```
train_repo[repo_pointer][0] = current_cos_theta;
train_repo[repo_pointer][1] = current_sin_theta;
train_repo[repo_pointer][2] = current_theta_dot;
train_repo[repo_pointer][3] = action;
```

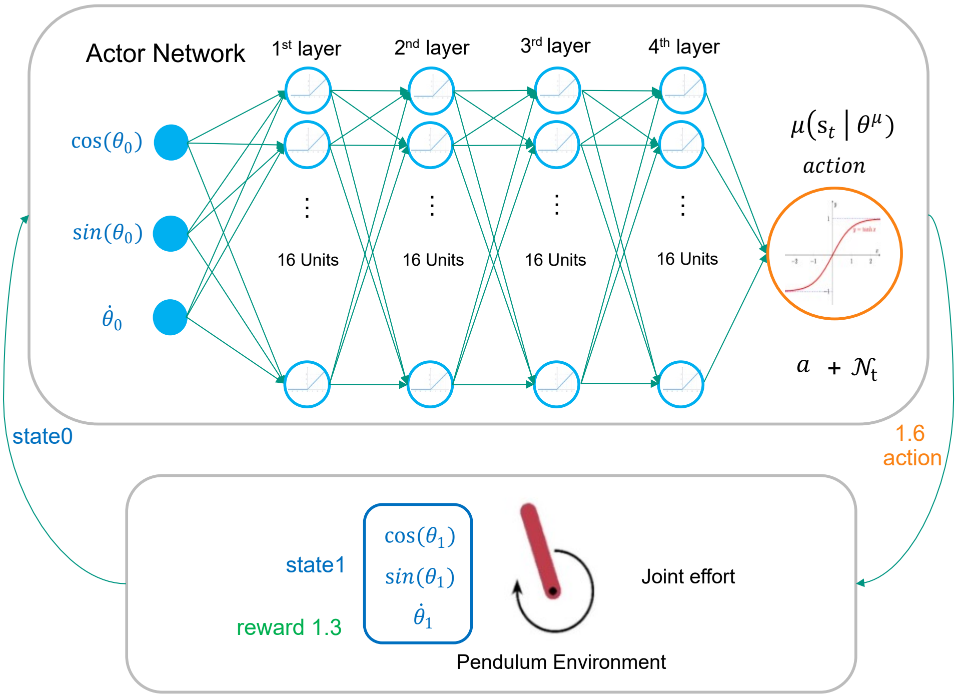
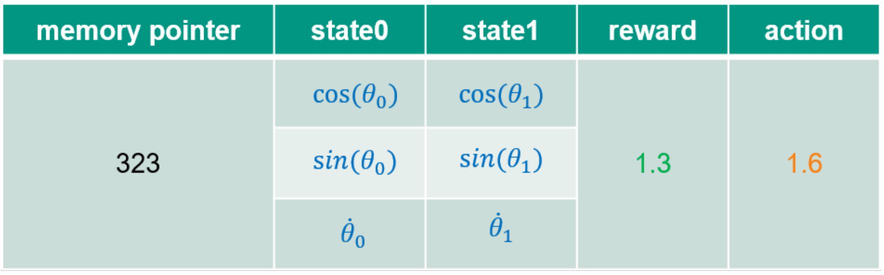


Figure 8.5: The DDPG actor interacts with pendulum environment



memory pointer	state0	state1	reward	action
323	$\cos(\theta_0)$	$\cos(\theta_1)$	1.3	1.6
	$sin(\theta_0)$	$sin(\theta_1)$		
	$\dot{\theta}_0$	$\dot{\theta}_1$		

Figure 8.6

```
train_repo[repo_pointer][4] = reward;
train_repo[repo_pointer][5] = next_cos_theta;
train_repo[repo_pointer][6] = next_sin_theta;
train_repo[repo_pointer][7] = next_theta_dot;
```

This memory buffer is ued in training stage of Alg (8) from step $11th$ step to $15th$. In step 11, a random batch of transition will be extracted from the memory buffer, for example, 32 images of different Fig (8.6). This is the training data for four networks of DDPG.

8.3 Train critic network

The training process of *critic network* requires a forward pass and backpropagation.

The forward pass has two parts: the *critic network* and the *target critic network*. The state signal S and action signal A from the training data set is feed into the *critic network*, then the output is Q. Next, the "next state" signal $S\_$ from the training data set is feed into *target critic network*, get a "target action", then this "target action" together with $S\_$ is feed into *target critic network*, get so called Q_target.

The Q_target, Q, and R from training data set constitute the *TD_error* shown clearly in

Fig (8.7). This is the loss function for training *critic network*. Then same with other training process, this loss goes back from *critic network* to get the partial derivative of loss function with respect to the *critic network* parameters. Adam optimizer is also used here to update the weights and biases of *critic network*. The forward pass and backpropagation functions also located at *actor_parameters.h* and *critic_parameters.h*. The Adam optimizer functions for *critic network* are located at *critic_parameters.h*.

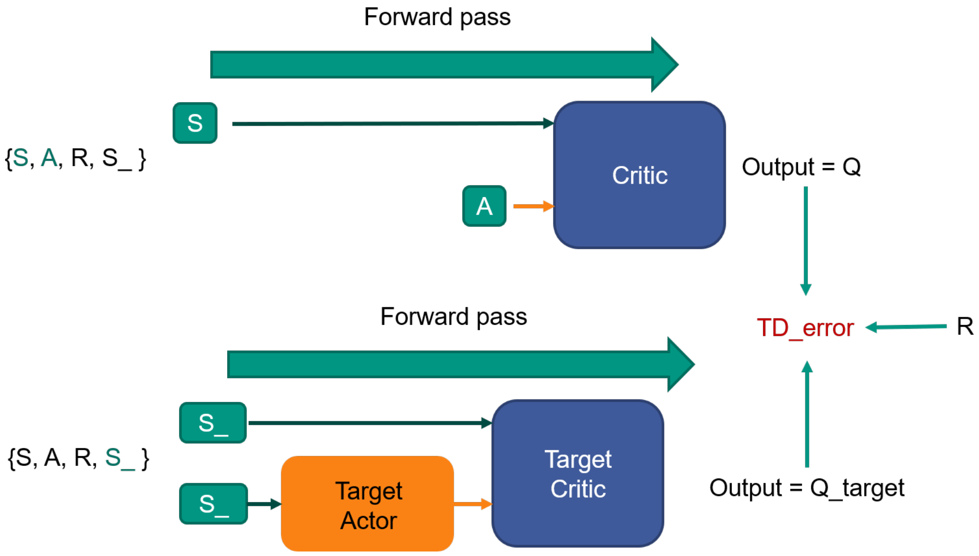


Figure 8.7: Overview of critic training (forward pass part)

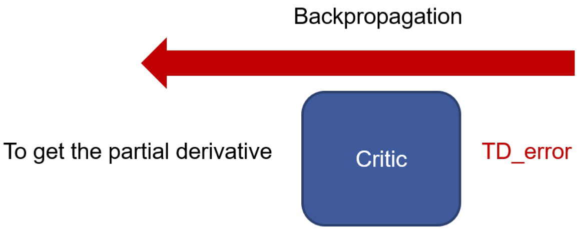


Figure 8.8: Overview of critic training (backpropagation part)

At the begining of this section, needs to review our basic approach to updating a neural network. It is not only suitable for reinforcement leanring here but also suitable for supervised leanring:

1) calculate the loss function using the forward propagation

2) use the information of forward propagation and use backpropagation to calculate the gradient of parameters

3) use the gradient to update the parameters through Adam, gradient descent method, or other optimizer

Before the training process, the training data needs to be extracted from the memory buffer, which is one single transition information: $\{s_t, a_t, r_t, s_{t+1}\}$ from one training batch of data.

This section will mainly discuss the step 12 and 13 in Alg (8). The target value of *critic network* is q_{target}, which is:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})\theta^{Q'}) \tag{8.1}$$

This is the reference Q value, and it is coming from the *target critic network* Q'. According to the Equ (8.1), the network Q' needs two inputs: the s_{i+1} and $\mu'(s_{i+1} \mid \theta^{\mu'})$. s_{i+1} could be directly get from memory buffer, and the $\mu'(s_{i+1} \mid \theta^{\mu'})$ is the forward propagation result when feed s_{i+1} to the *target actor network* shown in the bottom left of Fig (8.9). As shown in Fig (8.9), only use the information state1 s_{i+1} we could have the target Q value q'. Then the y_i in Equ (8.9) will be:

$$y_i = r_i + \gamma q' \tag{8.2}$$

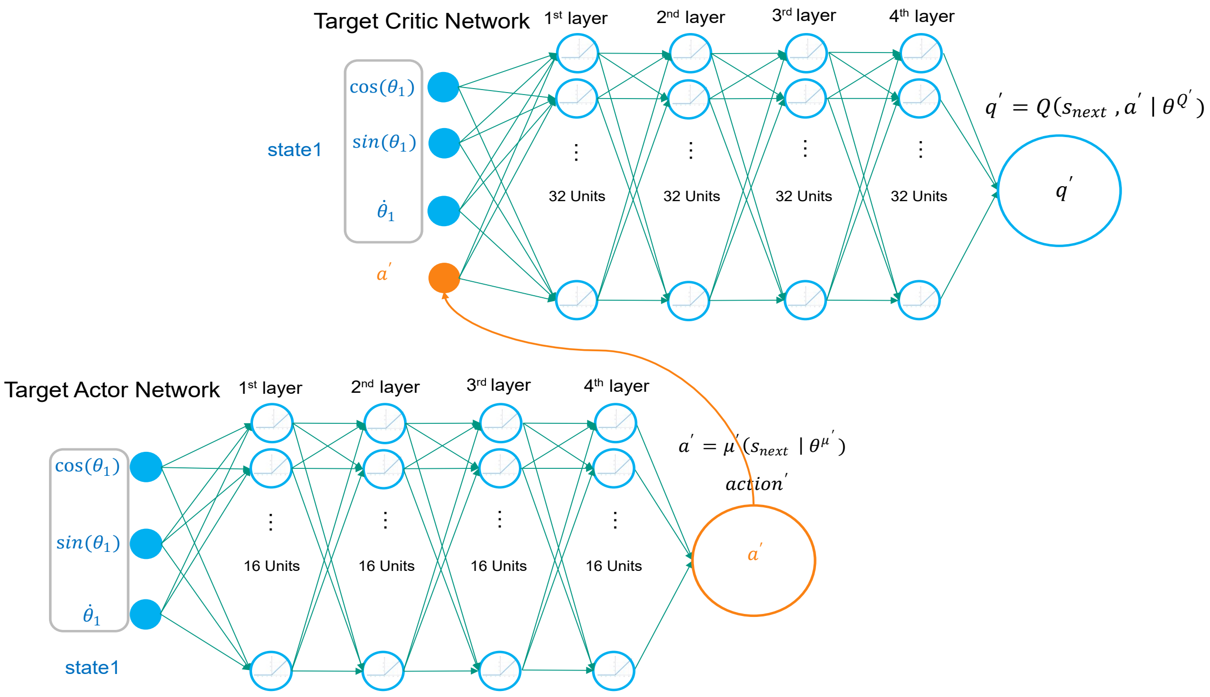


Figure 8.9: Calculation of target Q value

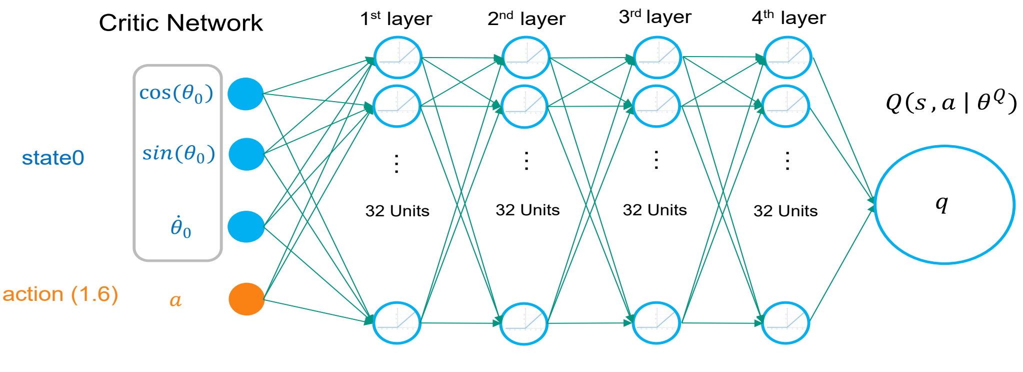


Figure 8.10: Q value forward propagation

The calculation of Q value q could use the *critic network* as shown in the Fig (8.10). We use the action from the memory buffer state0 and action that the agent taken in the history (1.6 shown in memory buffer Fig (8.6)).

```
// ******** the Critic Net ************** //

critic_input_layer( \
// actor_layer2_tanh.result,
critic_layer1_dense.action_input,
critic_layer1_dense.state_input, \
critic_layer1_dense.result,  \

critic_layer1_dense.action_weights, \
critic_layer1_dense.state_weights, \
critic_layer1_dense.biases
);

critic_layer1_active_relu(   \
critic_layer1_dense.result,  \
critic_layer1_relu.result  \
);

critic_dense_layer2(
critic_layer1_relu.result,  \
critic_layer2_dense.result, \
critic_layer2_dense.weights,  \
critic_layer2_dense.biases);

critic_layer2_active_relu(  \
critic_layer2_dense.result,  \
critic_layer2_relu.result  \
);

critic_dense_layer3(
critic_layer2_relu.result,  \
critic_layer3_dense.result, \
critic_layer3_dense.weights,  \
critic_layer3_dense.biases);

critic_layer3_active_relu(  \
critic_layer3_dense.result,  \
critic_layer3_relu.result  \
);

critic_dense_layer4(
critic_layer3_relu.result,  \
critic_layer4_dense.result, \
critic_layer4_dense.weights,  \
critic_layer4_dense.biases);

critic_layer4_active_relu(  \
critic_layer4_dense.result,  \
critic_layer4_relu.result  \
);

critic_dense_layer5(
critic_layer4_relu.result,  \
critic_layer5_dense.result, \
critic_layer5_dense.weights,  \
critic_layer5_dense.biases);

critic_layer5_linear_forward(
critic_layer5_dense.result, \
critic_layer5_linear.result,
1);
```

The code shows above is the *critic network* forward pass, for critic target network, that is the same. Only to notice that the input of these two network is different during training.

According to step $13rd$ of Alg (8), the loss function is determined by N batch of data, we only sample one training data set, then the loss function is modified to:

$$L = (y_i - Q(s_i, a_i \mid \theta^Q))^2 \tag{8.3}$$

```
// q_target = self.R + 0.9(Q_TARGET_GAMMA) * q_
critic_q_target = \
train_cost + \
Q_TARGET_GAMMA * \
critic_layer5_target_linear.result[0];

critic_diff_start = \
2 * (critic_layer5_linear.result[0] \
- critic_q_target);

critic_layer5_linear.diff_in[0] \
=  critic_diff_start;
```

This part of code clearly shows how to calculate the loss function for *critic network* training. The critic_diff_start in the code is the starting point of the backpropagation. This is passed to the last layer of *critic network*, then it continue to go back to the first layer.

Then update the *critic network* through the loss function, and according to the backpropagation mentioned in the policy gradient chapter, this will reflect the relationship between *critic network* parameter θ^Q and critic loss function. Secondly one can review the loss function in Equ (8.3), the *critic network* parameter θ^Q only have the relationship with $Q(s_i, a_i \mid \theta^Q)$, not y_i, the y_i here is similar with the labeled data in supervised learning, could be regarded as a fixed value. As the last layer in *critic network* is a linear function, between $4th$ layer and the output layer, the fully-connected connection has 32 weights and 1 biases belonging to θ^Q that is trainable.

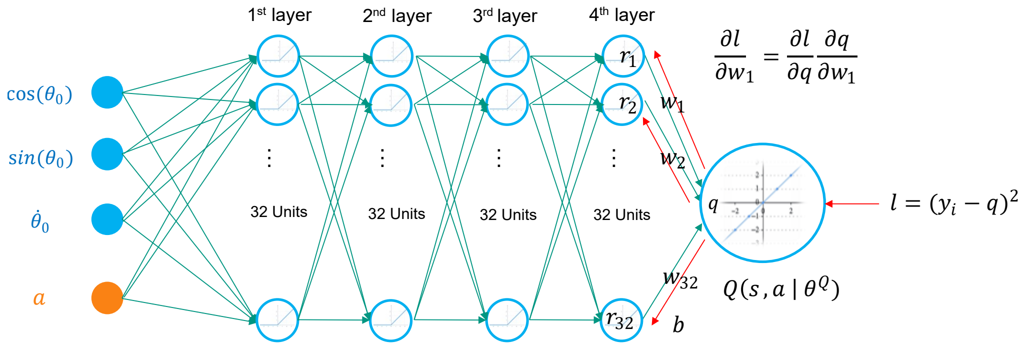


Figure 8.11: Illustration of backpropagation between 4th layer and 5th layer

As mentioned in policy gradient chapter, the purpose of backpropagation is to calculate the gradient (particial derivative) of trainable parameters. As shown in the Fig (8.11), the gradient to w_1 is very easy to deduce. The red arrow from the loss function pointed backward is the backpropagation data flow. It will moves from right to the left. The Fig (8.11) only shows the parameter update solution between $4th$ layer and output layer.

$$\begin{aligned} \frac{\partial l}{\partial w_1} &= \frac{\partial l}{\partial q}\frac{\partial q}{\partial w_1} \\ &= 2(y_i - q)\frac{\partial(w_1 r_1 + w_2 r_2 + \cdots + w_{32} r_{32} + b)}{\partial w_1} \\ &= 2(y_i - q) r_1 \end{aligned} \tag{8.4}$$

Where the y_i is coming from Equ (8.2), and the q is *critic network* output. The r_1 is the first output of $4th$ layer when feed state0 to *critic network* as shown in the Fig (8.11) on the $4th$ layer unit. This result is calculate from the forward propagation explained in Fig (8.10).

The derivative of bias b with respect to loss function is similar with Equ (8.5).

$$\begin{aligned} \frac{\partial l}{\partial b} &= \frac{\partial l}{\partial q}\frac{\partial q}{\partial b} \\ &= 2(y_i - q)\frac{\partial(w_1 r_1 + w_2 r_2 + \cdots + w_{32} r_{32} + b)}{\partial b} \\ &= 2(y_i - q) 1 \end{aligned} \tag{8.5}$$

After get the gradient between $4th$ layer and output layer, we need to continue to calculate other parameters in the neural networks. We consider what information is necessary to update the parameters between $3th$ layer and $4th$ layer. The answer is $\partial l/\partial r_1$. This partial derivative must be passed back.

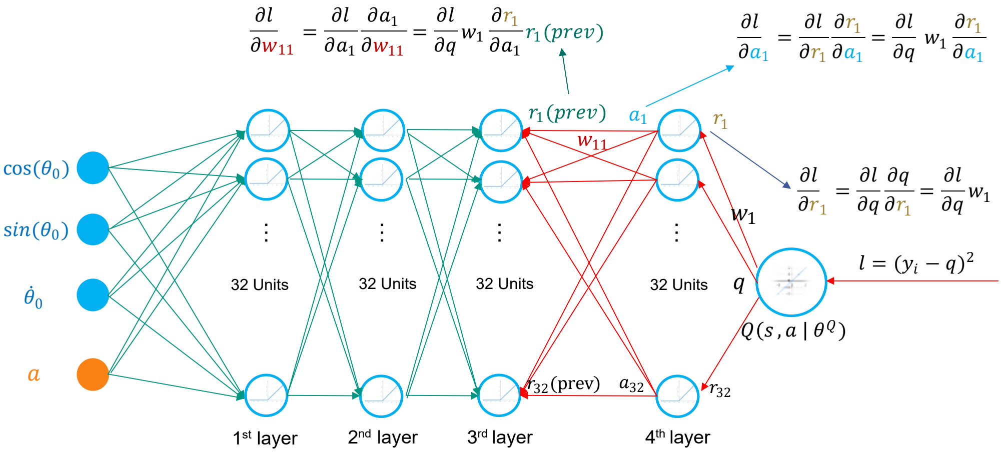

$\frac{\partial l}{\partial w_{11}} = \frac{\partial l}{\partial a_1}\frac{\partial a_1}{\partial w_{11}} = \frac{\partial l}{\partial q} w_1 \frac{\partial r_1}{\partial a_1} r_1(prev)$
$\frac{\partial l}{\partial a_1} = \frac{\partial l}{\partial r_1}\frac{\partial r_1}{\partial a_1} = \frac{\partial l}{\partial q} w_1 \frac{\partial r_1}{\partial a_1}$
$r_1(prev)$
a_1
r_1
w_{11}
$\frac{\partial l}{\partial r_1} = \frac{\partial l}{\partial q}\frac{\partial q}{\partial r_1} = \frac{\partial l}{\partial q} w_1$
$cos(\theta_0)$
$sin(\theta_0)$
$\dot{\theta}_0$
a
w_1
32 Units
32 Units
32 Units
32 Units
q
$l = (y_i - q)^2$
$Q(s, a \mid \theta^Q)$
r_{32}(prev)
a_{32}
r_{32}
1st layer
2nd layer
3rd layer
4th layer

Figure 8.12: Illustration of backpropagation between 3th layer and 4th layer

For the derivative of weight w_{11} in Fig (8.12), it always includes two parts. The first part, $\partial l/\partial a_1$, could be calculated locally. The second part, $\partial a_1/\partial w_{11}$, is transfered from backward pass during backward propagation. It is same for all parameters which is located on the backward pass. From Fig (8.12), it is clearly to observe the derivative of weight w_{11} with respect to loss function. Notice that the subscript $_{11}$ denotes that this weight connects the $1st$ unit on

$3rd$ layer with $1st$ unit on $4th$ layer. As we need the $\partial l/\partial w_{11}$, which is $\partial l/\partial a_1 \cdot \partial a_1/\partial w_{11}$. The $\partial l/\partial a_1$ is coming from the backward data pass, on the other hand, the $\partial a_1/\partial w_{11}$ could be calculated locally:

$$\begin{aligned}\frac{\partial a_1}{\partial w_{11}} &= \frac{\partial(r_1(prev)w_{11} + r_2(prev)w_{21} + \cdots + r_{32}(prev)w_{321} + b_1)}{\partial w_{11}} \\ &= r_1(prev)\end{aligned} \tag{8.6}$$

The $r_1(prev)$ is different from the r_1 as shown in the Fig (8.12). The $r_1(prev)$ means the $3rd$ layer ReLU activation output, and the r_1 is the first unit output of $4th$ layer ReLU activation. They are different output at different location as shown in Fig (8.12).

After having the local gradient, we need the partial gradient $\partial l/\partial a_1$, and this information is coming from the backpropagation.

$$\frac{\partial l}{\partial a_1} = \frac{\partial l}{\partial r_1}\frac{\partial r_1}{\partial a_1} \tag{8.7}$$

Again, the $\partial l/\partial r_1$ is coming from the backpropagation, and the $\partial r_1/\partial a_1$ is the local gradient. The relationship between r_1 and a_1 is depending on the ReLU function, if the input value a_1 is bigger than 0, then $\partial r_1/\partial a_1 = 1$, otherwise is 0. Where the $\partial l/\partial r_1$ could be calculated in Equ (8.8).

$$\begin{aligned}\frac{\partial l}{\partial r_1} &= \frac{\partial l}{\partial q}\frac{\partial q}{\partial r_1} \\ &= \frac{\partial l}{\partial q}w_1\end{aligned} \tag{8.8}$$

Here the w_1 is the first parameter between $4th$ layer and last layer, it is different from w_{11}. Until now, we have the full information to update the w_{11}. As we conbine the partial derivative from Equ (8.6), Equ (8.7) and Equ (8.8), we could have the Equ (8.9).

$$\begin{aligned}\frac{\partial l}{\partial w_{11}} &= \frac{\partial l}{\partial a_1}\frac{\partial a_1}{\partial w_{11}} \\ &= \frac{\partial l}{\partial q}w_1\frac{\partial r_1}{\partial a_1}r_1(prev)\end{aligned} \tag{8.9}$$

As to providing the partial derivative to the front layer, the $\partial l/\partial r_1(prev)$ needed to deduce.

$$\begin{aligned}\frac{\partial l}{\partial r_1(prev)} &= \frac{\partial l}{\partial a_1}\frac{\partial a_1}{\partial r_1(prev)} \\ &= \frac{\partial l}{\partial q}w_1\frac{\partial r_1}{\partial a_1}w_{11}\end{aligned} \tag{8.10}$$

```

critic_layer5_linear_back(
critic_layer5_linear.diff_in , \
critic_layer5_linear.diff_out , \
1);

critic_dense_layer5_back(
critic_layer4_relu.result,  \
critic_layer5_dense.result, \
critic_layer5_linear.diff_out, \
critic_layer5_dense.diff_out, \
critic_layer5_dense.weights, \
critic_layer5_dense.biases, \
critic_layer5_dense.diff_weights, \
critic_layer5_dense.diff_biases);

critic_layer4_relu_back(
critic_layer5_dense.diff_out,  \
critic_layer4_relu.diff_out, \
critic_layer4_dense.result);
critic_layer3_relu.result,  \
critic_layer4_dense.result, \
critic_layer4_relu.diff_out, \
critic_layer4_dense.diff_out, \
critic_layer4_dense.weights, \
critic_layer4_dense.biases, \
critic_layer4_dense.diff_weights, \
critic_layer4_dense.diff_biases);

critic_layer3_relu_back(
critic_layer4_dense.diff_out,  \
critic_layer3_relu.diff_out, \
critic_layer3_dense.result);

critic_dense_layer3_back(
critic_layer2_relu.result,  \
critic_layer3_dense.result, \
critic_layer3_relu.diff_out, \
critic_layer3_dense.diff_out, \
critic_layer3_dense.weights, \
critic_layer3_dense.biases, \
critic_layer3_dense.diff_weights, \
critic_layer3_dense.diff_biases);

critic_layer2_relu_back(
critic_layer3_dense.diff_out,  \
critic_layer2_relu.diff_out, \
critic_layer2_dense.result);

critic_dense_layer2_back(
critic_layer1_relu.result,  \
critic_layer2_dense.result, \
critic_layer2_relu.diff_out, \
critic_layer2_dense.diff_out, \
critic_layer2_dense.weights, \
critic_layer2_dense.biases, \
critic_layer2_dense.diff_weights, \
critic_layer2_dense.diff_biases);

critic_layer1_relu_back(
critic_layer2_dense.diff_out,  \
critic_layer1_relu.diff_out, \
critic_layer1_dense.result);

critic_layer1_back(
critic_layer1_dense.state_input,\
critic_layer1_dense.action_input,\
critic_layer1_dense.result, \
critic_layer1_relu.diff_out, \
critic_layer1_dense.state_weights, \
critic_layer1_dense.action_weights, \
critic_layer1_dense.biases, \
critic_layer1_dense.diff_state_weights, \
critic_layer1_dense.diff_action_weights, \
critic_layer1_dense.diff_biases\
);
```

The above codes is the backpropagation of *critic network*. Once this is done, at each layer will get a "diff_weights" and "diff_biases". This is the partial derivative of loss function with respect to weights and biases. The Adam optimizer will be used after this stage.

As shown in the Fig (8.12), at every red arrow stage of the backward, the backpropagation function have two major tasks:

1. move the partial derivative from the back to the front of current layer
2. calculate the gradient of current layer parameters with respect to the loss function

Other part of the backpropagation stage, for example the ReLU unit, only have one task that move the partial derivative to the next stage. One typical backpropagation function is shown below:

```
void critic_layer2_back(
float datainput[CRITIC_SECOND_LAYER_UNI],
float result[CRITIC_OUTPUT_LAYER_UNI],

float diff_in[CRITIC_OUTPUT_LAYER_UNI],
float diff_out[CRITIC_SECOND_LAYER_UNI],

float weights[CRITIC_SECOND_LAYER_UNI]
             [CRITIC_OUTPUT_LAYER_UNI],
float biases[CRITIC_OUTPUT_LAYER_UNI],

float diff_weights[CRITIC_SECOND_LAYER_UNI]
                  [CRITIC_OUTPUT_LAYER_UNI],
float diff_biases[CRITIC_OUTPUT_LAYER_UNI]
)
{
```

```
Product_diff_weights:
for(int ii = 0; ii<CRITIC_SECOND_LAYER_UNI; ii++)
{

    Product2: for(int jj=0;
    jj<CRITIC_OUTPUT_LAYER_UNI; jj++)
    {
        diff_weights[ii][jj] =
        datainput[ii]*diff_in[jj];
    }
}

Product_diff_bias:
for(int jj = 0; jj<CRITIC_OUTPUT_LAYER_UNI; jj++)
{
    diff_biases[jj] = diff_in[jj];
}

// calculate the diff_out
Reset_diff_out:
for(int ii = 0; ii<CRITIC_SECOND_LAYER_UNI; ii++)
{
    diff_out[ii] = 0;
}

Product_diff_out:
for(int ii = 0; ii<CRITIC_SECOND_LAYER_UNI; ii++)
{

    Product_diff_in:
    for(int jj=0;
    jj<CRITIC_OUTPUT_LAYER_UNI; jj++)
    {
        diff_out[ii] =
        diff_out[ii] +
        diff_in[jj]*weights[ii][jj];
    }
}


}
```

By using the same method, moves the partial derivative from the last layer to the first layer, and get all gradients of neural network parameters with respect to the loss function.

The optimization method of *critic network* is Adam [126]. Now have the gradient of weights and bias between $4th$ layer and output layer. Using gradient descent to update the parameter is simple, but in DDPG, the Adam optimizer is required. Because in the real implementation of solving pendulum problem, the gradient descent could not bring the DDPG training process into convergence. It is worth mentioning that in the most cutting-edge academic papers, there is no constant use of Adam or NAdam (Nesterov ac-celerated Adam) [127, 128] and other accepted adaptive optimization algorithms. What is the effective optimization method is depending on the problem itself.

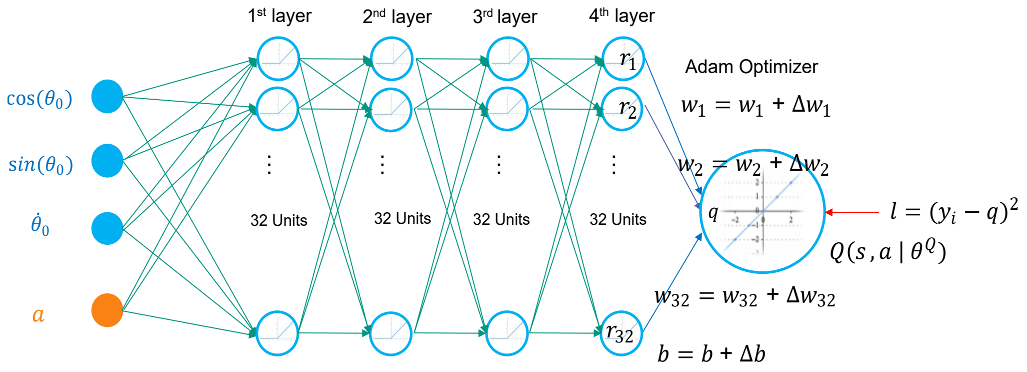


Figure 8.13: Overview of the optimizer used to update the value of weights and biases of network

For any kind of optimization method, they use parameter gradient to loss function as shown in Fig (8.13). The difference is how the optimizer use this gradient information. First of all, for Adam optimizer, the input of the function is weights, biases, the gradient of weights and biases `diff_weights` and `diff_biases` that calculated before. Each parameter have its own gradient (m_t) and the squared gradient (v_t), so there is `mt_weights` and `vt_weights` belongs to w_1 to w_{32}, and `mt_biases` and `vt_biases` belongs to b. And finally the learning rate. In following will explain how the Adam is implemented.

First task is calculate the update for m_t, which is explained in the Equ (8.11), it is to update the biased first moment estimate. The g_t is the gradient of parameters. The step t is to count how many Adam optimizer is been called. Every time update needs to increment the step t.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{8.11}$$

Second task is the update for v_t. which is explained in the Equ (8.12), it is to update the biased first moment estimate. The g_t is the gradient of parameters.

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) {g_t}^2 \tag{8.12}$$

The third step is to update the parameters using the intermediate result above. After calculate bias-corrected first moment estimate and bias-corrected second raw moment estimate, which explained in Equ (8.13)

$$\begin{aligned} \hat{m}_t &= m_t / (1 - {\beta_1}^t) \\ \hat{v}_t &= v_t / (1 - {\beta_2}^t) \\ \theta &= \theta - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) \end{aligned} \tag{8.13}$$

Here in Equ (8.13) the α is the *learning rate*, θ is the neural network parameters weights and biases. The ϵ is to prevent if the denominator is zero, here we choose 0.00000001 according to the Keras DDPG agent implementation by reverse engineering. Because through thorough reverse engineering, the hardware implementation could hole closely aligned with Keras implementation that keeps the correctness and is easy for debugging.

Up to now, the step 12 and 13 in Alg (8) is finished for *critic network*, in the next section will discuss how to update *actor network*.

8.4 Train actor network

This section will mainly focus on the training of *actor network*, which steps 14 in Alg (8). The same with *critic network* training, the training data is coming from the memory buffer, as shown in Fig (8.6). The purpose of policy gradient is to increase the $Q(s, a)$. Thus the Q is called as *objective function*, and $-Q$ as *loss function*.

Before one training process, the first is to get the training data from the repository as shown in Fig (8.14). The dataset S, A, R, S is one training data. It is one of the history movement, illustrated in step 13 in Alg (9).

To train the *actor network*, the first task is to do one time forward pass as illustrated clearly in the previous subsection (Train *critic network*). The value used here is S. After applying the state to the *actor network*, the output of *actor network* will be directly fed into the *critic network* as one of the input. Another input for the *critic network* is this state signal S. Finally, the *critic network* will generate the Q value. The forward pass functions are located in the *actor_parameters.h* and *critic_parameters.h*.

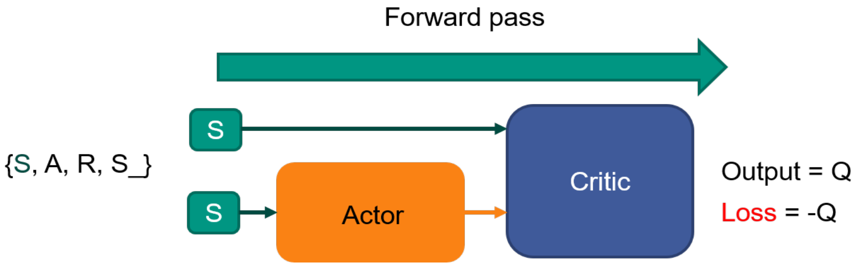


Figure 8.14: Overview of actor training (forward pass part)

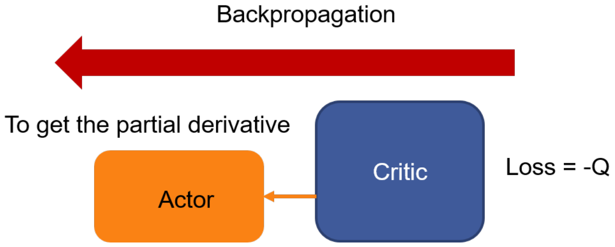


Figure 8.15: Overview of actor training (backpropagation part)

The minus of Q value is the loss function Fig (8.15). Then this loss will go back from the critic until the actor. Through this, one could get the partial derivative of loss function with respect to the *actor network* parameters. Then the Adam will be used to update the *actor network*. The forward pass and backpropagation functions also located at *actor_parameters.h* and *critic_parameters.h*. The Adam for *actor network* is located at *actor_parameters.h*.

As shown in Fig (8.16), the training for actor needs first feed the training data set$\{s_t, a_t, r_t, s_{t+1}\}$ for a forward pass. The state0 in Fig (8.16) is from the training date set s_t. And most important part in real implementation, is the *action signal* that feed into the *critic network* is the action value from the *action network*. So the forward pass for *actor network* is:

$$a = \mu(s_t \mid \theta^{\mu}) \tag{8.14}$$

The forward pass for *critic network* is:

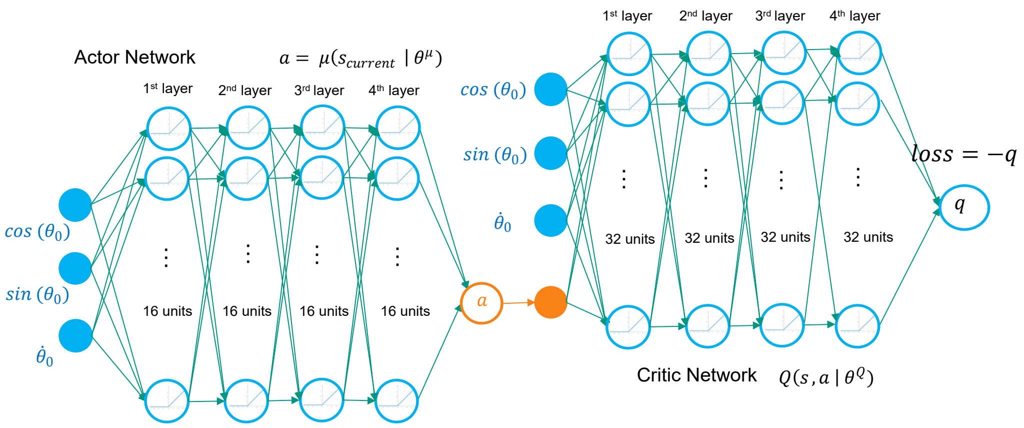


Figure 8.16: The forward pass between *actor network* and *critic network*

$$q = \mu(s_t, a_t \mid \theta^Q) \tag{8.15}$$

Our target is to maximize the Q, and to update the *actor network* θ^μ. Because the Q value is influenced by the *actor network* and *critic network*, first the partial derivative is passed through the *critic network* and back to the *actor network*. As the goal is to update the *actor network* parameter θ^μ the only difference is when the partial derivative pass through *critic network*, there is no needed to update the parameters for *critic network*, but only need pass through the partial derivative. Because the forward and backpropagation will pass through two networks, the code example is too long to put, thus it is located at the Appendix F and G.

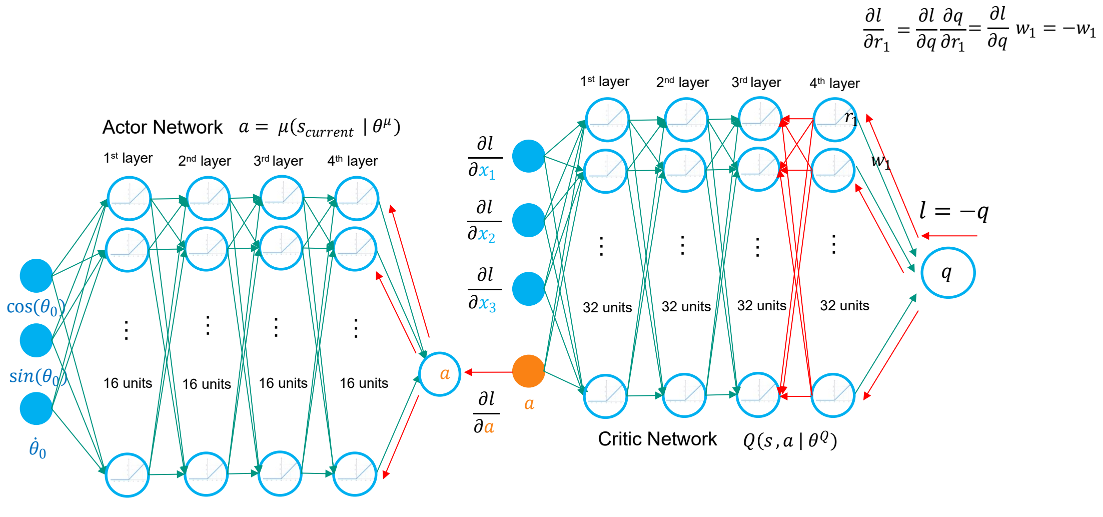

$\frac{\partial l}{\partial r_1} = \frac{\partial l}{\partial q}\frac{\partial q}{\partial r_1} = \frac{\partial l}{\partial q} w_1 = -w_1$
Actor Network
$a = \mu(s_{current} \mid \theta^{\mu})$
1st layer
2nd layer
3rd layer
4th layer
$\cos(\theta_0)$
$sin(\theta_0)$
$\dot{\theta}_0$
16 units
16 units
16 units
16 units
a
$\frac{\partial l}{\partial x_1}$
$\frac{\partial l}{\partial x_2}$
$\frac{\partial l}{\partial x_3}$
$\frac{\partial l}{\partial a}$
a
1st layer
2nd layer
3rd layer
4th layer
r_1
w_1
32 units
32 units
32 units
32 units
$l = -q$
q
Critic Network
$Q(s, a \mid \theta^{Q})$

Figure 8.17: The backward pass between *actor network* and *critic network*

From the beginning, the partial derivative $\partial l / \partial r_1$ we could easily deduce:

$$\begin{aligned} \frac{\partial l}{\partial r_1} &= \frac{\partial l}{\partial q}\frac{\partial q}{\partial r_1} \\ &= -w_1 \end{aligned} \tag{8.16}$$

Then later the backpropagation continuous the same as training *critic network*. As the partial derivative is passed to the first layer of *critic network*, we will get the information as shown in Fig (8.17), the $\partial l/\partial x_1$, the $\partial l/\partial x_2$, the $\partial l/\partial x_3$, and the $\partial l/\partial a$. Only the $\partial l/\partial a$ is used to continue transfering to the *actor network*. The only different part is in *critic network*, the final unit is linear activation, but in *actor network* the last unit use tanh activation and a multiplying unit 2. The forward pass function is:

$$\begin{aligned} tanh(x) &= 2\frac{e^x - e^{-x}}{e^x + e^{-x}} \\ &= 2\frac{1 - e^{-2x}}{1 + e^{-2x}} \end{aligned} \tag{8.17}$$

To deduce the backward pass, we need to know the derivative of this function.

$$\begin{aligned} tanh'(x) &= 2\frac{-e^{-2x}(-2)(1+e^{-2x}) - (1-e^{-2x})e^{-2x}(-2)}{(1+e^{-2x})^2} \\ &= 2\frac{2e^{-2x} + 2e^{-4x} + 2e^{-2x} - 2e^{-4x}}{(1+e^{-2x})^2} \\ &= 2\frac{4e^{-2x}}{(1+e^{-2x})^2} \\ &= 2\frac{(1+e^{-2x})^2 - (1-e^{-2x})^2}{(1+e^{-2x})^2} \\ &= 2 - 2\frac{(1-e^{-2x})^2}{(1+e^{-2x})^2} \\ &= 2 - 2tanh^2(x) \end{aligned} \tag{8.18}$$

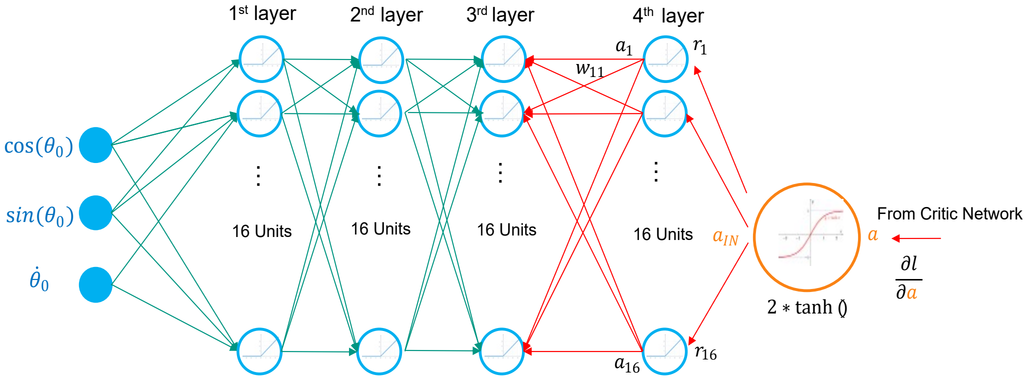


Figure 8.18: The $tanh$ function backward pass

As shown in the Fig (8.18), the input of last unit of *actor network* is a_{IN}, and output is a. The $\partial l/\partial a_{IN}$ is Equ (8.19).

$$\begin{aligned} \frac{\partial l}{\partial a_{IN}} &= \frac{\partial l}{\partial a}\frac{\partial a}{\partial a_{IN}} \\ &= \frac{\partial l}{\partial a}(2 - 2tanh^2(a_{IN})) \end{aligned} \tag{8.19}$$

The a_{IN} is calculated by forward pass when feed the state0 to *actor network*. The $\partial l/\partial a_{IN}$ will continue going back. The *actor network* parameters between $4th$ and $5th$ layer could be calculated the same as did in *critic network*.

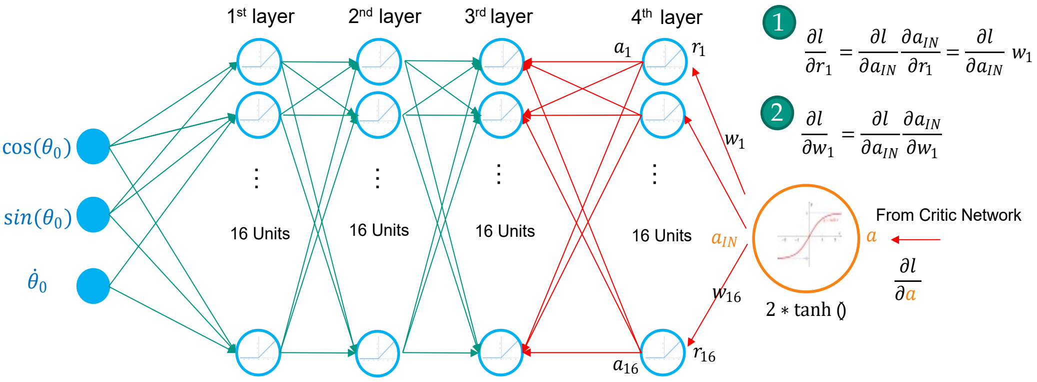

1st layer
2nd layer
3rd layer
4th layer
$\cos(\theta_0)$
$\sin(\theta_0)$
$\dot{\theta}_0$
16 Units
16 Units
16 Units
16 Units
a_1
r_1
w_1
a_{IN}
w_{16}
a_{16}
r_{16}
2 * tanh ()
1
$\frac{\partial l}{\partial r_1} = \frac{\partial l}{\partial a_{IN}} \frac{\partial a_{IN}}{\partial r_1} = \frac{\partial l}{\partial a_{IN}} w_1$
2
$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial a_{IN}} \frac{\partial a_{IN}}{\partial w_1}$
From Critic Network
a
$\frac{\partial l}{\partial a}$

Figure 8.19: Each backpropagation has two tasks, one is to calculate the local gradient, another is to pass the partial loss to the front layer

$$\begin{aligned}\frac{\partial l}{\partial w_1} &= \frac{\partial l}{\partial a_{IN}} \frac{\partial a_{IN}}{\partial w_1} \\ &= \frac{\partial l}{\partial a_{IN}} r_1\end{aligned} \tag{8.20}$$

The r_1 is the output result from the ReLU activation of $4th$ layer of *actor network*. And also, the partial derivative of loss function need to continue to pass backward.

$$\begin{aligned}\frac{\partial l}{\partial r_1} &= \frac{\partial l}{\partial a_{IN}} \frac{\partial a_{IN}}{\partial r_1} \\ &= \frac{\partial l}{\partial a_{IN}} w_1\end{aligned} \tag{8.21}$$

After this part, the later backpropagation is the same as *critic network*, even the network structure is also similar. After the partial derivative is passed through both *critic network* and *actor network*, all the parameters at *actor network* could be updated by Adam as described in last section.

Up to now, the step $14th$ of Alg (8) is finished.

8.5 Soft-replacement for target networks

In the step $15th$ of Alg (8) is to slowly update the *target actor network* and *target critic network*. This method could improve the stability of training process. The implementation method is easy:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \tag{8.22}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \tag{8.23}$$

The τ is 0.001, means the target networks parameter $\theta^{Q'}$ and $\theta^{\mu'}$ will remain fairly static.

8.6 Pendulum solution description

The pendulum experiment is done with a Linux based PC, and HighFlex2. The pendulum on PC is implemented by Keras-RL [125]. To ensure the same environment for Zynq version test, the hardware version also uses the same pendulum environment by Keras-RL on PC. The *critic network*, *actor network*, *target critic network* and *target actor network* is located on the HighFlex2. The HighFlex2 communicate with the pendulum environment through ethernet. Two functions, *forward()* and *backward()* are implemented on HighFlex2, to replace their counterparts inside Keras-RL. The *forward()* is used for policy network, say, the *actor network* for action selecting. This is described in the previous *actor network* build section. The *backward()* represents the training process and is called after each step. This is described in the previous train *actor network* and train *critic network* sections.

Algorithm 9 Modified DDPG Hardware Implementation

```
Randomly initialize critic network $Q(s,a \mid \theta^Q)$ and actor network $\mu(s \mid \theta^\mu)$ with weights $\theta^Q$
and $\theta^\mu$
Initialize target networks $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize reply bufer $R$
for episode=1,M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for $t = 1, T$ do
        Select action $a_t = \mu(s_t \mid \theta^\mu)$ from HighFlex2 forward()
        Apply the exploration noise on the action $a_t = \mu(s_t \mid \theta^\mu) + \mathcal{N}_t$
        Execute action $a_t$ with Keras-RL
        Observe reward $r_t$ and observe new state $s_{t+1}$ from Keras-RL
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$ in Keras-RL
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$ in Keras-RL
        num_steps = num_steps + 1
        if total num_steps > warmup_steps then
            Transmit these minibatch from PC to HighFlex2 backward()
            repeat
                Use the $(s_t, a_t, r_t, s_{t+1})$ to calculate the Q value and Target Q value
                Update critic by minimizing the loss
                $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i \mid \theta^Q))^2$
                Backpropagation, and Update the critic network and its target network
                Use the $(s_t)$ to calculate the Q value
                Update the actor policy using the sampled policy gradient:
                $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a \mid \theta^Q) \mid_{s=s_i,a=\mu(s_i)} \nabla\theta^\mu \mu(s \mid \theta^\mu) \mid_{s_i}$
                Backpropagation, and Update the actor network and its target network
            until Trained the whole batch of transition
        end if
    end for
end for
return $\theta$;
```

The detailed hardware implementation flow for DDPG and its testbench is shown in Alg (9). It is the modified version based on Alg (8). The Alg (9) demonstrated the detail interactive mode between Keras-RL and HighFlex2. From step 4 to step 23 is the interaction stage between agent and environment. The first part is from step 8 to step 14, which denotes how the agent in HighFlex2 interact with the environment in the Keras-RL PC. The environment has the current state s_t, and transmit this information to HighFlex2. The HighFlex2 function *forward()* works like an Echo server, which response to the user client request, then transfer the *actor network* output a_t back to Keras-RL PC. The environment Pendulum on OpenAI will response to this action a_t, then transfer from state s_t to next state s_{t+1}, meanwhile, environment will generate one scalar reward r_t. This transition (s_t, a_t, r_t, s_{t+1}) will be stored in Keras-RL at step 13.

The algorithm will first judge if the memory buffer has enough data for training or not. Imagine if the agent only moves 18 steps, then it could not provide a 32 batch of transition data for training. The warmup steps are set to 100, in this case, to provide the memory buffer with a little training dataset. In step 16, the data set is transferred from Keras-RL to HighFlex2 *backward()* function. From step 17 to step 24, the algorithm loops over the training data set in the *backward()* function of HighFlex2. The batch size could be the number ranging from 1 to 32. The training procedure from step 18 to step 20 is for updating *critic network* and *target critic network*. The 21th step to 23 step is the process for updating *actor network* and *target actor network*.

This testbench is also used for Chapter 9 beam dynamic simulation. The only difference is to replace the environment related signals: state, action, and reward. This could also can use for other DDPG application testbench.

8.7 Training curve of DDPG

The parameter M at step 4 of Alg (9) is set to 250, where denotes the maximum episode the pendulum could run. The following two figures shows the pure software version on GPU and hardware version on HighFlex2 Alg (9) version. Before to give out the reward trend, the reward signal needs to explain first. As mentioned in the problem description, the purpose of the pendulum problem is to keep the pole standing as long as possible, and with the least effort.

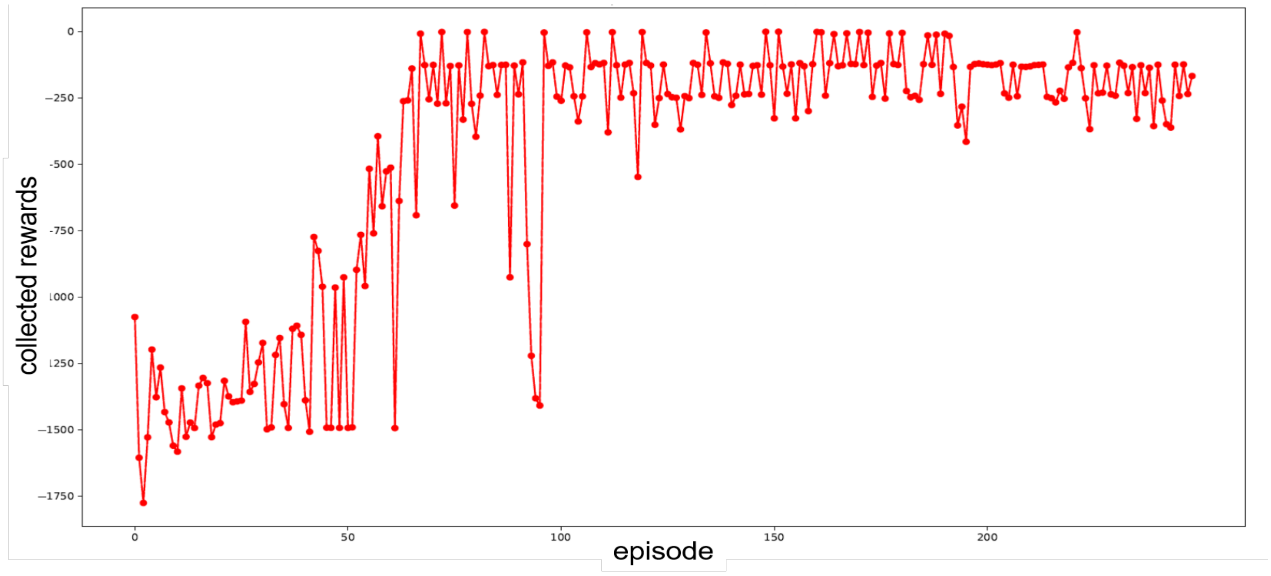


Figure 8.20: The pure software version (Keras-RL)

$$r = -(\theta^2 + 0.1 * \dot{\theta}^2 + 0.001 * a^2) \tag{8.24}$$

The theta is the angle from the vertical direction to the pole clockwise. The theta is range from -π to π. The absolute value of theta is used in Equ (8.24) to calculate the immediate scalar reward during each step. From this reward equation, one could deduce that the position of the pole has a significant influence on the value of the reward. If the pole falls down, then the reward will be a relatively large negative value. If the pole moves fast, then the accelerated velocity $\dot{\theta}$ will be also large, resulting in a large negative reward. Also, the small effort means the a^2 should be small, this force the agent to learn a policy that moves the pendulum to the balance point as fast as possible. Thus, when the reward is close to 0 is an ideal situation. The episodic accumulated reward is the sum of the immediate step reward that the agent collected during this episode. If the scalar immediate reward in each step is a small negative value, then the total accumulated reward will also have a relatively small absolute value (because the reward is always a negative value).

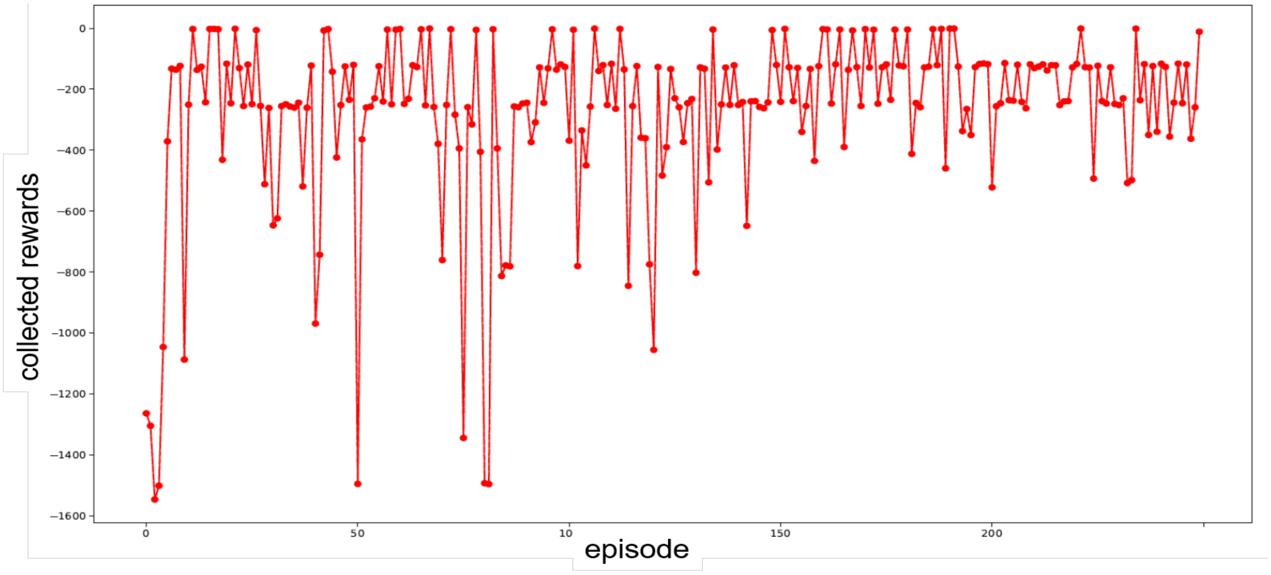


Figure 8.21: The hardware version (HighFlex2 and Keras-RL)

Fig (8.20) is the learning curve of the DDPG agent for the pendulum problem. The episodic accumulated reward is trying to climb from rough -1750 to -200. This means the agent has learned the skill to keep the pole to be balanced (from the simulation video). The same trend could see from the hardware version Fig (8.21), where the four networks running on hardware takes the agent's responsibility to solve the pendulum in the same way with Keras-RL.

8.8 DDPG training performance

When the agent memory buffer has collected enough transitions, the agent could go into the training process according to the step 15 to 25 in Alg (9). The test bench on CPU/GPU and Zynq will study the training latency on this part.

The DDPG training process is continuously did 50 times to compare the training latency. The Fig (8.22) shows the detail training latency for DDPG during 50 times. Each time take one transition data. The GPU have the worst performance, where have a mean latency 6522 µs, and the largest standard error 108. The CPU and Zynq have also dramatic difference in mean and standard error (SE) comparison. Zynq have lowest training latency with only 299.44 µs and CPU have 1625 µs. Zynq is much more stable that have only 0.1343 µs SE but on the other hand, CPU have around 16 µs SE.

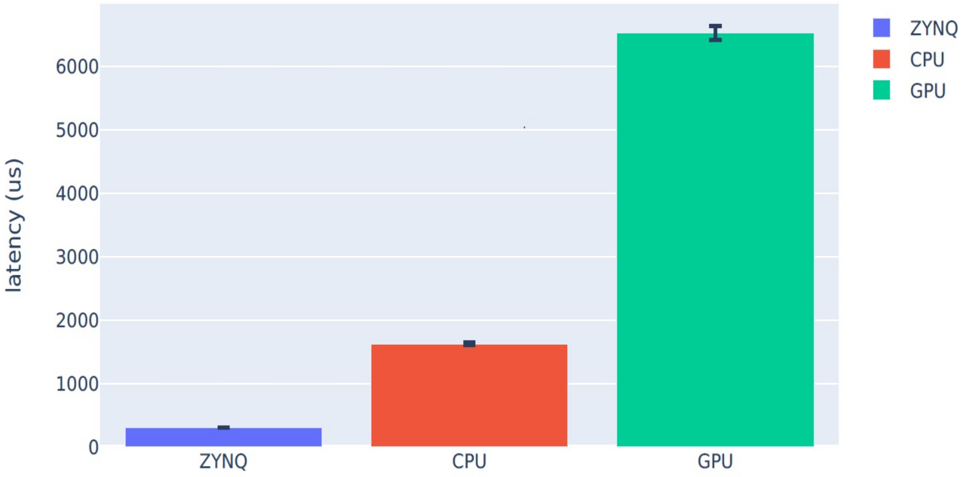


Figure 8.22: The comparison between CPU GPU and Zynq

8.9 Conclusion

Reinforcement Learning (RL) focuses on optimal control under Markov Decision Processes. Due to this property, the training process of the RL algorithm needs to acquire data from the environment at each step continuously. For some physics experiments, this interaction behaviour needs to be completed in a short time slot, thus a low-latency reinforcement learning training platform is demanded. This chapter gives a complete hardware-software implementation solution on HighFlex2 for the DDPG RL training process. The derivation formula and method in the article can also be used as a guide for other hardware platform DDPG based applications. The software framework is implemented in C on the HighFlex2, and a testbench is provided to verify the framework behaviour. The performance is demonstrated and compared with the standard Keras-RL implementation on CPU/GPU. The result shows a dramatical improvement of the performance on the HighFlex2 RL. For RL inference, the inference of the neural network could be deployed on the PL side. The next chapter will illustrate this method. If the parameters need to be updated, the HighFlex2 RL could update the parameters in the PL through the bare-metal connection between the processor and FPGA programable logic.

Chapter 9

Reinforcement Learning design at KARA

9.1 Introduction

This chapter focuses on the development of an intelligent data processing algorithm to control the fast beam dynamics at KARA. The complexity of the data processing and the real-time requirement for the fast feedback control imposes the deployment of reinforcement learning on the Zynq.

With the increasing demand for compact, energy- and cost-efficient accelerator systems, in addition to tailored photon emission matched to the often extreme requirements of experiments in physics and photon science, the control systems have to cope with increasing complexity, high sensor data output rates, large data volumes as well as the desire for fast feedbacks and extensive beam control.

Driven by the interaction of short electron bunches with their own emitted coherent synchrotron radiation (CSR), this instability leads to the formation of dynamically changing microstructures within the longitudinal charge distribution of the bunch. This will limit the use of the emitted THz light in experiments. Given its dynamic nature, a *fast* and *adaptive* feedback system is required to establish extensive control over the longitudinal beam dynamics.

Reinforcement learning reduces the effort and complexity of operating a control system, where it may eventually control an accelerator autonomously. [129] illustrates how reinforcement learning can be applied to this task, and also propose a longitudinal feedback loop. An undirected method to probe the longitudinal dynamics of the beam consists of the measurement of the CSR intensity emitted. To detect the CSR emitted at KARA, a dedicated front-end electronics readout card has been developed. KAPTURE-2 is able to sample a CSR pulse with a sampling time down to 3 ps and pulse repetition rates up to 1 GHz. The platform consists of two boards, the KAPTURE-2 front-end electronics [11] that samples the pulse from the accelerator, and high-end HighFlex2 data acquisition board that which readout and process in real-time the data generated by the KAPTURE-2 board. A fast light-weight reinforcement learning framework is developed on HighFlex2. To provide a proof of concept, the Inovesa simulation (act as an environment under RL concept) is connected with HighFlex2 to test the performance of the reinforcement learning algorithm on hardware.

9.2 Karlsruhe Research Accelerator

The Karlsruhe Research Accelerator (KARA) is an electron storage ring and serves as a platform for the development and testing of the new beam and acceleration technologies, pooling research of new accelerator concepts, and development of new detectors. The accelerator's circumference is 110 meters long. The electron energy range from 0.5 GeV to 2.5 GeV. The storage ring can operate with an arbitrary bunch pattern filling scheme. This scheme could range from a single-bunch to a multi-filling scheme up to 184 bunches with a time-space of 2 ns. Around the accelerators there are several detectors deployed, which is KAPTURE [10], KAPTURE2 [11] and KALYPSO [130]. These detectors are useful for a real-time resolved phase-space tomography, which is fundamental for the feedback control loop.

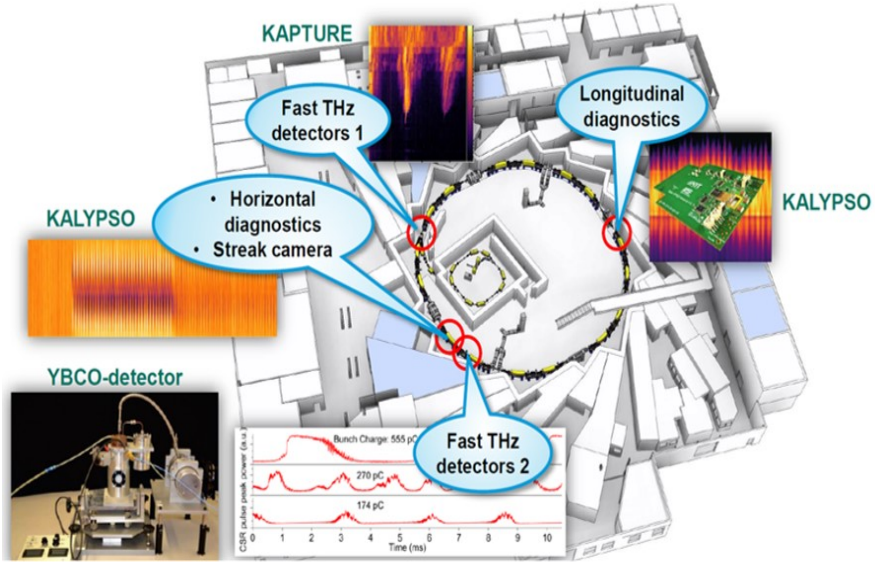


Figure 9.1: The storage ring is equipped with advanced beam diagnostics with fs time resolution

9.3 Micro-bunching instability

The coherent synchrotron radiation (CSR) is produced by electron bunches of a length of few millimetres at the synchrotron light source. At KARA there are two basic operation modes: the low α_c optics, and the normal mode. As shown in Fig (9.2), the normal mode is running in the X-rays frequency domain. The low α_c optics mode is interesting for users because, under this operation mode, it can generate coherent synchrotron radiation in the THz frequency range with high intensity. However, the short bunch mode also leads to microstructures appearing in the bunch, which causes fluctuations in the emitted CSR.

The longitudinal compression of particles within the bunch increases the intensity of coherent emission in the THz frequency range. The *self-interaction* of the bunch with the emitted CSR leads to the formation of micro-structures, which is referred to as *micro-bunching instability*. Due to the self-interaction of the bunch, the micro-bunching instability has a particular threshold current for the stable operation with the constant intensity of CSR emission. If the current exceeds the threshold, then the emitted radiation will change rapidly and persistently, which is shown in Fig (9.4).

Exceeding a certain threshold current, the CSR self-interaction of short electron bunches will result in a dynamically changing longitudinal charge distribution and thus to fluctuating CSR emission (illustrated in Fig. 9.3). This underlying longitudinal dynamics can be simulated

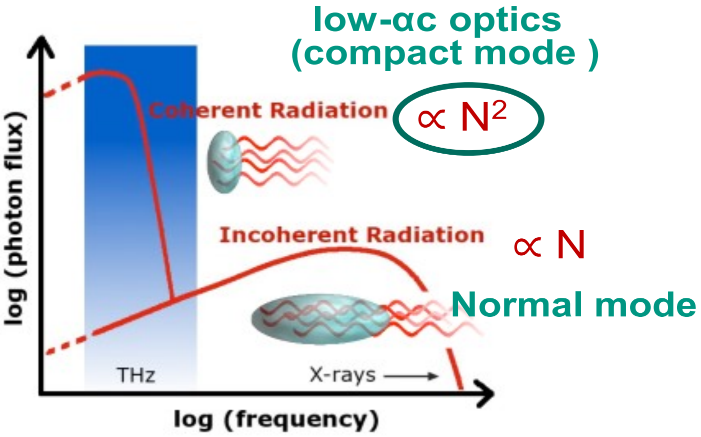


Figure 9.2: In the compact mode, the power is propotional to the square of number of paticles; in normal mode, the power is propotional to the number of paticles.

by Vlasov-Fokker-Planck (VFP) equation [131]. The CSR self-interaction could be described by the CSR wake potential,

$$V_{\mathrm{CSR}}(q) = \int_{-\infty}^{\infty} \widetilde{\rho}(\omega) Z_{\mathrm{CSR}}(\omega) e^{i\omega q} \mathrm{d}\omega \ , \tag{9.1}$$

which can be added as a perturbation to the Hamiltonian. Here, $q = (z - z_s)/\sigma_{z,0}$ denotes the generalized longitudinal position, $\widetilde{\rho}(\omega)$ the Fourier-transformed longitudinal bunch profile and $Z_{\mathrm{CSR}}(\omega)$ the CSR-induced impedance of the storage ring. At the KIT storage ring KARA, such simulations using the VFP solver Inovesa [132] have shown great qualitative agreement with measurements of the emitted CSR power [133]. Therefore the VFP solver Inovesa simulation will be the most significant avenue and reference to verify the control algorithm or machine learning approaches.

The additional potential in Equ (9.1) can be interpreted as a perturbation to the accelerating RF potential, and then cause a perturbation of the synchrotron motion within the bunch. This will lead to the formation of microstructures and their dynamic evolution at time scales that comparable to the *synchrotron period*. This physical property of synchrotron period T_s is the time unit of Fig (9.3). This is the major reason for the necessity of real-time control in the hardware and software design of the feedback loop.

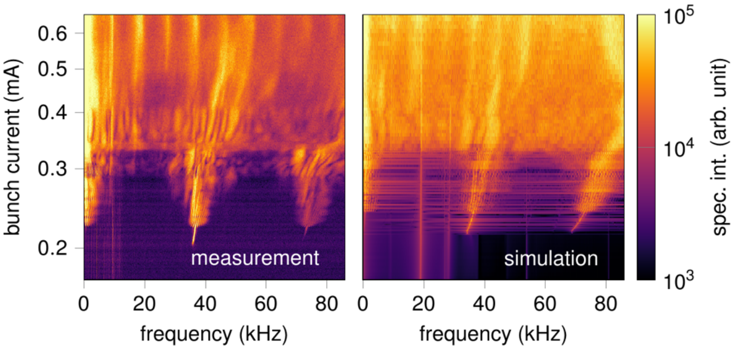
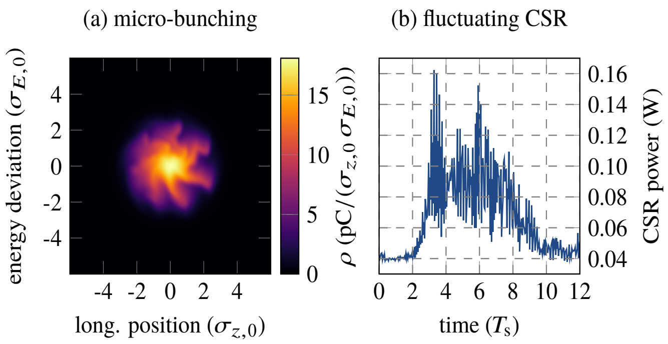


Figure 9.4: A comparison between the real data acquired by KAPTURE and the simulated behaviour by Innovesa. Image from IBPT, courtesy of Tobias Boltz.



(a) micro-bunching

(b) fluctuating CSR

Figure 9.3: (a) The CSR self-interaction of the bunch causes the formation of micro-structures in the longitudinal phase space density. (b) Their continuous variation leads to fluctuations in the emitted CSR power. The illustrated dynamics are simulated with the VFP solver Inovesa. Image from IBPT, courtesy of Tobias Boltz.

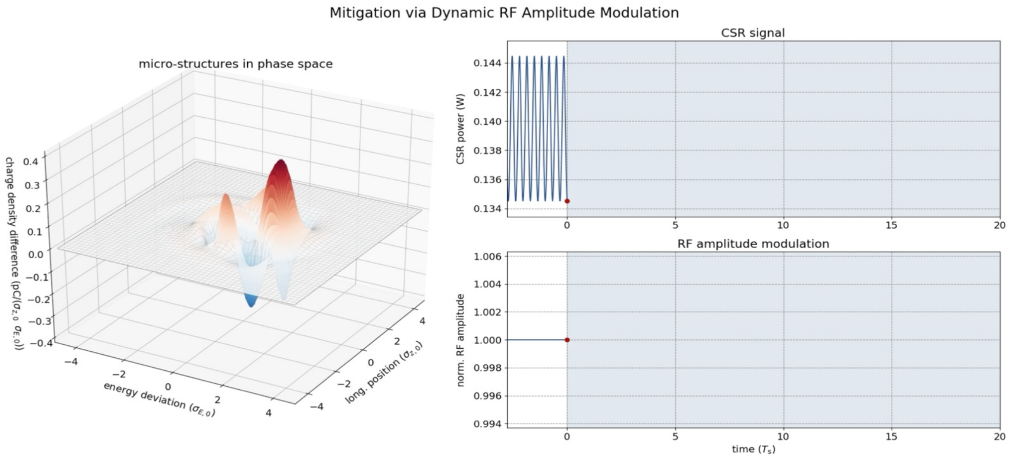


Figure 9.5: The natural behaviour of the instability leads to dynamic micro-structures in phase space (left subgraph) and fluctiating CSR power (right subgraph). Image from IBPT, courtesy of Tobias Boltz.

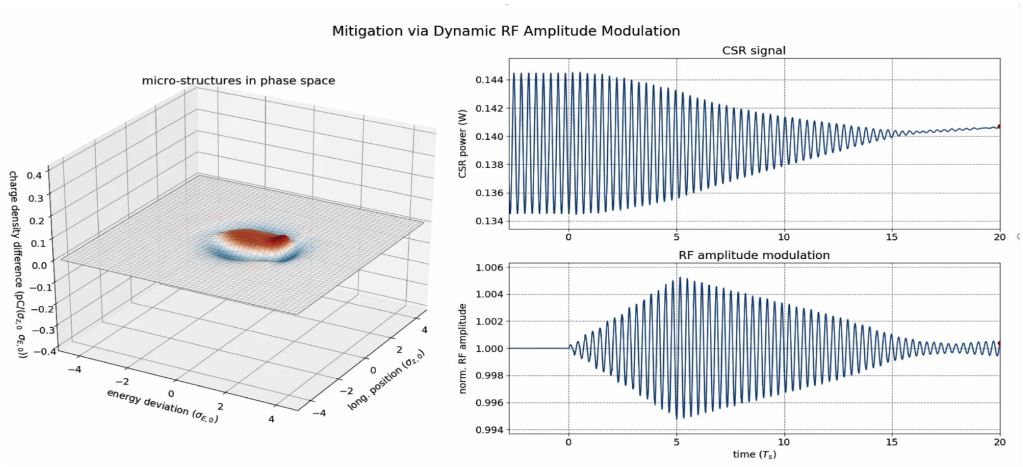


Figure 9.6: This CSR signal is measured by the KAPTURE system, while such manual control will be replaced by the reinforcement learning deployed on the HighFlex2. Image from IBPT, courtesy of Tobias Boltz.

As an extension study of [129] at the Institute for Beam Physics and Technology (IBPT), preliminary simulation with Inovesa showed a possibility of manual control over the CSR signal illustrated as Fig (9.5) and Fig (9.6). Fig (9.5) present that the CSR have a high variance initially. Through proper manual control by using the RF amplitude modulations, the CSR could reach a relatively high average value with low variance as shown at the end-point of the CSR signal in Fig (9.6). The goal is to use reinforcement learning to holds high level of reproducibility of such *ideal* manual control.

9.4 Implementation of feedback loop

In order to apply reinforcement learning (RL) methods to control the micro-bunching instability, the problem should be a Markov Decision Process (MDP) that suitable for reinforcement learning basic requirement. Thus the basic component of RL needs to be defined in the following: the state signal, the action signal and the reward signal.

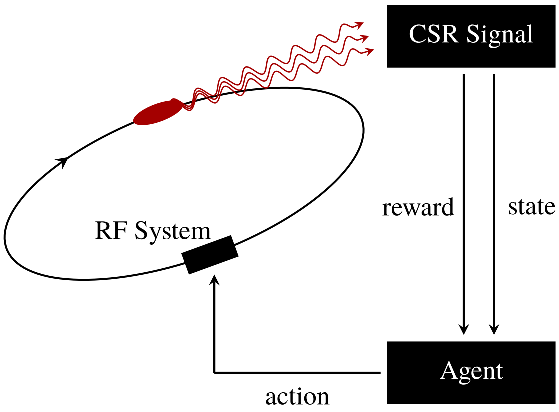


Figure 9.7: General feedback scheme using the CSR power signal to construct both, the state and reward signal of the Markov decision process (MDP). [129]

9.4.1 State signal

Mentioned in [129], under the simulation situation of the longitudinal dynamics by VFP solvers, an initial charge distribution and a set of constant parameters (e.g. machine parameters of the storage ring) will define an initial state. The temporal evolution of this charge distribution will be simulated by iteratively solving the VFP equation. At each time step, the calculation of the next step is entirely based on the charge distribution of the current step. This could make sure the definition of the state signal is fully described by a Markov process, described in Equ (6.2). Thus, the definition of state signal in RL will be a sequence of longitudinal charge distributions:

$$S_t \doteq \psi_t(z, E) \tag{9.2}$$

The Equ (9.2) is the ideal state signal. Although the first efforts towards phase space tomography are made at KARA, the longitudinal bunch profile is measured by an electro-optical near-field setup combined with an ultrafast line-camera, "KAYLPSO" [130]. The system provides a turn-by-turn basis diagnostics measurement with femtosecond time resolution. The horizontal bunch profile is measured by a fast-gated intensified camera or by KALYPSO. The system provides the precise energy spread measurement of each bunch. The fluctuation of emitted CSR power is strongly influenced by the micro-bunching structures, therefore, in reality, the state signals could be indirectly probed by CSR power emitted at the THz spectral range [11]. Two components serve for this purpose: a commercial zero-biased Schotty barrier diode detector and KAPTURE-2. The Schotty barrier diode detector could provide a spectral sensitivity from 50 GHz to 2 THz [134]. The hardware connection is demonstrated at Fig (9.10). This detector is then connected to the fast digitizer board KAPTURE-2. Then finally transferred to HighFlex2 for post processing.

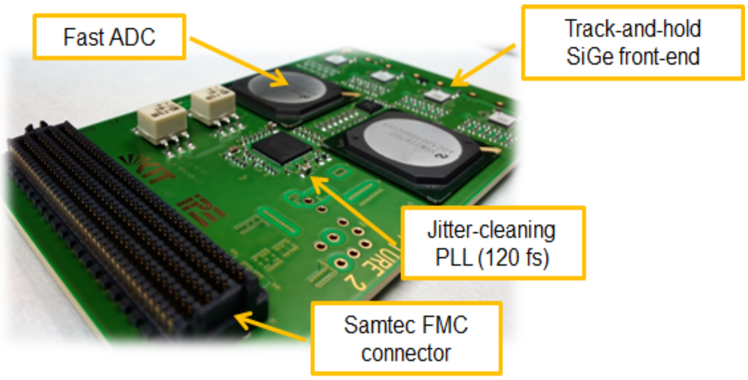


Figure 9.8: KAPTURE-2

KAPTURE (Karlsruhe Pulse Taking Ultra-fast Readout Electronics) [10] has been integrated as a permanent diagnostic device at KARA. KAPTURE-2 [11] is a picosecond sampling system for THz pulses with a high repetition rate (2 ns) in the accelerator. The system is able to acquire and sample the pulse shape with 3 ps resolution by 4-channel simultaneously and continuously with a data rate up to 4 x 1.8 GS/s @ 12 bit per single THz pulse. Two KAPTURE boards can be connected together to increase the number of sampling point to be acquired for each THz pulse. The KAPTURE card is integrated within a heterogeneous FPGA-GPUs framework, which provides a precise reconstruction of the THz pulse and the measurement of both the pulse amplitude (mV) and the peaking time with a picoseconds time resolution. Furthermore, a precise calculation of the fluctuation of both the intensity and the arrival time of the CSR pulse is executed. A real-time FFT and measurement of CSR fluctuation could be provided. This yields a useable observation for the state signal in the RL approach to micro-bunching control.

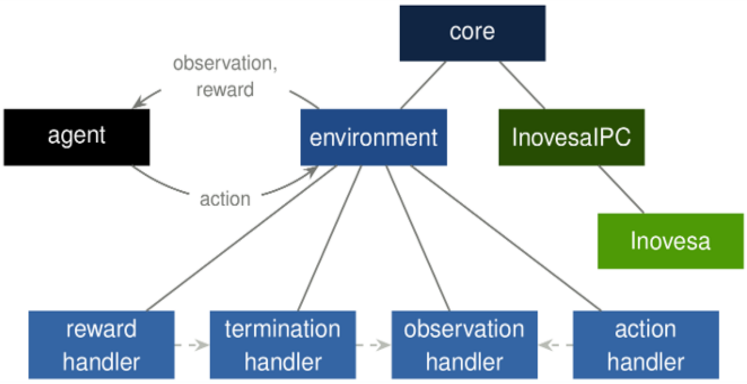


Figure 9.9: InovesaRL structure
courtesy of T. Boltz from IBPT

Under the support of hardware, a hand-crafted feature vector is constructed from the CSR time signal:

$$S_t \doteq (\mu_{t':t}, \sigma_{t':t}, m_{t':t}, a\_f_{main}, f_{main}, \phi\_f_{main}, \delta(theta_{v1}), termination)^{\mathrm{T}} \ , \tag{9.3}$$

To prove the hardware working properly before going to the field, the HighFlex2 RL implementation needs to be verified under the simulation. This simulation environment is called InovesaRL, and the definition of these 8 state signals defined in Equ (9.3) is located at InovesaRL simulation shown in Fig (9.9). The InovesaRL is a reinforcement learning training testbench that uses Inovesa as the environment. The "core" component is the abstraction layer that provides the user with the basic interface of an RL environment. The user could use such a way to define different RL algorithm and interact with the beam dynamics. The reward and action signal used for the verification is also defined in InovesaRL.

9.4.2 Reward signal

To obtain goal-directed learning from the environment, a proper definition of the reward signal is fundamental cause this will lead the agent to learn in the correct direction. As the primary goal is to emit CSR power with high average power but low variance, a reward function has been defined which is based on the time evolution of the CSR power intensity.

$$R_t \doteq R_t(P_{t,\mathrm{CSR}}) \ . \tag{9.4}$$

As the reward function has been defined in order to stabilize the CSR intensity, one possible set up for reward signal could be:

$$R_t \doteq \omega_1 \mu_{t':t} - \omega_2 \sigma_{t':t} \ , \tag{9.5}$$

where $\mu_{t':t}$ and $\sigma_{t':t}$ denote the normalized mean and standard deviation of the time series $P_{t,\mathrm{CSR}}$ in the time interval $[t', t]$, and $\omega_{1,2} > 0$ are weighting factors. The weighting factor is used to adjust the importance of mean and standard deviation. This definition of reward allows having a CSR power signal of high intensity and low fluctuation, which corresponds to a smooth charge distribution that is not significantly changing in time. The reward function could be defined, the Equ (9.5) is one possible definition, which seems to show a reasonable behaviour during the simulations. The simulation setup used for verification of RL in this chapter is to use Equ (9.5).

9.4.3 Action signal

The last signal to be defined is the action. The action signal is the output of the *actor network* which drives the RF cavity. Because the additional CSR wake potential in Equ (9.1) acts as a perturbation to the RF potential, one prominent approach seems to act a modulation centered to the main component of the RF system in order to compensate the CSR perturbation. This compensation should optimize the fast micro-bunching dynamics. Thus, one straightforward choice of the action space would be act on the RF amplitude modulation. The RF amplitude modulation is denoted in:

$$V_{RF} = \hat{V}(t) sin(2\pi f_{RF} t), \tag{9.6}$$

$$\hat{V}(t) = \hat{V}_0 A_{mod} sin(2\pi f_{mod} t + \phi_{mod}), \tag{9.7}$$

$$A_t \doteq (A_{mod}, f_{mod})^{\mathrm{T}} \ , \tag{9.8}$$

where V_{RF} denotes the RF amplitude and $\hat{V}(t)$ is the RF amplitude modulation. The A_{mod} is the amplitude of an RF amplitude modulation and f_{mod} the modulation frequency of the amplitude modulation. Dynamically modifying these two parameters should provide the agent with a substantial amount of control over the RF system. This impact on the micro-bunching dynamics has also been verified experimentally at [135, 136]. Both parameters are sent to a commercial signal processor (model iGp12-720F by Dimtel, Inc. [137]).

9.4.4 Algorithm choosen

In this paragraph, the selection of the reinforcement algorithm to control the micro-bunching instability by interacting with the state and reward signal to generate proper actions is discussed. The pendulum balance problem in the previous chapter has a similar structure insofar as the agent needs to generate proper actions to keep the episode running. Deep Deterministic Policy Gradient is the first attempt and many other method are also in discussion like Policy Gradient, Proximal Policy Optimization (PPO) [80], trust Region Policy Optimization (TRPO) [138], etc.

9.5 Hardware feedback loop

A detailed description of the General feedback shown in Fig (9.7) is depicted in Fig (9.10). The front-end electronics KAPTURE2 will sample the CSR signal, then the information will be sent to HighFlex2 DAQ. The HighFlex2 will extract several important features from the acquired CSR pulses, like: the mean value and the standard deviation of the CSR intensity, the mean value and the standard deviation of the arrive time with a resolution of picoseconds and many others. The reinforcement learning is running on HighFlex2, and will generate appropriate action signal (RF modulation) to the commercial model iGp12-720F for bunch-by-bunch (BBB) feedback, where it finally reaches a small RF kicker cavity.

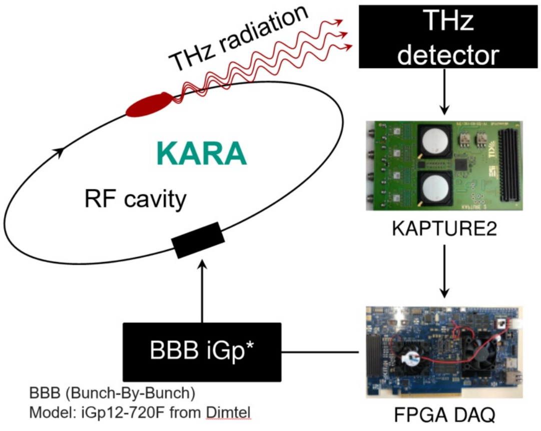


Figure 9.10: Implementation of the feedback loop to the RF cavity of KARA.

9.6 Simulation result of training

Reinforcement learning (RL) training convergence could indicate a successful RL implementation and will be of prime importance to the functionality correctly implementation. To have a proof of concept, the RL algorithm (DDPG training part on HighFlex2) is connected with Inovesa simulation (on a standard PC) directly by Ethernet. The HighFlex2 acts as a server to respond to the demand from the environment (Inovesa Beam Dynamics) shown in Fig (9.11). In principle, the inference and training should be located at PL and PS respectively in the field. Because the purpose is to prove the training part, the inference part could be implemented in the PS part. The detailed implementation methods and their corresponding codes have been elaborated in Chapter 8. The implementation to solve pendulum and to solve the beam dynamics have a high degree of consistency. The only difference is the input size of the state, the action space, and the definition of reward. The internal principle of RL training working the same.

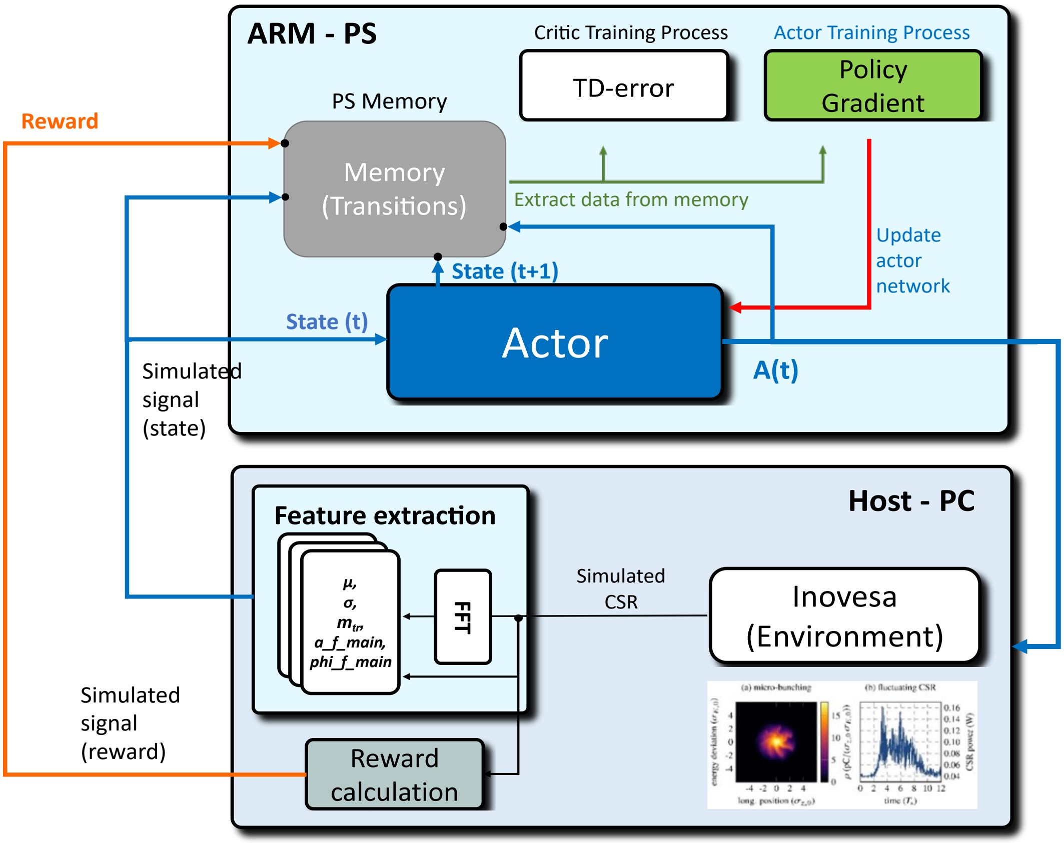


Figure 9.11: The setup of proof of concept with Inovesa simulation. The feature extraction and calculation of reward are processed in the PC. Only the RL part (actor and critic) is implemented at HighFlex2 ARM processor.

The networks used for KARA micro-bunching instability control are remarkably similar to those networks used at Chapter 8 Pendulum control. Two of the major signal is required to modify because the Pendulum and CSR state is different.

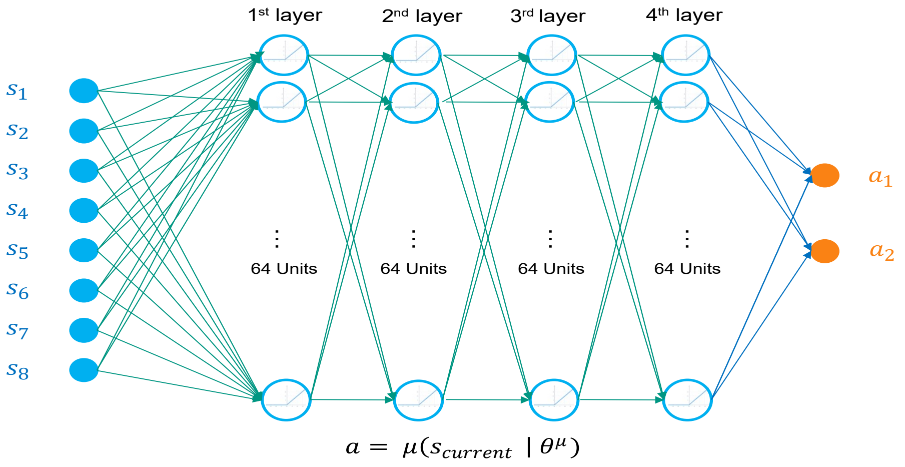


Figure 9.12: The DDPG *actor network* under Inovesa simulation

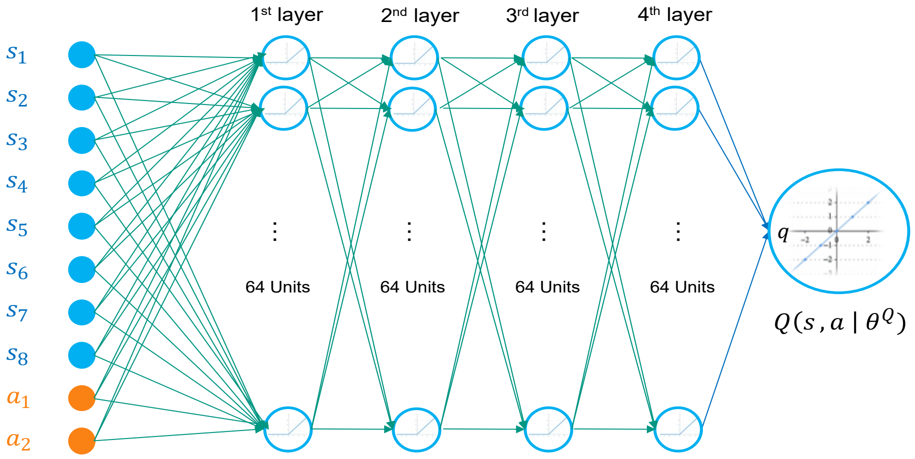


Figure 9.13: The DDPG *critic network* under Inovesa simulation

The *actor network* in DDPG will use these 8 state signal to predict the action signal in Fig (9.12). Action space is a two-dimensional vector as delivered at Equ (9.8)

Where in Equ (9.8), the A_{mod} is the amplitude of RF modulation and f_{mod} is the RF modulation frequency. The RF phase remains untouched. This is the yellow point of Fig (9.12) (*actor network* output) and Fig (9.13) (*critic network* input). The *actor network* has four fully-connected dense layers and each layer have activation ReLU. The final unit is a linear activation. Each layer has 64 units.

The *critic network* in DDPG receives the actions and states, to predict the Q value of this (a, s) pair as shown in Fig (9.13). The *critic network* has four fully-connected dense layers and each layer have activation ReLU. Each layer has 64 units. The final unit is a linear activation. The target networks (*target critic network* and *target actor network*), have the same structure as its counterpart.

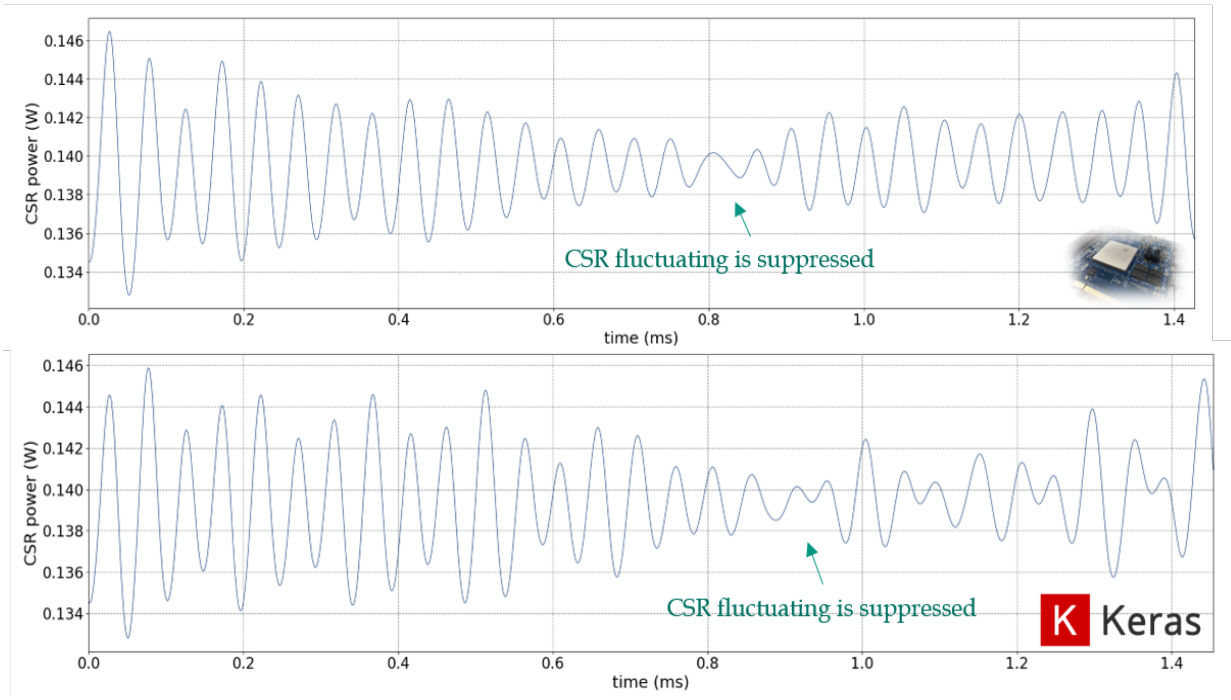


Figure 9.14: The CSR signal result after HighFlex2 (top) and the Keras-RL (bottom) interacts with Inovesa simulation.

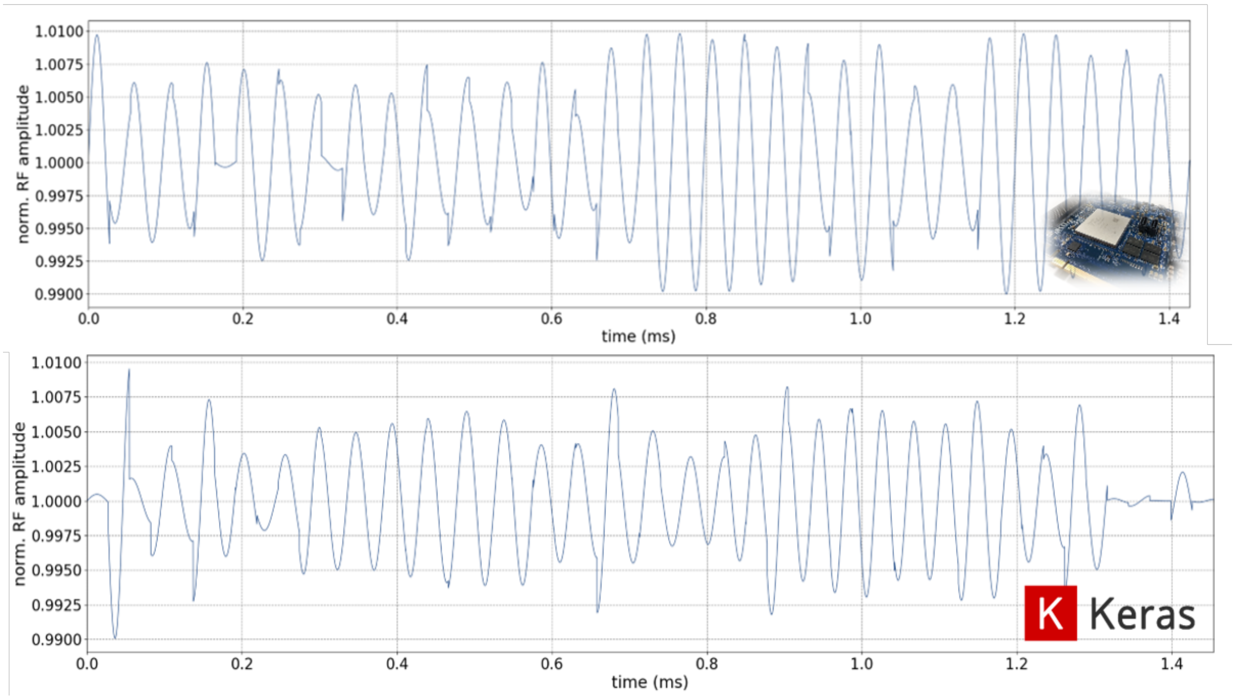


Figure 9.15: The RF amplitude modulation result after HighFlex2 (top) and the Keras-RL (bottom) interacts with Inovesa simulation.

As defined in Chapter 6, one episode consists of many steps. One episode could also be called a complete chain or an attempt at the interaction between agent and environment. It is a series of steps. As the result of the simulated RF modulation, the time behaviour of one episode of the CSR dynamics changes is demonstrated in the plots of Fig (9.14). The episode is terminated when the fluctuation of the CSR reaches a certain programmable threshold as shown in both software version and hardware version above 1.4 ms. More than 3500 of such episodes have been simulated with Inovesa. Fig (9.14) illustrates one of those. The comparison of the hardware implementation and Keras on GPU in Fig (9.14) shows a similar behaviour and they suppress the CSR fluctuation at around 0.8 ms and 0.9 ms respectively.

Fig (9.15) shows the comparison between the amplitude modulation generated by the Actor on HighFlex2 and on GPU. The hardware implementation generates the similar action compared to Keras.

9.7 Hardware training performance

A performance test is implemented according to the steps 15 to 25 in Alg (9). The test bench on CPU/GPU and Zynq will study the training latency on this part similar to the RL chapter above.

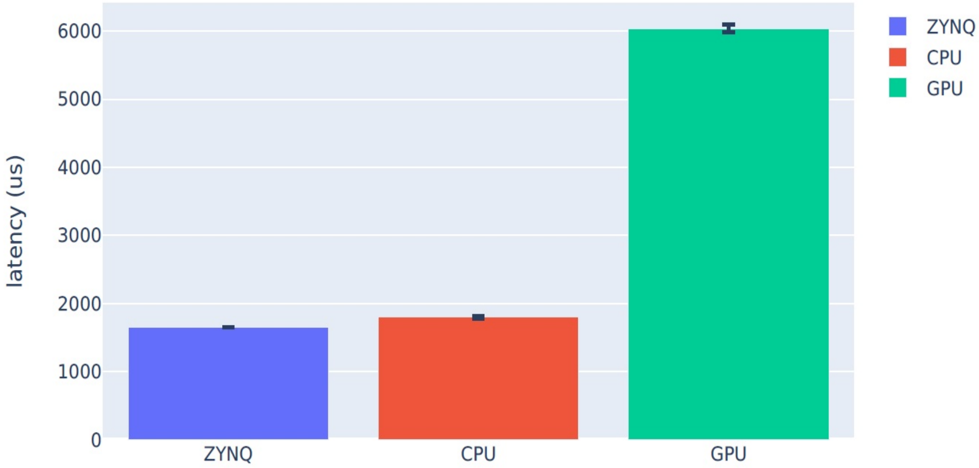
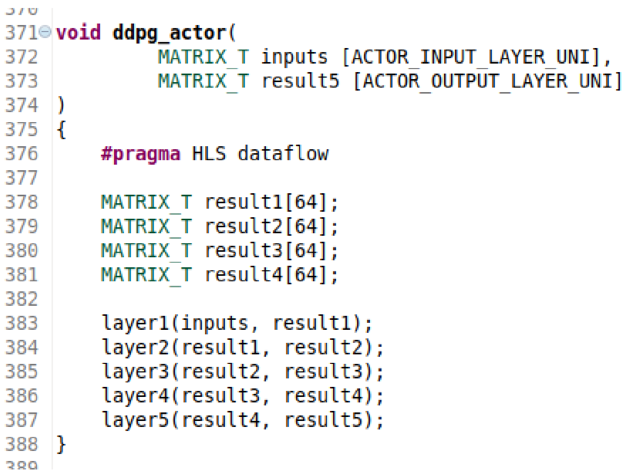


Figure 9.16: The comparison between CPU GPU and Zynq

The DDPG training process for KARA is the same as the pendulum problem testbench. Each training process is continuously done 50 times to compare the training latency. Fig (9.16) shows the detail training latency for DDPG during 50 times for KARA configuration. The GPU have the worst performance, where have a mean latency of 6037 µs, and the largest standard error 55 µs. The CPU and Zynq have also a dramatic difference in mean and standard error (SE) comparison. Zynq have the lowest training latency with only 1648 µs and the CPU have rough 1800 µs. Zynq have only 0.1343 µs SE but on the other hand, the CPU have around 16 µs SE. This SE means the Zynq could keep a much more stable training time.

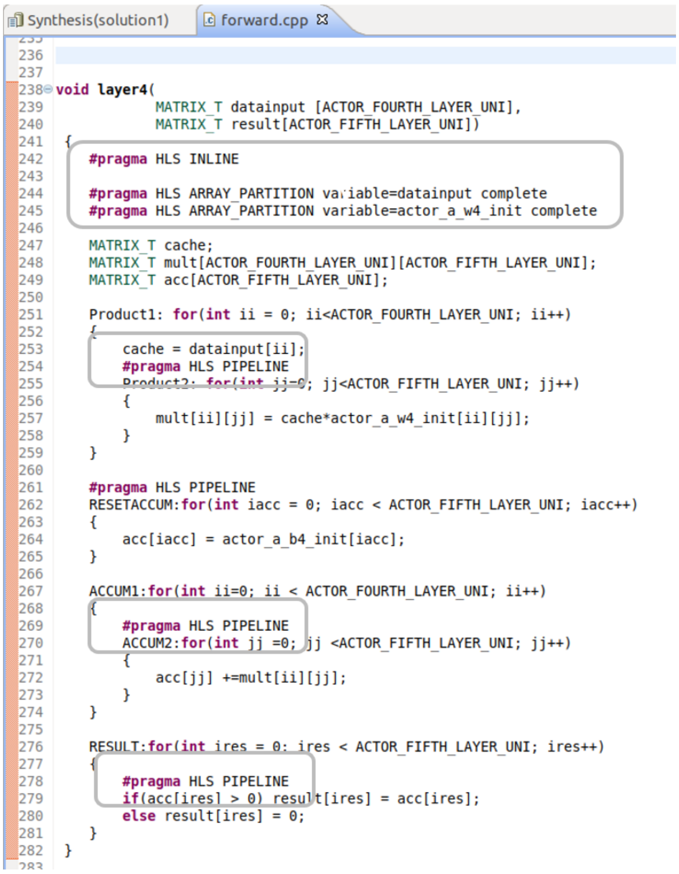
9.8 DDPG inference at PL

The inference of DDPG denotes only the *actor network* implementation because it is the neural network that used to generate the control action. To enable the low-latency of inference and fast development time, thesis decides to use Vivado HLS to implement the *actor network*.



```
void ddpg_actor(
        MATRIX_T inputs [ACTOR_INPUT_LAYER_UNI],
        MATRIX_T result5 [ACTOR_OUTPUT_LAYER_UNI]
)
{
    #pragma HLS dataflow

    MATRIX_T result1[64];
    MATRIX_T result2[64];
    MATRIX_T result3[64];
    MATRIX_T result4[64];

    layer1(inputs, result1);
    layer2(result1, result2);
    layer3(result2, result3);
    layer4(result3, result4);
    layer5(result4, result5);
}
```

Figure 9.17: The developed ddpg *actor network* forward pass top function

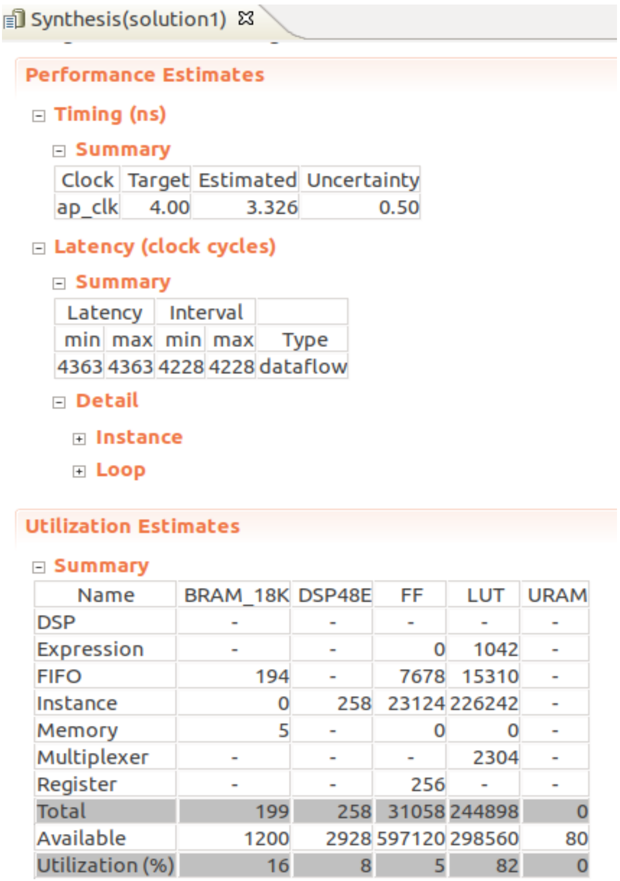


```
void layer4(
            MATRIX_T datainput [ACTOR_FOURTH_LAYER_UNI],
            MATRIX_T result[ACTOR_FIFTH_LAYER_UNI])
 {
    #pragma HLS INLINE

    #pragma HLS ARRAY_PARTITION variable=datainput complete
    #pragma HLS ARRAY_PARTITION variable=actor_a_w4_init complete

    MATRIX_T cache;
    MATRIX_T mult[ACTOR_FOURTH_LAYER_UNI][ACTOR_FIFTH_LAYER_UNI];
    MATRIX_T acc[ACTOR_FIFTH_LAYER_UNI];

    Product1: for(int ii = 0; ii<ACTOR_FOURTH_LAYER_UNI; ii++)
    {
        cache = datainput[ii];
        #pragma HLS PIPELINE
        Product2: for(int jj=0; jj<ACTOR_FIFTH_LAYER_UNI; jj++)
        {
            mult[ii][jj] = cache*actor_a_w4_init[ii][jj];
        }
    }

    #pragma HLS PIPELINE
    RESETACCUM:for(int iacc = 0; iacc < ACTOR_FIFTH_LAYER_UNI; iacc++)
    {
        acc[iacc] = actor_a_b4_init[iacc];
    }

    ACCUM1:for(int ii=0; ii < ACTOR_FOURTH_LAYER_UNI; ii++)
    {
        #pragma HLS PIPELINE
        ACCUM2:for(int jj =0; jj <ACTOR_FIFTH_LAYER_UNI; jj++)
        {
            acc[jj] +=mult[ii][jj];
        }
    }

    RESULT:for(int ires = 0; ires < ACTOR_FIFTH_LAYER_UNI; ires++)
    {
        #pragma HLS PIPELINE
        if(acc[ires] > 0) result[ires] = acc[ires];
        else result[ires] = 0;
    }
 }
```

Figure 9.18: The layer4 of *actor network* forward pass function

As shown in Fig (9.17), it clearly define the 5 layers' structure. All layers have a similar function. Take the layer4 as an example that shown in Fig (9.18). The forward pass normally maps the input vector to the output. Inside which the input vector is doing matrix multiplication and interact with weights and biases. This has original parallelism. The Fig (9.18) shows the usage of HLS hardware optimization directive "pragma pipeline" to enable the pipeline stage. And pipeline will automatically unroll the for loop.

Fig (9.19) demonstrates the result of ddpg_actor function. The interval for the next data coming is 4233 clock cycle at 4 ns clock period. Therefore if combine performance testbench



Synthesis(solution1)

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	3.326	0.50

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
4363	4363	4228	4228	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1042	-
FIFO	194	-	7678	15310	-
Instance	0	258	23124	226242	-
Memory	5	-	0	0	-
Multiplexer	-	-	-	2304	-
Register	-	-	256	-	-
Total	199	258	31058	244898	0
Available	1200	2928	597120	298560	80
Utilization (%)	16	8	5	82	0

Figure 9.19: The *actor network* HLS synthesis result.

that uses the forward pass for *actor network* to test 50 times, a comparison between three platforms is illustrated in Fig (9.20).

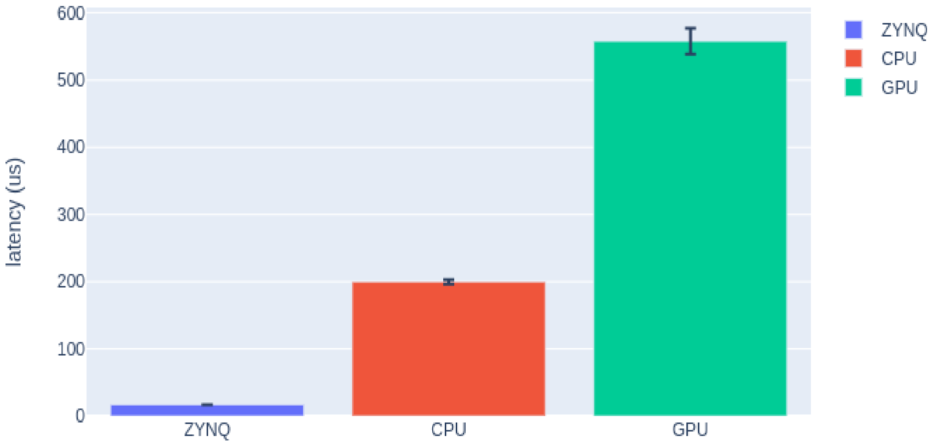


Figure 9.20: The comparison between CPU GPU and Zynq.

The Fig (9.20) concludes the last inference training performance. The GPU inference time have about 557 µs with 19 µs SE. And CPU inference time owns around 200 µs with 3.52 µs SE. The FPGA inference time is done by Vivado HLS, with a fixed latency of 4233 clock cycles means Zynq inference have 16.932 µs and no SE. The PL part inference shows an extremely low latency compared with CPU/GPU. This method to share the training task and inference work between PS and PL, that take a thorough usage of the Zynq device for RL.

9.9 Conclusion

Driven by CSR self-interaction, the micro-bunching instability at storage rings is caused by a fast and dynamic perturbation that depends on the longitudinal charge distribution. The goal is to stabilize the coherent synchrotron radiation emitted in the THz range by controlling longitudinal phase-space dynamics of the beam. To establish extensive control over dynamics, a reinforcement learning (RL) based feedback loop is proposed and developed that can react to small changes in the charge distribution and adjust the RF system accordingly. In this chapter, two major results are delivered: the overall closed-loop hardware design for the KARA experimental station, and the performance on physics simulation. As the closed-loop design, in detail, the fast THz beam intensity is measured by new KAPTURE-2 front-end electronics. To validate the concept, the full simulation of the physics of the beam which includes fast dynamics of the THz emissions has been set up by InovesaRL and interacts with reinforcement learning on HighFlex2. The HighFlex2 (RL agent) reacts to mimic the micro-bunching instability through Inovesa. The reinforcement learning implementation on HighFlex2 has been compared with Keras-RL on CPU/GPU. Both the Zynq RL implementation and the CPU/GPU RL implementation show a similar prediction of the behaviour and the same trend of reward collection, which confirms that the HighFlex2 RL is working as expected. Furthermore, the training performance has improved in both training and inference when compared to the standard CPU version. At the same time, a relatively fixed latency is guaranteed in training and fixed latency in inference which ensures a safety feedback training time for a physics experiment.

Chapter 10

Conclusion

The data acquisition system and data processing approach is critical for the success of large-scale experiments. The increasing data volume and complex task ask for novel hardware and software implementation. Two core contributions of this thesis are the design and evaluation of the data acquisition system HighFlex2, and the machine learning implementation capability on HighFlex2. Together, they make the HighFlex2 a multifunctional intelligent data acquisition board.

High-throughput data acquisition system

High-throughput is the major feature of the data acquisition system for a large-scale experiment. For this requirement, the main processing core is Xilinx Zynq UltraScale+ ZU11EG. The processing system and programmable logic on ZU11EG MPSoC provide the pluripotential for different task and data movement capability. The ZU11EG is hard-wired with 16 lanes PCIe gen3/4, 12 lanes FireFly duplex connection, and VITA 57.4 FMC+ standard connector. Thus the new DAQ has multiple connections and topologies possibility with other ATCA, AMC and uTCA based DAQ systems and front-end electronics. All the properties enable HighFlex2 to become the ideal hardware platform for the implementation of both traditional data acquisition and machine learning approach.

Webserver and EPICS

The Webserver and EPICS application is built through the Yocto Project. The users could access HighFlex2 programmable logic easily through a web page or EPICS. This makes the FPGA part much more open and friendly to non-hardware experts. By Yocto Project, the same method could be applied to the HDL logic related to front-end electronics. The important and critical signal could directly be accessed and monitored by the processing system. Therefore, it is also very helpful for hardware engineering developers because a more transparent way of controlling and debugging can be developed. The machine learning implementation benefited from the cooperative work between PS and PL.

Supervised Learning Implementation

The thesis provides an example design by Xilinx DPU. This method has the acceleration components for the commonly used operations like convolution and deconvolution, pooling, ReLU and softmax activation layers, and so on. The DPU is located at PL side and PS in used for coordinating data transfering and control. With DPU on HighFlex2, plenty of convolutional neural networks could be depolyed on HighFlex2, for example, VGG, ResNet, GoogLeNet, etc. The Zynq UltraScale+ device supports more aggressive convolution architectures than the Zynq 7000 series. For example more DPU cores, 4096 operations per clock, high RAM usage, etc. Using the command line under the embedded Linux environment to manipulate the CNN on HighFlex2 offers the machine learning engineer a quick model transfer from the CPU/GPU. From the experience of SL development at IPE, ARM is also a powerful and flexible computation node. It has enough abilities to implement the forward pass of some of the small size of neural network or convolutional neural network, LSTM, graphic neural network, etc. Therefore, a future work will be a thorough investigation around comparison between CPU/GPU, FPGA (PL), and ARM (PS). A detailed suggestion for careful hardware selection should be given in the following work.

Reinforcement Learning Implementation

Because many applications require an online-training reinforcement learning (RL) on the scene, this thesis proposes one solution to implement the training process of RL on HighFlex2 which is close to the data source. The policy gradient and deep deterministic policy gradient is verified and tested. One of the major tasks during these two implementations is the mathematical derivation of the training process. Among the training process of reinforcement learning, the most significant part is the certification of capability of the backpropagation on PS part of Zynq on HighFlex2. This provides the hardware engineers with a theoretical reference to deploy their own reinforcement learning for different algorithms and applications. Some components like forwarding pass, backpropagation, optimizer by gradient descent or Adam, can also be used for lightweight supervised learning training process. The functions are implemented with both C and C++, thus they could be easily transferred to another CPU/GPU/ARM-based hardware platform. Through the reference design, the FPGA-based RL training process also should be considered.

One could discover a complementary property between reinforcement learning (RL) and supervised learning (SL). Because the implementation of inference and training has a lot of similarity between RL and SL. The inference part of SL and RL normally is the forward pass of one kind of neural network, and what network structure in chosen depends on the signal the application is dealing with. Thus, the implementation of inference of RL could reuse the existing technology used at SL. For example, if a trained RL model uses a convolutional neural network (CNN) for the actor of Deep Deterministic Policy Gradient, then the aforementioned Xilinx DPU could be deployed at Programmable Logic (PL) part for the RL inference.

Bibliography

[1] Tom Shanley.x86 Instruction Set Architecture. MindShare press, 2010.

[2] Juan Jose Rodriguez Andina, Eduardo De la Torre Arnanz, and Maria Dolores Valdes. *FPGAs: Fundamentals, Advanced Features, and Applications in Industrial Electronics 1st Edition.* CRC Press, 2017.

[3] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. General-purpose graphics processor architectures. *Synthesis Lectures on Computer Architecture*, 13(2):1–140, 2018.

[4] Stephen Bo Furber. *ARM System-on-Chip Architecture.* pearson Education, 2000.

[5] T Odajima, Y Kodama, M Tsuji, M Matsuda, Y Maruyama, and M Sato. Preliminary performance evaluation of the Fujitsu A64FX using HPC applications. *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 523–530.

[6] Ralph Hintemann. Energy consumption of data centers worldwide. `http://ceur-ws.org/Vol-2382/ICT4S2019_paper_16.pdf` Accessed October 29, 2020.

[7] Vivian Delplace, Pierre Manneback, Frédéric Pinel, Sébastien Varrette, and Pascal Bouvry. Comparing the performance and power usage of GPU and ARM clusters for map-reduce. In *2013 International Conference on Cloud and Green Computing*, pages 199–200. IEEE, 2013.

[8] BERTEN. GPU vs FPGA performance comparison. `http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf` Accessed October 29, 2020.

[9] Meghana Patil, M Caselle, E Bründermann, Stefan Funkner, Benjamin Kehrer, G Niehues, W Wang, A S Müller, M Weber, D Makowski, et al. An Ultra-Fast and Wide-Spectrum Linear Array Detector for High Repetition Rate and Pulsed Experiments In *10th International Particle Accelerator Conf.(IPAC'19), Melbourne, Australia*, 2019.

[10] M Caselle, M Brosi, S Chilingaryan, T Dritschler, N Hiller, V Judin, A Kopmann, A.-S. Müller, J Raasch, L Rota, et al. A Picosecond Sampling Electronic "KAPTURE" for Terahertz Synchrotron Radiation, *Proceedings of IBIC2014, Monterey, CA, USA*, 2014.

[11] M Caselle, LE Ardila Perez, M Balzer, A Kopmann, L Rota, M Weber, M Brosi, J Steinmann, E Bründermann, and A.-S. Müller. The International School for Advanced Studies (SISSA), find out more
KAPTURE-2. A picosecond sampling system for individual THz pulses with high repetition rate. *Journal of Instrumentation*, 12(01):C01040, 2017.

[12] Xilinx. Zynq Ultrascale+ MPSoC data sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf Accessed June 29, 2020.

[13] M Caselle, LE Ardila Perez, M Balzer, T Dritschler, A Kopmann, H Mohr, L Rota, M Vogelgesang, and M Weber. A high-speed DAQ framework for future high-level trigger and event building clusters. *Journal of Instrumentation*, 12(03):C03015, 2017.

[14] Matthias Vogelgesang, Lorenzo Rota, LE Ardila Perez, Michele Caselle, Suren Chilingaryan, Andreas Kopmann, and Marc Weber. A heterogeneous FPGA/GPU architecture for realtime data analysis and fast feedback systems. In *Proc. International Beam Instrumentation Conference*, 2017.

[15] L Rota, M Caselle, S Chilingaryan, A Kopmann, and M Weber. A PCIe DMA architecture for multi-gigabyte per second data transmission. *IEEE Transactions on Nuclear Science*, 62(3):972–976, 2015.

[16] W Esmail, T Stockmanns, and J Ritman. Machine learning for track finding at PANDA. *arXiv preprint arXiv:1910.07191*, 2019.

[17] T Boltz, W Wang, E Bründermann, A Kopmann, W Mexner, and AS Müller. Accelerating machine learning for machine physics (An AMALEA project at KIT), *ICALEPCS2019*, New York.

[18] Kim Albertsson, Piero Altoe, Dustin Anderson, John Anderson, Michael Andrews, Juan Pedro Araque Espinosa, Adam Aurisano, Laurent Basara, Adrian Bevan, Wahid Bhimji, et al. Machine learning in high energy physics community white paper. *arXiv preprint arXiv:1807.02876*, 2018.

[19] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

[20] Wikipedia. Go (game). https://en.wikipedia.org/wiki/Go_(game) Accessed October 22, 2020.

[21] Patrick T Komiske, Eric M Metodiev, and Matthew D Schwartz. Deep learning in color: towards automated quark/gluon jet discrimination. *Journal of High Energy Physics*, 2017(1):110, 2017.

[22] CMS Collaboration et al. The CMS Experiment at the CERN LHC, *Journal of Instrumentation*, Volume 3, August, 2008.

[23] Isaac Henrion, Johann Brehmer, Joan Bruna, Kyunghyun Cho, Kyle Cranmer, Gilles Louppe, and Gaspar Rochette. Neural message passing for jet physics. 2017.

[24] Pasi Huovinen and Péter Petreczky. QCD equation of state and hadron resonance gas. *arXiv preprint arXiv:0912.2541*, 2009.

[25] Josef Sollfrank, Pasi Huovinen, Markku Kataja, PV Ruuskanen, Madappa Prakash, and Raju Venugopalan. Hydrodynamical description of 200A GeV/c S+ Au collisions: hadron and electromagnetic spectra. *Physical Review C*, 55(1):392, 1997.

[26] Long-Gang Pang, Kai Zhou, Nan Su, Hannah Petersen, Horst Stöcker, and Xin-Nian Wang. An equation-of-state-meter of quantum chromodynamics transition from deep learning. *Nature Communications*, 9(1):1–6, 2018.

[27] Carsten Schwarz, PANDA Collaboration, et al. The PANDA experiment at FAIR. In *Journal of Physics: Conference Series*, volume 374, page 012003. IOP Publishing, 2012.

[28] PANDA Cooperation. PANDA Annual Report 2019. `https://panda.gsi.de/system/files/user_uploads/k.peters/RE-MGM-2020-002.pdf`, 2019.

[29] Vivienne Sze, Yu-Hsin Chen, Joel Emer, Amr Suleiman, and Zhengdong Zhang. Hardware for machine learning: Challenges and opportunities. In *2017 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8. IEEE, 2017.

[30] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8248–8254. IEEE, 2019.

[31] Xilinx. Virtex-4 Family Overview. `https://www.xilinx.com/support/documentation/data_sheets/ds112.pdf`, 2010.

[32] Xilinx. Virtex-5 Family Overview. `https://www.xilinx.com/support/documentation/data_sheets/ds100.pdf`, 2015.

[33] Xilinx. Zynq-7000 SoC Data Sheet: Overview. `https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`, 2018.

[34] W Ketchum, S Amerio, D Bastieri, M Bauce, P Catastini, S Gelain, K Hahn, YK Kim, T Liu, D Lucchesi, et al. Performance study of GPUs in real-time trigger applications for HEP experiments. *Physics Procedia*, 37:1965–1972, 2012.

[35] L Rota, M Vogelgesang, LE Ardila Perez, M Caselle, S Chilingaryan, T Dritschler, N Zilio, A Kopmann, M Balzer, and M Weber. A high-throughput readout architecture based on PCI-express Gen3 and DirectGMA technology. *Journal of Instrumentation*, 11(02):P02007, 2016.

[36] HiTech Global. HTG-930: Virtex Ultrascale+ PCI Express Development Platform. `http://www.hitechglobal.com/Boards/Virtex_UltraScale+_Vita57.4.htm`, 2018.

[37] HiTech Global. HTG-z920: Xilinx Zynq Ultrascale+ MPSoC PCI Express Development Platform. `http://www.hitechglobal.com/Boards/MPSOC_UltraScale+.htm`, 2018.

[38] Xilinx. Zynq Ultrascale+ MPSoC zcu102: Overview. `https://www.xilinx.com/products/boards-and-kits/zcu102.html` Accessed June 29, 2020.

[39] Xilinx. Zynq Ultrascale+ MPSoC ZCU104: Overview. `https://www.xilinx.com/products/boards-and-kits/zcu104.html` Accessed June 29, 2020.

[40] Xilinx. Zynq Ultrascale+ MPSoC ZCU106: Overview. https://www.xilinx.com/products/boards-and-kits/zcu106.html Accessed June 29, 2020.

[41] Xilinx. Ultrascale+ Devices Integrated Block for PCI Express v1.3. https://www.xilinx.com/support/documentation/ip_documentation/pcie4_uscale_plus/v1_3/pg213-pcie4-ultrascale-plus.pdf Accessed June 29, 2020.

[42] Xilinx. Ultrascale Architecture and Product Data Sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf, 2020.

[43] Samtec. Firefly. http://suddendocs.samtec.com/ebrochures/firefly-brochure.pdf Accessed June 29, 2020.

[44] Xilinx. Zynq Ultrascale+ MPSoC Product Tables and Product Selection Guide. https://www.xilinx.com/support/documentation/selection-guides/zynq-ultrascale-plus-product-selection-guide.pdf Accessed June 29, 2020.

[45] Xilinx. High Speed Serial. https://www.xilinx.com/products/technology/high-speed-serial.html#overview Accessed June 29, 2020.

[46] Xilinx. Zynq Ultrascale+ Packaging and Pinouts. https://china.xilinx.com/content/dam/xilinx/support/documentation/user_guides/ug1075-zynq-ultrascale-pkg-pinout.pdf Accessed March 22, 2021.

[47] Silicon Labs. Clockbuilder Pro Software. https://www.silabs.com/products/development-tools/software/clockbuilder-pro-software, 2018.

[48] Xilinx. Ultrascale architecture PCB design. https://www.xilinx.com/support/documentation/user_guides/ug583-ultrascale-pcb-design.pdf Accessed June 29, 2020.

[49] LINEAR. Dual 25A or single 50A DC/DC micro-module regulator. https://www.analog.com/media/en/technical-documentation/data-sheets/LTM4650.pdf Accessed June 29, 2020.

[50] Xilinx. Zynq Ultrascale+ MPSoC Data Sheet: DC and AC switching characteristics. https://www.xilinx.com/support/documentation/data_sheets/ds925-zynq-ultrascale-plus.pdf Accessed June 29, 2020.

[51] National Laboratory Argonne. Experimental Physics and Industrial Control System. https://epics.anl.gov/ Accessed June 29, 2020.

[52] Xilinx. Zynq Ultrascale+ MPSoC Processing System v2.0. https://www.xilinx.com/support/documentation/ip_documentation/zynq_ultra_ps_e/v2_0/pg201-zynq-ultrascale-plus-processing-system.pdf, 2016.

[53] Yocto project. Website, 2020. https://www.yoctoproject.org/.

[54] National Laboratory. EPICS base. `https://epics.anl.gov/base/index.php` Accessed June 29, 2020.

[55] Martin R. Kraimer. EPICS application developer's guide - EPICS base release 3.16.1, 2017. `https://epics.anl.gov/base/R3-16/1-docs/AppDevGuide/AppDevGuide.html` Accessed June 29, 2020.

[56] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[57] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[58] Yoav Freund. Boosting a Weak Learning Algorithm by Majority. *Information and Computation*, 121(2):256–285, 1995.

[59] Sam T Roweis and Lawrence K Saul. Nonlinear Dimensionality Reduction by Locally Linear Embedding. *Science*, 290(5500):2323–2326, 2000.

[60] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[61] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[62] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, 1989.

[63] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507, 2006.

[64] Kunihiko Fukushima and Sei Miyake. Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Visual Pattern Recognition. In *Competition and Cooperation in Neural Nets*, pages 267–285. Springer, 1982.

[65] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Communications of the ACM (Association for Computing Machinery)*, Volume 60, Issue 6, June 2017, pages 84–90, https://doi.org/10.1145/3065386

[66] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. ImageNet Large Scale Visual Recognition Challenge *International Journal of Computer Vision*, 115(3):211–252, 2015.

[67] Matthew D Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. In *European Conference on Computer Vision*, pages 818–833. Springer, 2014.

[68] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[69] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[70] William Rowan Hamilton On a General Method in Dynamics. *Philosophical Transactions of the Royal Society*, part II for 1834, pages 247–308.

[71] Carl GJ Jacobi. Über die Reduktion der integration der partiellen Differentialgleichungen erster Ordnung zwischen irgend einer Zahl Variablen auf die Integration eines einzigen Systemes gewöhnlicher Differentialgleichungen. *Journal für die reine und angewandte Mathematik*, 17:97–162, 1837.

[72] Richard Bellman. Dynamic Programming. *Science*, 153(3731):34–37, 1966.

[73] Richard Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, pages 679–684, 1957.

[74] Ronald A Howard. Dynamic Programming and Markov Processes. 1960.

[75] Ian H Witten. *Learning to Control.* PhD thesis, University of Essex, 1976.

[76] Ian H Witten. An Adaptive Optimal Controller for Discrete-Time Markov Environments. *Information and Control*, 34(4):286–295, 1977.

[77] Christopher John Cornish Hellaby Watkins. Learning from Delayed Rewards. 1989.

[78] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*, 2013.

[79] Vijay R Konda and John N Tsitsiklis. Actor-Critic Algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[80] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[81] 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, 2016.

[82] 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.

[83] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation Functions: Comparison of Trends in Practice and Research for Deep Learning. *arXiv preprint arXiv:1811.03378*, 2018.

[84] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521(7553):436–444, 2015.

[85] Math HJ Bollen and Irene YH Gu. *Signal Processing of Power Quality Disturbances*. Wiley-IEEE Press, 2006.

[86] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[87] Alexis Perrier. *Effective Amazon Machine Learning*. Packt Publishing Ltd, 2017.

[88] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[89] Tao B Schardl and Siddharth Samsi. TapirXLA: Embedding Fork-Join Parallelism into the XLA Compiler in TensorFlow Using Tapir In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2019.

[90] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: End-to-End Optimization Stack for Deep Learning. *arXiv preprint arXiv:1802.04799*, 11:20, 2018.

[91] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. A Survey of FPGA-based Neural Network Accelerator. *arXiv preprint arXiv:1712.08934*, 2017.

[92] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. DLAU: A Scalable Deep Learning Accelerator Unit on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2016.

[93] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284, 2014.

[94] Xilinx. DPU for Convolutional Neural Network v3.0 – DPU IP Product Guide. `https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_0/pg338-dpu.pdf` Accessed June 6, 2020.

[95] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

[96] Swagat Kumar. Balancing a Cartpole System with Reinforcement Learning–a Tutorial. *arXiv preprint arXiv:2006.04938*, 2020.

[97] Gordon E Moore et al. Cramming more Components onto Integrated Circuits, 1965.

[98] NVIDIA, Peter Vingelmann, and Frank H.P. Fitzek. CUDA, release: 10.2.89, 2020.

[99] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[100] NVIDIA. Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf` Accessed September 29, 2020.

[101] NVIDIA. NVIDIA Tesla K40 Module. https://www.pny.eu/data/products/brochures/nvidia%20tesla%20k40m%20by%20pny%20datasheet.pdf Accessed September 29, 2020.

[102] Xilinx. Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html Accessed June 6, 2020.

[103] Xilinx. Zynq Ultrascale+ MPSoC Product Tables and Product Selection Guide. 2016.

[104] Xilinx. Petalinux Tools Documentation Reference Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1144-petalinux-tools-reference-guide.pdf Accessed June 6, 2020.

[105] Scott Rifenbark. Yocto Project Application Development and the extensible Software Development Kit(eSDK). https://www.yoctoproject.org/docs/2.6/sdk-manual/sdk-manual.html Accessed June 6, 2020.

[106] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the Game of Go without Human Knowledge. *Nature*, 550(7676):354–359, 2017.

[107] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An Open Urban Driving Simulator. *arXiv preprint arXiv:1711.03938*, 2017.

[108] Iulian V Serban, Chinnadhurai Sankar, Mathieu Germain, Saizheng Zhang, Zhouhan Lin, Sandeep Subramanian, Taesup Kim, Michael Pieper, Sarath Chandar, Nan Rosemary Ke, et al. A Deep Reinforcement Learning Chatbot. *arXiv preprint arXiv:1709.02349*, 2017.

[109] Mike Lewis, Denis Yarats, Yann N Dauphin, Devi Parikh, and Dhruv Batra. Deal or no deal? end-to-end learning for negotiation dialogues. *arXiv preprint arXiv:1706.05125*, 2017.

[110] Yuxi Li. Deep Reinforcement Learning: An Overview. *arXiv preprint arXiv:1701.07274*, 2017.

[111] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. MIT Press, 2011.

[112] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[113] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning Internal Representations by Error Propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[114] Long-Ji Lin. Reinforcement Learning for Robots using Neural Networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[115] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Advances in Neural Information Processing Systems*, pages 1057–1063, 2000.

[116] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, (5):834–846, 1983.

[117] Cartpole-v1. `https://gym.openai.com/envs/CartPole-v1/`, 2020.

[118] Russell Reed and Robert J MarksII. *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press, 1999.

[119] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic Differentiation in PyTorch, *31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA* , 2017.

[120] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[121] Pendulum-v0. `https://github.com/openai/gym/wiki/Pendulum-v0/`, 2020.

[122] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153. IEEE, 2009.

[123] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[124] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[125] Matthias Plappert. Keras-RL. `https://github.com/keras-rl/keras-rl`, 2016.

[126] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[127] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the Importance of Initialization and Momentum in Deep Learning. In *International Conference on Machine Learning*, pages 1139–1147, 2013.

[128] Timothy Dozat. Incorporating Nesterov Momentum into Adam. *International Conference on Learning Representations (ICLR) 2016.*

[129] Tobias Boltz, Tamim Asfour, Miriam Brosi, Erik Bründermann, Bastian Härer, Peter Kaiser, Anke-Susanne Müller, Christoph Pohl, Patrick Schreiber, Minjie Yan, et al. Feedback Design for Control of the Micro-bunching Instability Based on Reinforcement Learning. In *10th Int. Partile Accelerator Conf.(IPAC'19), Melbourne, Australia, 19-24 May 2019*, pages 104–107. JACOW Publishing, Geneva, Switzerland, 2019.

[130] L Rota, M Caselle, E Bründermann, S Funkner, Ch Gerth, B Kehrer, A Mielczarek, D Makowski, A Mozzanica, A-S Müller, et al. KALYPSO: Linear Array Detector for High-Repetition Rate and Real-Time Beam Diagnostics. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 936:10–13, 2019.

[131] Robert L Warnock and James A Ellison. A General Method for Propagation of the Phase Space Distribution, with Application to the Saw-Tooth Instability. In *The Physics of High Brightness Beams*, pages 322–348. World Scientific, 2000.

[132] Patrik Schönfeldt, Miriam Brosi, Markus Schwarz, Johannes L Steinmann, and Anke-Susanne Müller. Parallelized Vlasov-Fokker-Planck solver for Desktop Personal Computers. *Physical Review Accelerators and Beams*, 20(3):030704, 2017.

[133] Johannes L Steinmann, Tobias Boltz, Miriam Brosi, Erik Bründermann, Michele Caselle, Benjamin Kehrer, Lorenzo Rota, Patrik Schönfeldt, Marcel Schuh, Michael Siegel, et al. Continuous Bunch-by-Bunch Spectroscopic Investigation of the Microbunching Instability. *Physical Review Accelerators and Beams*, 21(11):110705, 2018.

[134] ACST. Advanced Compound Semiconductor Technologies (ACST). `http://www.acst.de/` Accessed June 6, 2020.

[135] Y Shoji and T Takahashi. Coherent Synchrotron Radiation Burst from Electron Storage Ring under External RF Modulation. In *Proc. 11th European Particle Accelerator Conf (EPAC 08)*, pages 178–180, July 2008.

[136] Johannes Leonhard Steinmann. *Diagnostics of Short Electron Bunches with THz Detectors in Particle Accelerators*. KIT Scientific Publishing, 2019. Ph.D. Thesis.

[137] D. Teytelman. iGp12-720F Signal Processor Technical User Manual. `https://www.dimtel.com/_media/support/manuals/manual12_720f_screen.pdf` Accessed June 6, 2020.

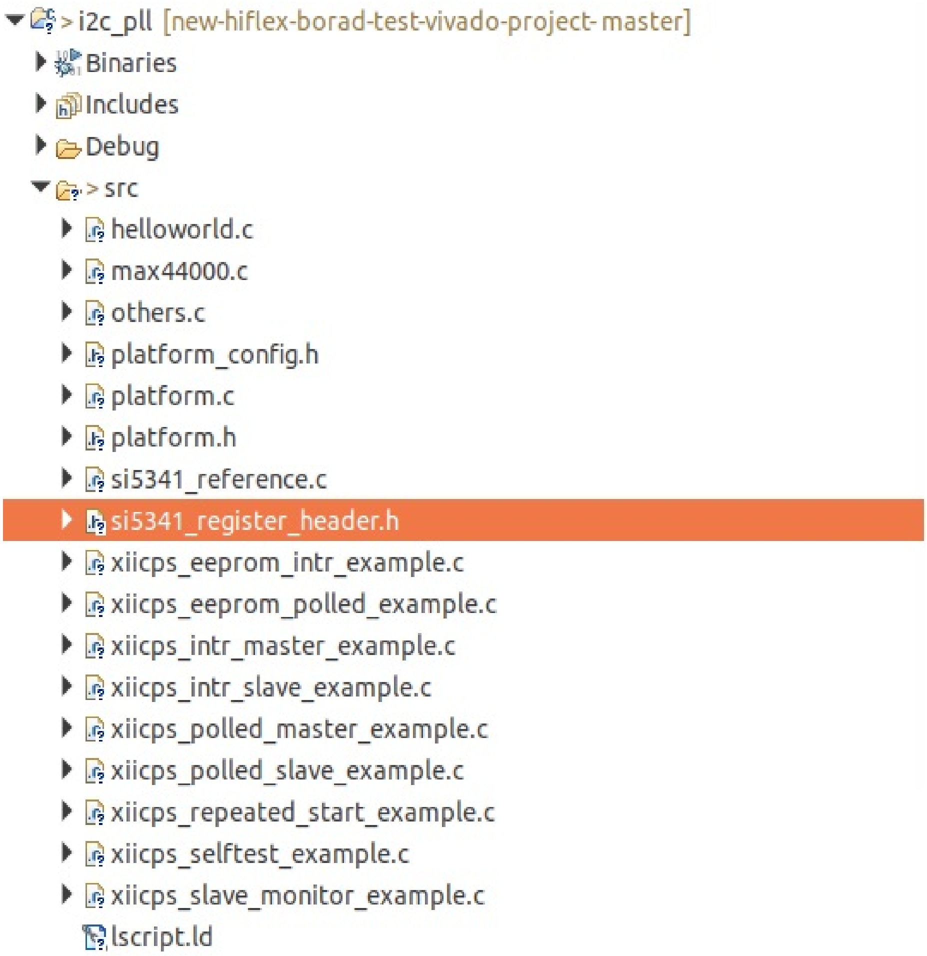
[138] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

List of Abbreviations

ASIC	Application-specific Integrated Circuit
CMS	Compact Muon Solenoid
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSR	Coherent Synchrotron Radiation
DAQ	Data Acquisition
DL	Deep Learning
DMA	Direct Memory Access
DMIPS	Dhrystone Million Instructions executed Per Second
DPU	Deep Learning Processing Unit
DSP	Digital Signal Processing
ECL	Emitter Coupled Logic
FPGA	Field Programmable Gate Array
GNN	Graphic Neural Network
GPU	Graphics Processing Unit
IBERT	Integrated Bit Error Ratio Tester
KARA	Karlsruhe Research Accelerator
LVCMOS	Low-Voltage Complementary Metal Oxide Semiconductor
LVDS	Low-Voltage Differential Signaling
LVPECL	Low-Voltage Positive Emitter-Couple Logic
MDP	Markov Decision Process
ML	Machine Learning
MRP	Markov Reward Process

PCIe	Peripheral Component Interconnect Express
PL	Programmable Logic
PLL	Phase Locked Loop
PS	Processing System
RAM	Random Access Memory
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SDK	Software Development Kit
SL	Supervised Learning
TPU	Tensor Processing Unit
TTL	Transistor-Transistor Logic

Appendix A



```
> i2c_pll [new-hiflex-borad-test-vivado-project- master]
  Binaries
  Includes
  Debug
  > src
    helloworld.c
    max44000.c
    others.c
    platform_config.h
    platform.c
    platform.h
    si5341_reference.c
    si5341_register_header.h
    xiicps_eeprom_intr_example.c
    xiicps_eeprom_polled_example.c
    xiicps_intr_master_example.c
    xiicps_intr_slave_example.c
    xiicps_polled_master_example.c
    xiicps_polled_slave_example.c
    xiicps_repeated_start_example.c
    xiicps_selftest_example.c
    xiicps_slave_monitor_example.c
    lscript.ld
```

Figure 1: This figure is the software structure under Xilinx SDK. The I2C relevant bare-metal drivers are inside different files for different I2C operation. For the user, it is the highlighted *.h* file that required to copy into SDK environment. It is generated from the ClockBuilder. This file will vary and depending on the clock chip itself. The functions will drive the PS I2C Controller to generate the serial I2C protocol to the SI5341. These I2C driver functions are mainly in the *xiicps_polled_master_example.c*. The reset, start, other operation are all in the main function (in the *helloworld.c*). It is relatively easy to configure because only the PS part is needed. Once the clock chip is configured, all the 10 clocks are available.

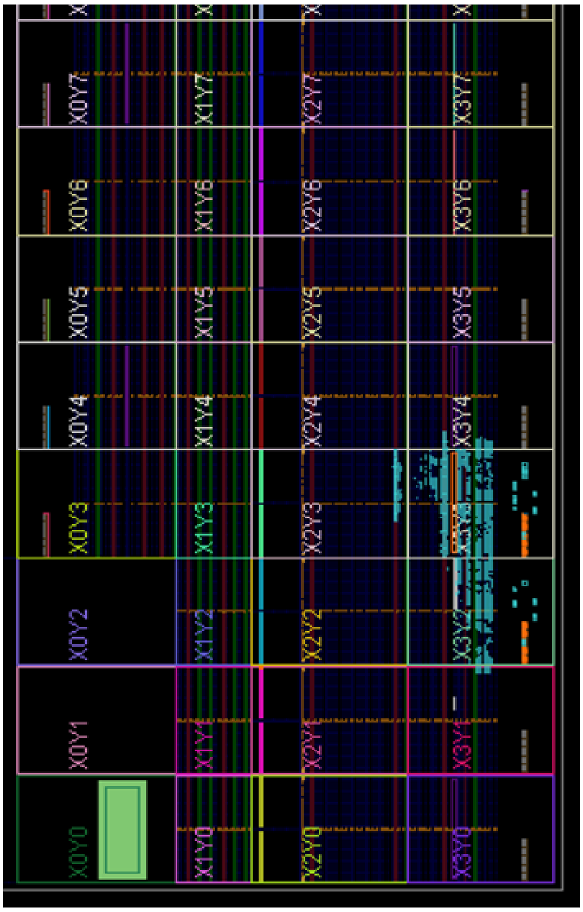


Figure 2: FPGA implementation of a PCIe core, generation 3/4 x 8 lanes (top)

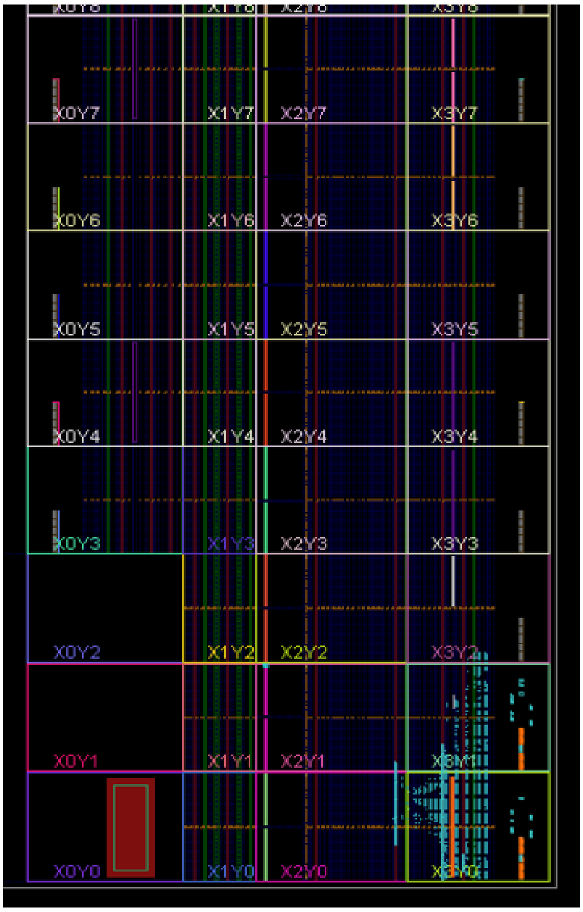


Figure 3: FPGA implementation of a PCIe core, generation 3/4 x 8 lanes (bottom)

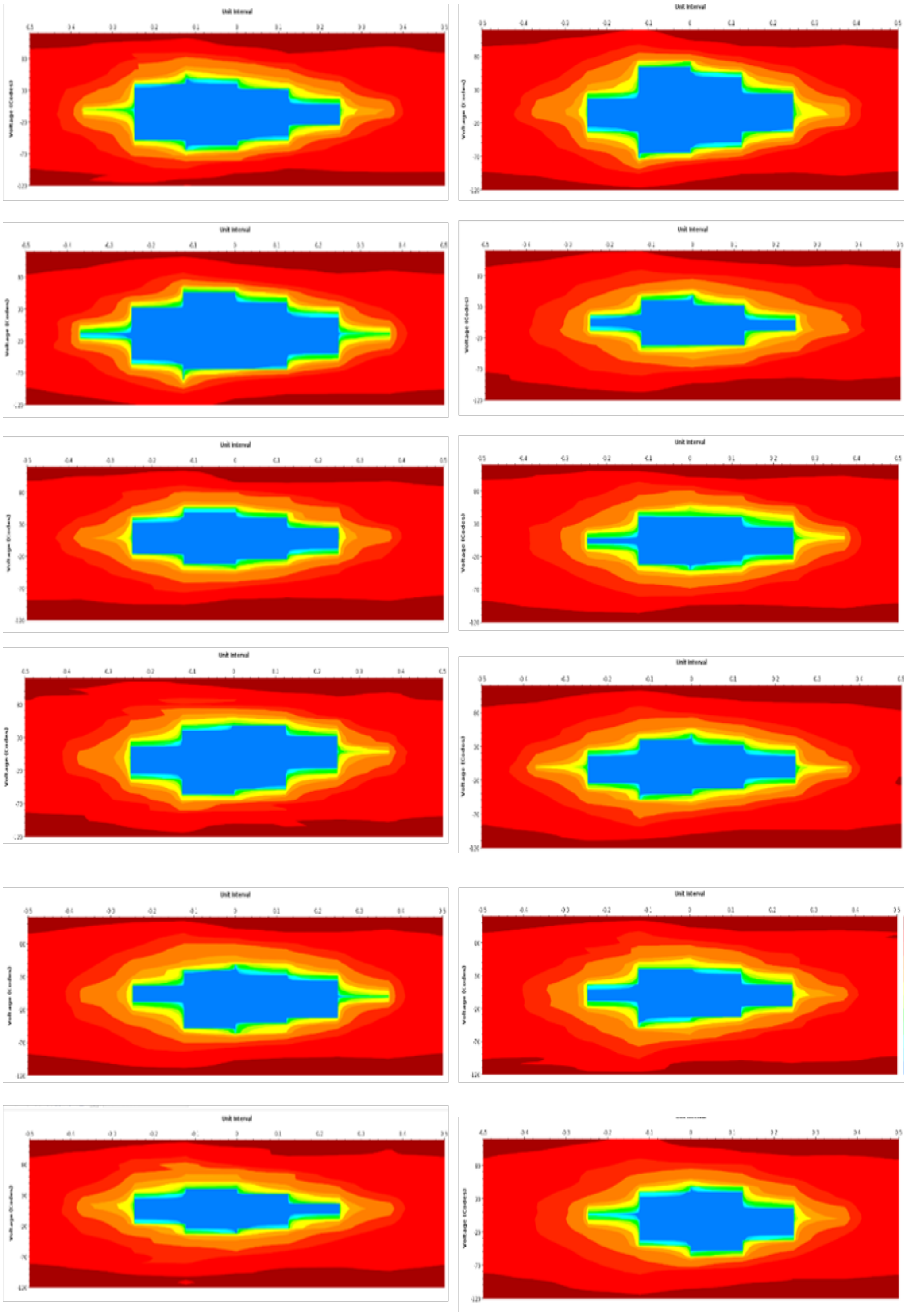


Figure 4: The figure demonstrates 12 eye diagrams of GTY lanes running at 20 Gbps. The subfigures from top-left to bottom-right correspond to the lane 0 to 11.

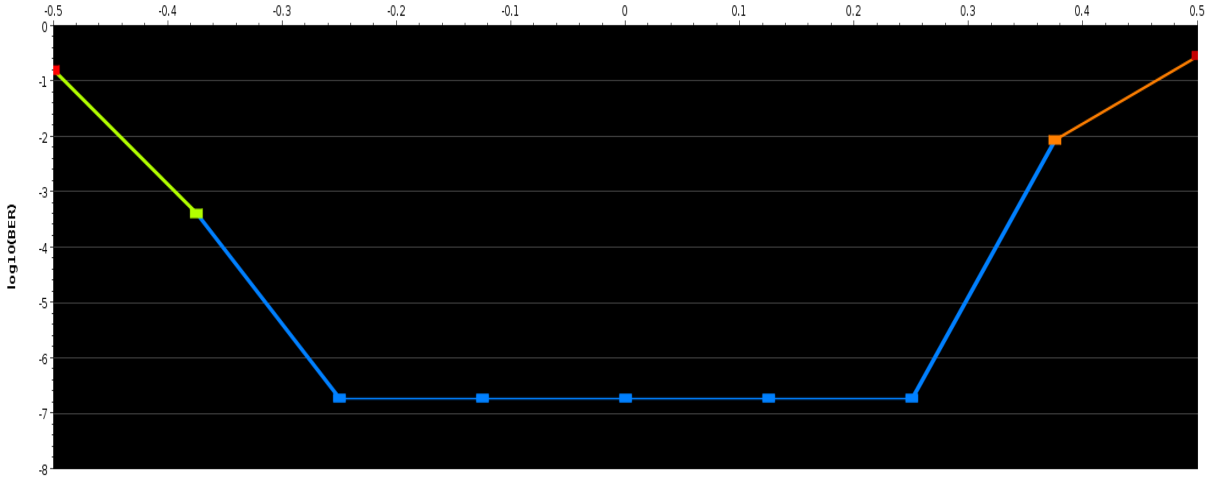


Figure 5: The bathtub curve diagram of GTY lane 0.

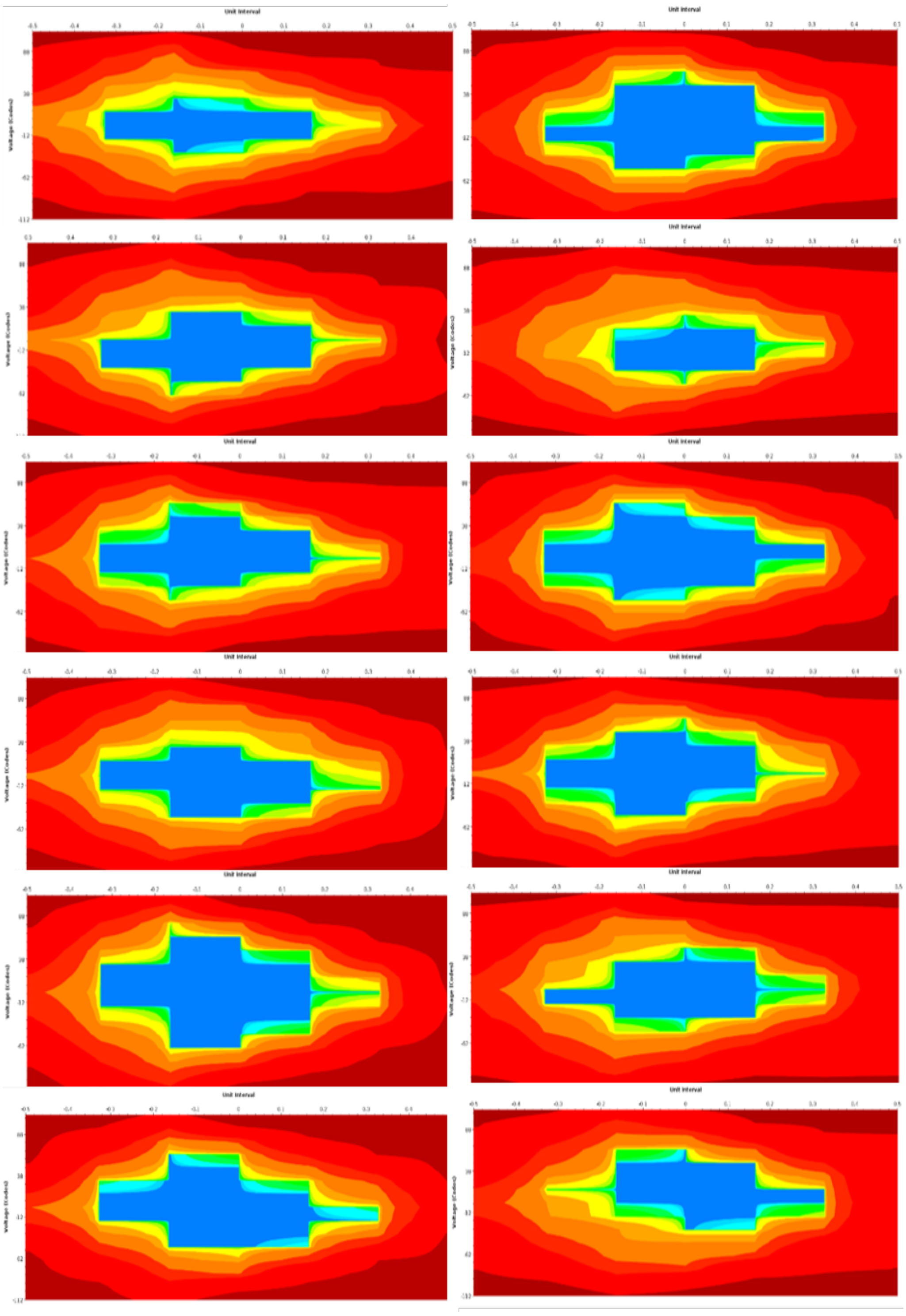


Figure 6: The figure demonstrates 12 eye diagrams of 12 different GTY lanes running at 25 Gbps. The subfigures from top-left to bottom-right correspond to the lane 0 to 11.

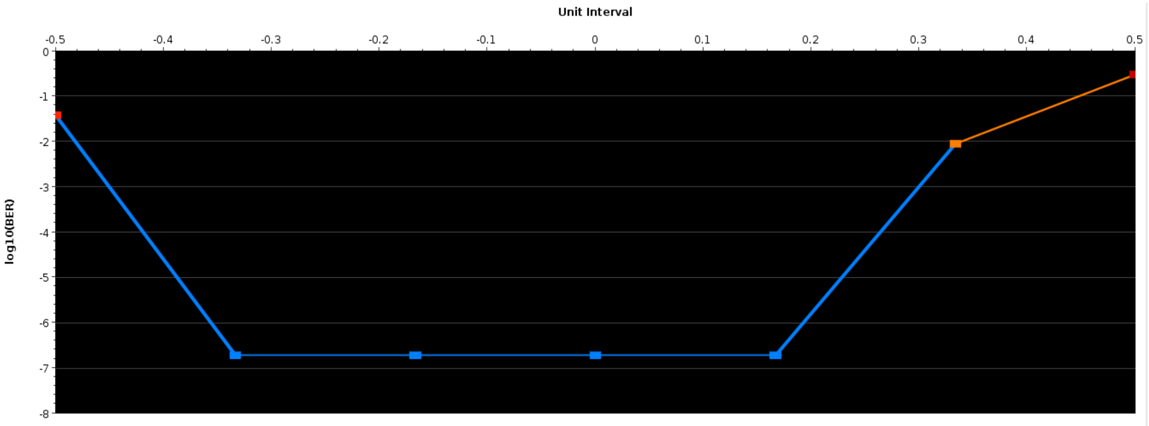
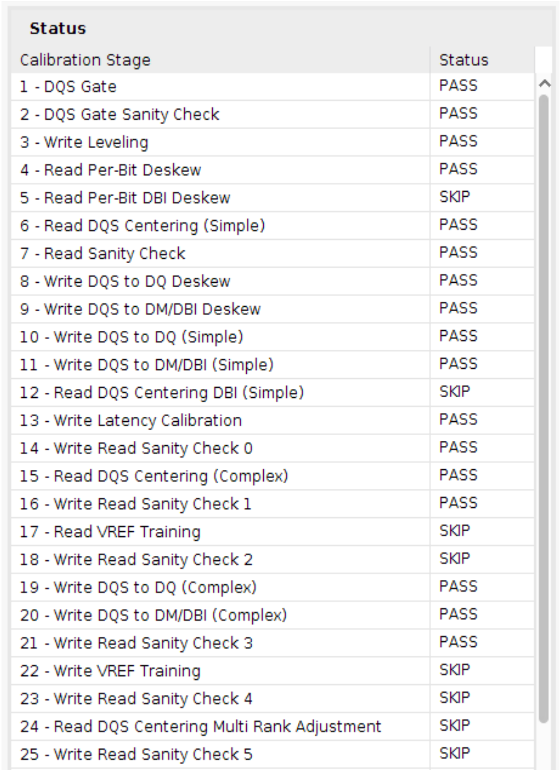


Figure 7: The bathtub curve diagram of GTY lane 1.



Status

Calibration Stage	Status
1 - DQS Gate	PASS
2 - DQS Gate Sanity Check	PASS
3 - Write Leveling	PASS
4 - Read Per-Bit Deskew	PASS
5 - Read Per-Bit DBI Deskew	SKIP
6 - Read DQS Centering (Simple)	PASS
7 - Read Sanity Check	PASS
8 - Write DQS to DQ Deskew	PASS
9 - Write DQS to DM/DBI Deskew	PASS
10 - Write DQS to DQ (Simple)	PASS
11 - Write DQS to DM/DBI (Simple)	PASS
12 - Read DQS Centering DBI (Simple)	SKIP
13 - Write Latency Calibration	PASS
14 - Write Read Sanity Check 0	PASS
15 - Read DQS Centering (Complex)	PASS
16 - Write Read Sanity Check 1	PASS
17 - Read VREF Training	SKIP
18 - Write Read Sanity Check 2	SKIP
19 - Write DQS to DQ (Complex)	PASS
20 - Write DQS to DM/DBI (Complex)	PASS
21 - Write Read Sanity Check 3	PASS
22 - Write VREF Training	SKIP
23 - Write Read Sanity Check 4	SKIP
24 - Read DQS Centering Multi Rank Adjustment	SKIP
25 - Write Read Sanity Check 5	SKIP

Figure 8: The tests during calibration.

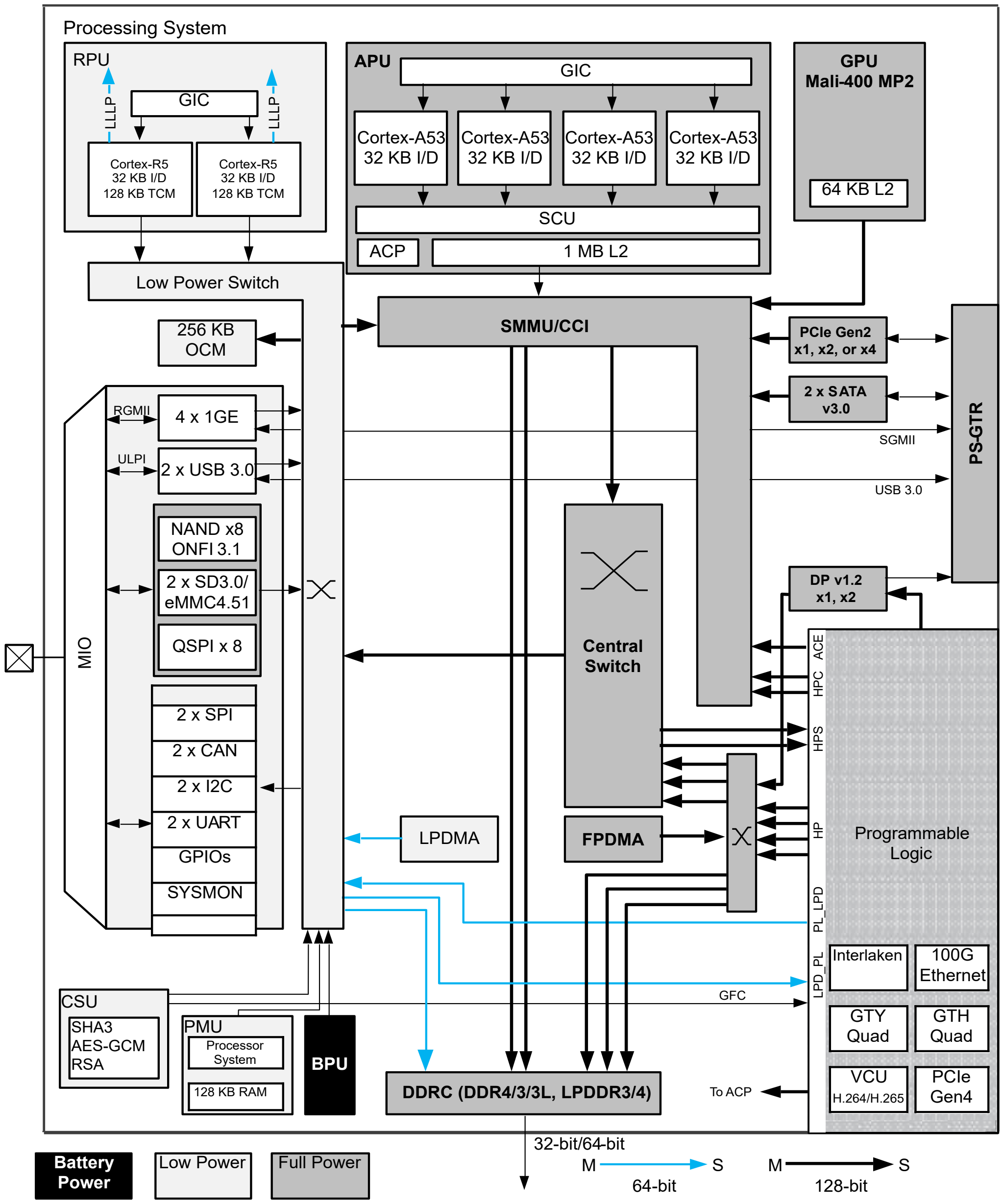
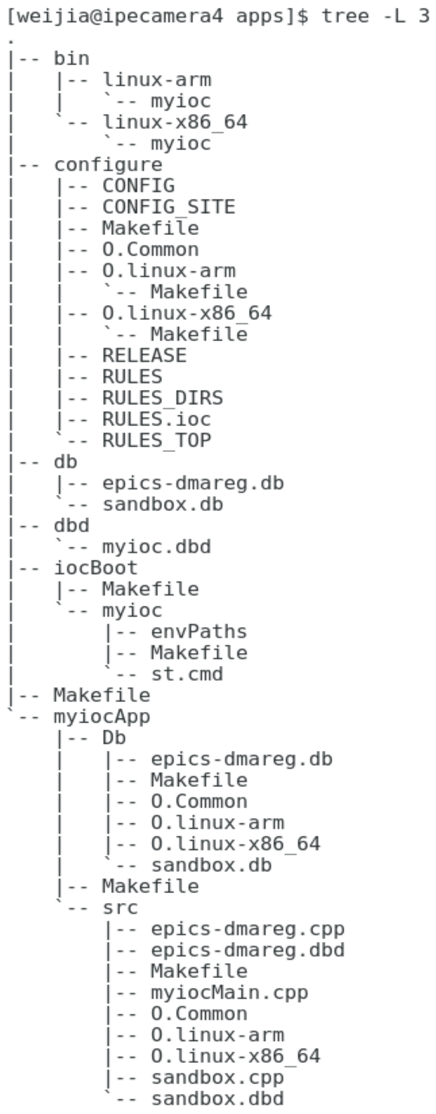


Figure 9: Zynq UltraScale+ MPSoC Top Level Block Diagram [52]

Appendix B

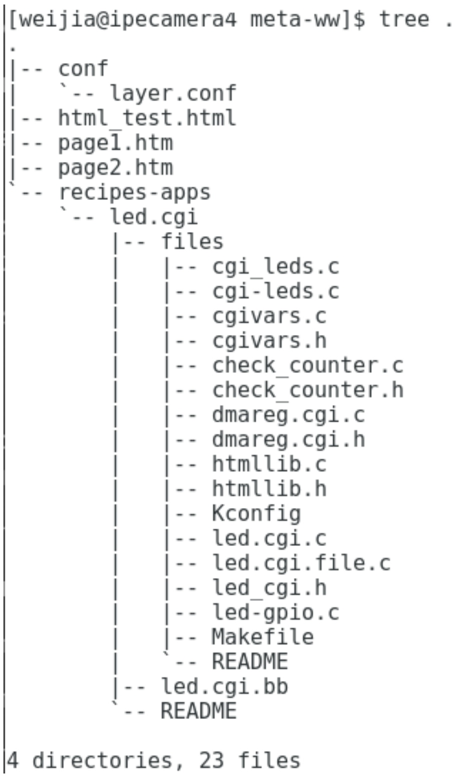


```
[weijia@ipecamera4 apps]$ tree -L 3
.
|-- bin
|   |-- linux-arm
|   |   `-- myioc
|   `-- linux-x86_64
|       `-- myioc
|-- configure
|   |-- CONFIG
|   |-- CONFIG_SITE
|   |-- Makefile
|   |-- O.Common
|   |-- O.linux-arm
|   |   `-- Makefile
|   |-- O.linux-x86_64
|   |   `-- Makefile
|   |-- RELEASE
|   |-- RULES
|   |-- RULES_DIRS
|   |-- RULES.ioc
|   `-- RULES_TOP
|-- db
|   |-- epics-dmareg.db
|   `-- sandbox.db
|-- dbd
|   `-- myioc.dbd
|-- iocBoot
|   |-- Makefile
|   `-- myioc
|       |-- envPaths
|       |-- Makefile
|       `-- st.cmd
|-- Makefile
`-- myiocApp
    |-- Db
    |   |-- epics-dmareg.db
    |   |-- Makefile
    |   |-- O.Common
    |   |-- O.linux-arm
    |   |-- O.linux-x86_64
    |   `-- sandbox.db
    |-- Makefile
    `-- src
        |-- epics-dmareg.cpp
        |-- epics-dmareg.dbd
        |-- Makefile
        |-- myiocMain.cpp
        |-- O.Common
        |-- O.linux-arm
        |-- O.linux-x86_64
        |-- sandbox.cpp
        `-- sandbox.dbd
```

Figure 10: EPICS embedded software files tree

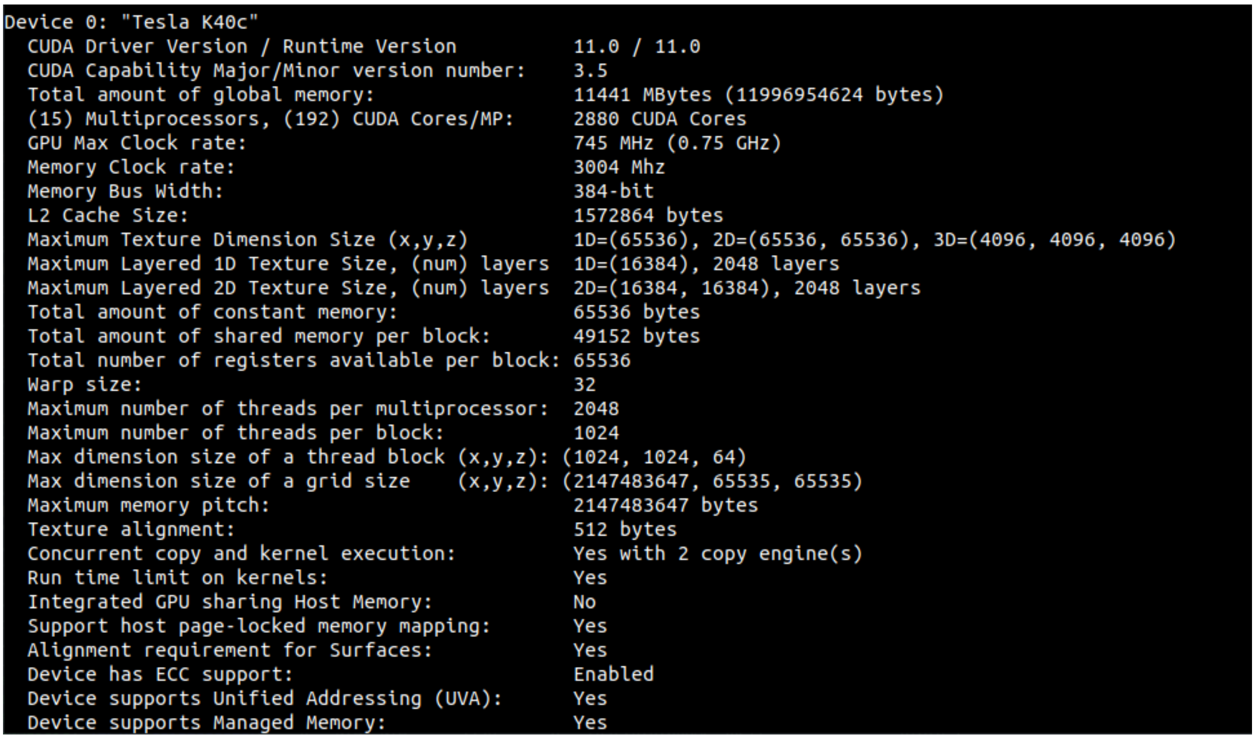
The folder */apps* is under the corresponding Yocto layer. First for cross-compiler employed on Zynq device, demostrate in Fig (10), in file `CONFIG_SITE` located at */configure*, the architecture variable needs to set to *linux-arm* to adapt to Zynq device:

```
CROSS_COMPILER_TARGET_ARCHS
```



```
[weijia@ipecamera4 meta-ww]$ tree .
.
|-- conf
|   `-- layer.conf
|-- html_test.html
|-- page1.htm
|-- page2.htm
`-- recipes-apps
    `-- led.cgi
        |-- files
        |   |-- cgi_leds.c
        |   |-- cgi-leds.c
        |   |-- cgivars.c
        |   |-- cgivars.h
        |   |-- check_counter.c
        |   |-- check_counter.h
        |   |-- dmareg.cgi.c
        |   |-- dmareg.cgi.h
        |   |-- htmllib.c
        |   |-- htmllib.h
        |   |-- Kconfig
        |   |-- led.cgi.c
        |   |-- led.cgi.file.c
        |   |-- led_cgi.h
        |   |-- led-gpio.c
        |   |-- Makefile
        |   `-- README
        |-- led.cgi.bb
        `-- README

4 directories, 23 files
```

Figure 11: Tree of files belongs to layer webserver application under the developing host PC. These files finally is been compiled into Yocto layers as a built embedded Linux application on HighFlex2.



```
Device 0: "Tesla K40c"
  CUDA Driver Version / Runtime Version          11.0 / 11.0
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 11441 MBytes (11996954624 bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
  GPU Max Clock rate:                            745 MHz (0.75 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
```

Figure 12: NVIDIA K40C GPU specification

Appendix C

```

// extract the state signal from memory
x1 = obv_repo[current_step][0];
x2 = obv_repo[current_step][1];
x3 = obv_repo[current_step][2];
x4 = obv_repo[current_step][3];

// extract the forward pass result ()
Y1 = layer2_softmax.result_repo[current_step][0];
// the first unit output (y1)
Y2 = layer2_softmax.result_repo[current_step][1];
// the second unit output (y2)

diff_start1 = -discounted_ep_rs[current_step] \
*one_hot_action_repo[current_step][0];
// 5.5203 (value in the demonstrated example episode, onehot[1]*vt)
diff_start2 = -discounted_ep_rs[current_step] \
*one_hot_action_repo[current_step][1];
// 0      (value in the demonstrated example episode, onehot[2]*vt)

// the num_step here is 13 for this example
diff_soft11 = (1-Y1)/ num_step;
diff_soft12 = -Y2 / num_step;
diff_soft21 = -Y1 / num_step;
diff_soft22 = (1-Y2)/ num_step;

diff_soft[0][0] = diff_soft11 * diff_start1;
diff_soft[0][1] = diff_soft12 * diff_start1;
diff_soft[1][0] = diff_soft21 * diff_start2;
diff_soft[1][1] = diff_soft22 * diff_start2;

layer2_softmax.diff_out[current_step][0] = \
(diff_soft[0][0] + diff_soft[1][0]);
layer2_softmax.diff_out[current_step][1] = \
(diff_soft[0][1] + diff_soft[1][1]);

// caculation of the diff out for the second layer
for (int i=0; i<=SECOND_LAYER_UNI-1;i++)
{
    layer2_dense.diff_out[current_step][i] = \
    ((diff_soft[0][0] + diff_soft[1][0]) \
    * layer2_dense.weights[i][0] +  \
    (diff_soft[0][1] + diff_soft[1][1]) \
    * layer2_dense.weights[i][1]);
}

// backpropogate the partial loss to the front layers
// used later for updating first layer weights
for(int i=0; i <= SECOND_LAYER_UNI-1; i++)
{
    layer1_tanh.diff_out[current_step][i] = \
    (1-layer1_tanh.result_repo[current_step][i] \
    *layer1_tanh.result_repo[current_step][i]);
}

// get the differential value respect to layer2 weights.
for (int i=0; i<=SECOND_LAYER_UNI-1;i++)
{
    layer2_dense.delta_weight[current_step][i][0] = \
    layer2_softmax.diff_out[current_step][0] \
    * (layer1_tanh.result_repo[current_step][i]);
    layer2_dense.delta_weight[current_step][i][1] = \
    layer2_softmax.diff_out[current_step][1] \
    * (layer1_tanh.result_repo[current_step][i]);
}

layer2_dense.delta_bias[current_step][0] = \
layer2_softmax.diff_out[current_step][0] ;
layer2_dense.delta_bias[current_step][1] = \
layer2_softmax.diff_out[current_step][1] ;


for(int input_count = 0; input_count <= \
INPUT_LAYER_UNI-1; input_count++)
{
    for(int output_count=0; output_count <= \
    SECOND_LAYER_UNI-1; output_count++)
    {
        layer1_dense.delta_weight[current_step]
        [input_count][output_count] = \
        (layer2_dense.diff_out[current_step][output_count]) \
        * (layer1_tanh.diff_out[current_step][output_count]) \
        * (obv_repo[current_step][input_count]);
    }
}


for(int output_count = 0; \
output_count <= INPUT_LAYER_UNI-1; output_count++)
{
        layer1_dense.delta_bias[current_step][output_count] = \
        (layer2_dense.diff_out[current_step][output_count]) * \
            (layer1_tanh.diff_out[current_step][output_count]);
}
```

Appendix D

The following code is the *critic network* forward part. It is used for predicting the Q value of the state-action pair. Thus there are two input unit for *critic network*: the state and action. The output is a scalar value.

```
// critic network dense layer 1
critic_input_layer(
        // actor network result,
      critic_layer1_dense.action_input, \
      critic_layer1_dense.state_input, \
      critic_layer1_dense.result,  \

      critic_layer1_dense.action_weights, \
      critic_layer1_dense.state_weights, \
      critic_layer1_dense.biases
);

critic_layer1_active_relu(            \
      critic_layer1_dense.result,  \
      critic_layer1_relu.result   );

// critic network dense layer 2
critic_dense_layer2(critic_layer1_relu.result,  \
      critic_layer2_dense.result, \
      critic_layer2_dense.weights,  \
      critic_layer2_dense.biases);

critic_layer2_active_relu(              \
      critic_layer2_dense.result,  \
      critic_layer2_relu.result);

// critic network dense layer 3
critic_dense_layer3(critic_layer2_relu.result,  \
      critic_layer3_dense.result, \
      critic_layer3_dense.weights,  \
      critic_layer3_dense.biases);

critic_layer3_active_relu(                       \
      critic_layer3_dense.result,  \
      critic_layer3_relu.result);

// critic network dense layer 4
```

```
critic_dense_layer4(critic_layer3_relu.result,  \
        critic_layer4_dense.result, \
        critic_layer4_dense.weights,  \
        critic_layer4_dense.biases);

critic_layer4_active_relu(                \
        critic_layer4_dense.result,  \
        critic_layer4_relu.result );

// critic network dense layer 5
critic_dense_layer5(critic_layer4_relu.result,  \
        critic_layer5_dense.result, \
        critic_layer5_dense.weights,  \
        critic_layer5_dense.biases);

critic_layer5_linear_forward(critic_layer5_dense.result, \
        critic_layer5_linear.result,  1);
```

The slight difference between normal dense layer and critic input dense layer is that require to calculate the action input and state input seperately, then combine then together. Also in this layer, the weights are divided into action's weights and state's weights. In the training part, these two parts also handled in different ways.

```
void critic_input_layer(
     float action_input[CRITIC_INPUT_ACTION_UNI],
     float state_input[CRITIC_INPUT_STATE_UNI],
     float result[CRITIC_SECOND_LAYER_UNI],
     float action_weights[CRITIC_INPUT_ACTION_UNI] \
     [CRITIC_SECOND_LAYER_UNI],
     float state_weights[CRITIC_INPUT_STATE_UNI] \
     [CRITIC_SECOND_LAYER_UNI],
     float  biases[CRITIC_SECOND_LAYER_UNI]
    )
{
     float cache;

     float action_mult[CRITIC_INPUT_ACTION_UNI] \
     [CRITIC_SECOND_LAYER_UNI];
     float action_acc[CRITIC_SECOND_LAYER_UNI];
     float action_result[CRITIC_SECOND_LAYER_UNI];

     float state_mult[CRITIC_INPUT_STATE_UNI] \
     [CRITIC_SECOND_LAYER_UNI];
     float state_acc[CRITIC_SECOND_LAYER_UNI];
     float state_result[CRITIC_SECOND_LAYER_UNI];


     //   action part calculation
     Action_Product1: for(int ii = 0; \
     ii <CRITIC_INPUT_ACTION_UNI; ii++)
     {
         cache = action_input[ii];
         Action_Product2: for(int jj=0; \
         jj <CRITIC_SECOND_LAYER_UNI; jj++)
         {
             action_mult[ii][jj] = cache*action_weights[ii][jj];
         }
     }

     Action_RESETACCUM: for(int iacc = 0;   \
     iacc < CRITIC_SECOND_LAYER_UNI; iacc++)
     {
         action_acc[iacc] = (float) 0;
     }

     Action_ACCUM1: for(int ii=0; \
     ii < CRITIC_INPUT_ACTION_UNI; ii++)
     {
         Action_ACCUM2: for(int jj =0; \
         jj <CRITIC_SECOND_LAYER_UNI; jj++)
         {
             action_acc[jj] += action_mult[ii][jj];
         }
     }

     Action_RESULT: for(int ires = 0; \
     ires < CRITIC_SECOND_LAYER_UNI; ires++)
     {
         action_result[ires] = (float) action_acc[ires];
     }

     //     state part calculation
     State_Product1: for(int ii = 0; \
     ii <CRITIC_INPUT_STATE_UNI; ii++)
     {
         cache = state_input[ii];
         State_Product2: for(int jj=0; \
         jj <CRITIC_SECOND_LAYER_UNI; jj++)
         {
             state_mult[ii][jj] = cache*state_weights[ii][jj];
         }
     }

     State_RESETACCUM: for(int iacc = 0; \
     iacc < CRITIC_SECOND_LAYER_UNI; iacc++)
     {
         state_acc[iacc] = (float) 0;
     }

     State_ACCUM1: for(int ii=0; \
     ii < CRITIC_INPUT_STATE_UNI; ii++)
     {
         State_ACCUM2:for(int jj =0; \
         jj <CRITIC_SECOND_LAYER_UNI; jj++)
         {
             state_acc[jj] += state_mult[ii][jj];
         }
     }

     State_RESULT:for(int ires = 0; \
     ires < CRITIC_SECOND_LAYER_UNI; ires++)
     {
         state_result[ires] = (float) state_acc[ires];
     }

     Adding_Bias: for(int ires = 0; \
     ires < CRITIC_SECOND_LAYER_UNI; ires++)
     {
         result[ires] =
         (float) state_result[ires] + action_result[ires] \
         + biases[ires];
     }
 };
```

Appendix E

The following code snippet is the *actor network* forward part. It is used for predicting the action from state signal. Thus the input for Pendulum state here is $\cos(\theta)$, $\sin(\theta)$, and $\dot{\theta}$, that represent the full information of Pendulum state. The output is the action, here is the joint force, that indicate how the agent moves the pendulum to keep balanced.

```
// get the state signal from environment
actor_layer1_dense.datainput[0] =  current_cos_theta;
actor_layer1_dense.datainput[1] =  current_sin_theta;
actor_layer1_dense.datainput[2] =  current_theta_dot;

/ a = actor.choose_action(s)
// the Actor Eval Net (forward propagation)  //
// actor network dense layer 1
actor_dense_layer1(actor_layer1_dense.datainput,  \
                   actor_layer1_dense.result,  \
                   actor_layer1_dense.weights,  \
                   actor_layer1_dense.biases);

actor_layer1_active_relu(actor_layer1_dense.result, \
                         actor_layer1_relu.result);

// actor network dense layer 2
actor_dense_layer2(actor_layer1_relu.result,  \
                   actor_layer2_dense.result,\
                   actor_layer2_dense.weights,\
                   actor_layer2_dense.biases);


actor_layer2_active_relu(actor_layer2_dense.result,  \
                   actor_layer2_relu.result);

// actor network dense layer 3
actor_dense_layer3(actor_layer2_relu.result,  \
                   actor_layer3_dense.result,\
                   actor_layer3_dense.weights,\
                   actor_layer3_dense.biases);

actor_layer3_active_relu(actor_layer3_dense.result,    \
                   actor_layer3_relu.result);

// actor network dense layer 4
```

```
actor_dense_layer4(actor_layer3_relu.result,   \
                   actor_layer4_dense.result,\
                   actor_layer4_dense.weights,\
                   actor_layer4_dense.biases);

actor_layer4_active_relu(actor_layer4_dense.result,      \
                   actor_layer4_relu.result);

// actor network dense layer 5
actor_dense_layer5(actor_layer4_relu.result,   \
                   actor_layer5_dense.result,\
                   actor_layer5_dense.weights,\
                   actor_layer5_dense.biases);

actor_layer5_active_tanh(actor_layer5_dense.result,      \
                   actor_layer5_tanh.result);
```

The detail implementation of function *actor_dense_layer1()* is shown below, it is simply a matrix multiplication that calculate the fully connected layer forward pass.

```
void actor_dense_layer1(
    float datainput[ACTOR_INPUT_LAYER_UNI],
    float result    [ACTOR_SECOND_LAYER_UNI],

    float weights[ACTOR_INPUT_LAYER_UNI]    \
                 [ACTOR_SECOND_LAYER_UNI],
    float biases[ACTOR_SECOND_LAYER_UNI]
   )
{
    float cache;

    float mult[ACTOR_INPUT_LAYER_UNI][ACTOR_SECOND_LAYER_UNI];
    float acc[ACTOR_SECOND_LAYER_UNI];

    Product1: for(int ii = 0; ii<ACTOR_INPUT_LAYER_UNI; ii++)
    {
        cache = datainput[ii];
        Product2: for(int jj=0; jj<ACTOR_SECOND_LAYER_UNI; jj++)
        {
            mult[ii][jj] = cache*weights[ii][jj];
        }
    }

    RESETACCUM: for(int iacc = 0; \
    iacc < ACTOR_SECOND_LAYER_UNI; iacc++)
    {
        acc[iacc] = (float) biases[iacc];
    }

    ACCUM1: for(int ii=0; ii < ACTOR_INPUT_LAYER_UNI; ii++)
    {
        ACCUM2:for(int jj =0; jj <ACTOR_SECOND_LAYER_UNI; jj++)
        {
            acc[jj] +=mult[ii][jj];
        }
    }

    RESULT:for(int ires = 0; \
    ires < ACTOR_SECOND_LAYER_UNI; ires++)
    {
        result[ires] = (float) acc[ires];
    }

}
```

Appendix F

```


actor_layer1_dense.datainput[0] = current_s1;
actor_layer1_dense.datainput[1] = current_s2;
actor_layer1_dense.datainput[2] = current_s3;

actor_dense_layer1(
actor_layer1_dense.datainput,   \
actor_layer1_dense.result,   \
actor_layer1_dense.weights,   \
actor_layer1_dense.biases);

actor_layer1_active_relu(
actor_layer1_dense.result, \
actor_layer1_relu.result); \

actor_dense_layer2( \
actor_layer1_relu.result,   \
actor_layer2_dense.result,\
actor_layer2_dense.weights,\
actor_layer2_dense.biases);

actor_layer2_active_relu( \
actor_layer2_dense.result, \
actor_layer2_relu.result);

actor_dense_layer3( \
actor_layer2_relu.result,   \
actor_layer3_dense.result,\
actor_layer3_dense.weights,\
actor_layer3_dense.biases);

actor_layer3_active_relu(
actor_layer3_dense.result, \
actor_layer3_relu.result);


actor_dense_layer4(
actor_layer3_relu.result,   \
actor_layer4_dense.result,\
actor_layer4_dense.weights,\
actor_layer4_dense.biases);

actor_layer4_active_relu(
actor_layer4_dense.result , \
actor_layer4_relu.result);

actor_dense_layer5(
actor_layer4_relu.result ,  \
actor_layer5_dense.result ,\
actor_layer5_dense.weights ,\
actor_layer5_dense.biases);

actor_layer5_active_tanh(
actor_layer5_dense.result ,     \
actor_layer5_tanh.result);
// End of the Actor Net forward prop

// Begin of the Critic Net forward prop

//   pass the actor output to the critic input

critic_layer1_dense.action_input[0]
= actor_layer5_tanh.result[0];

//   ******** the Critic Net **************   //
critic_input_layer(   \
critic_layer1_dense.action_input ,
critic_layer1_dense.state_input , \
critic_layer1_dense.result ,  \
critic_layer1_dense.action_weights , \
critic_layer1_dense.state_weights , \
critic_layer1_dense.biases
);

critic_layer1_active_relu(   \
critic_layer1_dense.result ,  \
critic_layer1_relu.result  \
);

critic_dense_layer2(
critic_layer1_relu.result ,  \
critic_layer2_dense.result , \
critic_layer2_dense.weights ,  \
critic_layer2_dense.biases);

critic_layer2_active_relu(  \
critic_layer2_dense.result ,  \
critic_layer2_relu.result  \
);

critic_dense_layer3(
critic_layer2_relu.result,  \
critic_layer3_dense.result, \
critic_layer3_dense.weights,  \
critic_layer3_dense.biases);

critic_layer3_active_relu(  \
critic_layer3_dense.result,  \
critic_layer3_relu.result  \
);


critic_dense_layer4(
critic_layer3_relu.result,  \
critic_layer4_dense.result, \
critic_layer4_dense.weights,  \
critic_layer4_dense.biases);

critic_layer4_active_relu(  \
critic_layer4_dense.result,  \
critic_layer4_relu.result  \
);

critic_dense_layer5(
critic_layer4_relu.result,  \
critic_layer5_dense.result, \
critic_layer5_dense.weights,  \
critic_layer5_dense.biases);

critic_layer5_linear_forward(
critic_layer5_dense.result, \
critic_layer5_linear.result, \
1);
```

Appendix G

```
// training for Actor Eval

// initialize the diff in from the start
critic_layer5_linear.diff_in[0] =  -1;

critic_layer5_linear_back(
critic_layer5_linear.diff_in, \
critic_layer5_linear.diff_out, \
1);

critic_dense_layer5_actor_back(
critic_layer5_linear.diff_out, \
critic_layer5_dense.diff_out, \
critic_layer5_dense.weights);

critic_layer4_relu_back(
critic_layer5_dense.diff_out,  \
critic_layer4_relu.diff_out, \
critic_layer4_dense.result);

critic_dense_layer4_actor_back(
critic_layer4_relu.diff_out, \
critic_layer4_dense.diff_out, \
critic_layer4_dense.weights);

critic_layer3_relu_back(
critic_layer4_dense.diff_out,  \
critic_layer3_relu.diff_out, \
critic_layer3_dense.result);

critic_dense_layer3_actor_back(
critic_layer3_relu.diff_out, \
critic_layer3_dense.diff_out, \
critic_layer3_dense.weights);

critic_layer2_relu_back(
critic_layer3_dense.diff_out,  \
critic_layer2_relu.diff_out, \
critic_layer2_dense.result);

critic_dense_layer2_actor_back(
critic_layer2_relu.diff_out, \
critic_layer2_dense.diff_out, \
critic_layer2_dense.weights);

critic_layer1_relu_back(
critic_layer2_dense.diff_out,  \
critic_layer1_relu.diff_out, \
critic_layer1_dense.result);

critic_dense_layer1_actor_back(
critic_layer1_relu.diff_out, \
critic_layer1_dense.actor_diff_out, \
critic_layer1_dense.action_weights );

// then back from critic to actor
actor_layer5_tanh_back(
critic_layer1_dense.actor_diff_out, \
actor_layer5_tanh.diff_out, \
actor_layer5_dense.result );

actor_dense_layer5_back(
actor_layer4_relu.result,  \
actor_layer5_dense.result, \
actor_layer5_tanh.diff_out, \
actor_layer5_dense.diff_out, \
actor_layer5_dense.weights, \
actor_layer5_dense.biases, \
actor_layer5_dense.diff_weights, \
actor_layer5_dense.diff_biases);

actor_layer4_relu_back(
actor_layer5_dense.diff_out, \
actor_layer4_relu.diff_out, \
actor_layer4_dense.result);

actor_dense_layer4_back(
actor_layer3_relu.result,  \
actor_layer4_dense.result, \
actor_layer4_relu.diff_out, \
actor_layer4_dense.diff_out, \

actor_layer4_dense.weights, \
actor_layer4_dense.biases, \

actor_layer4_dense.diff_weights, \
actor_layer4_dense.diff_biases);

actor_layer3_relu_back(
actor_layer4_dense.diff_out, \
actor_layer3_relu.diff_out, \
actor_layer3_dense.result);
```

```
actor_dense_layer3_back(
actor_layer2_relu.result,  \
actor_layer3_dense.result, \
actor_layer3_relu.diff_out, \
actor_layer3_dense.diff_out, \

actor_layer3_dense.weights, \
actor_layer3_dense.biases, \

actor_layer3_dense.diff_weights, \
actor_layer3_dense.diff_biases);

actor_layer2_relu_back(
actor_layer3_dense.diff_out, \
actor_layer2_relu.diff_out, \
actor_layer2_dense.result);

actor_dense_layer2_back(
actor_layer1_relu.result,  \
actor_layer2_dense.result, \
actor_layer2_relu.diff_out, \
actor_layer2_dense.diff_out, \

actor_layer2_dense.weights, \
actor_layer2_dense.biases, \

actor_layer2_dense.diff_weights, \
actor_layer2_dense.diff_biases);

actor_layer1_relu_back(
actor_layer2_dense.diff_out, \
actor_layer1_relu.diff_out, \
actor_layer1_dense.result);
```

All purpose for backpropagation is to get the diff_weights and diff_biases (partial derivative). Once the above code finish, for *actor network*, all the parameters have the partial derivative of loss function. Then it could be used for any optimizer methods to update the parameters.