Sample

AI NPU System Design with Python and Verilog

Building from Scratch: A Complete Guide to Modeling, Custom ISA, Compiler, and FPGA Implementation

Roger Kim

Professor,
Department of Next-Generation Semiconductor Soongsil University, Korea

First Edition: February 2026

Resources & Contact Information

The source code and project files for the practice exercises in this book are released as below.

- Source Code Repository: https://github.com/estlit/AI_NPU_System_Design_v1 (Please refer to **Appendix A** for usage guide.)
- Author Contact: For academic inquiries, please contact the author at [estlit@daum.net].

DEDICATION

"To my wife, for her endless encouragement and support throughout this journey"

Preface

AI Semiconductors Are Proven by Bits, Not Concepts

Artificial Intelligence is already deeply embedded in our lives. While bookstores and the internet overflow with resources on AI algorithms, practical guides for designing Neural Processing Units (NPUs) from theory to silicon remain rare. It is difficult to find resources that guide you through the entire process to a working system.

Many young engineers I meet have a deepening understanding of algorithms but often feel lost when implementing them on actual hardware. This book addresses that gap. My goal is to move beyond abstract explanations and show the concrete path where concepts turn into actual designs.

I also faced numerous trials and errors, but I persisted by verifying every step—from Python models to RTL translation and FPGA checks. Design is only complete when logic transforms into real signals working correctly from input to output. I felt a distinct thrill when the FPGA results matched the Python model without a single bit of error, confirming that AI semiconductors are proven by bit-accuracy.

This book records those experiences to help you walk this path faster. It covers the entire NPU design lifecycle: Python-based reference modeling, fixed-point design, custom ISA definition, mini-compiler creation, and final RTL implementation with FPGA verification. I have personally verified every process included here.

All source code is public. While the NPU presented here is compact, it offers a solid foundation for expansion. Please use the provided code as a stepping stone to build something far more sophisticated. Now, I invite you to step into the world of design where everything is proven by bits.

Roger Kim

About the Author

Roger Kim is currently a Professor in the Department of Next-Generation Semiconductor at Soongsil University in Seoul, Korea, where he bridges the gap between academic theory and industrial practice.

His career began at the Semiconductor Division of Samsung Electronics in 1991. For the first decade, he specialized in Library Development within the ASIC Technology team. His expertise in Design Automation was recognized early on, earning him the prestigious **Samsung Technology Silver Award** from the Chairman in 2000. Notably, he served as the **first lecturer** for the Verilog Synthesis course at the Samsung Advanced Technology Training Institute, laying the educational foundation for logic design engineers. He was also honored with the **Company Excellence Award** from the CEO in 2001.

Expanding his expertise to the global stage, he led the European System LSI business for Samsung Semiconductor in London and Frankfurt from 2004 to 2009. He subsequently served as a Group Leader for Marketing & Sales in the System LSI division and later in Foundry Strategic Marketing, managing global business operations.

He is the author of **"Semiconductor School" (Amazon, 2022)**, a book widely appreciated for explaining complex semiconductor technologies in accessible language. With a Bachelor's degree in Electronics from Kyungpook National University, Prof. Kim possesses a unique and extensive background that combines deep engineering technicality in system semiconductors with global business insight.

CONTENTS

Part 1

The Era of Artificial Intelligence: From Concepts to Large Language Models

"Artificial Intelligence is not merely a new technology. It is a paradigm shift that fundamentally redefines how we build software and the hardware that executes it."

Introduction

The term Artificial Intelligence has become a buzzword, but for engineers, it requires a clear technical understanding beyond media definitions. Part 1 explores the engineering imperatives behind this revolution, moving from "rule-based" logic to "data-driven" learning. We will examine why this algorithmic shift mandates a physical evolution in silicon—the NPU—and trace the architectural journey from basic Perceptrons to the Transformers powering today's Large Language Models.

Chapter 1. The Map of Artificial Intelligence: The World of AI, ML, and DL

1.1 Technical Hierarchy and Evolutionary History of AI

Although **Artificial Intelligence (AI), Machine Learning (ML), and Deep Learning (DL)** are often used interchangeably as marketing terms, for an engineer, these three concepts possess distinct **Technical Hierarchies** and **Contexts**. Understanding their relationship is the first step in surveying the overall design of an AI system.

Artificial Intelligence (AI): The Vast Universe

- **Definition:** Artificial Intelligence is the overarching concept, defined as "a broad field of computer science attempting to implement human intellectual capabilities (learning, reasoning, perception, language understanding, etc.) into computer systems." It corresponds to the 'Final Goal' we aim to reach rather than a specific technology.
- **Historical Divergence: Symbolism vs. Connectionism** The history of AI is dotted with the conflict between two philosophies on 'how to implement intelligence.'
    - **Symbolism:** *"Intelligence is the manipulation of Symbols and Rules."*
        - This was the dominant school of thought from the 1950s to the 1980s. It attempted to inject human knowledge into computers via logical rules (If-Then).
        - Success Case: IBM's **'Deep Blue'**, which defeated Garry Kasparov in 1997, is a prime example. Deep Blue did not win by learning on its own, but through search algorithms and tens of thousands of rules input by chess masters.
        - Limitation: It was ineffective against problems without clear rules (e.g., recognizing cat photos, natural conversation). This failure led to the **'First AI Winter.'**
    - **Connectionism:** *"Intelligence emerges from neuronal connections in the brain."*

- This approach attempts to learn from data via **Artificial Neural Networks (ANN)** mimicking the human brain. Although initially ignored due to a lack of computing power, it has become the mainstream of modern AI with the advent of Deep Learning.

AI, ML, and DL: Conceptual Boundaries and Engineering Differences

From an engineering perspective, AI, ML, and DL show clear differences in Approach to problem-solving, System Complexity, and Hardware Requirements.

1. Machine Learning (ML): Data-Centric Inductive Reasoning

Machine Learning is a concrete Methodology to achieve AI. In 1997, Professor Tom Mitchell defined it as follows:

> *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."*

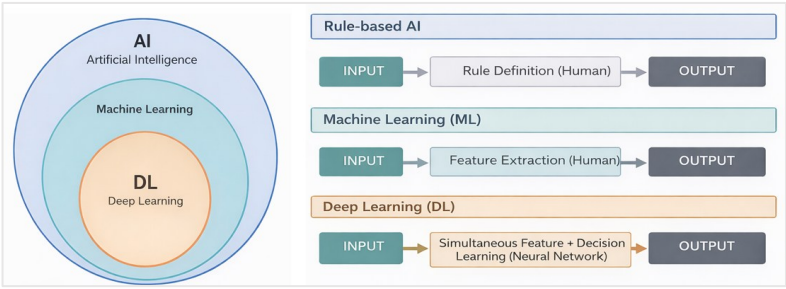
The core of Machine Learning lies in Inductive Reasoning. While traditional programming is deductive (Rule → Result), Machine Learning inversely identifies rules from data.

Limitation (The Bottleneck of Feature Engineering): Traditional Machine Learning algorithms (such as SVM, Random Forest, etc.) required humans to manually extract 'Features' from data. For example, to distinguish a dog from a cat, humans had to explicitly define and quantify features like "shape of ears" or "form of pupils."

2. Deep Learning: Representation Learning and End-to-End

Deep Learning is a subset of Machine Learning that utilizes Deep Neural Networks (DNN) to learn the data's 'Representation' itself.

- **Technological Innovation (Representation Learning):** The most decisive difference distinguishing Deep Learning from traditional Machine Learning is the 'Automation of Feature Extraction.' Deep learning models accept data (pixels) as is and hierarchically learn from low-level features (dots, lines) to high-level features (eyes, nose, face). Because it processes from input to output at once without human intervention, it is also called End-to-End Learning.



[Figure 1.1] Relationship and Comparison of AI, ML, and DL

3. Comparative Analysis Summary (From an NPU Design Perspective)

Comparison Item	Traditional ML	Deep Learning
Core Algorithms	Decision Trees, SVM, Bayesian	CNN, RNN, Transformer
Data Requirements	Low ~ Medium (Thousands)	Vast (Millions ~ Billions)
Hardware Dependency	CPU Sufficient (Good for complex logic)	GPU/NPU Essential (Parallel matrix ops)
Feature Extraction	Manual (Designed by humans)	Automatic (Learned by algorithms)
Impact on NPU Design	Handling complex control logic (Branch) is important	MAC Performance and Memory Bandwidth are key

[NPU Designer's Perspective] Why Do We Focus on Deep Learning?

Traditional Machine Learning algorithms rely heavily on logic branching (If-Else), making them suitable for CPUs. In contrast, over 90% of Deep Learning computations consist of Matrix Multiplication and addition. The NPU is hardware born specifically to accelerate this 'simple, repetitive matrix computation' of Deep Learning.

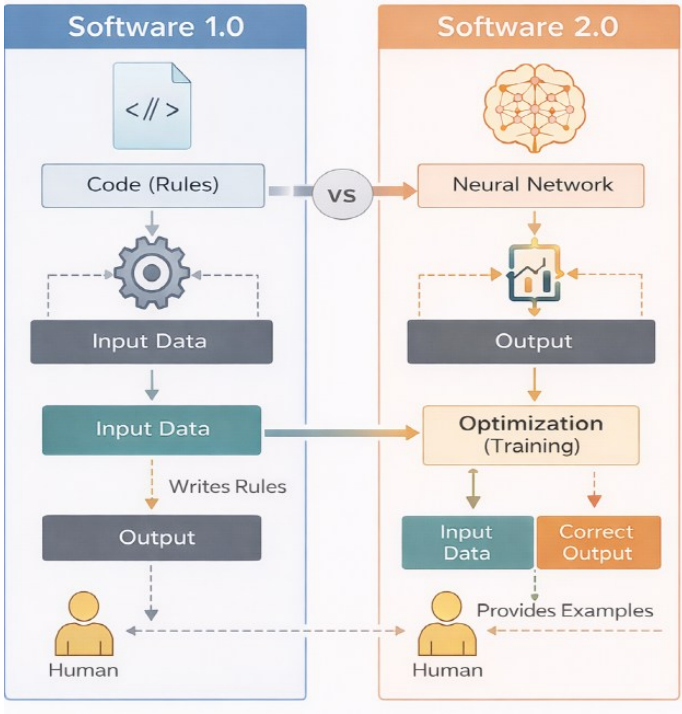
The Rise of Deep Learning and Historical Milestones

The fact that Deep Learning has become the mainstream of modern AI is no coincidence. It is the result of the convergence of three factors: Big Data, Hardware, and Algorithms.

- **AlexNet (2012):** It ignited the 'Deep Learning Boom' by winning the ImageNet competition (ILSVRC) with an overwhelming margin against traditional methods. This was the first time the parallel processing power of GPUs truly shined.
- **AlphaGo (2016):** Google DeepMind's AlphaGo defeated Lee Sedol 4:1, conquering the game of Go, which was considered a realm of intuition and creativity. This was a great victory demonstrating the combination of Reinforcement Learning and Deep Learning.
- **Transformer (2017):** Google announced the Transformer architecture in the paper *"Attention is All You Need."* By parallelizing sequential data processing, it laid the foundation for the birth of today's Large Language Models (LLM) like the GPT series.

1.2 From Rule-Based to Data-Driven: The Great Paradigm Shift (Software 2.0)

From the perspective of software engineering, the rise of AI signifies a fundamental shift in the programming paradigm, far beyond mere technological advancement. Andrej Karpathy defined this transition as moving from 'Software 1.0' to 'Software 2.0'.



[Figure 1-2] Transition to Software 2.0

Software 1.0: Humans Write the Code (Rule-Based)

Software 1.0 refers to the traditional coding we have practiced using languages like C++, Java, and Python. This approach is effective when the developer clearly understands the logic required to solve a problem.

- **Example: Spam Mail Filter (Software 1.0)**

```
Python
# Developer manually writes the rules
def is_spam(email):
    if "advertisement" in email.subject:
        return True
    if "deposit" in email.body and sender == "unknown":
        return True
    return False
```

- **The Complexity Barrier:** The world is far too complex. Consider 'autonomous driving'. Distinguishing between a crumpled bag and a rock using only if-else statements is nearly impossible when writing

complex rules like 'stop for rocks but not for plastic bags'. As rules increase, the code becomes 'spaghetti,' making maintenance impossible.

Software 2.0: Data Writes the Code (Data-Driven)

In Software 2.0, developers do not write the code themselves. Instead, they collect 'Data' and define an 'Objective Function'. An optimization algorithm (Optimizer) then automatically generates the 'code' in the form of neural network weights (Weights).

- **Working Principle:**
  - **Input:** Problem (Image) + Answer (Label)
  - **Process:** Finding weights (W) that minimize error through tens of thousands of trials (Training)
  - **Result:** A mathematical model that works perfectly, even if it is not human-readable (Black-box)
- **Industry Case: Financial Fraud Detection System (FDS)**
  In the past, simple rules like "block overseas payments at 2 AM" were used, but criminals easily bypassed them. A machine learning-based FDS analyzes hundreds of millions of transaction records to identify and defend against complex patterns, such as "*a $30 payment at Mall A followed by a $1,000 payment attempt at Site B five minutes later*".

Training and Inference: The Core of NPU Design

From a hardware perspective, the Software 2.0 process is divided into two entirely different tasks. Understanding this difference is the key to NPU design.

Category	Training (Learning)	Inference (Execution)
Metaphor	Student studying text books	Student taking an exam
Goal	Searching for optimal weights (W)	Predicting results using fixed weights
Computation	Bi-directional (Forward + Backprop)	Uni-directional (Forward Only)
Precision	High (FP32, BF16)	Low (INT8)
Hardware	GPU Clusters (e.g., H100)	Efficient NPU (Edge Devices)

This book will focus on designing an **'Inference NPU'** to run completed models efficiently on edge devices such as smartphones and automobiles.

1.3 Why Do We Need Hardware Accelerators (NPU)?

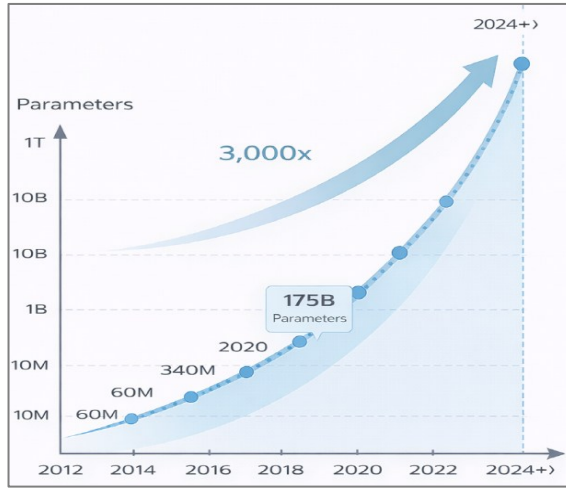
As the software paradigm shifted, the volume and nature of computations required by computers changed completely. We have reached an era of 'computational explosion' and an 'energy crisis' that general-purpose processors (CPU, GPU) can no longer handle.

Explosive Growth of Deep Learning Models

The performance of AI models tends to increase in proportion to the number of parameters.

- **2012 AlexNet:** Approx. 60 million parameters.
- **2018 BERT:** Approx. 340 million parameters.
- **2020 GPT-3:** A staggering **175 billion** parameters (a 3,000x increase in 8 years).
- **2024 and Beyond:** The latest models possess **Trillion**-scale parameters.

This means GPT-3 should retrieve 175 billion numbers from memory and perform computations just to output a single word.



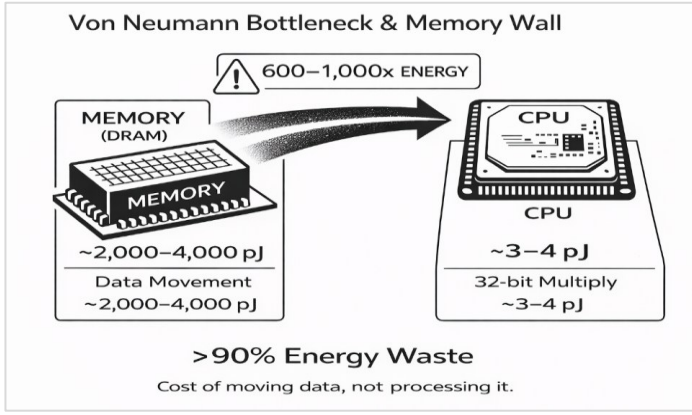
[Figure 1-3] Explosive Growth of Parameters

Von Neumann Bottleneck and the Memory Wall

The greatest enemy of modern computer architecture is 'data movement'. No matter how fast a CPU is, the overall performance is capped by the memory speed if the memory (DRAM) storing the data is slow. This is known as the Von Neumann Bottleneck.

- **The Shocking Energy Truth:** The energy required to fetch data from DRAM inside a semiconductor is approximately **600 to 1,000 times higher** than the energy used to perform a multiplication with that data.
    - One 32-bit multiplication: Approx. **3~4 pJ** (picojoules)
    - One DRAM data access: Approx. **2,000~4,000 pJ**

In other words, over 90% of AI computational energy is wasted on 'moving data' rather than actual calculation. General-purpose structures like CPUs or GPUs have limitations in reducing this data movement.



[Figure 1-4] Von Neumann Bottleneck and Energy Consumption

Limitations of CPU and GPU

- **CPU (Central Processing Unit): The Smart but Slow Conductor**
    - The CPU is a 'Serial Processing' device optimized for executing operating systems and complex logic control.
    - It is inefficient for the simple and repetitive large-scale matrix operations found in deep learning. It is akin to asking a math professor to solve 1 billion simple addition problems.
- **GPU (Graphics Processing Unit): A Powerful Truck with Low Fuel Efficiency**
    - Originally designed for 3D graphics rendering, the GPU is a 'Parallel Processing' device equipped with thousands of small

cores.

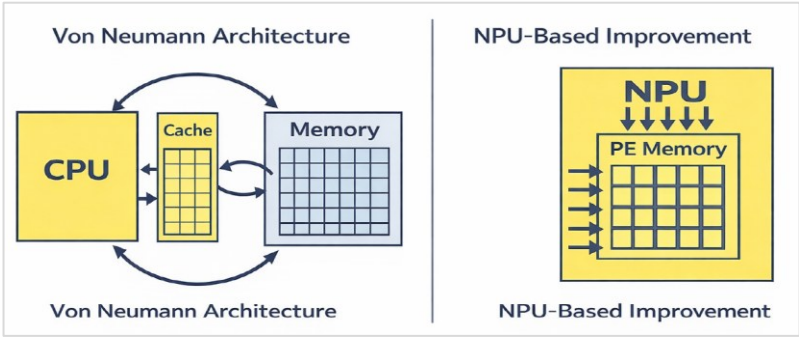
- While it has become the savior of deep learning, it still contains unnecessary functions for graphics (e.g., texture mapping), wasting chip area and power. Its power consumption is too high for mobile devices.

The Inevitable Rise of NPU: An Energy Efficiency Revolution

The **NPU (Neural Processing Unit)** emerged to solve these problems.

- **Abandoning General Purpose:** It boldly discards graphics processing and OS management functions.
- **Maximizing Computation Density:** Most of the chip area is filled with **MAC (Multiply-Accumulate)** units.
- **Data Reuse:** By adopting a **Systolic Array** structure that reuses data (weights) fetched from memory as much as possible within the chip, it drastically reduces the most expensive cost: 'memory access'.

Industry Case: Since 2016, Google has introduced its own NPU, the TPU (Tensor Processing Unit), into its data centers. As a result, it improved energy efficiency (**Performance per Watt**) by **30 to 80 times** compared to conventional GPUs.



[Figure 1-5] Energy Efficiency of NPU Architecture

[NPU Designer's Perspective] We Respond to the Demands of the Era

The goal of semiconductor design is no longer just a 'faster chip'. Our mission is to create a chip that 'processes more intelligence with less power' for the sake of 'Sustainable AI'. The techniques discussed in this book—**Quantization, Sparsity, and On-chip Memory Optimization**—are all fierce solutions devised by engineers to maximize this energy efficiency.

[Chapter1 Summary] The Map of AI and the Inevitability of NPU

Technical Hierarchy:

- AI, ML, and DL are not separate technologies but are nested relationships (AI ⊃ ML ⊃ DL).
- NPUs are dedicated hardware designed to accelerate the matrix operations of 'Deep Learning (DL),' which is the most computationally intensive.

Paradigm Shift (Software 2.0):

- The software development method has shifted from 'humans writing rules (Rule-based)' to 'data creating rules (Data-driven)'.
- This has led to an explosion in computational requirements that general-purpose processors (CPUs) can no longer sustain.

Hardware Evolution (Why NPU?):

- CPU: Smart but slow serial processing (for Operating Systems).
- GPU: Fast but power-hungry parallel processing (for Training).
- NPU: Low-power, high-efficiency accelerator with unnecessary functions removed (for Inference).
- Especially in the era of Large Language Models (LLM), energy efficiency has become a matter of **survival**, not choice.

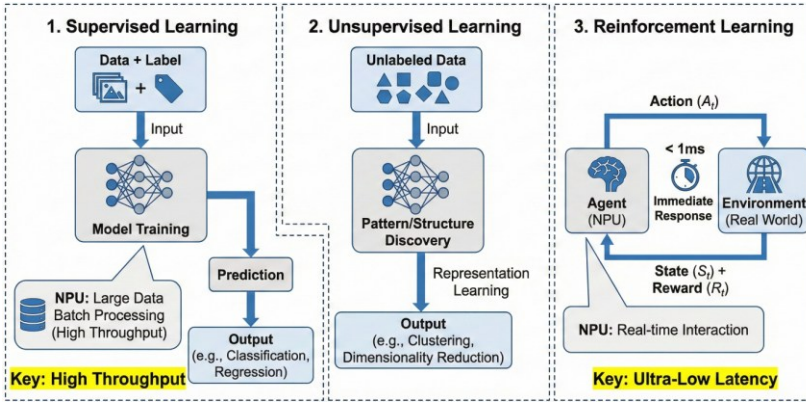
Chapter 2. How Machines Learn: Learning and Data

> *"Data is the new oil, but like oil, it's useless until it's refined."*
> — Clive Humby

If the **Model** is the engine in an AI system, **Data** is the fuel. When the input data or learning methodology is inadequate, even a superior NPU architecture cannot perform to its full potential. In Chapter 2, we define the three core paradigms of machine learning (Supervised, Unsupervised, Reinforcement) from a mathematical perspective and deeply explore how hardware engineers should preprocess and optimize data to maximize NPU performance.

2.1 Paradigms of Learning: In-depth Analysis of Supervised, Unsupervised, and Reinforcement Learning

Machine learning algorithms are broadly classified into three categories based on the **'Form of Feedback'** and **'Objective.'** This classification goes beyond simple software differences; it is a critical factor determining the nature of computations and memory access patterns that the NPU should handle.



[Figure 2-1] Three Learning Paradigms of Machine Learning and Key NPU Design Factors

- **Supervised Learning:** Learns from labeled data. **High Throughput** is critical for processing large batches of data at once.
- **Unsupervised Learning:** Discovers patterns or structures in data without labels. Used for data compression and feature extraction.
- **Reinforcement Learning:** Immediate feedback occurs within an interaction loop with the environment. **Ultra-Low Latency** is vital for real-time response.

Supervised Learning: Learning Data Distribution and Function Approximation

- **[Definition]:** Finding the Mapping between Input and Output
- **[Academic Explanation]** Supervised learning is the process of **approximating** a function f that best explains the relationship between input space X and output space Y. Using a labeled dataset, we statistically estimate parameters that minimize the **Loss** (error) between the model's prediction and the actual value.
- **[For Example]** It is like a student solving **'practice problems with an answer key.'** The student (model) solves the problem, checks the answer, analyzes why they got it wrong, and modifies their knowledge to get it right next time. Through repetitive learning, AI becomes capable of inferring answers even for unseen problems (new data).

Key Task ①: Regression

- **[Academic Explanation: Estimation of Continuous Variables]** Regression deals with problems where the output data is a **Continuous Value**. It expresses the trend of output values with respect to inputs using linear or non-linear models. It typically uses Mean Squared Error (MSE) to measure the geometric distance between the predicted value and the ground truth coordinate.
- **[For Example] "Predicting tomorrow's temperature."** If the actual temperature is 25.0°C and AI predicts 24.5°C, the numerical distance (0.5°C difference) becomes the error. The answer is one of the continuous numbers like 25.1°C or 30.5°C, not discrete categories.

[NPU Design Perspective: Computational Efficiency]
Regression models are relatively economical to implement in hardware. The model's final output stage ends with a **Linear combination** without complex

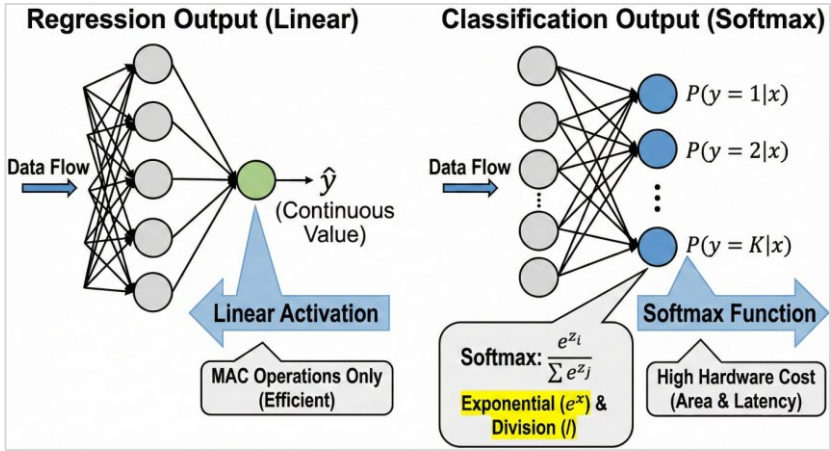
transformations. This can be processed using only the NPU's basic **MAC (Multiply-Accumulate)** units, requiring no separate Special Function Units (SFU).

Key Task ②: Classification

- **[Academic Explanation: Probability Prediction of Discrete Classes]** Classification is the problem of identifying an input as one of K pre-defined **Discrete Classes**. The model estimates the conditional probability $P(y|x)$ that the input belongs to each class and reduces the difference between the predicted probability distribution and the actual distribution using Cross-Entropy based on information theory.
- **[For Example] "Is this picture a dog or a cat?"** AI cannot answer "0.5 dogs." Instead, it answers with probabilities like "80% probability of a dog, 20% probability of a cat." If the answer is 'dog', it learns to increase the certainty (80%) of being a 'dog'.

[NPU Design Perspective: Hardware Bottleneck]
Classification models require caution during hardware design.

- **Cost of Softmax:** The Softmax function used to convert scores into probabilities involves exponential functions (e^x) and division (/). These are high-cost operations that occupy significant area in digital circuits and slow down processing speed (Latency).
- **Design Tip:** Therefore, in inference-only NPUs, if precise probability values are not needed, Softmax is often omitted and replaced with a simple **Argmax** circuit that just finds the 'largest value' to optimize performance.



[Figure 2-2] Comparison of Hardware Computational Costs by Output Layer Type

- **Left (Regression):** Output is calculated via simple Linear Activation, allowing efficient processing with basic MAC units.
- **Right (Classification):** Requires the Softmax function for probability conversion. The exponential function (e^x) and division (/) imply high-cost operations in terms of area and latency.

Unsupervised Learning: Discovering Intrinsic Structures

- **[Definition]:** Learning without Labels and Representation
- **[Academic Explanation]** Unsupervised learning learns the **Intrinsic Structure** of the data's probability distribution P(x) given only input x without label y. While supervised learning focuses on 'Prediction', unsupervised learning aims for **'Representation Learning'** to efficiently summarize or cluster high-dimensional data. Representative examples include Dimensionality Reduction and Clustering.
- **[For Example] "Categorizing foreign news articles."** Suppose a person who doesn't know the language receives thousands of newspaper articles. Although they don't know the content (no answer), they can classify similar ones by looking at the shapes of letters or repeating word patterns, saying "This pile looks like economic news, and that pile looks like sports news." This is the process of finding patterns in the data itself.

[NPU Design Perspective: Bandwidth Reduction]
The **Autoencoder** structure in unsupervised learning is key to NPU system

efficiency.

- **Problem: Raw Data** from CCTVs or autonomous driving sensors is too large, consuming excessive memory bandwidth.
- **Solution:** Place an unsupervised learning-based lightweight compressor (**Encoder**) at the sensor edge.
    - Instead of raw images (pixels), only the compressed **Latent Vector** is transmitted to the main NPU.
    - This drastically reduces data transmission, building a smart interface that solves the **Memory Wall**.

Reinforcement Learning: Optimization via Trial and Error

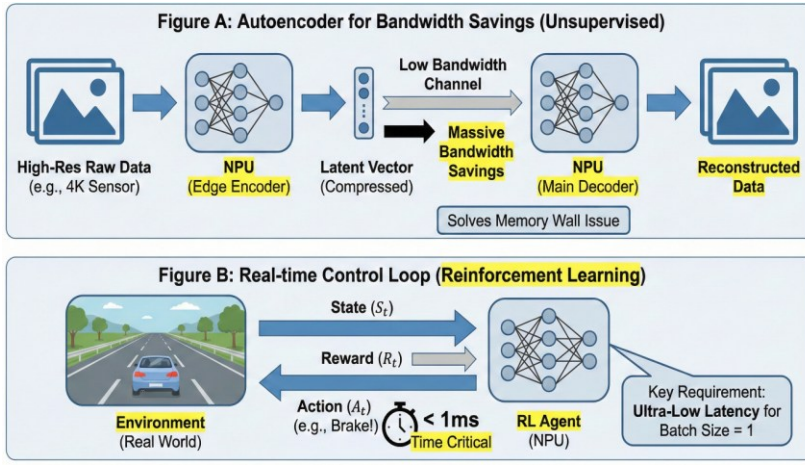
- **[Definition]:** Sequential Decision Making
- **[Academic Explanation]** Reinforcement Learning is a process where an **Agent** interacts with an **Environment** to learn an optimal **Policy (π)** that maximizes cumulative **Reward**. Mathematically, this is modeled as a Markov Decision Process (MDP). Deep RL (e.g., DQN) combined with deep learning allows determining optimal actions through approximation functions even in environments with infinitely large state spaces (e.g., Go, autonomous driving roads).
- **[For Example] "Learning to ride a bicycle."**
    - **State:** Current tilt of the bike, speed.
    - **Action:** Turning the handle or pedaling.
    - **Reward:** Pain if you fall (Penalty), feeling good if you move forward (Reward).
    - Initially, you keep falling (Trial and Error), but eventually, your body learns the method (Policy) to ride long without falling. You learn it yourself without anyone telling you "Turn the handle 30 degrees" (Supervised Learning).

[NPU Design Perspective: The Battle against Latency]
An NPU for Reinforcement Learning should have a completely different design philosophy from a Supervised Learning NPU.

- **Supervised Learning (Throughput Focus):** It doesn't matter if 100 photos are processed at once (**Batch Processing**). Only throughput matters.
- **Reinforcement Learning (Latency Focus):** An autonomous car detecting a car ahead and braking should happen **'Right Now.'**
    - Therefore, it should be designed to maintain computational efficiency even when data comes in one by one (**Batch Size**

= 1).

- Many GPUs or NPUs are optimized for large batches, causing performance to drop sharply at batch size 1. The core technology for RL-dedicated NPUs is reducing this response speed (Latency) to the microsecond (μs) level.



[Figure 2-3] NPU Use Cases from the Perspective of Data Bandwidth and Response Speed

- **Figure A (Unsupervised):** Illustrates a structure solving the bandwidth problem by compressing high-resolution sensor data into small Latent Vectors using an Autoencoder.
- **Figure B (Reinforcement):** Shows that in real-time environments like autonomous driving, the Agent (NPU) should take immediate Action. Thus, **Ultra-Low Latency** at **Batch Size 1** is essential rather than mass processing capability.

2.2 Data Preprocessing: Normalization for Hardware

Preprocessing: Scaling and Standardization

- **[Definition]**: Scaling and Standardization
- **[Academic Explanation]** Preprocessing is the process of converting data with different physical units into a comparable scale. A representative technique is **Z-Score Normalization**, which shifts the

probability distribution to a Gaussian form by making the data mean (μ) 0 and variance (σ) 1.

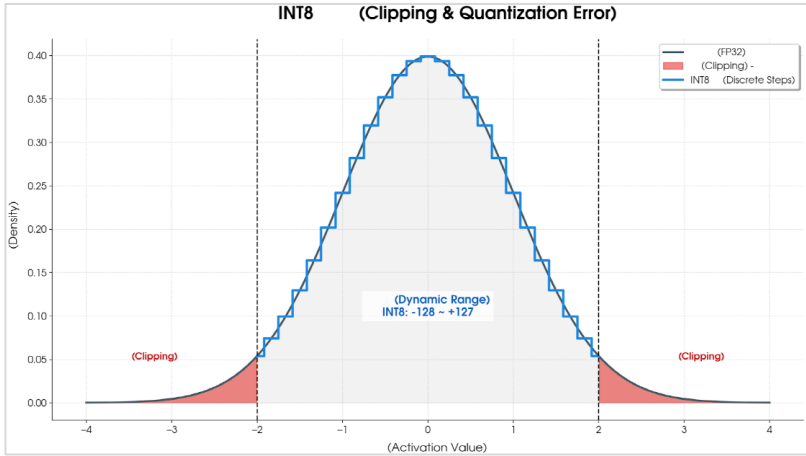
$$z = \frac{x - \mu}{\sigma}$$

This prevents specific input variables from having excessive influence during the optimization of the loss function and improves the convergence stability of model training.

- **[For Example] "Analyzing health with Height (175cm) and Vision (1.0)."** Calculating without normalization causes the AI to misunderstand that 'Height is everything' because the value 175 is overwhelmingly larger than 1.0. It's like adding the weight of an elephant and an ant without unit conversion. Both values should be adjusted to small numbers around 0 for a fair comparison.

[NPU Design Perspective]: Prerequisite for Quantization
Why do NPU designers want data to be gathered around '0'? It is precisely because of **INT8 Quantization**.

- **Constraint:** High-performance NPUs use **8-bit integers (INT8, -128 ~ +127)** instead of 32-bit floating-point (FP32) for power efficiency. There are only 256 containers to hold numbers.
- **Problem:** If data is skewed (Bias) or the range is too wide (e.g., 0 ~ 10000), information gets crushed or cut off (**Clipping**) when forced into 256 slots.
- **Solution:** Preprocessing should gather data beautifully around 0 to use the limited 8-bit resolution fully, minimizing **Quantization Error**.



[Figure 2-4] Data Distribution and Limitations of INT8 Quantization

- The blue steps on the normal distribution curve show the discontinuity (error) when 32-bit floats are converted to 8-bit integers.
- The red areas at both ends visualize the **Clipping (Information Loss)** zones where data exceeds the INT8 expression range (-128 ~ 127).

2.2.2 Evolution of Feature Engineering

- **[Definition]:** From Manual Design to Learned Features
- **[Academic Explanation]** Feature extraction is the process of reducing dimensions by selecting only information meaningful for problem-solving from **Raw Data**. In the past, **Hand-crafted Features**, where domain experts directly designed algorithms, were dominant. However, in the deep learning era, the paradigm has shifted to **Learned Features**, where neural networks discover the weights of filters from data by themselves.
- **[For Example] "Finding a cat in a photo."**
    - **Past (Manual):** Programmers manually wrote code to "find triangular ear shapes" or "find whisker-like lines."
    - **Present (Deep Learning):** When AI is shown 1,000 cat photos, it realizes on its own, "Ah, pointed ears and whiskers are important for cats," and extracts those features.

[NPU Design Perspective: From Dedicated Circuits to General-Purpose Units]

This change completely overhauled hardware architecture.

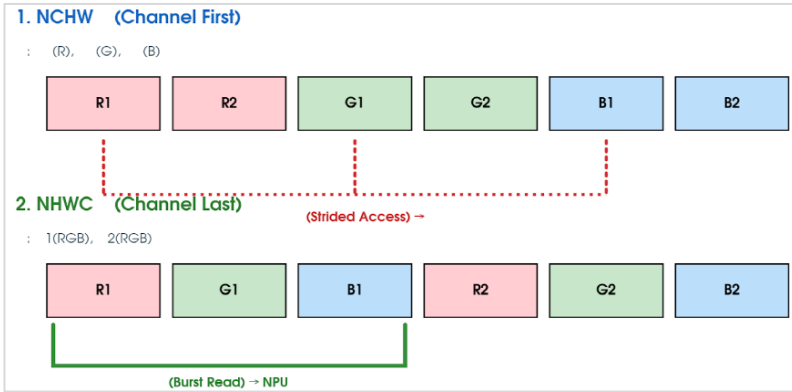
- **Past:** Specific algorithm-dedicated hardware (**Hard-wired Logic**) like Sobel Filter or Canny Edge Detector was required. Flexibility was low.
- **Present:** The NPU doesn't need to know if the filter is looking for 'ears' or 'whiskers'. It just needs to multiply the values in memory (learned weights) with the input data.
- Therefore, the core of NPU design boils down to how fast it can process massive **Matrix Multiplications (GEMM Engine)** rather than implementing 'specific functions'.

2.2.3 Tensor Shape and Memory Layout

- **[Definition]:** Physical Storage Order of Multi-dimensional Data
- **[Academic Explanation]** It is the method of arranging logically 3D data (Image: Height H, Width W, Channel C) into 1D computer memory.
    - **NCHW:** Prioritizes Channel (C). (Batch, Channel, Height, Width)
    - **NHWC:** Places Channel (C) at the very end. (Batch, Height, Width, Channel)
- **[For Example] "Organizing RGB colored papers."**
    - **NCHW:** Collect all red papers (R), then green (G), then blue (B), storing them separately.
    - **NHWC:** Stack red, green, and blue sheets one by one (RGB) to store them as a **'single pixel set.'**

[NPU Design Perspective: Cache Hit Rate and Bandwidth]
Most modern NPUs prefer the **NHWC** format.

- **Reason:** Convolution operations involve a **Dot Product** that fetches all channel (R, G, B) values of an input pixel simultaneously and adds them up.
- **Effect:** In NHWC format, R, G, B data are attached **Contiguously** in memory.
    - Therefore, necessary data can be fetched all at once with a single memory request (**Burst Read**), maximizing efficiency.
    - In contrast, NCHW has R and G data far apart, requiring multiple sparse memory reads, which slows down speed.



[Figure 2-5] NCHW vs. NHWC (Memory Layout Comparison)

- **NCHW (Top):** Emphasizes cache inefficiency by showing R, G, B data separated, requiring 'jumping' (dotted arrows) to access.
- **NHWC (Bottom):** Clearly shows R, G, B are continuous as a single pixel unit, enabling one-time **'Burst Read'** (bold solid arrow), explaining why it is optimized for NPUs.

2.3 Overfitting and Regularization: Basic Engineering of Model Optimization

The problem of **Overfitting**, where a model obsesses too much over training data and performs poorly on new data, is an eternal challenge in machine learning. Interestingly, **Regularization** techniques to solve this are directly linked to NPU hardware **Compression** and **Power Efficiency** technologies.

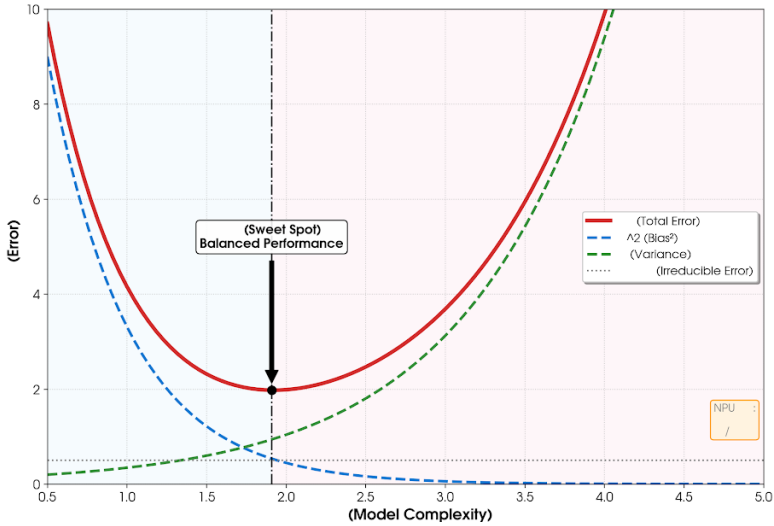
Bias-Variance Tradeoff

- **[Definition]:** Components of Error and Equilibrium Point
- **[Academic Explanation]** A model's Total Error is decomposed into the sum of **Bias**, **Variance**, and **Irreducible Error**.
    - **High Bias (Underfitting):** The model is too simple to capture the underlying patterns of the data.
    - **High Variance (Overfitting):** The model is excessively complex and has memorized even the random **Noise** of the training data.
    - Our goal is to find the optimal **Sweet Spot** between the two to maximize generalization performance.

- **[For Example] "Studying for an exam."**
    - **Underfitting (High Bias):** Studied insufficiently and doesn't even know basic formulas. Gets everything wrong on the test.
    - **Overfitting (High Variance):** Memorized the answers to practice problems perfectly, including the **'typos'** in the question paper. If the problem changes slightly on the actual test (new data), they get it wrong.

[NPU Design Perspective: Model Complexity and Memory]
The tradeoff is directly linked to hardware resource management.

- **High Variance Model:** If you recklessly increase the number of parameters to boost performance, the NPU's **Memory Footprint** and **Computations (MACs)** explode.
- **Strategy:** Therefore, hardware designers should suppress model size through appropriate regularization to induce optimal performance within limited **On-chip Memory (SRAM)**.



[Figure 2-6] Bias-Variance Tradeoff Graph

- **Blue Dotted Line (Bias²):** Error is high when the model is too simple (left) but decreases rapidly as it becomes complex.
- **Green Dotted Line (Variance):** Error increases as the model becomes complex (right) by learning data noise.
- **Red Solid Line (Total Error):** The sum of both. The lowest point of the U-shaped curve is the optimal model (**Sweet Spot**) we seek.

- **Background Color:** Intuitively distinguishes **Underfitting** (blue area) and **Overfitting** (red area) zones.
- **NPU Note:** Added a comment on the bottom right (Complexity Increase = Cost Explosion) to connect with the book's context.

L1, L2 Regularization

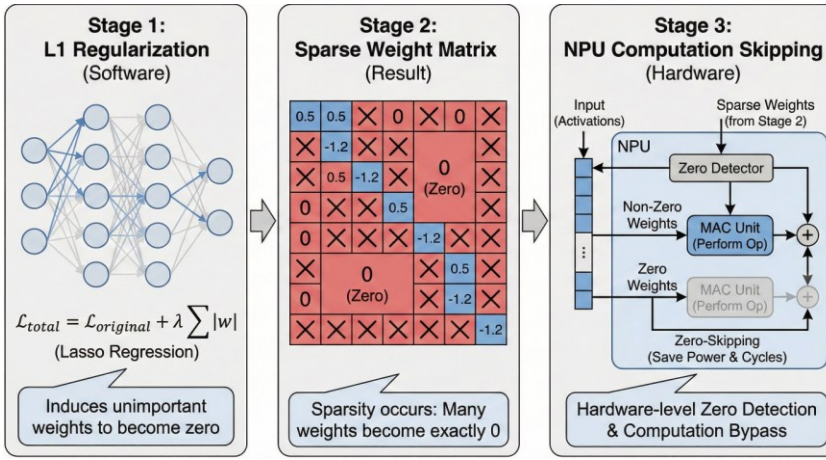
- **[Definition]:** Putting Brakes on Weights
- **[Academic Explanation]** Adds a **Penalty Term** regarding the size of weights (w) to the Loss Function to prevent specific weights from becoming abnormally large.

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{original}} + \lambda \sum |w|$$

    - **L2 Regularization (Ridge):** Adds the square of weights. Makes weights generally small but not zero.
    - **L1 Regularization (Lasso):** Adds the absolute value of weights. Has the property of converging unimportant weights exactly to **0 (Zero)** during mathematical optimization.
- **[For Example] "Budget Cut Policy."**
    - **L2 (Uniform Cut):** "Cut every department's budget by 10%." (Everyone survives, but scale shrinks.)
    - **L1 (Restructuring):** "Make budgets 0 for departments with no performance and eliminate them." (Only important departments remain, others disappear.)

[NPU Design Perspective: Sparsity and Zero-Skipping]
L1 Regularization is the key to NPU hardware acceleration.

- **Zero-Skipping:** For hardware, since X x 0 = 0, there is no need to perform multiplication if the weight is 0.
- **Sparsity Accelerator:** Modern NPUs are equipped with logic that detects 0 values and **Skips** them entirely from the computation pipeline.
    - **Effect:** Shortens computation cycles (Speed Up) + Prevents unnecessary switching (Reduces Power Consumption).
    - In short, **"Software's L1 Regularization maximizes Hardware's power efficiency."**



[Figure 2-7] Sparsity Induction via L1 Regularization (Lasso) and NPU Zero-Skipping

- **Stage 1 (Software):** Adds L1 regularization term to loss function during training to impose weight penalty.
- **Stage 2 (Result):** Generates a **Sparse Matrix** where unnecessary weights converge exactly to '0' due to mathematical properties.
- **Stage 3 (Hardware):** When the NPU's Zero Detector detects a 0 value, it **Bypasses** the MAC unit, reducing power consumption and increasing processing speed.

2.3.3 Dropout: Ensemble Effect

- **[Definition]:** Learning through Random Absence
- **[Academic Explanation]** A technique that **Deactivates** random neurons with probability p during the learning process. Since it trains a partial network (Sub-network) with a different structure every time, it achieves a generalization effect similar to **Ensembling** thousands of neural networks.
- **[For Example] "Soccer Team Training."**
    - If you only pass to one ace player (specific neuron), the team is doomed if that player gets injured. If the coach randomly benches the ace during practice games, the other players learn how to score on their own, making the whole team stronger.

[NPU Design Perspective: Weight Fusing during Inference]
NPU designers should keep in mind that Dropout is used only during 'Training' and not during 'Inference'.

- **No RNG Needed:** Inference-only NPUs do not need complex

Random Number Generator (RNG) circuits for Dropout.

- **Preprocessing (Weight Scaling):** Instead, since weight values have grown by the proportion (p) they were turned off during training, scaling by multiplying (1-p) to the total weights is necessary during inference. By pre-multiplying (**Fusing**) and storing this in weights at the compiler stage before NPU execution, runtime computations can be reduced.

2.3.4 Early Stopping

- **[Definition]:** Catching the Inflection Point of Overfitting
- **[Academic Explanation]** As training progresses, training error continues to decrease, but the error of the **Validation Set** eventually stops decreasing and starts increasing again. This inflection point is where overfitting begins, and this technique stops training immediately at this moment.
- **[For Example] "Cooking Ramen."**
    - Boiling noodles indefinitely doesn't make them tastier. If you boil past the point of being 'al dente' (optimal point), the noodles get soggy (Overfitting). The skill is turning off the heat when it's tastiest.

[NPU Design Perspective: Most Economic Resource Management] This is the most economical regularization method preventing unnecessary waste of computational resources. Especially for NPUs supporting **On-device Learning**, it is crucial to implement Early Stopping logic linked with hardware status registers to reduce battery consumption.

2.4 Model Verification and Evaluation Metrics: Software Scores and Hardware Reality

When designing an NPU or building a system, we receive two report cards. One is **"How accurate is it? (Accuracy)"** and the other is **"How fast and efficient is it? (FPS, Watt)."** This section interprets how software evaluation metrics affect hardware operation methods.

Confusion Matrix and the Trap of Accuracy

- **[Definition]:** Four Quadrants of Correct and Incorrect
- **[Academic Explanation]** The Confusion Matrix summarizes the

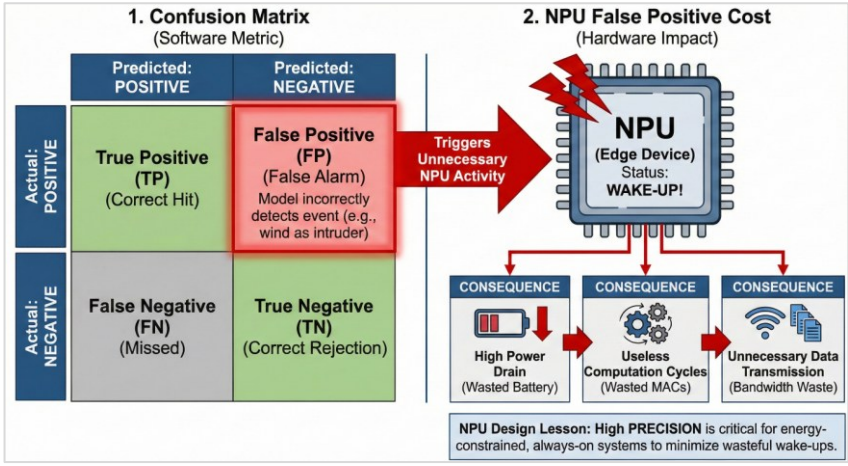
prediction results of a classification model into four cases: **TP (True Positive), TN (True Negative), FP (False Positive), and FN (False Negative)**. The most intuitive metric, **Accuracy**, is the ratio of correctly predicted data among total data.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

- **[For Example] "The Boy Who Cried Wolf."**
  - **TP (Truth):** Wolf came, boy shouted "Wolf!" (Correct)
  - **FP (False Alarm):** Wolf didn't come, shouted "Wolf!" (Villagers suffer in vain)
  - **FN (Miss):** Wolf came, stayed quiet because asleep. (Sheep eaten - Critical)

[NPU Design Perspective: Accuracy Trap and Sparse Data]
Assume a "Defect Detection Process" where the defect rate is 0.1%. Even if the NPU naively outputs "Normal" for everything, Accuracy comes out to 99.9%. However, this NPU provides no practical value.

- **Hardware Implication:** Simply having a model with high **Accuracy** does not make a good NPU system. In environments with severe data imbalance (e.g., CCTV anomaly detection), **Recall** or **Precision** (discussed next) become more critical standards for setting the NPU's alarm **Threshold** than accuracy.



[Figure 2-8] Impact of Classification Model Prediction Results

The figure visualizes the impact of classification results on hardware resources

in 4 quadrants.

- **False Positive (False Alarm):** Predicts yes when it's no. Causes **Power Waste** by unnecessarily waking up the main processor or transmitting data.
- **False Negative (Miss):** Predicts no when it's yes. Means a **Critical Failure** of the system, like a security camera missing a thief.

Precision and Recall: Tug-of-war between Cost and Safety

- **[Definition]:** Reliability vs. Coverage
- **[Academic Explanation]**
    - **Precision:** The ratio of actual True instances among those predicted as True by the model. Asks: *"Is my prediction certain?"*

$$\text{Precision} = \frac{TP}{TP + FP}$$

    - **Recall (Sensitivity):** The ratio of instances correctly identified by the model among all actual True instances. Asks: *"Did I miss anything?"*

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **[For Example]**
    - **Precision Focus (Spam Filter):** Classifying a normal email as spam (FP) is a disaster. Strategy: "Even if we miss some spam, let's only put certain spam in the spam folder."
    - **Recall Focus (Cancer Diagnosis, Fire Alarm):** Diagnosing cancer as normal (FN) kills the patient. Strategy: "Even if there are misdiagnoses (FP), we should find everything suspicious."

[NPU Design Perspective: False Positive and Power Consumption]
Consider an 'Always-on' security camera with an NPU.

- **High FP (Low Precision):** Recognizes waving branches (FP) as a thief, **Waking up** the main processor and transmitting video to the cloud.
    - **Result:** Battery drains rapidly and network bandwidth is wasted due to unnecessary communication and computation.
- **High FN (Low Recall):** Misses the real thief (TP).
    - **Result:** Failure of function as a security system.
- **Design Strategy:** Therefore, for battery-based NPUs, increasing **Precision** to reduce wasted computations (FP) is key to power

efficiency.

F1-Score and mAP: Harmonious Performance Evaluation

- **[Definition]:** The Judge of Imbalanced Data
- **[Academic Explanation]** Precision and Recall are in a trade-off relationship. We use the **F1-Score**, the Harmonic Mean of the two, to evaluate balanced performance. In Object Detection, **mAP (mean Average Precision)**, the area under the Precision-Recall curve at various thresholds, is used as the standard.

$$\mathrm{F1} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

- **[For Example] "Physical Fitness Test."**
  - Being 1st in running but getting 0 in sit-ups doesn't make a good soldier. You need to do well in both to get a high overall score (F1-Score).

[NPU Design Perspective: Standard of Benchmarks]
When verifying NPU performance in AI semiconductor benchmarks like **MLPerf**, we don't just look at "How many frames per second (IPS)."

- **Conditional Performance:** Set a **Quality Floor** like "Max IPS achieved while maintaining mAP 0.5 or higher."
- **Quantization Loss:** Running an FP32 model on an INT8 NPU increases speed but drops mAP. Minimizing this mAP drop (**Accuracy Recovery**) is the measure of NPU compiler technology.

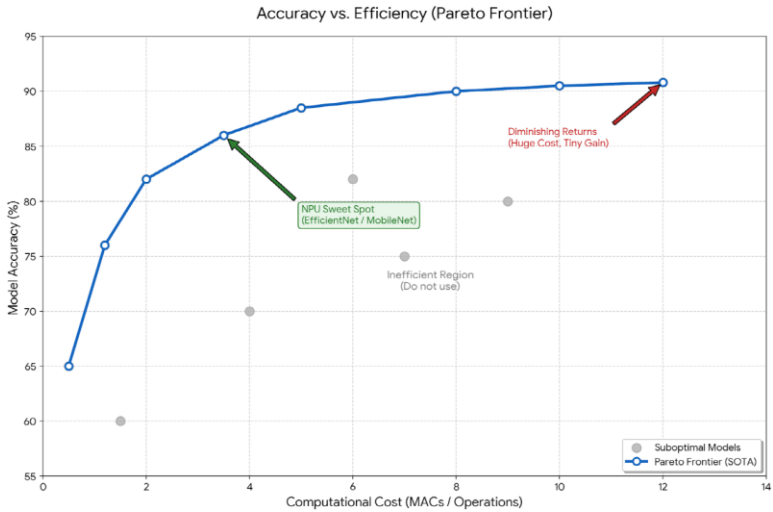
Accuracy vs. Efficiency Trade-off

- **[Definition]:** What to Sacrifice for 0.1% Accuracy?
- **[Academic Explanation]** Modern models exponentially increase parameters to maximize accuracy. However, hardware resources are limited. The **Accuracy-Efficiency Trade-off** is the engineering decision to find the point yielding the highest accuracy within given hardware constraints (Power, Latency).
- **[For Example] "Solving SAT Problems."**
  - **Student A:** Checked answers 10 times and got 100, but took 5 hours. (Accuracy Best, Practicality Fail)
  - **Student B:** Solved intuitively without checking, got 98, finished in 1 hour. (Accuracy Slightly Lower, Speed Best)
  - In real-time systems, Student B is much more competent.

[NPU Design Perspective: Pareto Optimality]

NPU designers should weigh "Cost-effectiveness."

- **Pareto Frontier:** If doubling model size (computations) only increases accuracy by 0.1%, the NPU should not adopt it.
- **Justification for Light-weighting:** "Accuracy drops by 1%, but speed quadruples and power is halved?" → For Mobile/Edge NPUs, this is much more beneficial. This is why we should aggressively apply the **Regularization** and **Pruning** learned in Section 2.3.



[Figure 2-9] Pareto Efficiency Curve of Accuracy vs. Computational Cost

- **Pareto Frontier (Blue):** The line connecting the highest achievable accuracy for a given computational cost. Models on this line represent the **State-of-the-Art**.
- **NPU Sweet Spot (Green):** The point NPU designers should focus on. The inflection point where the graph rises steeply and then flattens, offering 'Best Value' with high accuracy for a small increase in computation.
- **Diminishing Returns (Red):** The right end shows massive computation investment for a mere 0.1% accuracy gain. An uneconomical zone to avoid from a hardware efficiency perspective.

[Chapter2 Summary] Learning Data and Hardware Optimization

Types of Learning and Inference Focus

- While various paradigms like Supervised, Unsupervised, and Reinforcement Learning exist, the core target of NPU hardware design is accelerating the **'Inference'** of trained models.
- Inference-only NPUs can remove complex circuits and memory for Backpropagation, allowing for a much simpler and power-efficient design than training processors.

Data Preprocessing (Preprocessing & Quantization)

- **Normalization:** Gathering the distribution of input data around 0 goes beyond software convenience; it is a prerequisite for minimizing information loss (**Clipping**) during hardware **INT8 Quantization**.
- **Memory Layout (NHWC):** Maximize **Burst Read** efficiency by using the NHWC format, which places data adjacently in channel (C) order, allowing the systolic array to fetch data chunks in a single access.

Regularization and Sparsity

- **Zero-Skipping:** L1 Regularization (Lasso) or ReLU activation functions mathematically turn weights and output values into '0'.
- **Hardware Acceleration:** NPUs detect these '0's to **Skip** unnecessary multiplication operations, increasing operation speed (**Latency**) and drastically reducing power consumption (**Power**). In short, software regularization techniques become the key to activating hardware sparsity acceleration.

Model Verification and Trade-off

- **Cost of Metrics:** Simply high Accuracy is not everything. In Edge NPUs, since False Positives cause power waste (battery drain) and False Negatives cause critical failure, the balance of **Precision** and **Recall** should be tuned to hardware specs.
- **Pareto Efficiency:** NPU design is a battle of 'Accuracy vs. Efficiency.' Instead of investing excessive computations for 0.1% accuracy gain, engineering decisions are needed to find the appropriate compromise (**Sweet Spot**) and achieve optimal performance with lightweight models.

[Appendix A] Source Code Structure & Usage Manual

The practice examples in this book are organized into four distinct stages based on the learning progression: **Concept (00) → Single Verification (01) → Mass Verification (02) → Processor Extension (03)**.

[Source Code Repository] All source codes and project files used in this book are available on the official GitHub repository. Please clone or download the latest version to your local workspace before proceeding.

- **GitHub URL:**
  https://github.com/estlit/AI_NPU_System_Design_v1

1. System Environment

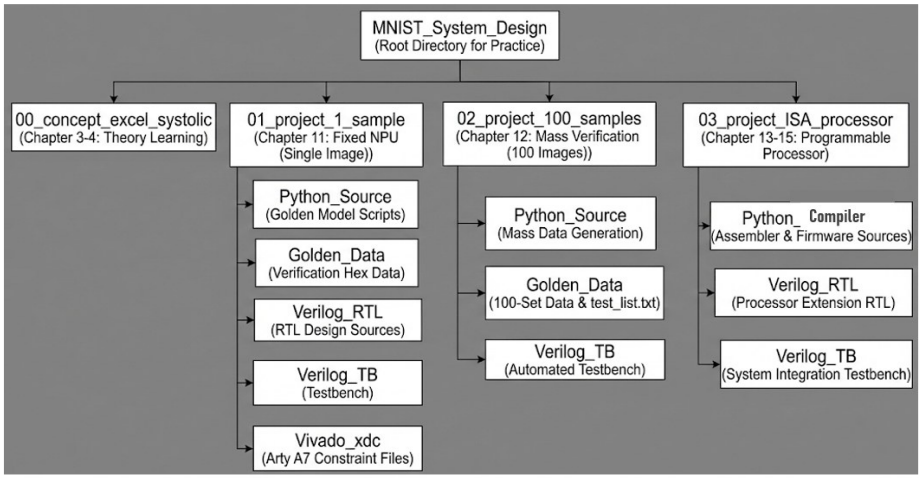
This project has been developed and verified in the following system environment. To reproduce the results exactly as described in the book, we recommend setting up a similar environment.

Category	Item	Specification
Operating System	OS	Windows 11 Pro
	Version	24H2 (OS Build 26100.7623)
	System Type	64-bit operating system, x64-based processor
Software & Language	Distribution	Anaconda (packaged by Anaconda, Inc. | main, Jun 12 2025)
	Python Version	3.13.5 [MSC v.1929 64 bit (AMD64)]
	Required Library	numpy, tensorflow, matplotlib

2. Directory Structure

The project is divided into four main directories to separate logic verification from hardware implementation and automation.

- **00_concept_excel_systolic**: Mathematical verification of systolic array logic.
- **01_project_1_sample**: RTL design and verification of a fixed-function NPU using a single sample.
- **02_project_100_samples**: Automation of mass regression testing using 100 samples.
- **03_project_ISA_processor**: Implementation of a programmable NPU system based on a custom ISA.



3. Detailed File Description

This section defines the technical functions and roles of the core files in each stage. The **"Modification Points & Guide"** column is particularly important; it highlights specific lines of code (such as data load paths, hardware parameters, or compiler settings) that you must adjust according to your local development environment to avoid path-related errors during simulation.

00_concept_excel_systolic (Architecture Theory)

This stage involves numerical verification of the systolic array's data flow using spreadsheet logic.

Category	File Name	Technical Description	Modification Points & Guide
Root	systolic_matrix.xlsx	A visualization tool for tracing the operation of each Processing	**[Optional]** Update input matrices A and B to verify automatic result

Category	File Name	Technical Description	Modification Points & Guide
		Element (PE).	calculation logic.

01_project_1_sample (Core NPU Design)

This stage focuses on hardware core design for single-sample inference and golden data mapping.

Category	File Name	Technical Description	Modification Points & Guide
Golden_Data	conv1_bias.hex	Bias data for Convolution Layer 1.	Add to Design Sources
	conv1_out.hex	Expected output after Layer 1 operation (Golden).	-
	conv1_pool_out.hex	Expected output after Pooling operation.	-
	conv1_weights.hex	Weight data for Convolution Layer 1.	Add to Design Sources
	fc_argmax.hex	Expected final classification index.	-
	fc_bias.hex	Bias data for the Fully-Connected (FC) layer.	Add to Design Sources
	fc_input.hex	Input data for the FC layer.	-
	fc_scores.hex	Expected scores before Softmax.	-
	fc_weights.hex	Weight data for the FC layer.	Add to Design Sources
	input_img.hex	Input image for 1 sample	Add to Design Sources
Python_Source	conv_weight_converter.py	Utility to convert training data into NPU-compatible formats.	-
	npu_golden_model.py	Script to generate golden data based on the reference model.	-
Verilog_RTL	argmax10.sv	Module to extract the	-

Category	File Name	Technical Description	Modification Points & Guide
		index of the maximum value among 10 results.	
	arty_conv1_conv2_flatten_top.sv	Top RTL module for layer integration and data flattening.	-
	conv1_core.sv	Core logic for Convolution Layer 1 operations.	-
	conv1_fsm.sv	Finite State Machine for Layer 1 control.	-
	conv1_lastpixel_helper.sv	Helper module for handling boundary conditions.	-
	conv1_layer.sv	Integrated module for Layer 1 computation and memory.	-
	conv1_top.sv	Top-level interface module for Layer 1.	-
	conv1_wb_regs_tb.sv	RTL for testing the register bank.	-
	conv2_lastpixel_helper.sv	Helper module for Layer 2 boundary processing.	-
	conv2_ram_buffer.sv	RAM for Layer 2 data buffering.	-
	conv2_ram_write_controller.sv	Control logic for RAM write operations.	-
	conv2_top.sv	Top-level interface module for Layer 2.	-
	fcl.sv	Integrated module for the Fully-Connected layer.	-
	fc_bias_rom.sv	ROM module for storing FC bias data.	-
	fc_core.sv	Core logic for FC layer operations.	-
	fc_input_buffer.sv	Temporary storage for	-

Category	File Name	Technical Description	Modification Points & Guide
		FC input data.	
	fc_weight_rom.sv	ROM module for storing FC weights.	-
	fpga_top2.sv	**[Top]** Overall top-level module for Project 01 FPGA.	-
	line_buffer_module.sv	Line buffer module for Convolution operations.	-
	maxpool2x2_stream.sv	Streaming unit for Max-Pooling processing.	-
	relu_unit.sv	Processing unit for the ReLU activation function.	-
	rom_image_mem.sv	ROM module for loading the test image.	**[Modify]** Update the $readmemh statement: use input_img.hex for single-sample testing and test_000_input.hex for 100-sample verification.
	systolic_array_4pe_k3.sv	4-PE systolic array for 3x3 kernel operations.	-
Verilog_TB	conv1_top_tb.sv	Unit testbench for Layer 1.	**[Modify]** Verify the file path in the $readmemh statement.
	fcl_tb.sv	Unit testbench for the FC layer.	**[Modify]** Verify the file path in the $readmemh statement.
	fpga_top2_npu_integrity_tb.sv	Integrity verification testbench for the entire system.	**[Modify]** Verify the file path in the $readmemh statement.
	fpga_top2_tb.sv	This TB focuses on	-

Category	File Name	Technical Description	Modification Points & Guide
		checking Verification Scenarios (User Button/Switch sequence) for FPGA implementation.	
Vivado_xdc	arty_conv1_demo.xdc	Physical pin mapping constraints for Arty A7 board.	-

02_project_100_samples (Mass Verification)

This stage involves automatic system reliability verification using a set of 100 images.

Category	File Name	Technical Description	Modification Points & Guide
Golden_Data	conv1_bias.hex conv1_weights.hex fc_bias.hex fc_weights.hex	Trained weight/bias data for the 100-sample set. 4 weight/bias files for 100 input image testing	Must be added to Design Sources
	test_000_input.hex to test_099_input.hex	Data set containing 100 input images .	1.test_000_input.hex must be added to Design Sources. **2. [Modify]** Update $readmemh in rom_image_mem.sv to point to test_000_input.hex.
	test_000_golden.hex to test_099_golden.hex	Golden result files for each of the 100 images .	-
	test_list.txt	Index list mapping filenames to their golden labels.	-
Python_Source	01_mnist_npu_train_gen.py	Generator for the 100-sample data and the test list.	-

Category	File Name	Technical Description	Modification Points & Guide
	02_mnist_npu_hw_patch.py	Tool for automatically patching TB paths.	**[Optional]** If auto-detection fails, set the BASE_DIR variable.
Verilog_TB	fpga_top2_100_auto_tb.sv	Testbench for 100-sample loop verification.	**[Modify]** The number of NUM_TESTS parameter and BASE_DIR variable.

03_project_ISA_processor (ISA NPU System)

This stage implements a programmable NPU system based on a custom Instruction Set Architecture (ISA).

Category	File Name	Technical Description	Modification Points & Guide
Python_Compiler	isa_compiler.py	Compiler to translate ISA source into machine code.	-
	program.isa	High-level NPU control sequence defined by the user.	**[Optional]** Design custom command sequences for different scenarios.
	program.hex	Compilation output (Machine code binary).	-
	program.asm	Assembly view generated during the compilation process.	-
	compile_report.txt	Debugging report containing instruction mapping.	-
Verilog_RTL	fpga_top2_isa.sv	**[Top]** Top-level module for the ISA processor system.	-
	instruction_rom.sv	Memory module that	**[Modify]** Ensure the

Category	File Name	Technical Description	Modification Points & Guide
		stores the program.hex.	path in initial $readmemh points to the correct **program.hex**.
	isa_controller_v2.sv	Core control unit for instruction fetch and decode.	-
	opcode_defs.sv	Opcode definitions shared between compiler and hardware.	-
Verilog_TB	fpga_top2_isa_tb.sv	Integrated verification for the ISA system.	**[Modify]** The number of NUM_TESTS parameter and BASE_DIR variable.

4. Path Configuration & Troubleshooting Guide

"More than 90% of errors during NPU hardware simulation and data generation stem from File Path mismatches." This section provides a guide to understanding file loading logic and modifying paths to suit your specific environment.

4.1. Choosing Your Path Strategy

Readers can choose between the following two strategies to manage paths depending on their system environment.

Strategy	Definition	Pros & Cons
Relative Path	Paths based on the current file location (e.g., ../Golden_Data/).	**Pros:** Portability; paths remain valid if the project folder is moved. **Cons:** Execution errors may occur depending on where the simulator is launched.
Absolute	The full path from the drive root	**Pros:** Maximum reliability

Strategy	Definition	Pros & Cons
Path	(e.g., C:/NPU_AI/project/...).	regardless of the execution context. **Cons:** All paths must be manually updated if the folder structure changes.

4.2. Path Modification Points

If the simulation fails to start, "File Not Found" errors occur, or if **the FPGA output differs from the simulation**, check the $readmemh or open() statements in the following files and correct the paths.

Stage	Target File	Code to Inspect	Note
Project 01	fpga_top2_tb.sv	Load paths for weights (.hex) and input images.	Verify $readmemh statements for Testbench verification.
Project 01/02	**rom_image_mem.sv**	**$readmemh for input image.**	**[Critical]** Ensure the path points to the specific test image (e.g., test_000_input.hex) for FPGA synthesis.
Project 02	02_mnist_npu_hw_patch.py	base_path variable setting.	Input your local project root path for automatic patching.
Project 02	fpga_top2_100_auto_tb.sv	Load path for BASE_DIR variable.	Check the file indexing loop for mass verification.
Project 03	instruction_rom.sv	Load path for program.hex.	Match with the compiler's output directory.
Project 03	fpga_top2_isa_tb.sv	Load paths for golden verification data.	Ensure the path matches the Python generator's output.

4.3. Quick Troubleshooting Checklist

- **Simulator Execution Context:** Ensure your simulator is running inside the Verilog_TB directory.
- **Compiler Output Verification:** For Project 03, verify that program.hex has been successfully generated in the Python_Compiler folder before attempting the hardware simulation.
- **[New] Simulation-FPGA Mismatch:** If the simulation passes but the FPGA hardware produces incorrect results (e.g., Output '1' instead of '7'):
    - **Verify RTL Source:** Check if the $readmemh statement in rom_image_mem.sv is pointing to the correct image file.
    - **Check Synthesis Dependency:** Vivado may not automatically detect changes in .hex files. Re-run **Synthesis** and **Implementation** after any data file modification.
    - **Bit-Exact Consistency:** Ensure that the same weight/bias files are used for both the Python Golden Model and the FPGA Bitstream.

Epilogue: From Logic to Silicon

We have been on a long journey together at the peak of the AI era. We worked hard to build a 'Vessel for Intelligence.' Now, as we close the last page, let's look back at our path and look forward to the future.

Engineering Milestones We Achieved Together

This book was not just a manual on 'how to make an NPU.' We explored *why* AI needs new hardware, moving beyond the old computer structures. In Part 1, we saw the shift to 'Software 2.0.' In Part 2, we learned how engineering can turn physical limits—like energy use and memory speed—into intelligence.
In Part 3, we went through the tough process of 'Quantization.' We moved Python's exact numbers into the strict world of 8-bit integers. We built real circuits step-by-stitch using a 'Bit-True' strategy, allowing zero errors. Finally, in Part 4, we gave the hardware its own 'Language (ISA)' and built a compiler. We successfully made a Full-Stack AI Processor that can judge and calculate on its own.

A Round of Applause for You

In engineering, there is a big gap between 'understanding' something and 'building' it. You probably faced many errors and frustrations while turning flexible Python code into rigid hardware signals.
But if you felt a thrill when you saw the text "SUMMARY: PASS=100" after overcoming those walls, you have grown. You are no longer just a simple developer; you are now an 'NPU Architect' who sees the whole system. I send you a big round of applause for your patience and passion. You didn't give up.

A Foundation for the Future

Now, the knowledge you gained from this book should not stay only in your head. Every time you optimize a MAC unit or a data path, you help save energy. This brings humanity one step closer to a Sustainable AI Era.
Right now, the semiconductor industry is changing from focusing on 'Computing' to focusing on 'Data.' I truly hope your hard work becomes the seed for the next generation of high-speed NPU processors built in Korea. I look forward to the day when your designs become the control centers of the future, making the world a better place.

I wish you great success.


HBM
NPU
INTELLIGENT CONTROL
MAC ARRAY
Roger Kim
AI NPU System Design with Python and Verilog
Full Source Code Included
Python & SystemVerilog
AI NPU System Design with Python or Verilog