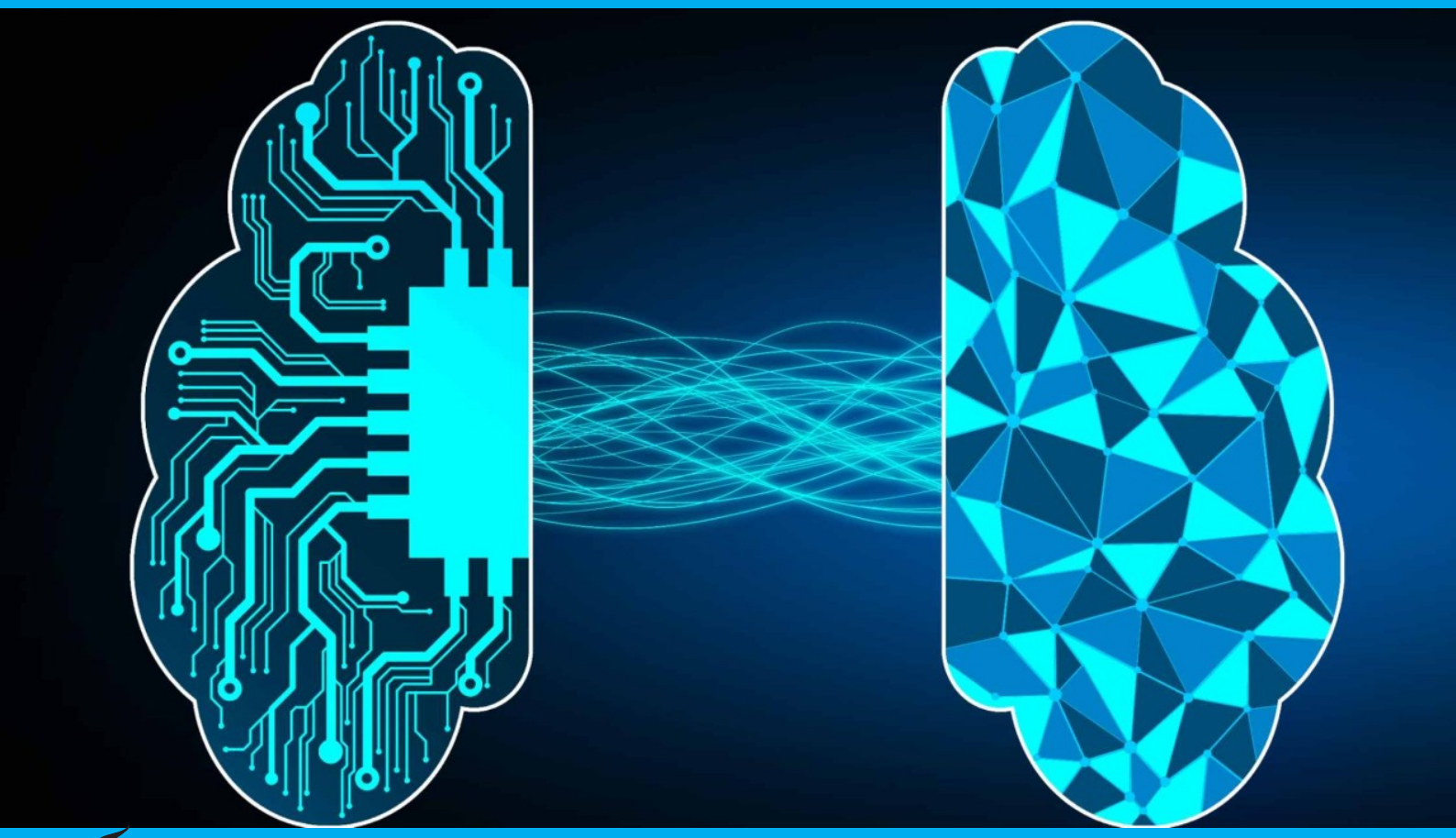
Hardware Accelerator Design Based on Hierarchical Temporal Memory

Ruopu Wu



TU Delft

Hardware Accelerator Design Based on Hierarchical Temporal Memory

by

Ruopu Wu

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday, August 26, 2025 at 11:00 AM.

Student number:	5925010	
Project duration:	September 1, 2024 – September 1, 2025	
Thesis committee:	Dr. ir. C. Frenkel,	TU Delft, supervisor
	Dr. ir. R. Bishnoi,	TU Delft
	M.Sc. D. Casnici	TU Delft, daily supervisor
	M.Sc. D. Layh,	TU Delft, daily supervisor

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

The rapid growth in the energy consumption of artificial intelligence (AI) models has made low-power, high-efficiency, brain-inspired computing hardware a central research focus. Hierarchical temporal memory (HTM) offers robustness and energy efficiency via its spatial pooler (SP) and the use of sparse distributed representations (SDRs), but its temporal memory (TM) is highly sensitive to input perturbations. Moreover, most existing hardware efforts target isolated modules rather than a complete, efficient inference loop. To address these issues, we employ the HTM SP as a sparse encoder and replace TM with deep neural networks, proposing a dual-path SP+FCNN/RNN framework. On sequential MNIST, this scheme achieves 96.5% classification accuracy and markedly improves generalization.

Targeting the three computational bottlenecks revealed in the algorithmic analysis: SP overlap dot products, k-WTA sorting, and neural network multiply-accumulate operations, this work designs a 25-lane parallel binary dot-product accelerator (MAC-SP), a hierarchical min-heap sorting accelerator (Heap-Sort), and a 16×16 systolic-array accelerator (MAC-NN). All three accelerators are integrated, with a unified (memory-mapped input/output) MMIO interface, into the open-source CROC SoC built around an Ibex core. Logic synthesis in a 40 nm process shows that the complete SoC occupies only $0.112\,mm^2$, consumes 1.078 mW, and achieves a maximum clock frequency of 400 MHz. Relative to the pure software baseline, the three core operations achieve speedups of 28.7×, 23.5×, and 38.1×, respectively. End-to-end inference for a single MNIST image drops from 3.78×10^7 to 1.43×10^6 clock cycles, for an overall 26.4× speedup.

Contents

Abstract iii

1 Introduction 1

2 Background 5
2.1 Fully Connected Networks . . . 5
2.1.1 McCulloch-Pitts (MP) Neuron Model . . . 5
2.1.2 Perceptron . . . 6
2.1.3 Multilayer Perceptron and Fully Connected Neural Network . . . 7
2.2 Recurrent Neural Network . . . 9
2.2.1 LSTM . . . 10
2.2.2 GRU . . . 10
2.2.3 minGRU . . . 11
2.2.4 Enhancing RNN Performance with Attention Mechanism . . . 12
2.3 Hierarchical Temporal Memory . . . 14
2.3.1 Sparse Distributed Representations . . . 15
2.3.2 Comparison of High-Dimensional Representations: HDC vs. SDR . . . 17
2.3.3 Spatial Pooler . . . 17
2.3.4 Temporal Memory . . . 21
2.4 Existing Hardware Implementations of HTM . . . 24
2.4.1 Reconfigurable Neuromorphic HTM architecture . . . 24
2.4.2 Scalable Memristor Cross-Array for HTM Spatial Pooler . . . 26
2.4.3 Challenges in HTM Hardware Implementation and Future Design Directions . . . 28

3 Methodology 31
3.1 Research Objectives . . . 31
3.2 Software Architecture . . . 31
3.2.1 Hardware Implementation Methodology . . . 32
3.3 Evaluation Method . . . 32
3.3.1 Performance Metrics . . . 32
3.3.2 Experimental Dataset . . . 33
3.3.3 Baselines . . . 34

4 Software Implementation 35
4.1 Implementation of HTM . . . 35
4.2 Combination of SP and FCNN . . . 38
4.3 Combination of SP and GRU . . . 39
4.4 Implementation of minGRU . . . 41
4.5 Applying the Attention Mechanism . . . 42
4.6 Comparison for Hardware Implementation . . . 43

5 Hardware Implementation 47
5.1 System overview . . . 47
5.2 Detailed Design of Accelerator Blocks . . . 49
5.2.1 Architecture of the SP Accelerator . . . 49
5.2.2 Architecture of the Heap Sort Accelerator . . . 53
5.2.3 Architecture of the MAC-NN Accelerator . . . 57
5.3 System Integration and Functional Verification . . . 60
5.4 Hardware Synthesis and Resource Utilisation Analysis . . . 63
5.5 Design Comparison with Prior HTM Work . . . 64

6 Conclusion and Future Work **67**
6.1 Conclusion 67
6.2 Future Work 67

A Appendix **69**
A.1 Spatial Pooler Parameters 69
A.2 Temporal Memory Parameters 70

List of Figures

1.1 Estimate for AI energy consumption for 2025-2030 [13]. 2

2.1 The model of an MP neuron, where $x_1, x_2, ..., x_i, ..., x_n$ are the input data to the model; $\omega_1, \omega_2, ..., \omega_i, ..., \omega_n$ is the calculation parameter of the neuron model, which is used to respond to the weight of each dimension of the input data; θ denotes the output threshold of the neuron, y is the output result of the neuron model [24]. 6

2.2 Illustration of a single-layer perceptron. The input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ is linearly combined with the corresponding weights α_i and a bias term β, forming the weighted sum. The result is then passed through an activation function to produce the final output 7

2.3 Structure of a fully connected neural network with two hidden layers. Each layer applies a weight matrix (W_1, W_2, W_y) and a non-linear activation function to transform the input vector $\mathbf{x}$ into the final output y [32]. 8

2.4 Illustration of an LSTM cell across three time steps ($t-1$, t, $t+1$), x_t is the input at t time step, and y_t is the output. The cell state maintains long-term dependencies, while the forget (F_t), input (I_t), and output (O_t) gates regulate information flow. The candidate state ($\tilde{C}_t$) is updated via a combination of the gates, allowing effective handling of long-range dependencies and mitigating the vanishing gradient problem [43]. 11

2.5 Illustration of a GRU, showing the update (Z_t) and reset (R_t) gates controlling the hidden state (H_t). The candidate hidden state ($\tilde{H}_t$) is generated via the reset gate and combined with the previous hidden state to capture long-range dependencies efficiently [43]. 12

2.6 Comparison between GRU and minGRU. The standard GRU (left) consists of an update gate z_t, a reset gate r_t, and a candidate hidden state $\tilde{h}_t$, where r_t modulates the contribution of the previous hidden state h_{t-1} in the computation of $\tilde{h}_t$. In contrast, minGRU (right) removes the reset gate, simplifying the computation of candidate hidden states to depend solely on the current input x_t [45]. 12

2.7 A model of the attention mechanism in a sequence-to-sequence framework. At each decoding step t, the model computes attention weights $\alpha_{t,i}$ by comparing the decoder state s_{t-1} with each encoder hidden state h_i. These weights determine how much each h_i contributes to the context vector, which is then combined with the decoder state s_{t-1} to update s_t [50]. 14

2.8 An illustrative example of a SDR. The figure shows a binary vector containing 2000 bits, with only a small fraction of bits set to "1" (active), while the remaining bits are "0" (inactive). 15

2.9 Illustration of overlapping active bits between SDRs. SDR A is used as the reference vector with $n = 1024$ and $w = 10$. SDR B overlaps with SDR A on 5 active bits, which exceeds the threshold $\theta = 4$, thus SDR B $\in \Omega_A(n = 1024, w = 10, b = 4)$. In contrast, SDR C only overlaps with SDR A on a single active bit, so SDR C $\notin \Omega_A(n = 1024, w = 10, b = 4)$. 15

2.10 Illustration of the union property of SDRs. The figure shows multiple individual SDRs. By forming a bitwise logical OR operation (union) of these SDRs, a new composite SDR (Union_SDR) is generated, which preserves critical information from all constituent SDRs. Due to the inherent sparsity and distributed nature of SDRs, membership tests can be efficiently performed by verifying the overlap of active bits, as highlighted by the green rectangles. 16

2.11 Architecture of HTM. **Left**: Input data are encoded through stacked levels, where cells form vertical columns and only a sparse subset of columns becomes active. **Right**: An enlarged column shows each HTM cell modeled after a cortical pyramidal neuron, receiving feed-forward input on proximal dendrites (brown), lateral context on distal dendrites (red), and top-down feedback on apical dendrites (blue). This arrangement allows HTM to combine spatial pattern encoding with context-dependent temporal prediction [70]. . 18
2.12 The learning process of spatial pooling. It shows that each column has a corresponding receptive field, or region, called potential connections. The connection weights between this region are modified during the learning process. Also, it shows the local inhibition function in the algorithm. Within a certain range of an active mini-column, another active column will not appear [63]. 19
2.13 The figure shows the curve of the relationship between activation frequency and boost factors. In the SP module of HTM, the system applies a 'boosting' mechanism to inactive columns to ensure that all columns have the opportunity to participate in representing the input. The horizontal axis shows the average activation frequency of a column (usually calculated using a sliding window), and the vertical axis shows the boost factor for that column. A higher boost factor indicates that the column will be enhanced to improve its competitiveness [71]. 21
2.14 The figure illustrates the encoding and prediction processes of the Temporal Memory module in HTM for sequential inputs at various time points. Each column represents a spatial location and contains multiple cells representing different temporal contexts. The figure shows three time points: $t-1$ (first pattern), t (current time point), and $t+1$ (last pattern). These are arranged from left to right to indicate the progression of time. . . . 22
2.15 This figure illustrates how the HTM algorithm distinguishes between input sequences in different contexts [64]. It specifically shows the changes in state before and after learning. (A) Structural overview: Each column contains multiple cells, represented by small dots in the figure, which represent different temporal contexts at the same spatial location. (B) Before learning: Before learning, the model can only recognise spatial patterns. This means that patterns A, B, C, and D, as well as patterns X, B, C, and Y, activate the same columns. This indicates that the same SDR data is received. All cells in each column are activated, forming bursting columns. (C) After learning: After temporal learning, the model activates different cells for each context. Only the first input has a bursting column. Now, 'B' activates different cells in different contexts (labelled 'B' and 'B', respectively), rather than activating the entire column. 24
2.16 The figure shows the architecture of the MCU: The encoder writes the input sequentially into the input buffer in 8-bit packets. Subsequently, each column is returned via a 7-bit pipeline bus with 4-bit overlap values and cached in the overlap RAM. The built-in k-winner-take-all comparison chain selects the active columns based on this, writing the 7-bit column index and 11-bit column-cell identifier to the Winning Columns RAM and Winning Cells RAM, respectively. Finally, the results are broadcast to the column array via an 11-bit H-Bridge bidirectional network, forming an efficient closed-loop data flow from encoding to learning. In the memory-bank labels, the notation rows Œ word-width (bits) is used: for example, Winning Columns RAM 20 Œ 7 holds 20 words, each 7 bits wide [19]. 25
2.17 Memristor-crossbar realisation of an HTM SP. A 20 × 20 MNIST image is applied to the 400 input rows ($X_0 X_{399}$). Each vertical slice represents an HTM column with 25 potential synapses stored as low-resistance (black, LRS) or high-resistance (white, HRS) memristors. Column currents are modulated by programmable boost resistors (b) and fed to local winner-take-all circuits; the earliest-spiking 2% columns form the SDR at the output [20]. 27

3.1 Proposed software implementation. The original MNIST image undergoes binarization and is then fed into the SP to generate an SDR. The resulting SDR is evaluated by four classification modules in parallel: the original TM of the HTM algorithm, FCNN, GRU, and minGRU. The outputs of each classifier are used to compare recognition performance under the same input encoding. 32

3.2 Handwriting Images in the MNIST Dataset. 33

4.1 The diagram illustrates the inference process of the HTM algorithm. First, the binarized MNIST data is fed into the SP module. As shown, the SP generates an SDR of a given length, where each bit corresponds to a column in the TM module. When a bit in the SP output is set to 1, it indicates that the corresponding column in the TM should contain active cells at that particular time step; if no active cells are present, the column enters a bursting state. In addition, the active cells in the TM at the current time step evolve from the predictive cells generated at the previous time step. If the active columns match the predicted columns, the prediction is considered correct. Finally, active cells are used as the output of this time step. 36

4.2 Comparison of the MNIST image before and after binarisation. On the left is the original image and on the right is the image after binarisation. It can be seen that the image retains all the original information. 36

4.3 Prediction accuracy of HTM over time when processing Sequential MNIST images. The blue curve corresponds to predictions made on a digit 0 image across 30 time steps, while the red curve shows results on a digit 5 image. Accuracy is computed at each step based on overlap with the expected SDR for that digit. 37

4.4 For digit 5, the accuracy of the training image and one test image. For each similar but slightly different time step, relatively high accuracy can be achieved at the current time step, but these differences cause the new columns to enter a bursting state, which causes a significant drop in the accuracy of subsequent predictions. Additionally, the accuracy curve does not commence at time step 0. This is because no data is included in the first two rows of the MNIST image and no corresponding output is produced. 38

4.5 The diagram illustrates the combination of SP and FCNN. In the first instance, the MNIST image is binarized and input into SP, which is a departure from the previous method. Rather than dividing the MNIST image into 28 rows and inputting them sequentially, it is inputted into SP at one time. Subsequently, the SDR data generated by SP is input into FCNN. It is evident from the figure that the size of the FCNN input dimensions is equivalent to the length of the SDR data. Finally, the classification output is generated by the FCNN. 39

4.6 The relationship between the accuracy of the model integrating SP with FCNN and the size of the SDR produced by SP is demonstrated, where 79×79 is the baseline given in the source code [63]. It is evident that as the SDR size increases, the accuracy concomitantly improves. However, once the SDR size attains 128, no further enhancement in accuracy is discernible. 40

4.7 Illustration of how the output of the SP is converted into the input for the GRU. First, the binarized MNIST image is fed into the SP, which produces the corresponding SDR. Then, the indices of the active bits (1 bits) in the SDR are extracted and fed into the GRU model, with each index serving as the input for a single time step. 41

4.8 The diagram illustrates the process of generating classification results using GRU. Initially, the inputs are arranged in time steps, with the GRU module, in each time step, accepting the hidden state of the inputs from the previous time step, in addition to the inputs from the current time step. Ultimately, the hidden state of the final moment is output to the FCNN to generate the final classification results. 41

4.9 This figure demonstrates the change in the accuracy of the minGRU model before and after using the attention mechanism. The SDR size refers to the size of the SDR data that the minGRU model receives as output from the SP. It is evident that the accuracy of the minGRU model with SDR sizes of 784 and 4096 is enhanced through the integration of the attention mechanism, thereby substantiating the efficacy of the attention mechanism in conjunction with the minGRU model. 42

4.10 Figure (a) and Figure (b) show the accuracy (%) versus model parameter sizes, and the accuracy versus inference latency for different models (Baseline, FCNN, GRU, and minGRU), respectively. For the calculation of inference latency, the implementation hardware is assumed to be capable of performing 1024 MAC parallel computations per cycle with a memory bandwidth of 1024 bytes/cycle. Values in parentheses (e.g., 28×28) indicate the size of the input dimension used by the model. 44

5.1 Execution Cycles of SP-FCNN Algorithm on Ibex Core. It can be observed that the MAC operations in the SP component consume the largest number of cycles. This arises because the SP portion involves 4096×784 MAC operations, which cannot be executed in parallel on the Ibex core and therefore must be processed sequentially within a for loop, leading to significant overhead. Similarly, the MAC operations in the FCNN component account for the second-largest number of cycles. Consequently, our subsequent optimization efforts focus primarily on parallelizing these MAC operations. 48

5.2 This diagram illustrates the hardware system architecture and memory address mapping designed in this study. Based on the open-source RISC-V Ibex Core, the system uses an OBI crossbar to efficiently interconnect the instruction memory (512 KB), data memory (1 MB) and peripheral modules (UART, GPIO, timers, etc.). The blue area on the right represents the dedicated hardware accelerator, comprising MAC units and sorting modules for the SP and NN, as well as dedicated weight SRAM. Memory address space allocation for each module is shown on the far right, enabling the central processor to schedule and access them uniformly. 49

5.3 This is a block diagram of the SP column accelerator. Data is supplied to 25 parallel PEs from a 32-bit input FIFO and a 128-bit weight FIFO. Each PE performs bitwise AND operations and pop count, and its 6-bit output is reduced to a 16-bit dot product through an adder tree and cached in the output FIFO. A control state machine coordinates four pipeline stages: the simultaneous fetching of instructions A and B; the fetching of instruction B; decomposition; and the calculation of the output. 51

5.4 The six-state FSM of the SP accelerator. After the IDLE state, the controller enters the 'Fetch Inputs and Weights' state, initiating a three-cycle micro-loop. First, it issues a 128-bit SRAM read. Then, it moves to the 'Wait Weights from SRAM' state for the one-cycle latency. Finally, it reaches the 'Decompose Weights Vector' state, where it splits the returned 128-bit word into four 32-bit chunks. It then returns to the 'Fetch' state. This loop repeats until `weights_done` equals 1. The FSM then remains in the 'Fetch Inputs and Weights' state until `inputs_done` is also asserted, acquiring inputs only. Once both vectors are ready, the machine jumps to 'Process and Output', where 25 parallel PEs and an adder tree produce the dot product. Finally, it transitions through 'Finish' back to 'IDLE' to start the next column. 52

5.5 This is the waveform during the first half of the SP accelerator's operation after startup. After the start trigger, the state machine transitions from IDLE to a three-cycle pipelined flow comprising the 'fetch inputs and weights', 'fetch weights wait', and 'Decompose weights' stages. The index of the input vector increments in each 'Fetch inputs and weights' section, while the index of the weight vector is only updated in the 'fetch weights wait' section. This indicates that the 1-cycle delay of the 128-bit weight read is fully hidden within the fetch pipeline. 53

5.6 The waveform is produced after the SP accelerator has completed the data retrieval process. Once the weight index count is fixed at 6 (i.e., once 7 blocks have been retrieved), the state machine remains in the 'fetch input and weights' state and continues to retrieve the remaining 18 input blocks. When the index of the input vector reaches 18 and the `inputs_done` and `weights_done` conditions are satisfied, the FSM sequentially transitions to Process, then Output, and finally Finish. The result signal is pulled high at the first output point, followed by the busy signal being set to 0 and the state machine returning to IDLE. This verifies that the fixed delay for a single column is 25 clock cycles. 53

5.7 This is a schematic diagram showing the execution process of the heap sort algorithm [92]. The figure shows the various stages of the maximum heap sort process. Starting with the initial construction of the maximum heap (a), the top element is exchanged with the last element and sunk to maintain the maximum heap structure (bj). Each step moves the current maximum value to the end of the final sorted array. Figure (k) shows the final sorted result. Grey nodes represent elements that have already been sorted, while italicised nodes indicate elements currently involved in swapping or sinking operations. . 54

5.8 This diagram illustrates the process of data comparison and exchange within the Sort Node. It shows the specific steps involved in data exchange by the Sort Node during heap sorting in detail. In the initial stage, the current node (N) receives updated data from the upper node (U). The comparison stage then begins, determining the size relationship between the current node (N) and its left and right child nodes (L and R) to establish whether an exchange is necessary. Based on the results of the exchange, the next layer of nodes is notified to perform further data updates. 55

5.9 The hardware structure of a heap sorter is implemented to illustrate the transformation from independent storage elements to a RAM-based hierarchy. The top-level sorting node manages both the input and output data streams uniquely [95]. 56

5.10 Hardware architecture of a hierarchical minimum heap sorter implementation. Each level consists of a sort node that is responsible for comparing and updating heap elements, as well as dual-port SRAM blocks that enable parallel data access and address control logic that manages memory transactions. Update information is propagated upwards through the levels to dynamically maintain heap properties. The topmost level (Level 0) interfaces directly with external input and output, coordinating heap insertion and extraction operations. 57

5.11 Simulation waveforms and state diagrams of a 2-level minimum heap sorter during the initial stage of constructing the heap structure. As mentioned earlier, we only have one sort node on each layer. The upper figure shows how the sorting nodes behave when performing sequential comparisons and writing to the double-port RAM with input data of 0x209, 0x58, and 0x77. The `left_memory_we` and `right_memory_we` signals at Levels 0 and 1 perform valid write operations as the data sinks, while the `next_level_update` signal transmits the sorting results. The lower figure shows the corresponding heap structure diagram at that moment, illustrating the sorting process whereby larger values sink down and smaller values float up, in line with the heap construction logic shown in the diagram: If the child node is smaller than the current node, swap and update the next level. To save hardware resources, it is important to note that the implemented min-heap structure only requires the root node to be less than or equal to the left and right child nodes, without enforcing the size relationship between them. 58

5.12 The simulation waveform and schematic diagram of the heap structure changes during the flush phase of the min-heap sorter. The upper half of the figure shows the simulation waveform, illustrating the complete process: after the `flush_signal` is pulled high, the top elements of the heap are popped out one by one; the maximum value, `0xFFFFFFFF`, is pushed into the top of the heap; and then it propagates downward along the path. The `flush_cnt` corresponds to the count of each output; `output_data` outputs the sorted data; and `output_valid` indicates the validity of the output. The lower half of the figure shows the heap structure changes that correspond to each step of the flush process. Arrows indicate the propagation paths of smaller values rising and the maximum value sinking during the heapification process. Using the first flush as an example, once the top element (12) has been output, the maximum value propagates downwards and eventually settles in the left subtree RAM of the right child node at level 1. Subsequently, smaller values (such as 25, 53, and 58) float up to the top of the heap in sequence. 59

5.13 3×3 systolic array: the left onchip memory bank feeds input data row by row, which shift rightward across the PE grid every cycle; weights are injected at the top and stream downward column by column, multiplying with the current inputs in each PE and accumulating into partial sums that continue flowing downward until the bottom memory bank collects the final results [102]. 60

5.14 Timing diagram of a 3×3 systolic array multiplying two matrices: at $t = 0$, the first-row input vector is injected from the left and aligns with the first-column weights in the upper-left PEs to perform the initial MACs. Each subsequent cycle shifts the entire input wave one column to the right and the weight wave one row downward; the partial sums stored in each PE are updated with the new product from the left and passed to the PE below. At $t = 4$, the data wavefront has filled the array, after which a new final result emerges from the bottom-right PE every clock cycle [102]. 60
5.15 Block diagram of the MAC-NN systolic array accelerator. A controller (FSM) fetches 32bit weights from the external Weight SRAM into the Weight Buffer and streams 32bit inputs into the array via the Input Buffer; weights enter from the top, inputs from the left, and each PE performs MAC operations while propagating partial sums downward to the Result Buffer for output. 61
5.16 Illustration of the FSM of the MAC-NN accelerator. In `READW`, weights are fetched from the SRAM and stored in the weight buffer; in `READI`, the inputs are written into the corresponding input buffer. During CALCULATE, if `CAL_CNT` has not yet reached sixteen, the computation continues and intermediate results are stored in the result buffer. Finally, once computation is complete, all results are output sequentially from the result buffer. 61
5.17 Illustration how the CPU configures the registers of the MAC-SP accelerator via the OBI bus. First, the CPU writes the value 0x10 to address 0x04 (`ADDR_VEC_B`), setting the base address of vector B. Then, it writes the value 0x01 to address 0x08 (`ADDR_NUM_VEC`), specifying that one vector is to be processed. Each write operation is completed via the gnt handshake signal, and the corresponding values are latched in the internal registers `vec_B_addr_reg` and `num_vec_B_reg`. This indicates that the CPU can initialise and control the accelerator using standard STORE instructions alone. 62
5.18 This diagram illustrates the accelerator's entire start-up process, from the CPU writing the start command to address `0x00` (`ADDR_CTRL`) to outputting results. The CPU uses the STORE instruction to write the value 0x1 to the control register, which triggers the start pulse. The `MAC_SP_accel_busy` signal is then pulled high, indicating that the accelerator has started processing. Once the calculation is complete, the `MAC_SP_result_valid` signal is pulled high and the final dot-product result (`0x0188` in this case) is output via the FIFO. 63

List of Tables

2.1 Key Differences Between HDC and SDR 17

4.1 Comparison of GRU and minGRU Models on MNIST Dataset 42
4.2 Comparison of Different Models 45

5.1 Key modules of the hardware system and their specifications 50
5.2 Chunking parameters for the SP-column accelerator 50
5.3 MMIO Register Mapping of the MAC-SP Accelerator 62
5.4 CROC SoC Synthesis Summary 63
5.5 Speedup of Hardware Accelerators Compared to Ibex CPU 64
5.6 Clockcycle count for a single MNIST inference, before and after adding hardware accelerators 64
5.7 Comparison between the digital HTM–SP core by Zyarah *et al.* [19] and the CROC SoC proposed in this thesis 65

A.1 Parameters of Spatial Pooler. 69
A.2 Parameters of Temporal Memory. 70

1

Introduction

AI is currently one of the hottest research fields [1]. Since its inception, it has been linked to humanity's curiosity about, and exploration of, the nature of its intelligence. The human brain's exceptional sensory perception, memory, learning, and decision-making capabilities demonstrate far greater efficiency and adaptability than traditional computers in complex, dynamic real-world environments. This inherent advantage has sparked enthusiasm to understand and replicate its structure and functions to achieve similar or even superior intelligent performance in artificial systems.

Early research in AI focused on abstract models of biological neurons. The McCulloch-Pitts (MP) model [2] was among the first to translate neuronal signal processing into mathematical logic, laying the foundation for artificial neural networks (ANNs) [3]. Building on this, the Perceptron introduced a learnable structure. The development of the backpropagation (BP) algorithm [4] further enabled effective learning and generalization in neural networks.

With the growth of data availability and computing resources in the 21st century, deep neural networks (DNNs) have achieved significant progress. Architectures such as convolutional neural networks (CNNs) [5] and recurrent neural networks (RNNs) [6] have demonstrated strong performance in fields including image processing, speech recognition, and natural language understanding [7].

In several domains, AI systems have matched or surpassed human-level performance. For example, AlphaGo defeated expert Go players [8], showcasing the potential of AI in game theory. In medical imaging, AI can detect early signs of disease with high accuracy [9]. Autonomous driving systems have also achieved favorable results under specific traffic scenarios [10]. These outcomes underscore the practical value and ongoing research momentum of AI technologies.

Although traditional neural network structures have achieved significant success in many fields, their limitations have gradually become apparent as AI applications have become more complex and diverse. For instance, many traditional neural network architectures exhibit poor generalization ability and are susceptible to interference in dynamic, uncertain, environments [11]. Additionally, these methods require a large amount of data for adequate training, resulting in substantial computational demands [12]. Both the training and inference stages require considerable computational resources and energy. This is in stark contrast to the brain's ability to process information efficiently, quickly, and with low energy consumption in highly complex tasks.

Figure 1.1 illustrates Goldman Sachs Global Investment Researchs projection of worldwide data-centre electricity demand (2015 to 2030). The light-blue area represents conventional workloads (ex-AI), which held roughly steady at about 200 TWh per year until 2019. From 2020 onward, however, the curve bends upward, reaching nearly 800 TWh by 2030. The dark overlay shows the additional demand attributable to AI workloads. Although negligible before 2020, AI power consumption is expected to accelerate rapidly, climbing to almost 300 TWh by the end of the decade and pushing total data-center demand past the 1000 TWh mark.

In this context, deeper research into the neurons and structure of the human brain has led to the emergence of brain-inspired algorithms as a cutting-edge and promising research direction in AI. One such algorithm is the HTM algorithm, which is a typical brain-inspired computational model that aims to better mimic the operational mechanisms of the human brain's neocortex [14]. Rather than focusing on simplistic modelling of individual neurons, the HTM algorithm prioritises simulating

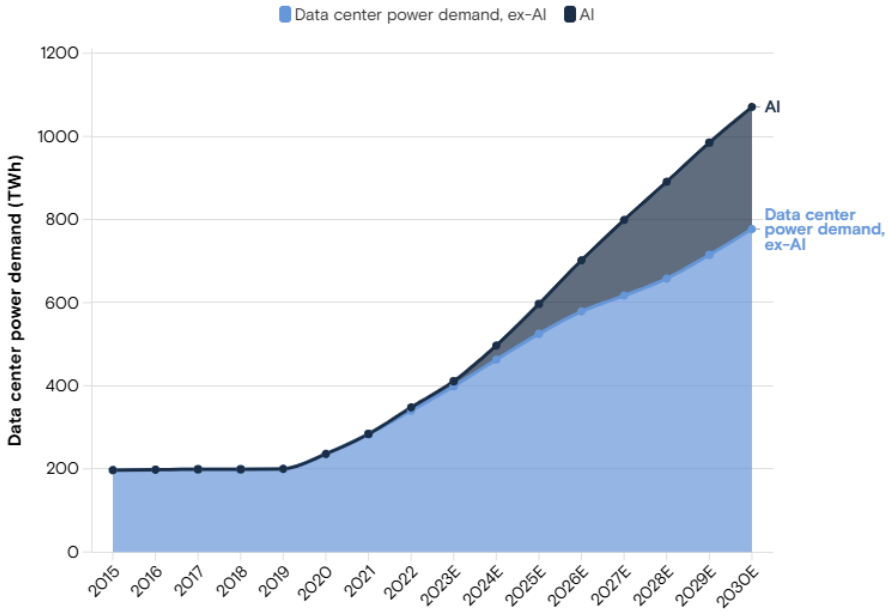


Figure 1.1: Estimate for AI energy consumption for 2025-2030 [13].

the overall structure of cortical regions and their dynamic spatiotemporal information processing. By incorporating the SDR mechanism, it can represent information efficiently and robustly, demonstrating significant advantages in highly dynamic or noisy environments [15].

The two primary HTM components-SP and TM respectively capture spatial patterns and temporal sequences, mirroring key functions of cortical circuitry [16]. Empirical studies show that this architecture adapts to salient features in streaming data and performs well on complex spatiotemporal tasks. Continued investigation of brain-inspired models such as HTM may address generalization and robustness challenges observed in many deep networks, while offering pathways toward low-power, high-performance computing systems that more closely reflect biological principles [17].

Although existing research, such as [18], has validated the feasibility of the HTM algorithm through hardware design, this design has been implemented using memristors, which will encounter challenges for large-scale integration and scalability. Therefore, the primary research objective of this paper is to conduct an in-depth exploration of the HTM algorithm and its implementation in digital circuit accelerators to advance low-power applications of AI technology in complex sequence data processing and real-time computing tasks. Although HTM is theoretically better than traditional ANNs for processing complex spatio-temporal data, its practical deployment still encounters key challenges such as efficient hardware mapping, alleviating computational bottlenecks, and enhancing robustness and generalization. In particular, there is currently a lack of systematic and in-depth exploration into how to efficiently implement the HTM algorithm at the digital circuit level while ensuring its performance in real-time inference scenarios. Therefore, the main research question of this study can be summarized as follows:

Is it possible to design a hardware accelerator that efficiently implements the HTM algorithm, thereby ensuring the robustness and generalization capabilities of the algorithm while meeting the requirements of real-time computing and low power consumption?

To better understand and address the above main research questions, this study further breaks them down into the following specific sub-questions:

1. *Does the SDR mechanism in the HTM algorithm offer significant advantages in terms of reducing hardware resource consumption and improving inference speed?*

2. *Does the HTM algorithm have any flaws? How can these flaws be addressed or optimized?*

3. *What computational bottlenecks exist in the hardware implementation of the HTM algorithm? How can these bottlenecks be efficiently addressed through algorithm-hardware co-optimization?*

4. *How do different hardware architecture design choices (e.g., parallelization strategies) affect the performance and power consumption of the HTM algorithm implementation?*

This paper makes two main contributions. First, it identifies a critical flaw in the HTM algorithm and proposes an effective solution. Specifically, the TM module is overly sensitive to changes in input data, which makes it difficult to generalize similar inputs effectively, ultimately leading to significant differences in output results. This thesis, therefore, proposes replacing the TM module with a deep learning network structure to retain the advantages of the SDR mechanism in data representation and avoid excessive sensitivity. Experimental validation demonstrated that this improved approach achieved an accuracy rate of 96.5% in the MNIST dataset classification task, enhancing the model's performance and generalization capabilities compared to the 86% accuracy rate achieved using the HTM algorithm alone.

Second, this study developed an application-specific integrated circuit (ASIC) accelerator module for the HTM algorithm and successfully integrated it into the open-source CROC system-on-chip (SoC) platform. This achieved efficient hardware implementation and acceleration of the HTM algorithm. This work provides reliable hardware support for the inference process of a single MNIST image of the HTM algorithm by 26.4×, laying the foundation for future brain-inspired algorithms in low-power, real-time computing scenarios.

Background introduces the theoretical foundations of brain-inspired computing and HTM. It delves into the core concepts of the HTM framework, with a focus on SDR. From a mathematical perspective, it establishes quantitative relationships between SDR's sparsity rate, dimension, overlap threshold, and mismatch probability, highlighting the advantages of SDR in terms of noise resilience, fast matching, and the union property. The chapter then splits into two sections detailing the two main HTM modules. For the SP, it introduces steps such as the potential synapse set, overlap calculation, local inhibitory competition, the Hebbian learning rule, and adaptive boosting. The TM section elaborates on the column-cell hierarchical structure, the predictive depolarization mechanism, and the synaptic plasticity rules. The chapter explains how the TM utilises contextual sequences to generate predictions and outlines its natural advantages and limitations in handling long-term dependencies and anomaly detection. Additionally, it introduces traditional deep learning frameworks. Starting with the MP neuron model and the perceptron, it reviews the early development of artificial neural networks. The hidden layer structure, nonlinear activation, and backpropagation algorithm of the multilayer perceptron/feed-forward neural network (FCNN) are then introduced, and their typical characteristics in terms of feature abstraction capability and parameter efficiency are analyzed. Recurrent neural networks are also introduced in this section, which covers the basic ideas of RNNs in processing sequential data and the issues of gradient vanishing and explosion. Then, it introduces the gated memory units of the long short-term memory (LSTM) network, the simplified structure of the gated recurrent unit (GRU) network, and the potential for parallelization of the minGRU network by removing the reset gate. It also explains how the attention mechanism overcomes the fixed-dimension bottleneck and improves the ability to capture long-range dependencies.

Software Implementation introduces the algorithmic process. First, it outlines the implementation details and performance baseline of native HTM on Sequential-MNIST and highlights the issue that the TM module is highly susceptible to input perturbations, resulting in suboptimal classification accuracy. It then introduces a scheme in which HTM-SP acts as a sparse encoder connected to an FCNN. This demonstrates that SDR outputs can accelerate model convergence and improve accuracy while maintaining sparsity. Building on this, the chapter explains how to temporalize the SDR and feed it into the GRU. It explores the advantages of recurrent structures in capturing long-term dependencies and compares standard GRU with simplified minGRU, using parameter count and inference latency as metrics. By removing the reset gate in minGRU, the model's computational complexity is reduced while accuracy is maintained, paving the way for hardware parallelization. To address the expressive bottleneck caused by the fixed dimension of hidden states, the chapter concludes by introducing an attention mechanism on top of minGRU. This enhances the capture of long-range dependency features, further improving classification accuracy under large-scale SDR. Finally, this chapter provides a comparison of the evaluation results of each algorithm in terms of computational complexity, model size,

inference latency, and memory bandwidth. Based on these results, the 'HTM-SP + FCNN' algorithm is selected as the target for subsequent ASIC design. This achieves a closed-loop transition from theoretical analysis to deployable software models and provides clear performance and resource benchmarks for hardware micro-architecture optimization in Chapter 3.

Hardware Implementation This chapter introduces the implementation of dedicated hardware accelerators based on the aforementioned software design. Through function-level cycle profiling of the Ibex RISC-V core, it was found that the three main components overlapping dot product in HTM-SP, k-Winner-Take-All sorting in sparse activation, and FCNN/GRU multiply-accumulate account for over 98% of the runtime. This establishes the design direction for the three custom accelerators. The microarchitecture, data chunking strategies, pipeline timing, and state machine designs of the 25-way parallel binary dot product MAC-SP, the hierarchical minimum heap Heap-Sort and the 16 $\times$ 16 systolic array for MAC-NN are detailed, along with simulation waveforms that verify key features such as fixed 25-cycle latency and online k-WTA selection. Next section explains how the three accelerators are mounted onto the CROC SoC's OBI Crossbar via a unified MMIO register mapping. Finally, we synthesized this design in a TSMC 40nm process, the CROC SoC occupies 0.112 mm^2, operates at a clock frequency of 400 MHz, achieves 26.4$\times$ acceleration and dissipates 1.078 mW in total.

Conclusion summarizes the main research findings of this paper and outlines areas for potential improvement in future research.

2

Background

This chapter provides the theoretical background to the research presented in this thesis. First, Section 2.1 introduces fully-connected neural networks (FCNN), a foundational model in deep learning. Despite their proven capability in pattern recognition tasks involving structured data, FCNN exhibit limitations in processing sequential information, prompting the development of RNNs.

Next, Section 2.2 elaborates on RNNs and their variants, including the LSTM and the GRU. The incorporation of recurrent architectures and gating mechanisms within these models enables them to overcome the limitations of traditional neural networks in processing sequential data, thereby enhancing their ability to model temporal dependencies. Furthermore, integrating the attention mechanism with RNN architectures leads to an enhancement in the ability to capture long-range dependencies, resulting in superior performance in the domain of complex sequence modeling tasks.

Subsequently, Section 2.3 discusses HTM, a biologically inspired model characterized by mechanisms analogous to those of the neocortex. HTM employs SDRs to encode input signals and effectively handle spatial-temporal information robustly. Specifically, the SP is responsible for extracting spatial patterns from input data, while the TM captures temporal dependencies within input sequences.

Finally, existing HTM hardware accelerator designs are summarised in Section 2.4. They are grouped into two categories: FPGA/ASIC designs, and compute-in-memory solutions. Zyarah and Kudithipudi's FPGA prototype integrates SP and TM, introducing 'synthetic synapses' to reduce wiring [19]. However, TM has only been validated in isolation due to resource and power limitations. Truong et al. map the entire SP to a memristor crossbar to demonstrate dense, low-latency analogue processing. However, they omit TM, only rely on SPICE simulation, and still encounter leakage and write-wear issues [20]. We summarize three open challenges from these works: (i) the absence of a full SPTM hardware closed loop, (ii)achieving a balanced trade-off among storage, bandwidth, and power, trade-off (iii) the lack of system-level integration with host SoCs and sensors.

2.1. Fully Connected Networks

The FCNN, also known as the multi-layer perceptron (MLP), is the fundamental architecture in modern deep learning systems. Its theoretical foundations can be traced back to interdisciplinary research in neuroscience in the 1950s. Building on the McCulloch-Pitts neuron model [2] and the Hebbian learning rule [21], Frank Rosenblatt introduced the perceptron model in 1957 [22]. This was the first mathematical formalization of an artificial neural system capable of learning. By abstracting the features of biological neurons into computational units, this model not only opened up the study of artificial neural networks but also laid the mathematical foundations for subsequent, more complex neural network architectures.

2.1.1. McCulloch-Pitts (MP) Neuron Model

The McCulloch-Pitts neuron model was proposed by Warren McCulloch and Walter Pitts in 1943 [2] and is considered to be the earliest mathematical model of artificial neurons. The model attempts to describe the information processing of neurons in the brain using mathematical logic and lays the theoretical foundation for subsequent research on artificial neural networks.

As shown in Figure 2.1, the basic structure of the MP neuron includes multiple inputs, weights, a threshold, and a binary output. Specifically, the neuron receives a set of input signals, each associated with a weight w_i. The neuron first computes the weighted sum of the inputs, given by $S = \sum w_i x_i$, and then compares it with a predefined threshold θ, where $x_i, w_i \geq 0$ for all i. If S is greater than or equal to the threshold, the neuron outputs 1; otherwise, it outputs 0, which can be expressed as:

$$y = \begin{cases} 1, & S \geq \theta \\ 0, & S < \theta, \end{cases} \tag{2.1}$$

Essentially, this model functions as a simple binary classifier, capable of implementing basic Boolean operations such as AND and OR. However, because it uses a step function as its activation function, it cannot handle more complex nonlinear problems, such as the XOR problem [23].

Although the MP neuron is a simplified abstraction of biological neurons and does not account for continuous input-output relationships or learning mechanisms, it laid the groundwork for the development of the perceptron. It also provided an important theoretical basis for later multi-layer neural networks, such as FCNN and CNN. The proposal of this model marks the starting point of research into artificial neural networks and has inspired further research in neural computing and pattern recognition.

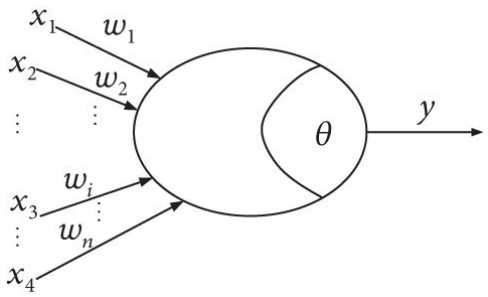


Figure 2.1: The model of an MP neuron, where $x_1, x_2, ..., x_i, ..., x_n$ are the input data to the model; $\omega_1, \omega_2, ..., \omega_i, ..., \omega_n$ is the calculation parameter of the neuron model, which is used to respond to the weight of each dimension of the input data; θ denotes the output threshold of the neuron, y is the output result of the neuron model [24].

2.1.2. Perceptron

The Perceptron is one of the earliest ANN models, proposed by Frank Rosenblatt in 1958 [22]. This model innovatively abstracted the biological electro-transmission mechanism of neurons into a threshold logic unit. Unlike the MP neuron model, the perceptron introduces a weight learning mechanism for learning and training. By using a weighted linear combination of input signals and an activation function, it achieved for the first time a binary classifier based on the supervised learning paradigm [25]. Its lies in the establishment of the trainable architecture paradigm of "input-weight-activation" and the use of an error-driven weight update algorithm, which successfully partitioned the feature space for linearly separable problems. As shown in Figure 2.2, the perceptron consists of the following core components:

- **Input Layer:** Consists of multiple input features $x_1, x_2, ..., x_n$, where each input corresponds to a weight $\alpha_1, \alpha_2, ..., \alpha_n$.
- **Weights:** Each input feature has an adjustable weight that determines its influence on the final decision.
- **Weighted Sum:** The inputs are multiplied by their corresponding weights and summed together. The calculation formula is as follows:

$$S = \sum_{i=1}^{n} \alpha_i x_i + \beta, \tag{2.2}$$

where β is the bias term, which adjusts the position of the decision boundary, and $x_i, \alpha_i, \beta_i \in \mathbb{R}$.

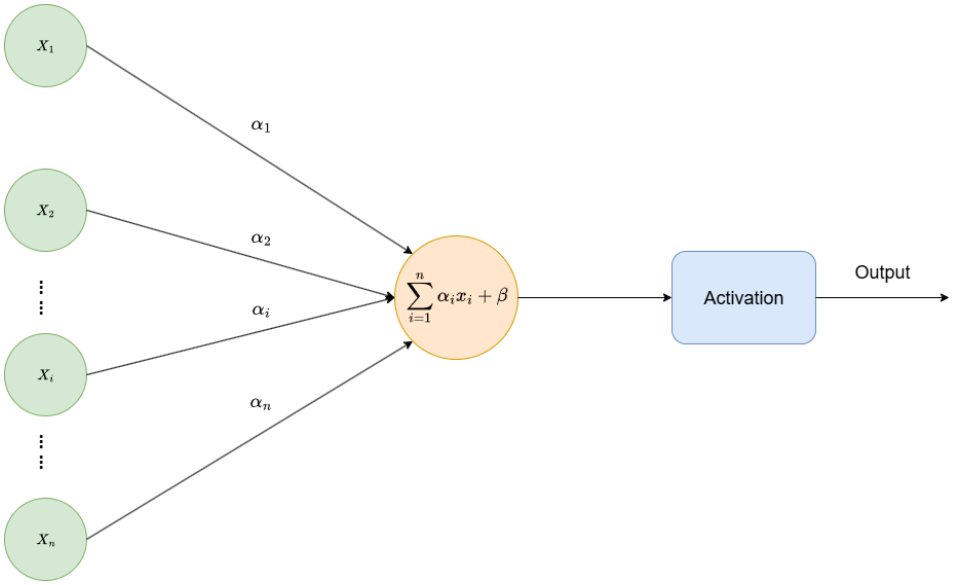


Figure 2.2: Illustration of a single-layer perceptron. The input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ is linearly combined with the corresponding weights α_i and a bias term β, forming the weighted sum. The result is then passed through an activation function to produce the final output

- **Activation Function:** The perceptron uses a step function as the activation function:

$$y = \begin{cases} 1, & S \geq \theta \\ 0, & S < \theta, \end{cases} \tag{2.3}$$

  where θ is the threshold.

Although the linear classification capability of perceptron machines was a breakthrough in simple pattern recognition tasks. The most fundamental limitation stems from the mathematical nature of its single-layer structure: constrained by a linear hyperplane partitioning mechanism, the system is unable to handle non-linearly differentiable problems such as XOR - a shortcoming that was demonstrated in 1969 [26]. A more fundamental flaw lies in the lack of feature abstraction capabilities of modern deep learning: the directly hardwired architecture of input and output layers allows the model to complete only linear combinations of raw features, without the ability to achieve multi-level refinement of data representations through hidden layers [27]. This architectural limitation restricts the expressiveness of the model but ultimately pushes researchers to break out of the single-layer architecture paradigm and open up deep learning research.

2.1.3. Multilayer Perceptron and Fully Connected Neural Network

Building on the foundation of the Perceptron, researchers introduced the concept of the hidden layer [28], which forms a hierarchical structure of "input-hidden-output", as shown in Figure 2.3. Each hidden layer unit uses a non-linear activation function, such as Sigmoid or ReLU, to perform a non-linear transformation of the linear combination of outputs from the previous layer [29]. As shown in Equation 2.4, the Sigmoid function squashes input values into the $(0, 1)$ interval, making it suitable for probabilistic interpretation [30], while the ReLU function outputs the input directly if it is positive and zero otherwise, enabling sparse activations and efficient gradient propagation [31]. The key feature of this architecture is that the first hidden layer learns low-level feature combinations from the raw input, while the second layer further abstracts these low-level features. Through progressive feature distillation across layers, the network ultimately approximates complex decision boundaries. Most importantly, the backpropagation (BP) algorithm propagates output layer errors backward through the network using the chain rule [4], allowing weight updates across layers.

$$\text{Sigmoid}: \quad \sigma(x) = \frac{1}{1 + e^{-x}}, \qquad \text{ReLU}: \quad \text{ReLU}(x) = \max(0, x). \tag{2.4}$$

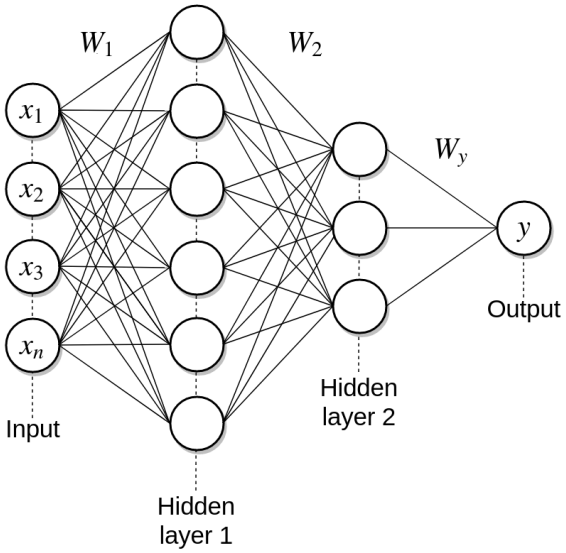


Figure 2.3: Structure of a fully connected neural network with two hidden layers. Each layer applies a weight matrix (W_1, W_2, W_y) and a non-linear activation function to transform the input vector $\mathbf{x}$ into the final output y [32].

An MLP/FCNN consists of multiple layers of neurons:

- **Input Layer:** Receives external input data, where each neuron corresponds to an input feature.
- **Hidden Layers:** Situated between the input and output layers, it may contain one or more hidden layers. Each neuron in a hidden layer is fully connected to all neurons in the previous layer.
- **Output Layer:** Produces the final prediction results based on the task requirements. For example, classification tasks output class probabilities, while regression tasks produce continuous numerical values.

FCNN computation includes forward propagation and backpropagation [33]. In forward propagation, each neuron receives input from the previous layer, computes the weighted sum, and applies an activation function for nonlinear transformation:

$$h^{(l)} = \sigma\left(W^{(l)}h^{(l-1)} + b^{(l)}\right), \tag{2.5}$$

where $h^{(l)}$ is the output of the neurons in the l-th layer, $W^{(l)}$ is the weight matrix for that layer, $b^{(l)}$ is the bias term, $\sigma(\cdot)$ is the activation function, such as ReLU, Sigmoid, or Tanh.

To optimize the neural network, FCNN employs gradient descent combined with the BP algorithm to adjust weights. BP computes the gradient of each layers error and updates the weights:

$$W^{(l)} = W^{(l)} - \eta\frac{\partial L}{\partial W^{(l)}}, \tag{2.6}$$

where L is the loss function (mean squared error (MSE) or cross-entropy loss), η is the learning rate, and $\frac{\partial L}{\partial W^{(l)}}$ is the gradient of the loss for the weights [34].

MSE is one of the most commonly used loss functions in machine learning and statistics. It is used to measure the discrepancy between predicted and actual values. MSE calculates the difference between each predicted and true value, squares these differences, and then takes the average of all the squared differences [35]. One advantage of the MSE is that it imposes a heavier penalty on larger errors, thus encouraging models to minimize significant prediction errors. Additionally, the MSE function is continuous and differentiable, facilitating optimization through gradient-based algorithms such as gradient descent. The mathematical expression of MSE is defined as follows:

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{2.7}$$

where n is the number of samples, y_i is the true value of the ith sample, $\hat{y}_i$ is the corresponding predicted value.

Cross-entropy loss is another widely used loss function, particularly in classification problems. It evaluates a model's classification performance by measuring the discrepancy between its predicted and actual label distributions [36]. Cross-entropy loss has several significant properties. Firstly, it imposes a non-linear penalty for misclassification. Therefore, when the model is highly confident in incorrect classes, the loss increases dramatically, thereby encouraging the model to produce more cautious and accurate probability estimations. Secondly, the cross-entropy loss function is convex and differentiable everywhere, providing a robust mathematical foundation for gradient-based optimization algorithms and mitigating the issue of local optima. Most importantly, cross-entropy loss provides a probability distribution that gives classification results and indicates the models confidence in its predictions [37]. This feature is extremely valuable in many practical applications. The mathematical formulation of the cross-entropy loss function is defined as follows:

$$L = -\frac{1}{N}\sum_{j=1}^{N}\sum_{i=1}^{C} y_{ij}\log(\hat{y}_{ij}), \tag{2.8}$$

where N is the total number of training samples and C is the number of output classes. y_{ij} denotes the ground-truth label for the j-th sample and the i-th class, and $\hat{y}_{ij}$ is the predicted probability for the same class, usually obtained via a softmax layer.

In summary, FCNNs originate from foundational neuron models such as the MP neuron, which have shaped the development of modern deep learning. Despite their robust performance in various structured pattern-recognition tasks, FCNNs inherently face challenges when modeling temporal or sequential information due to their static architectures. This inherent limitation has driven the development of more complex structures, such as RNNs, which incorporate recurrence to effectively capture temporal dependencies [38]. This has laid the foundation for the subsequent models that will be discussed in the following sections.

2.2. Recurrent Neural Network

While FCNNs are effective for structured data and static pattern recognition, they are not explicitly designed to model sequential dependencies. As a result, they often struggle to capture temporal structure in tasks involving time series or language data.

To address this problem, researchers in the 1980s began to explore a new type of network structure, RNNs, which can exploit the temporal information in data [39]. By introducing recurrent connections into the network, RNNs allow the output at each step not only to depend on the current input but also to incorporate the state information of the previous moments, thus building a memory mechanism. This mechanism allows the model to dynamically capture temporal dependencies and contextual information as it processes sequential data, providing a more coherent and enriched feature representation for subsequent tasks [6].

Early RNNs suffer from the vanishing and exploding gradient problem, which hampers the learning of long-term dependencies. To enable gradient-based training, backpropagation through time (BPTT) was introduced as a temporal unrolling of the standard back-propagation algorithm [40]. While BPTT makes end-to-end optimization possible, it does not prevent gradient degradation. To alleviate this difficulty, gated architectures-most notably the long short-term memory (LSTM) [41] and the gated recurrent unit (GRU) [42] were proposed. By employing learnable input, forget, and output gates, these models dynamically regulate the flow of information across time steps, thereby preserving salient context and substantially improving the modeling of long-range dependencies.

2.2.1. LSTM

Traditional RNNs tend to have gradient vanishing or gradient explosion in their backpropagation process when dealing with long sequence data, due to the repeated use of the same weight matrix between time steps. This makes it difficult for the model to capture important information from early inputs when the sequence is long, thus affecting the learning of long-term dependencies. To overcome this difficulty, LSTM was developed to allow information to be passed over long sequences by introducing an independent cell state and a series of gating mechanisms, including forgetting gates, input gates, and output gates [41]. In this way, the model not only effectively mitigates the problems of gradient vanishing and gradient explosion but also flexibly updates and filters unnecessary information while preserving critical long-term information, thus improving the modeling capability for long sequence data.

An LSTM augments a standard RNN with a memory cell c_t and three learnable gates, allowing the network to preserve or discard information across time steps. Given the current input x_t, the previous hidden state h_{t-1}, and the previous cell state c_{t-1}, the gates are defined as follows:

- **Forget Gate**: Determines how much information from the previous cell state should be forgotten:

$$f_t = \sigma\left(W_f \cdot \begin{bmatrix} h_{t-1} \\ x_t, \end{bmatrix} + b_f\right) \tag{2.9}$$

  where σ is the Sigmoid function, which outputs values in the range $[0, 1]$. W_f and b_f are the weight matrix and bias vector, respectively.

- **Input Gate and Candidate Cell State**: The input gate determines how much of the current input should be written to the cell state, while the candidate cell state is computed as follows:

$$i_t = \sigma\left(W_i \cdot \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_i\right), \tag{2.10}$$

$$\tilde{c}_t = \tanh\left(W_c \cdot \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_c\right), \tag{2.11}$$

  The hyperbolic tangent function (tanh) ensures that $\tilde{c}_t$ has values in the range $[-1, 1]$.

- **Output Gate and Hidden State**: The output gate determines how much of the cell state should be used for the final hidden state:

$$o_t = \sigma\left(W_o \cdot \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_o\right). \tag{2.12}$$

  The hidden state is then computed as:

$$h_t = o_t \odot \tanh(c_t). \tag{2.13}$$

  The cell state c_t carries the long-term information of the sequence and is updated by combining the retained portion of the previous cell state with the candidate content generated at the current time step:

$$c_t = f_t \odot c_{t-1} \; + \; i_t \odot \tilde{c}_t, \tag{2.14}$$

As shown in Figure 2.4, LSTM enables the network to store and propagate essential information over long sequences while filtering out unnecessary details by integrating these three gates, improving its ability to capture long-term dependencies.

2.2.2. GRU

GRU simplifies computation by directly combining the update and reset gates in the hidden state operations, making it more efficient than LSTM while achieving similar performance in certain applications [44]. Given the current input x_t at time step t and the previous hidden state h_{t-1} [42], the GRU computations are as follows:

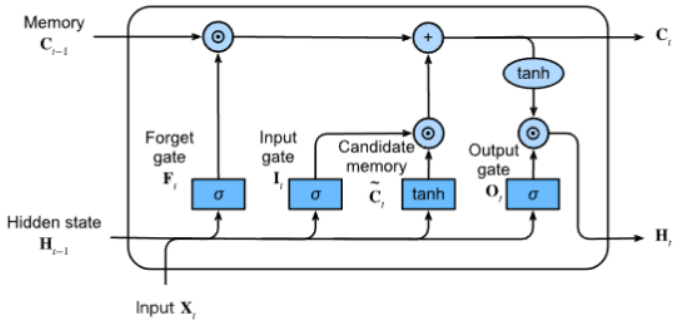


Figure 2.4: Illustration of an LSTM cell across three time steps $(t-1,\ t,\ t+1)$, x_t is the input at t time step, and y_t is the output. The cell state maintains long-term dependencies, while the forget (F_t), input (I_t), and output (O_t) gates regulate information flow. The candidate state $(\tilde{C}_t)$ is updated via a combination of the gates, allowing effective handling of long-range dependencies and mitigating the vanishing gradient problem [43].

- **Update Gate**: Determines how much of the previous hidden state should be retained while incorporating new information:

$$z_t = \sigma\left(W_z \cdot \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_z\right). \tag{2.15}$$

- **Reset Gate**: Controls how much of the previous hidden state should be combined with the current input to generate the candidate hidden state:

$$r_t = \sigma\left(W_r \cdot \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_r\right). \tag{2.16}$$

- **Candidate Hidden State**: The candidate hidden state is generated using the reset gate:

$$\tilde{h}_t = \tanh\left(W \cdot \begin{bmatrix} r_t \odot h_{t-1} \\ x_t \end{bmatrix} + b\right). \tag{2.17}$$

- **Final Hidden State Update**: The final hidden state is computed by merging the previous hidden state and the candidate hidden state using the update gate:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t. \tag{2.18}$$

2.2.3. minGRU

Although both GRU and LSTM models can handle long sequences, their formulations impose certain limitations on parallel computation. For example, in a GRU model, the final hidden state is computed only after both the update gate and the reset gate have been computed. This sequential dependency prevents efficient parallelization during training, limiting the runtime performance of the model. In response, Feng *et al.* in 2024 proposed an improved model called minGRU [45]. As shown in Figure 2.6, this model eliminates the reset gate entirely and instead computes the candidate hidden state directly from the previous hidden state and the current input, simplifying most computations and increasing the efficiency through parallel processing.

Compared to the standard GRU, minGRU simplify the computation of the candidate's hidden state:

$$\tilde{h}_t = \text{Linear}_{d_h}(x_t), \tag{2.19}$$

which now depends solely on the current input x_t, reducing the complexity of the network structure and computational overhead.

As shown in the Figure 2.6, minGRU retains only the update gate z_t, which controls the balance between the previous hidden state h_{t-1} and the candidate hidden state $\tilde{h}_t$:

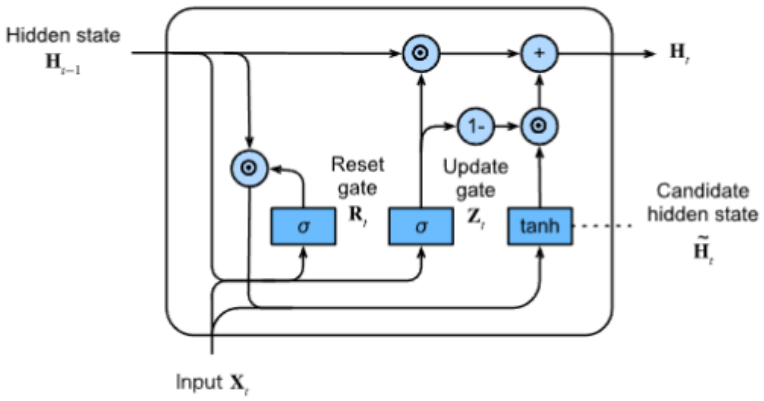


Figure 2.5: Illustration of a GRU, showing the update (Z_t) and reset (R_t) gates controlling the hidden state (H_t). The candidate hidden state ($\tilde{H}_t$) is generated via the reset gate and combined with the previous hidden state to capture long-range dependencies efficiently [43].

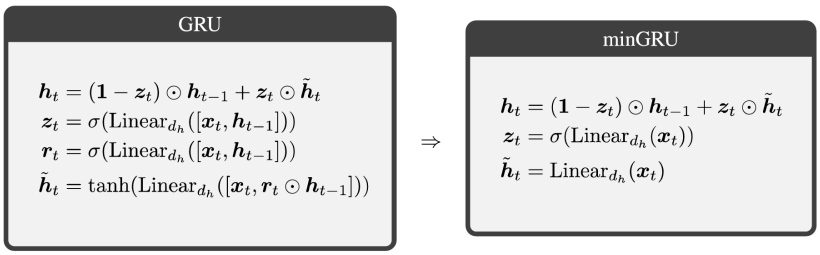


Figure 2.6: Comparison between GRU and minGRU. The standard GRU (left) consists of an update gate z_t, a reset gate r_t, and a candidate hidden state $\tilde{h}_t$, where r_t modulates the contribution of the previous hidden state h_{t-1} in the computation of $\tilde{h}_t$. In contrast, minGRU (right) removes the reset gate, simplifying the computation of candidate hidden states to depend solely on the current input x_t [45].

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t. \tag{2.20}$$

By removing the reset gate and eliminating the need for additional transformations on h_{t-1}, minGRU reduces both the parameter count and the sequential dependency of the model. This not only alleviates redundant dependencies in gradient propagation but also enhances parallel computing efficiency in hardware implementations. Additionally, during training, minGRU can mitigate overfitting while still effectively capturing historical information [45]. As a result, minGRU often achieves performance comparable to or even better than standard GRU models in practical applications.

2.2.4. Enhancing RNN Performance with Attention Mechanism

Traditional recurrent networks, including gated variants such as LSTM and GRU can alleviate, but do not eliminate, vanishing and exploding gradient effects. More critically for todays hardware, their update rule is inherently sequential: each hidden state depends on the previous one, forcing computations to proceed step by step along the time axis. This serial data dependency prevents the massive layer-wise parallelism that GPUs and TPUs are designed to exploit, so training and inference throughput scale poorly as sequence length increases. At the same time, the hidden state must still compress the entire past into a single vector, causing earlier details to fade and limiting the modelling of long-range dependencies [46].

To overcome these limitations, the attention mechanism was proposed. Rather than storing the entire history in a fixed-length state, this mechanism computes context-dependent weighted sums over all input positions, allowing the model to access relevant information directly, regardless of distance. First introduced in neural machine translation, attention has since become a core component of a wide range of NLP systemsincluding text generation, question answering, and speech recognitionowing to

its capacity for parallel computation and its ability to model long-range dependencies more effectively than traditional RNNs.

The core idea of the attention mechanism is that, instead of relying solely on the final hidden state, the model dynamically attends to different parts of the input sequence as needed, thereby enhancing its ability to capture long-range dependencies [47].

Given an input sequence $\{x_1, x_2, ..., x_T\}$ and its corresponding hidden states $\{h_1, h_2, ..., h_T\}$, the attention mechanism follows these steps:

- **Compute Attention Scores**: Each input hidden state is matched with the query at the current time step to compute its relevance score:

$$e_{t,i} = \text{score}(q_t, h_i), \tag{2.21}$$

where q_t denotes the query vector at time step t and h_i is the encoder hidden state at position i. The scalar $e_{t,i}$ quantifies how well the information stored in h_i aligns with the decoders current information need represented by q_t; a larger value indicates a greater relevance of h_i to the prediction being made at time t. The score function measures the relevance between the query vector q_t and each encoder hidden state h_i. It maps the pair (q_t, h_i) to a scalar $e_{t,i}$, with larger values indicating greater importance of h_i for the current decoding step [48]. Three widely used score formulations are:

$$\text{General Form: } e_{t,i} = q_t^\top W_a h_i, \tag{2.22}$$

$$\text{Scaled DotProduct: } e_{t,i} = \frac{q_t^\top h_i}{\sqrt{d}}, \tag{2.23}$$

$$\text{Additive Form: } e_{t,i} = v^\top \tanh(W_q q_t + W_h h_i), \tag{2.24}$$

In Equation 2.22, the general form inserts a trainable projection $W_a \in \mathbb{R}^{d\times d}$ between the query and key vectors, enabling a richer, learnable similarity measure than a plain inner product.

The scaled dotproduct score in Equation 2.23 is parameter-free; dividing by $\sqrt{d}$ (with d the hidden dimension) keeps the magnitude of the logits roughly constant as d grows, which stabilises gradients in high-dimensional spaces [49].

Finally, the additive form in Equation 2.24 employs two linear projections $W_q, W_h \in \mathbb{R}^{d_a\times d}$ followed by a non-linearity and a learned vector $v \in \mathbb{R}^{d_a}$, effectively realising a single-layer feed-forward network that can model complex alignments between q_t and h_i at the expense of additional parameters and computations.

- **Normalize Attention Weights**: The scores are normalized using the softmax function so that they sum to 1:

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{j=1}^{T} \exp(e_{t,j})}, \tag{2.25}$$

where $\alpha_{t,i}$ represents the normalized attention weight assigned to the i-th hidden state h_i at time step t, reflecting its relative importance with respect to the query q_t.

- **Compute Context Vector**: The weighted sum of all input hidden states is computed:

$$c_t = \sum_{i=1}^{T} \alpha_{t,i} h_i, \tag{2.26}$$

where c_t is the **context vector** at time step t, containing key information from the input sequence.

- **Generate the Final Output**: The context vector c_t is combined with the current hidden state q_t to produce the final output:

$$\tilde{h}_t = \tanh(W_c[c_t; q_t]). \tag{2.27}$$

The attention mechanism allows the model to flexibly focus on different parts of the input sequence at each time step, rather than being constrained by a hidden state [47]. In this work, we adopt a simple additive attention mechanism, where the relevance score between the decoder hidden state q_t and each encoder hidden state h_i is computed using a feedforward neural network with a single hidden layer.

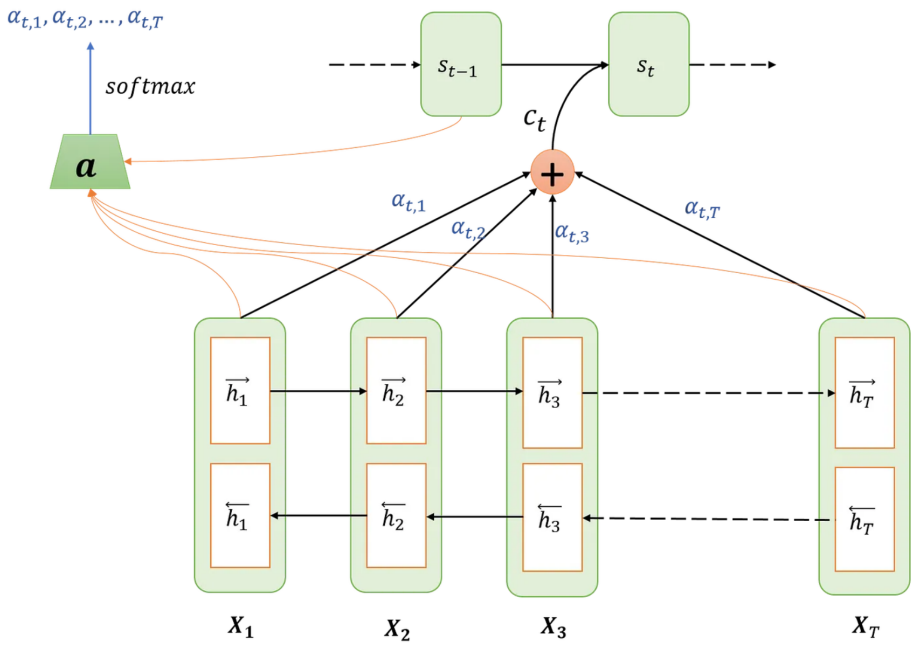


Figure 2.7: A model of the attention mechanism in a sequence-to-sequence framework. At each decoding step t, the model computes attention weights $\alpha_{t,i}$ by comparing the decoder state s_{t-1} with each encoder hidden state h_i. These weights determine how much each h_i contributes to the context vector, which is then combined with the decoder state s_{t-1} to update s_t [50].

In summary, RNNs are effective models for processing sequential data but suffer from issues such as vanishing or exploding gradients. Advanced variants, including LSTMs and GRUs, overcome these limitations through specialized gating mechanisms, effectively managing information flow across sequences. Additionally, the minGRU simplifies the GRU structure, achieving comparable performance with reduced computational complexity. The collective enhancement of these RNN models signifies a substantial augmentation in the capacity of neural networks to accommodate intricate temporal dependencies, a development that is particularly advantageous in scenarios involving sequential data. Moreover, the attention mechanism enhances traditional RNN models by selectively focusing on critical portions of input sequences, improving the capability of RNNs to capture long-range dependencies [51].

2.3. Hierarchical Temporal Memory

HTM is a biologically inspired machine learning algorithm modeled after the structural and functional principles of the neocortex. Unlike traditional neural networks that typically require dense computations and large labeled datasets, HTM utilizes sparse distributed representations (SDR) and predictive memory mechanisms. This approach enables robust information encoding, resistance to noise, and significant reductions in computational power consumption. Due to its inherent robustness from SDR and energy-efficient online learning capability, HTM is particularly suitable for scenarios such as anomaly detection, real-time event prediction, and incremental learning [52]. Consequently, HTM was selected in this work as an alternative to conventional deep learning models, offering both interpretability and computational efficiency.

This framewrok contain two primary components of HTM: the HTM SP and the HTM TM [53]. The HTM SP mainly performs the process of information encoding, converting inputs into SDR. At the same time, the HTM TM is responsible for learning and inference processes and is capable of generating

outputs based on historical inputs. The HTM algorithm will be introduced in three parts. Firstly, the SDR data type, which is widely used in the HTM algorithm, is introduced. Then, the two parts of the HTM algorithm, SP and TM, will be introduced.

2.3.1. Sparse Distributed Representations

Numerous studies, such as [54], have shown that cortical regions in the human brain primarily encode information through sparse activity patterns, typically involving only 1% to 10% of neurons at any given time. Inspired by this biological principle, the HTM framework formalises the concept of a SDRa high-dimensional binary vector in which only a small, fixed proportion of bits are active. As the example in Figure 2.8, only a handful of bits are 1 while the vast majority are 0 [55]; this extreme sparsity yields distinctive bit patterns that are easily distinguished, confer a vanishingly small collision probability, and provide strong robustness and scalability [15]. Sparsity also facilitates computational and storage efficiency, as operations need only involve active bits. Additionally, encoding information sparsely against a predominantly inactive background enhances noise tolerance and robustness. Crucially, sparsity makes measuring semantic similarity between SDRs, through the overlap of active bits, meaningful and discriminative.

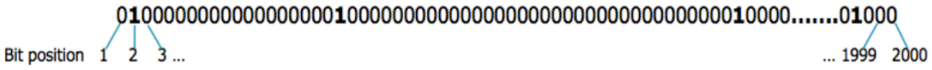
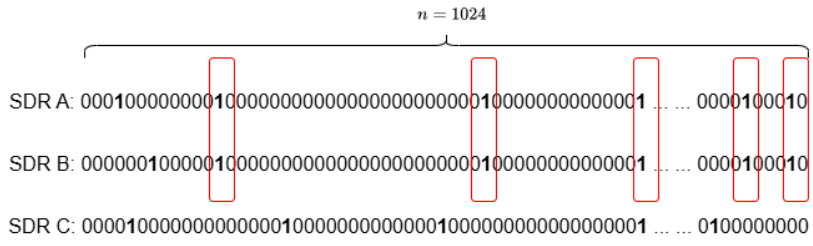


Figure 2.8: An illustrative example of a SDR. The figure shows a binary vector containing 2000 bits, with only a small fraction of bits set to "1" (active), while the remaining bits are "0" (inactive).

Moreover, with carefully chosen parameters, SDRs can achieve strong noise tolerance while producing only a minimal number of false matches. Given two SDRs, both with size n and sparsity w, a match is defined as having at least θ overlapping active bits, up to w bits. We define $\sum_{b=\theta}^{w} |\Omega_x(n, w, b)|$ as the total number of random SDRs that would be classified as a "match" against a specific target SDR. $\Omega_x(n, w, b)$ represents the set of all SDRs of size n and sparsity w that overlap exactly b active bits with a given reference SDR x. Formally, it can be expressed as:

$$\Omega_x(n, w, b) = \{y \mid |y| = w, |x \cap y| = b\}. \tag{2.28}$$

Figure 2.9 shows an example of 1024-bit SDR data. SDR A is used as the reference here, and it can be seen that SDR B has five overlapping activation bits with SDR A. When the 'matching' threshold is set to 4, SDR B is considered to be part of the set of SDRs that match SDR A. However, SDR C has only one overlapping active bit with SDR A, so it does not belong to the matching set [56].



$$\text{SDR B: number of matching 1 bits} = 5 > \theta = 4 \longrightarrow \begin{matrix} \text{SDR B} \in |\Omega_A(n = 1024, w = 10, b = 4)| \\ \text{SDR C} \notin |\Omega_A(n = 1024, w = 10, b = 4)| \end{matrix}$$

Figure 2.9: Illustration of overlapping active bits between SDRs. SDR A is used as the reference vector with $n = 1024$ and $w = 10$. SDR B overlaps with SDR A on 5 active bits, which exceeds the threshold $\theta = 4$, thus SDR B $\in \Omega_A(n = 1024, w = 10, b = 4)$. In contrast, SDR C only overlaps with SDR A on a single active bit, so SDR C $\notin \Omega_A(n = 1024, w = 10, b = 4)$.

Additionally, $\binom{n}{w}$ can be used to represent the total number of SDR codes with length n and exactly w bits of value 1. Among a total of $\binom{n}{w}$ patterns, the probability of a false positive is:

$$f_w^p(\theta) = \frac{\sum_{b=\theta}^{w} |\Omega_x(n, w, b)|}{\binom{n}{w}}. \tag{2.29}$$

This equation means that we calculate the total number of overlapping patterns from θ to w bits, and then divide by the total number of possible patterns [57]. This ratio represents the probability of a false match. For example, consider vectors with $n = 1024$ bits and sparsity $w = 4$. If the matching threshold is set to $\theta = 2$, allowing 50% noise, the probability of a false match is:

$$f_w^p(2) = \frac{\sum_{b=2}^{4} |\Omega_x(1024, 4, 2)|}{\binom{1024}{4}} = \frac{1}{14587}. \tag{2.30}$$

This indicates a relatively high chance of false matches under 50% noise. However, if w and θ are increased to 20 and 10, respectively, the probability of a false match is:

$$f_w^p(10) = \frac{\sum_{b=10}^{20} |\Omega_x(1024, 20, 10)|}{\binom{1024}{20}} = \frac{1}{10^{13}}. \tag{2.31}$$

This suggests that with a modest increase in w and θ, SDRs can achieve near-perfect robustness even with 50% noise [58].

In contrast to traditional binary data types, each digit in an SDR has its own meaning. In traditional binary data, only the complete code has a specific meaning. For example, in ASCII, the letter "m" is represented by the binary code "01101101", and any modification of any of these digits will completely change the meaning of this binary code. In SDR, however, there may be bits representing whether the letter is a consonant or a vowel, bits representing how the letter sounds, etc [59].

Next, several important mathematical properties of SDR data types will be introduced [60]. The first is the high storage efficiency of SDRs. Since SDR data is sparsely distributed, it contains only a very small number of 1's. Therefore, when storing SDR data, we only need to record the location of the 1 bits. This means that although SDR data has many bits of data, we only need to store the exact position of 1% to 10 % of the data. The second notable feature is the union property of SDRs. As illustrated in Figure 2.10, one can combine a set of SDRs into a new SDR by performing a bitwise OR on the active bits. This property is particularly useful for composite or cumulative representations in HTM-based systems, as it allows multiple patterns to be merged without losing their individual semantic information. The resulting "union" SDR contains all the active bits from each constituent SDR, facilitating set membership checking and robust feature aggregation, while retaining the noise tolerance and sparsity benefits inherent in the SDR framework [61].

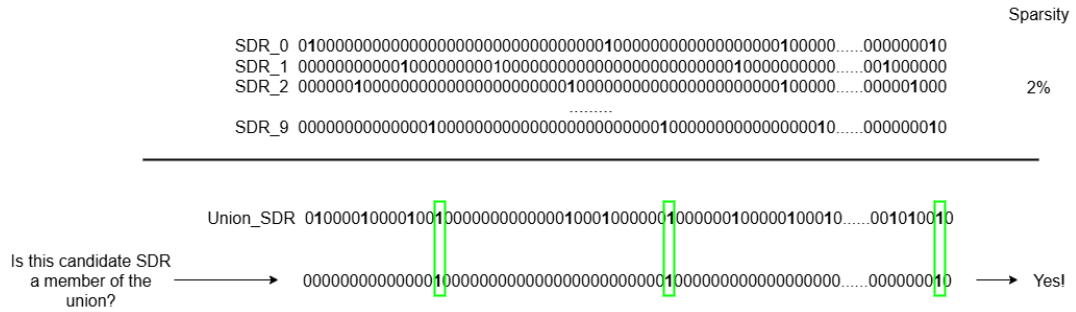


Figure 2.10: Illustration of the union property of SDRs. The figure shows multiple individual SDRs. By forming a bitwise logical OR operation (union) of these SDRs, a new composite SDR (Union_SDR) is generated, which preserves critical information from all constituent SDRs. Due to the inherent sparsity and distributed nature of SDRs, membership tests can be efficiently performed by verifying the overlap of active bits, as highlighted by the green rectangles.

2.3.2. Comparison of High-Dimensional Representations: HDC vs. SDR

While researching the HTM algorithm, we discovered another area of research called hyperdimensional computing (HDC), which also uses data similar to SDR for computation. The differences between SDR and HDC are outlined below.

SDR is a sparse, binary vector representation where only a small percentage of bits are active. This sparsity ensures high capacity for unique pattern encoding while maintaining robustness to noise and subsampling [62]. SDRs are commonly used in HTM systems and are designed to maintain semantic similarity overlapping active bits indicate semantic relatedness between inputs [63]. Their structure supports biologically inspired learning through mechanisms like Hebbian updates and synaptic permanence [64].

In contrast, HDC relies on high-dimensional dense binary or bipolar vectors (e.g., 10,000 bits) in which nearly all positions are active [65]. Computation is performed through algebraic operations such as XOR or elementwise multiplication (binding), summation or thresholding (bundling), and Hamming distance [66]. Learning in HDC is typically achieved through a few updates to prototype vectors and does not require maintaining or adjusting explicit synaptic weights [67]. The differences between HDC and SDR are shown in Table 2.1.

Table 2.1: Key Differences Between HDC and SDR

Aspect	**SDR**	**HDC**
Vector Density	Sparse (1% to 10% active bits)	Dense (most or all bits active)
Semantic Similarity	Based on overlapping active bits	Based on Hamming or cosine similarity
Learning Mechanism	Synaptic plasticity (e.g., Hebbian rule)	Vector overlay (e.g., summation, majority)
Biological Basis	Modeled on cortical circuits	Abstract, non-biological
Hardware Efficiency	Sparse, event-driven operations	Parallel bitwise operations

In summary, SDRs offer structured, biologically plausible encodings with fine-grained temporal dynamics [68], whereas HDC provides a flexible, hardware-efficient alternative that excels in fast, robust inference tasks [69].

2.3.3. Spatial Pooler

SP is one of the core components of the HTM, responsible for converting input data into SDRs. In brief, the SP learns feedforward synaptic connection patterns to transform arbitrary binary input patterns into distributed encodings with fixed sparsity. Figure 2.11 outlines HTMs hierarchical organisation. At each level, only a sparse subset of columns, which is vertical stacks of cells, wins the local inhibition competition, producing a SDR for the input. Zooming in on a single column, every cell acts like a pyramidal neuron with three dendritic zones: proximal segments (brown) receive feed-forward signals from the input space, distal segments (red) collect lateral context from neighbouring cells to represent temporal transitions, and apical segments (blue) convey top-down feedback from higher layers.

The primary function of the SP is to learn and extract the spatial patterns present in the input. It ensures that only a small subset of columns is activated for each input, thereby maintaining sparsity, while similar inputs result in overlapping sets of active columns and dissimilar inputs activate distinct sets [71]. Consequently, the SDR output not only preserves the essential informational patterns of the input but also exhibits a fixed sparsity level and robustness to noise, thereby providing a stable encoding foundation for the subsequent temporal memory module.

Typically, the input to the SP is represented as a fixed-length binary vector, where each bit corresponds to an input feature: a value of 1 indicates that the feature or percept is active, while 0 denotes inactivity [72]. These binary inputs may be obtained directly from sensor encodings or derived from raw data using HTM encoders. One of the roles of the SP is to transform these denser input representations into sparser ones. Regardless of the density of the input, the SP selects only a limited number of columns for activation, ensuring that the output SDR maintains a uniform sparsity [73]. Given a new input vector, the SP determines which columns become active through the following steps:

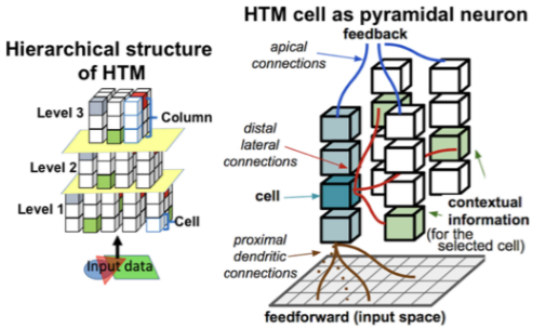


Figure 2.11: Architecture of HTM. **Left**: Input data are encoded through stacked levels, where cells form vertical columns and only a sparse subset of columns becomes active. **Right**: An enlarged column shows each HTM cell modeled after a cortical pyramidal neuron, receiving feed-forward input on proximal dendrites (brown), lateral context on distal dendrites (red), and top-down feedback on apical dendrites (blue). This arrangement allows HTM to combine spatial pattern encoding with context-dependent temporal prediction [70].

1. **Overlap Calculation**
   First, during the SP calculation, each column of cells is treated as a unified whole. As can be seen in Figure 2.12, the red columns represent the activated columns. As can be seen, all cells in the same column are activated together. Given an input vector $x \in \{0, 1\}$, SP first calculates the degree to which each column matches the input. Let the 'connected' synapse set of column i be:

$$S(i) = \{ j \mid p_{ij} \geq \tau \}, \tag{2.32}$$

   where τ is a fixed threshold, p_{ij} represents the permance value corresponding to the column between the inputs. SP calculates the overlap between each column and the current input. Overlap is defined as the number of synapses in that column that are both (a) "connected" and (b) linked to active bits (i.e., equal to 1) in the input. Formally, for column i, its overlap O_i can be expressed as:

$$O_i = \sum_{j \in \mathrm{S}(i)} x_j, \tag{2.33}$$

   where x_j is the j-th (0 or 1) value of the input vector. The SP then multiplies each column's overlap by its boost factor b_i (initialized to 1) to obtain an enhanced overlap [15]:

$$\hat{O}_i = b_i \, O_i, \tag{2.34}$$

   The boost factor is used later during the inhibition (competition) process. This step implies that a column's competitiveness is determined not solely by its instantaneous degree of match, O_i, with the current input, but is also dynamically modulated by its boost factor, b_i, is linked to the column's historical activity level. Specifically, if a column has been activated frequently, its boost factor tends to decrease. Conversely, if a column activates less frequently than other columns, its boost factor is increased. Therefore, the boost factor serves to balance the activation frequencies across different columns, enabling a more effective and representative selection of winning columns during the subsequent inhibition process.

2. **Stimulus Threshold Filtering**
   A column's overlap must reach or exceed the threshold θ_i for the column to be considered for activation. In other words, if a column's overlap is smaller than the θ_i (for instance, a threshold of 1 indicates that at least one input bit must match), its overlap is effectively set to zero, implying that it does not respond sufficiently to the current input. This step eliminates columns with minimal overlap, reducing unnecessary competition.

3. **Competitive Inhibition**
   Each minicolumn i is assigned a 2-D coordinate $\mathbf{y}_i$. We define its local inhibition neighborhood

$$\mathbf{N}_i \;=\; \Big\{ j \;\Big|\; \|\mathbf{y}_i - \mathbf{y}_j\| < \phi,\; j \neq i \Big\}, \tag{2.35}$$

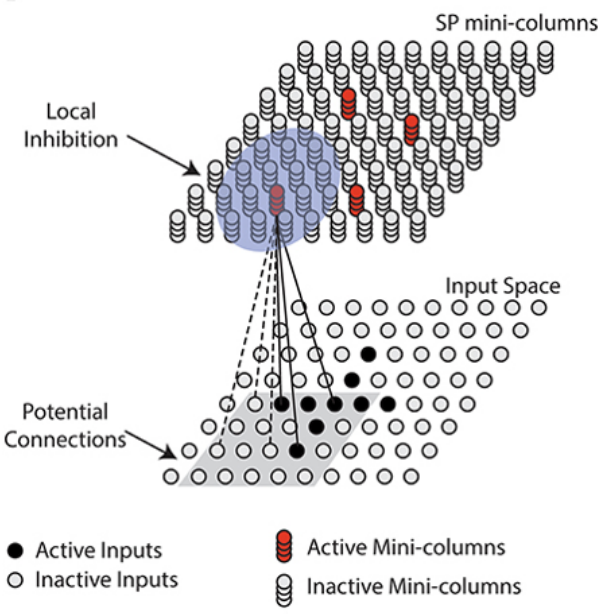


Figure 2.12: The learning process of spatial pooling. It shows that each column has a corresponding receptive field, or region, called potential connections. The connection weights between this region are modified during the learning process. Also, it shows the local inhibition function in the algorithm. Within a certain range of an active mini-column, another active column will not appear [63].

where $\|\cdot\|$ is the Euclidean distance and ϕ denotes the inhibition radius. Thus $\mathbf{N}_i$ comprises all mini-columns whose centers lie within ϕ of column i. Increasing ϕ enlarges the lateral-inhibition field; in the limit $\phi \to \infty$ the process reduces to a global competition, whereas $\phi = 0$ disables lateral inhibition entirely. In Figure 2.12 the blue circle depicts a local inhibition region, inside which only one column wins, and each mini-column connects synaptically to a subset of the input space.

Let ρ be the desired activation sparsity (e.g. $\rho = 0.02$ for 2%). The number of winners allowed inside $\mathbf{N}_i$ is therefore

$$k_i \;=\; \lceil \rho\, |\mathbf{N}_i| \rceil, \tag{2.36}$$

where $|\mathbf{N}_i|$ is the neighborhood size and $\lceil\cdot\rceil$ rounds up to the nearest integer. In words, we keep only the top ρ-fraction of columns (by overlap score) in each local neighborhood. Denote by θ_i the k_i-th largest boosted overlap $\{\hat{o}_j \mid j \in \mathbf{N}_i\}$.The activity status of a column is:

$$a_i = \begin{cases} 1, & \hat{O}_i \geq \theta_i \text{ and } O_i \geq O_{\min}, \\ 0, & \text{otherwise.} \end{cases} \tag{2.37}$$

For example, if $|\mathbf{N}_i| = 50$ and $\rho = 0.02$, then $k_i = 1$: only the column with the highest $\hat{o}_j$ in that neighborhood survives, all others are inhibited to 0.

4. **Output**
   The set of columns that survive inhibition constitutes the final SDR corresponding to the current input.

Once the set of active columns corresponding to the current input has been determined, the SP adapts synaptic connections based on these activations to capture the statistical regularities of the input data. The SP's learning rules follow the principle of Hebbian learning—"cells that fire together, wire together; those that do not fire together, lose their connection." In practice, this is implemented as follows:

1. **Synaptic Plasticity Updates**
   For each active column, the SP iterates over all of its potential synapses:

- If a synapse is linked to an input bit that is active in the current input and the column itself is also active, then this input-column pair is considered "co-active." According to the Hebbian rule, the synaptic connection is strengthened. In practice, its weight value is increased.
- If a synapse is linked to an input bit that is inactive while the column is active, then the synapse is considered "active at the column side but inactive at the input side." In this case, the synaptic connection is weakened, reducing its weights.
- Columns that are not active do not update their synapses.

For each inactive column, since the column itself does not respond to the current input, no learning process is performed.

Equation 2.38 represents the overall process of synaptic update. A column updates its synaptic permanence values only when it is active ($c_i(t) = 1$). If the connected input bit is simultaneously active ($x_j(t) = 1$), the permanence is increased by δ_{inc}; otherwise it is decreased by δ_{dec}. When the column is inactive ($c_i(t) = 0$), no adaptation occurs [74]. Figure 2.12 also shows the weight update process. Solid lines indicate weights that will increase during learning, while dashed lines indicate weights that will decrease during learning. It can be seen that the weights between active columns and active inputs will increase during learning, while the weights between active columns and inactive inputs will decrease during learning.

$$\Delta p_{ij}(t) = c_i(t)\left[x_j(t)\,\delta_{\text{inc}} - \big(1 - x_j(t)\big)\,\delta_{\text{dec}}\right], \qquad p_{ij}(t+1) = p_{ij}(t) + \Delta p_{ij}(t). \tag{2.38}$$

2. **Threshold and Clipping**
When adjusting the permanence values, the condition $0 \leq p_{ij} \leq 1$ must be maintained for each permanence p_{ij}. Any permanence updates that push values beyond this range are clipped. Furthermore, if a permanence value crosses the connection threshold θ or falls below θ, the status of that synapse toggles accordingly. Specifically, a permanence value that surpasses θ becomes a "connected" synapse, whereas dropping below θ causes the synapse to disconnect. As learning proceeds, each column's actual set of connected synapses, which is the weight matrix W_{ij}, is thus updated dynamically [15].

The SP continually refines each column's receptive field within the input space through the above learning rules. Columns that become active strengthen connections to input patterns consistent with the current input, while connections for non-active columns remain unchanged. Over time, each column (or group of columns) maintains or fine-tunes its connectivity with frequently occurring input bits. In this way, different sets of columns learn to encode different patterns, collectively forming a more efficient representation of the input space. Overall, the input patterns that frequently co-occur will experience slight adjustments in their connection strengths, allowing the SP to track changes in the input data distribution while preserving a stable and robust representation.

During the learning process, SP also applies a method called boosting. Boosting is a strategy to balance activation opportunities across columns in the long term. The rationale behind boosting is that columns with weak initial connectivity or slower learning speeds might rarely become active during competition, thus remaining permanently inactive. To make full use of each column and maintain representational capacity, the SP introduces adaptive boosting, dynamically adjusting individual excitability to ensure all columns have opportunities for activation.

Specifically, the SP maintains an "active duty cycle" for each column, reflecting its recent activation frequency, typically calculated as a moving average. The recent activation frequency $\overline{a}_i(t)$ over a sliding time window T is computed as follows:

$$\overline{a}_i(t) = \frac{(T-1)\overline{a}_i(t-1) + a_i(t)}{T} \tag{2.39}$$

Simultaneously, SP calculates the average activation frequency of column i's neighbors:

$$\langle \overline{a}(t) \rangle_{\text{Neighbor}(i)} = \frac{1}{|\text{Neighbor}(i)|} \sum_{j \in \text{Neighbor}(i)} \overline{a}_j(t) \tag{2.40}$$

Using these frequency values, the SP dynamically adjusts a column-specific boost factor b_i. A common updating rule for the boost factor is given by:

$$b_i = \exp\left(-\beta\left(\overline{a}_i(t) - \langle \overline{a}(t)\rangle_{\text{Neighbor}(i)}\right)\right), \tag{2.41}$$

where β is a parameter controlling the strength of boosting. Figure 2.13 shows the relationship curve between activation frequency and boost factors [74].

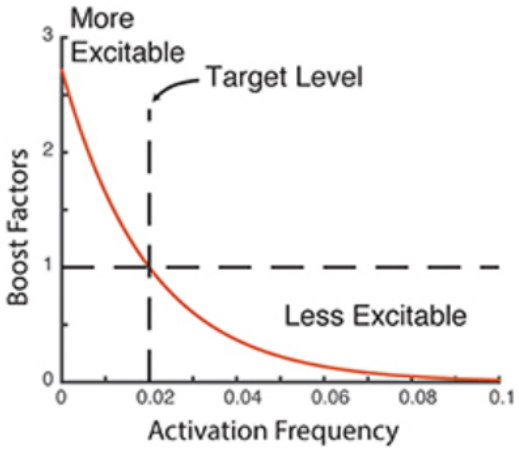


Figure 2.13: The figure shows the curve of the relationship between activation frequency and boost factors. In the SP module of HTM, the system applies a 'boosting' mechanism to inactive columns to ensure that all columns have the opportunity to participate in representing the input. The horizontal axis shows the average activation frequency of a column (usually calculated using a sliding window), and the vertical axis shows the boost factor for that column. A higher boost factor indicates that the column will be enhanced to improve its competitiveness [71].

From Equation 2.41, it can be seen that if column i's activation frequency is below the neighborhood average, which is $\overline{a}_i(t) < \langle \overline{a}(t)\rangle$, then the term $-\beta(\dots)$ becomes positive, causing $b_i > 1$, thus increasing the column's excitability. Conversely, if a column is frequently active, b_i is reduced to less than 1. Usually, b_i is limited within a certain range to prevent excessive deviations. Therefore, the primary goal of boosting is to increase the excitability of columns that are rarely active, rather than decreasing sensitivity in columns frequently activated [16].

With each input-driven update cycle, columns with historically low activation frequencies accumulate higher boost factors, increasing their overlap scores during subsequent competitions. Once these columns eventually respond strongly to some inputs, they rapidly form stable connections with current patterns, becoming more readily activated in future iterations. Thus, a positive feedback loop forms, allowing previously inactive columns to re-engage, increasing the network's representational capacity. Simultaneously, those previously overactive columns have reduced boost factors, preventing dominance by a small subset of columns. This dynamic balancing through boosting helps the SP achieve uniform resource allocation and avoid the presence of permanently inactive columns.

2.3.4. Temporal Memory

The TM algorithm of HTM is responsible for learning input sequences and making predictions based on the SDR generated by the SP. By introducing temporal context, it can represent and predict future inputs based on current inputs [17]. In TM, each column consists of multiple cells, which collectively represent temporal contexts at different time steps. Each cell at each discrete time step can be in several possible states:

1. **Active State:**
   The active state of a cell is indicated by its firing at the current time step t, as shown in blue in the Figure 2.14. In the context of predictive analysis, if at least one cell within a column was predicted during the previous time step, then only those predicted cells become active, signifying accurate predictions. Conversely, if no cell was predicted in the column, then all cells within the column become active simultaneously, resulting in a burst, as shown in the time $t-1$ in the

Figure 2.14. This phenomenon of bursting activity corresponds to a situation with no predictive context, indicating uncertainty about sequence context.

2. **Predictive State**
   The transition of a cell into a predictive state is initiated by the reception of sufficient input by its distal dendritic segments, which results in the surpassing of a predefined activation threshold. This process leads to the depolarisation of the cell, signifying its anticipation of activation in the subsequent time step. For a given cell, if any distal dendritic segment at time t possesses sufficient active synapses connected to currently active cells, exceeding a prediction threshold θ_s, this cell transitions into the predictive state at the subsequent time step $t+1$, as shown in pink in Figure 2.14.

   - If the predicted activation matches the actual input in the next step, the cell becomes active, which means it made a correct prediction.
   - If the predicted activation does not occur, the cell returns to a non-active state, and its synaptic permanence values decrease (reflecting biological phenomena such as long-term depression), allowing the system to continuously adapt and maintain accurate predictions.

3. **Learning State**
   Each active column selects a specific "winner cell" responsible for learning, forming, or updating distal dendritic segments representing context. When a column bursts (none of its cells were predicted), the SP chooses a winner cell to represent the current context, typically selecting the cell with the best matching distal dendritic segment to the recent context. If no suitable candidate exists, the least-used cell is chosen. The distal dendritic segment or newly formed segment on the winner cell then updates its synapses, reinforcing connections to cells active at the previous time step, thus encoding sequential continuity [75]. When the prediction of a cell matches the actual next input, which is the correct prediction, its corresponding distal synapses undergo further strengthening, enhancing the reliability of its prediction capability. Learning does not affect non-winner cells, but prolonged inactivity of synapses causes permanence values to decrease, thus pruning ineffective connections over time [76].

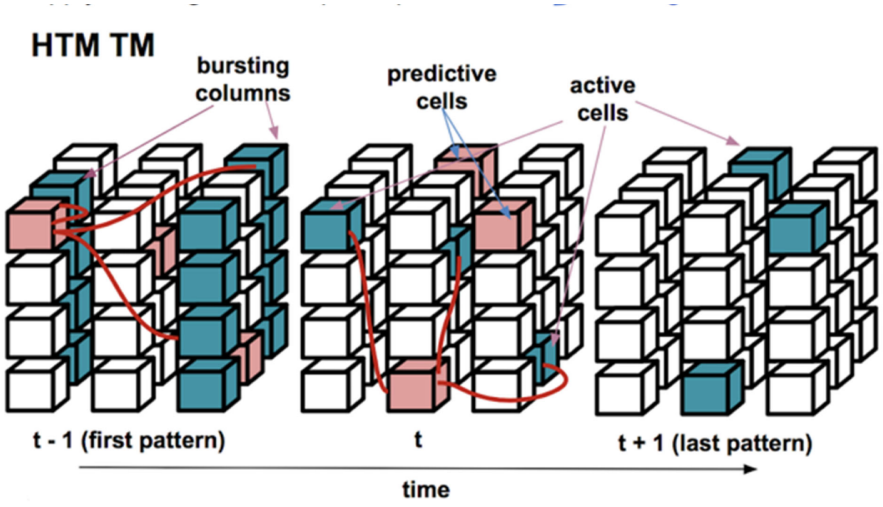


Figure 2.14: The figure illustrates the encoding and prediction processes of the Temporal Memory module in HTM for sequential inputs at various time points. Each column represents a spatial location and contains multiple cells representing different temporal contexts. The figure shows three time points: $t-1$ (first pattern), t (current time point), and $t+1$ (last pattern). These are arranged from left to right to indicate the progression of time.

At each discrete time step, the TM algorithm performs a series of computational processes to update cell activation states, encode temporal context, and form predictions:

1. **Receive Current Active Columns:**
   TM receives a set of active columns from the SP, which represents the SDR of the current input.

2. **Determine Activation Status:**
   For each active column, TM checks whether any cell within the column is in a predictive state from the previous time step.

   - If there are cells with successful predictions, as shown in blue at time t in Figure 2.14, then those cells are activated, indicating that the system has successfully memorised the temporal relationship between the preceding and following sequences.
   - If no cells were predicted in this column, the column experiences a burst: all cells within the column become active simultaneously, as shown in Figure 2.14, the cells on the left side of the column light up simultaneously at time $t-1$. These burst-activated cells thus represent the input without a previously learned context.

3. **Learning Updates:**
   For each currently active column, TM performs learning operations to update synaptic connections, thereby establishing a memory of the input sequence. Specifically:

   - If a burst occurs, TM selects a single "winner cell" within the columntypically the cell that has the strongest matching distal segment to the previous input contextor, if none exists, the least-used cell. TM then creates new distal dendritic segments (or updates existing segments) on the winner cell to encode the current temporal context.
   - If the column contains correctly predicted cells, TM strengthens the active segments and synapses corresponding to the predictive cells.
   - During these learning updates, synapses and segments are adjusted based on cell activity from the previous time step, following Hebbian principles: synapses linking to correctly predictive cells are strengthened, while those incorrectly predicting cells are weakened. This mechanism increases the permanence values of synapses connecting temporally correlated cells and decreases those related to incorrect predictions or noise. Over time, this leads to stable and precise sequence representations and memorization of historical context.

4. **Forming Predictions for Next Steps:**
   Finally, TM computes a predictive state for the next time step. For each cell, TM evaluates all distal dendritic segments, and if a segment's synaptic activation exceeds a predefined threshold, the corresponding cell transitions into a predictive state. These predictive cells represent TMs expectations for the next input. If a column contains predictive cells but is not subsequently activated by the actual input, the prediction is considered incorrect. Such prediction errors inform future learning steps, adjusting TMs internal model to achieve increasingly accurate predictions.

The TM module employs update rules to construct a robust representation of sequential patterns. This is achieved by associating temporal context with column activations at each time step. As a result, identical input patterns presented in different temporal contexts can activate distinct cell subsets within columns. This process enables the TM to represent temporal sequences effectively [70].

Figure 2.15 illustrates the learning and prediction processes of the HTM algorithm [64]. Panel B shows the state before learning. At this stage, the HTM algorithm can only activate the corresponding columns based on the SDR signals generated by the SP component. However, it cannot distinguish the semantic meaning of these activations in different contexts. For instance, in the input sequences $A \rightarrow B \rightarrow C \rightarrow D$ and $X \rightarrow B \rightarrow C \rightarrow Y$, the elements B and C activate the same column in different contexts, and all cells within that column are activated simultaneously. This indicates that the HTM has not yet developed an awareness of sequence structure, being able to perform only context-free spatial recognition. Panel C shows the results after temporal learning. Now, the same inputs B and C activate different cells within the same column in different contexts. For instance, in the sequence $A \rightarrow B \rightarrow C \rightarrow D$, B activates a particular cell in the column labeled B, whereas in the sequence $X \rightarrow B \rightarrow C \rightarrow Y$, B activates a different cell labeled B. This mechanism enables the HTM to recognize not only 'what is seen is B', but also 'under what circumstances B is seen', thereby forming context-dependent memory representations. This learning mechanism also enables the HTM to make predictions: after the $A \rightarrow B'$ sequence is recognized, the HTM can put certain C' cells in a predictive state based on previous learning records. If C is input next, the prediction is validated, and the synaptic connections are strengthened. If the prediction is incorrect, however, the corresponding synapses are

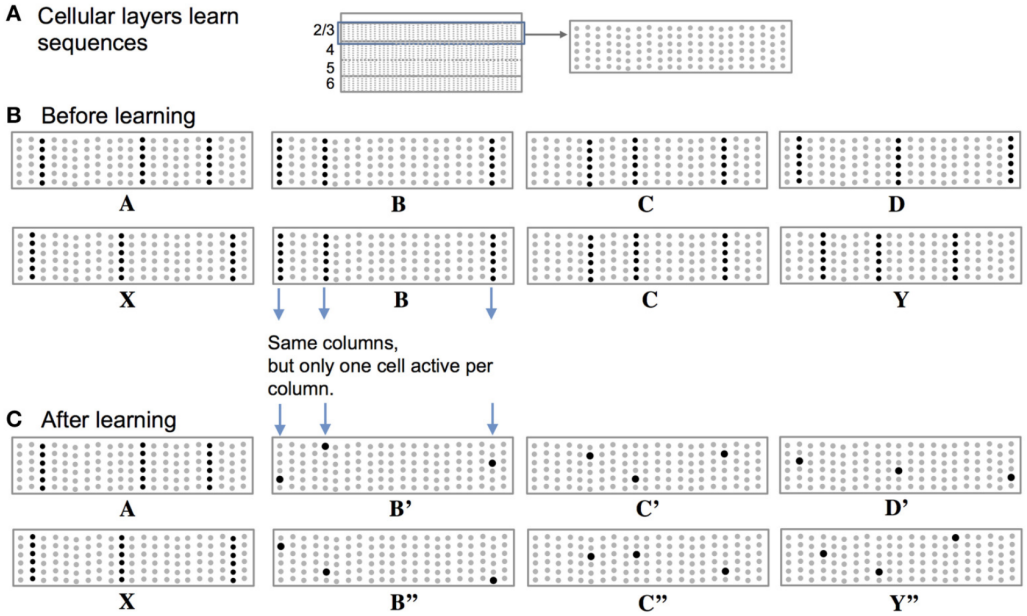


Figure 2.15: This figure illustrates how the HTM algorithm distinguishes between input sequences in different contexts [64]. It specifically shows the changes in state before and after learning. (A) Structural overview: Each column contains multiple cells, represented by small dots in the figure, which represent different temporal contexts at the same spatial location. (B) Before learning: Before learning, the model can only recognise spatial patterns. This means that patterns A, B, C, and D, as well as patterns X, B, C, and Y, activate the same columns. This indicates that the same SDR data is received. All cells in each column are activated, forming bursting columns. (C) After learning: After temporal learning, the model activates different cells for each context. Only the first input has a bursting column. Now, 'B' activates different cells in different contexts (labelled 'B' and 'B', respectively), rather than activating the entire column.

weakened.

In conclusion, HTM is a biologically inspired computational framework that has been modeled after the neocortex. It consists primarily of the SP and TM. The SP transforms input data into SDRs, ensuring robustness and noise resilience through overlap calculations, competitive inhibition, Hebbian learning, and adaptive boosting. TM extends these spatial encodings by incorporating temporal context, learning sequential patterns, and predicting future inputs. The integration of these two components, spatial encoding and temporal sequence learning, enables HTM to process complex spatio-temporal data effectively.

2.4. Existing Hardware Implementations of HTM

HTM algorithms have shown potential in pattern recognition, sequence prediction, and anomaly detection. However, their highly parallel and memory-intensive computational characteristics have exposed bottlenecks in energy consumption and latency when running on general-purpose CPUs/GPUs.

This section is organized around two representative works.

- The Reconfigurable Neuromorphic HTM Architecture by Zyarah and Kudithipudi, which demonstrates the design concept of "synthetic synapses" and flow networks [19];
- Second, the Memristor-Crossbar HTM SP by Truong et al., each column of synapses is implemented as a parallel memristor crossbar array fragment [20].

This section aims to summarize the key challenges and development trends of HTM hardware design by comparing the differences in architecture, storage organization, parallel granularity, and energy efficiency performance.

2.4.1. Reconfigurable Neuromorphic HTM architecture

In the field of digital reconfigurable circuits, Zyarah and Kudithipudi's work is representative [19]. They proposed the first end-to-end HTM FPGA solution that simultaneously covers SP and TM. The

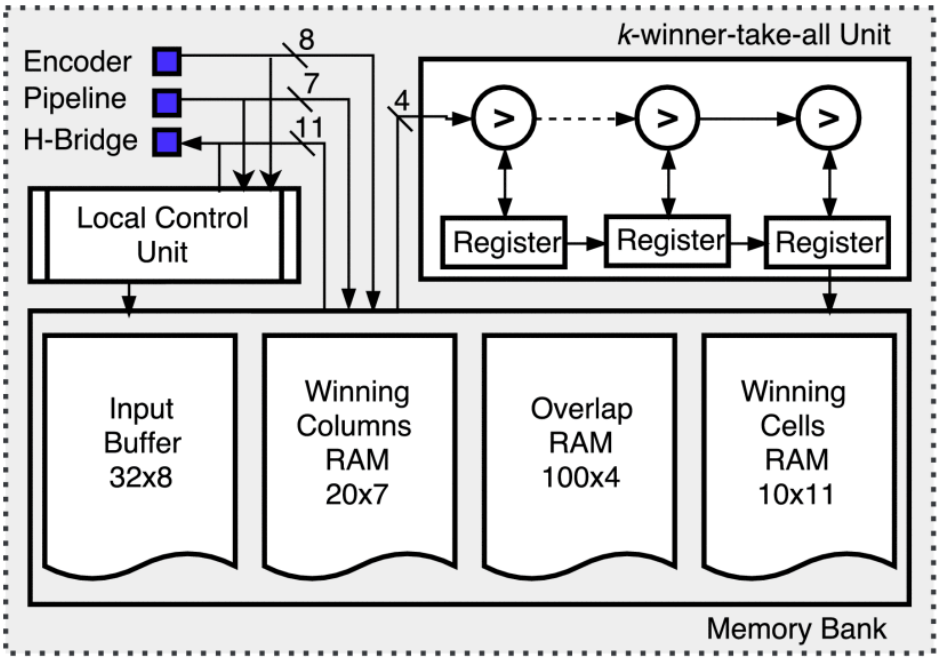


Figure 2.16: The figure shows the architecture of the MCU: The encoder writes the input sequentially into the input buffer in 8-bit packets. Subsequently, each column is returned via a 7-bit pipeline bus with 4-bit overlap values and cached in the overlap RAM. The built-in k-winner-take-all comparison chain selects the active columns based on this, writing the 7-bit column index and 11-bit column-cell identifier to the Winning Columns RAM and Winning Cells RAM, respectively. Finally, the results are broadcast to the column array via an 11-bit H-Bridge bidirectional network, forming an efficient closed-loop data flow from encoding to learning. In the memory-bank labels, the notation rows Œ word-width (bits) is used: for example, Winning Columns RAM 20 Œ 7 holds 20 words, each 7 bits wide [19].

core of their innovation is the 'synthetic synapse' concept, which involves compressing information about potential and dynamic connections into on-chip storage. LFSR or ROM are used to generate receptive fields, reducing wiring area while maintaining dynamic connectivity and enhancing scalability. The architecture employs a column-parallel pipeline and an H-bridge network to map a region of 100 columns, each with three cells, onto the Xilinx Zynq-7000, thereby supporting subsequent cascaded expansion. Figure 2.16 shows the architecture of the microcontroller unit (MCU) in the designed system. The local control unit (LCU) on the left receives control and data packets from the encoder, pipeline, and H-bridge. The memory bank at the bottom integrates four on-chip RAMs: input buffer RAM, overlap RAM, winner column RAM, and winner cell RAM. These are used to temporarily store input segments, column overlap degrees, and competition results. The k-winner-take-all unit in the top right corner consists of a pipeline comparator chain and a register array. This enables real-time selection of the 20% of columns with the highest activation levels after reading the overlap RAM. This list is then written back to the winning column and winning cell RAM and broadcast via the H-bridge to the columns and cells to complete the learning or prediction process.

In classification tasks, the FPGA version of SP + SVM achieved noise-robust accuracy rates of 91.16% and 90% on the MNIST and European numberplate font (EUNF) datasets, respectively. Performance remained stable even when the training set accounted for only 20% of the data, which demonstrates the advantages of HTM in terms of small sample sizes and robustness to noise. For sequence prediction, HTM can perform first- and second-order predictions on five-digit number sequences, achieving a maximum accuracy of 95%. When 25% noise is added, the accuracy decreases by approximately 12% to 20%, verifying the robustness of the SDR representation. In terms of performance, the design trains SP at 100 MHz in just 5.75 μs per sample, achieving an acceleration of approximately 1,364 $\times$ compared to the MATLAB software implementation. The resource overhead is 63 k logic units and 9.1 k memory cells, which is theoretically scalable to 300 columns. In a 65 nm ASIC estimation, the single-cell power consumption is approximately 1.39 mW, offering a significant advantage over similar works at the same scale.

Although this study successfully designed a hardware accelerator for the HTM algorithm and

achieved promising experimental results, there are still many shortcomings. Firstly, while the paper claims to have 'fully' implemented SP and TM on an FPGA, the authors separated the two components during validation: they first trained SP independently, feeding its sparse representation to external SVM and k-NN software classifiers to evaluate static recognition performance on MNIST and EUNF images. Then, TM was trained separately to perform first- and second-order predictions on short sequences of five digits.

Furthermore, the TM experiment itself is highly constrained. The network scale is limited to 100 columns by three cells, and the predicted sequence is reduced to a single-channel, five-frame synthetic digital stream. Each frame is an independent binary image, rather than a true multi-frame visual sequence or a task involving longer-term dependencies. Since there are only three cells per column, the amount of context that can be maintained in parallel is extremely limited. This means that higher-order or cross-sample sequence structures cannot be tested, which raises doubts about the scalability of TM in practical applications.

At the hardware level, there is also a trade-off between scalability and energy efficiency. Implementing a 100-column region requires 63,449 logic slices and 9,108 memory slices. In theory, the authors estimate that the number of columns can only be scaled up linearly to around 300 on the same Zynq-7045 device. This is still far from the thousands of columns commonly used in HTM. Furthermore, the power consumption figure of 1.39 mW per cell is derived from static estimates using a 65 nm back-end synthesis tool. To save routing area, the proposed 'synthetic synapse' maps potential connections to local LFSR/ROM; however, during computation, addresses must be generated and compared one by one. The paper acknowledges that this prolongs overlapping computation time; as the number of columns and receptive fields increases, real-time performance will deteriorate rapidly.

Overall, this work is an important step towards making HTM a reconfigurable digital core and achieving the design of an HTM digital circuit accelerator. However, more comprehensive and in-depth testing is still required for large-scale deployment targeting high-dimensional, long-sequence, low-power applications. The paper also identifies deficiencies in HTM about network scale, energy consumption models, and online adaptability.

2.4.2. Scalable Memristor Cross-Array for HTM Spatial Pooler

In addition to digital circuit accelerator designs for HTM, such as FPGA/ASIC, mentioned above, there are also studies that use primarily analogue circuit design to create HTM acceleration modules. For instance, Truong et al. proposed the Spatial-Pooling Memristor Crossbar [20], which maps HTM's spatial pooling unit in its entirety to a 400 × N analogue crossbar array, resulting in a compute-in-memory device computing model. The authors applied a 20 × 20 pixel perceptron vector directly to the 400-row crossbar array, enabling the array to autonomously generate SDRs, with an average of only around 2% of columns activated. Through continuous online training, the activation frequency of columns balances out over time. Even when the input is mixed with 50% noise, the similarity of the output SDR remains at 0.6, demonstrating significant robustness improvements. Even when the number of columns is increased to 4,096, the sparsity remains at 2%, while the recognition rate on MNIST increases to 95%; the 256-column version achieves a recognition rate of 76%, indicating a positive correlation between array size and representational capability.

Figure 2.17 illustrates the workflow of the proposed memristor cross-array spatial pooling module within a single inference-learning cycle. The IMG2VEC module on the left first converts a 20 × 20 MNIST grayscale image into a 400-dimensional vector. This vector is then sequentially applied to the 400 input rows ($X_0 - X_{399}$) of the crossbar array. Each vertical column in the array corresponds to a 'column' in the HTM. Within each column, only 25 randomly selected crosspoints are considered potential synapses. In memoristers, a low-resistance state (LRS) corresponds to a connected synapse with high conductance, while a high-resistance state (HRS) corresponds to a disconnected or weak synapse with low conductance. As shown in Figure 2.17, solid black circles (LRS) represent active synapses whose weights have exceeded the connection threshold, while hollow circles (HRS) represent synapses that are not yet connected or have been weakened. These two states are implemented in hardware using low/high-resistance memristors, which can switch between HRS and a LRS to represent inactive and active synapses respectively. When an input vector is applied to the memristor array, active synapses in each column simultaneously conduct, forming a column current (I_c). This current passes through a current-to-time converter at the columns base, where a programmable memristor (b) adjusts the current as a boosting factor. Less active columns receive a higher boost (lower resistance),

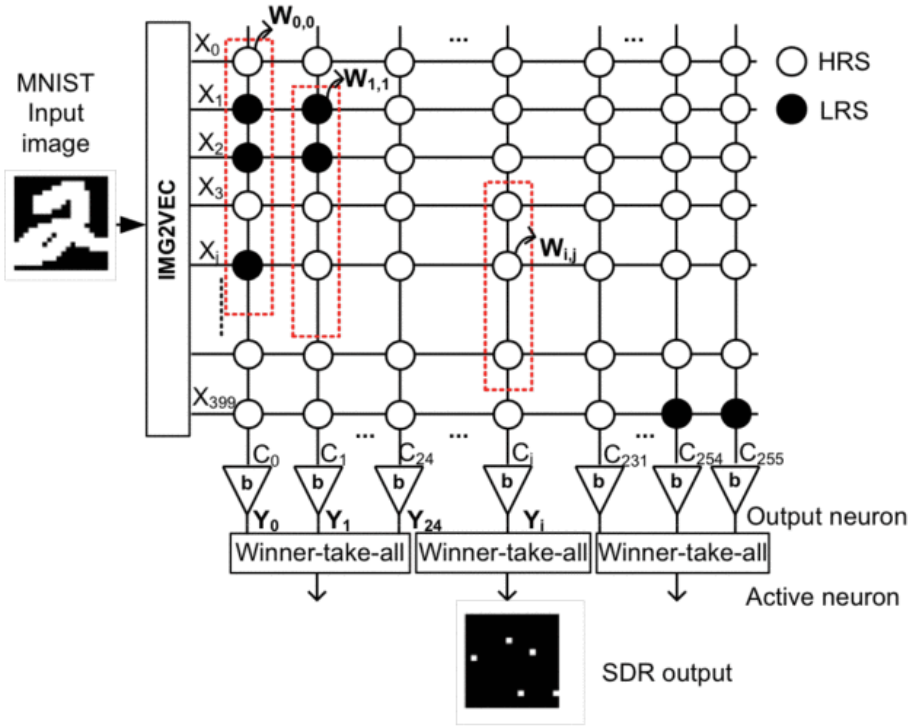


Figure 2.17: Memristor-crossbar realisation of an HTM SP. A 20 × 20 MNIST image is applied to the 400 input rows ($X_0 X_{399}$). Each vertical slice represents an HTM column with 25 potential synapses stored as low-resistance (black, LRS) or high-resistance (white, HRS) memristors. Column currents are modulated by programmable boost resistors (b) and fed to local winner-take-all circuits; the earliest-spiking 2% columns form the SDR at the output [20].

increasing their chance of activation, while frequently active columns receive less boost to promote balanced usage.

A Winner-Take-All inhibition network selects the earliest spiking 2% of columns in each group, marking them as active neurons. These winning columns, visualized as sparse white dots in the output SDR, have their synapses strengthened through Hebbian learning, completing the overlap-inhibit-learn cycle. Overall, leveraging memristors' non-volatile conductance enables direct and efficient transformation of input pixels into sparse HTM representations with semantic retention and adaptability.

This study used a memristor crossbar to accelerate the SP component of the HTM algorithm. Simulation verification showed that the SP component could achieve 95% accuracy in classifying the MNIST dataset using SDR data. However, the study also has some limitations. Firstly, only the HTM-SP component was implemented, not the TM component. While the authors acknowledged in the introduction that the TM component requires learning and predicting SDR sequences, this work only proposed using a memristor array to convert perceptual vectors into single-frame SDRs. There was no attempt to concatenate these SDRs to provide temporal context, nor was there any mention of a distal-synapse training circuit. Consequently, the current implementation is limited to single-frame feature extraction and cannot achieve complete sequence learning and prediction functionality. In order to extend to full HTM and larger-scale sequence tasks, issues such as device controllability, fine-grained weight updates, array interconnections and power consumption still need to be addressed.

Secondly, the verification methods are limited to SPICE behavioural simulation. All area, power consumption, and recognition rate data are based on device models obtained using Cadence Spectre on a 0.13 μm CMOS process in the simulation software. The chip has not been fabricated and the effects of ageing, write endurance, or temperature drift on weight drift have not been tested. Therefore, there is no experimental evidence to suggest whether column current and WTA decisions will remain reliable in the presence of real-world process variations, random resistance distributions, and write pulse noise.

Thirdly, scalability depends heavily on the characteristic of extreme sparsity and the local interconnection assumption. To maintain acceptable leakage current and metal line voltage drop limits, the design requires an SDR sparsity of 2% and restricts the WTA suppression window to 25 columns.

Nevertheless, the peak read current for a 100 × 100 array remains close to 20 mA at a read voltage of 1 V. The researchers also acknowledge in their paper that larger arrays or higher activation rates would rapidly amplify leakage current and IR drop issues. Furthermore, the simplified topology, which has a maximum of 25 potential synapses per column and 25 columns per group, weakens the receptive field and global inhibition capabilities. This limits its expressive power for high-resolution inputs or large-scale features.

Finally, the weight updates and learning granularity are limited. The circuit uses two-state or low-bit-count Hebbian adjustment. Switching to STDP would be more biologically realistic and capable of supporting higher-order sequence learning, but would require precise pulse generators and complex control logic. However, the gradual conductance adjustment of filament-type memristors is inherently difficult to implement stably. This limits weight resolution and makes achieving fine-grained increases or decreases in thousands of distal synapses challenging when scaling up to TM in the future.

2.4.3. Challenges in HTM Hardware Implementation and Future Design Directions

Based on our analysis of the two typical implementations - the digital reconfigurable/ASIC approach represented by Zyarah and Kudithipudi, and the memristor crossbar array approach proposed by Truong et al. we can identify current issues and bottlenecks in the research of hardware implementation for HTM accelerators. This allows us to propose potential research directions and considerations for future designs.

1. **Lack of HTM Integrity:** Although existing digital hardware implementations have successfully mapped HTM's SP and TM modules onto chips, the two modules are usually tested and verified separately. Meanwhile, solutions based on memristor crossbar arrays have only implemented the SP component, omitting the TM module, which is crucial for sequence learning. The absence of an online closed-loop verification process incorporating encoding sequence learning prediction decision-making prevents HTM from demonstrating its full temporal modelling capabilities and advantages on hardware platforms. This may be because current TM algorithms are incomplete and complex, causing existing research to either avoid or struggle to implement the full HTM workflow. Therefore, future research should first focus on refining and maturing the TM algorithm theoretically. Suppose the TM component of HTM continues to face algorithmic issues or limitations in the short term. In that case, it may be appropriate to introduce or borrow more mature sequence modelling methods from the deep learning field as an alternative or transitional solution. Additionally, the practical feasibility and scalability of the TM algorithm implementation at the hardware level must be thoroughly investigated, including key issues such as storage resource requirements, power consumption bottlenecks, the ability to finely adjust weight updates, and collaboration with the SP module for optimization.

2. **The difficult balance between storage density, bandwidth, and power consumption:** Existing FPGA or ASIC design solutions are usually limited by the capacity of the on-chip SRAM and registers, as well as the amount of global interconnect resources available. Once the number of columns in an HTM module exceeds a few thousand, resource bottlenecks are quickly approached. Although memristor crossbar array solutions offer significant advantages in terms of storage density, they are also severely constrained by sneak-path leakage current, metal line voltage drop (IR drop) and write energy consumption. Consequently, these bottlenecks must be mitigated through extremely sparse activation modes and localised WTA competition strategies. Future HTM hardware designs must conduct a more in-depth analysis of HTM algorithm requirements in areas such as data storage and weight updates. They should also explore measures such as reducing weight quantisation precision, enhancing parallel read/write capabilities, and optimising storage organisation structures. This will help to overcome current resource and energy constraints and achieve more efficient, scalable hardware implementation solutions.

3. **System-level integration:** For the HTM accelerator to be deployed in practical applications, it must be tightly integrated with system components such as the main control SoC, sensor modules, and edge MCUs. Previous research has focused primarily on the implementation methods and performance optimization of the HTM hardware itself without considering how the HTM accelerator can collaborate effectively with other modules in real-world scenarios. Therefore, future

research could consider incorporating lightweight, open-source RISC-V CPU cores to enable flexible control and task scheduling of the HTM accelerator. It is also essential to analyse critical issues such as control signal transmission strategies between the CPU and HTM accelerator, pipeline coordination design and software/hardware interface definitions, to ensure the HTM accelerator can be used efficiently and reliably in real-world applications.

Future research on the HTM algorithm and its hardware accelerators will build upon the analysis presented in this section, with more detailed discussions provided in the subsequent chapter.

3

Methodology

In the aforementioned literature review, we noted that existing HTM hardware accelerators primarily focus on SP. Systematic exploration of algorithmic improvements and hardware mapping for TM is lacking. This results in an inability to conduct more in-depth research and practical applications of the complete HTM algorithm. To address this, this study considers on the complete HTM framework, using MNIST handwritten digit classification as a benchmark task to evaluate the HTM TM component's learning and prediction capabilities for simple tasks.

3.1. Research Objectives

This study aims to address the lack of application of the full HTM algorithm and verify the feasibility of using the HTM algorithm or combining SP and deep learning algorithms, as a replacement for TM. The following research objectives have been established in this section and will be verified in subsequent implementations and experiments:

1. Use the full HTM algorithm to carry out simple image classification tasks with an accuracy rate of at least 90%.

2. If sufficient accuracy cannot be obtained with the full HTM algorithm, then under the condition that SP remains unchanged, introduce deep learning algorithms such as FCNN/GRU as TM functional replacements, and achieve classification accuracy equivalent to or higher than traditional deep learning algorithms.

3. Complete the mixed-architecture RTL design and 40 nm ASIC synthesis using a RISC-V CPU controller to enable real-time on-chip inference.

4. Compare the energy-throughput performance of ASIC accelerators with CPU.

3.2. Software Architecture

Figure 3.1 provides an overview of the software pipeline. The raw MNIST images are first binarized, converting each 28 x 28 grayscale frame into a 784-bit binary vector. This vector then passes through the SP, which acts as an encoder, transforming the high-dimensional input into an SDR. The resulting SDR is then forwarded to one of four sequence-learning back-ends: TM, FCNN, GRU or lightweight minGRU. Each back-end is trained to map the SDR to a class score vector. A soft-max layer then selects the predicted digit label. Thus, the framework enables us to compare the native HTM pipeline (SP + TM) with deep-learning alternatives while maintaining an unchanged SP front-end.

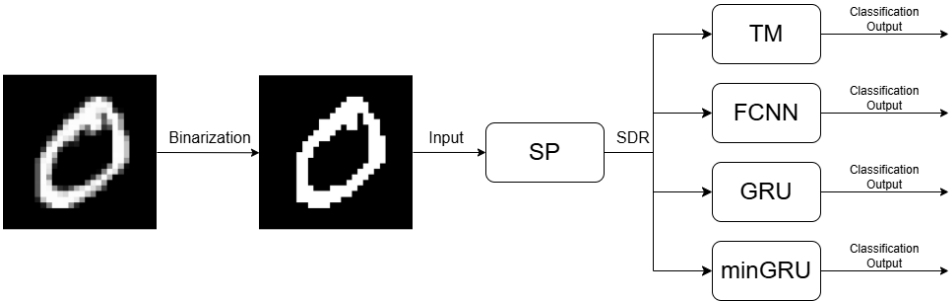


Figure 3.1: Proposed software implementation. The original MNIST image undergoes binarization and is then fed into the SP to generate an SDR. The resulting SDR is evaluated by four classification modules in parallel: the original TM of the HTM algorithm, FCNN, GRU, and minGRU. The outputs of each classifier are used to compare recognition performance under the same input encoding.

3.2.1. Hardware Implementation Methodology

To efficiently map the software pipeline onto an SoC, this work adopts the Ibex RISC-V processor as the control core and designs dedicated accelerators for the SP and NN computation chain. The development follows a top-down flow of "requirements → algorithm → architecture → RTL → verification":

1. **Cycleaccurate modelling**
   A C-based cycle-accurate model running on Ibex estimates loop counts, memory traffic, and MAC operations for every module, guiding resource allocation and parallelism decisions.
2. **RTL implementation and synthesis**
   Hardware blocks are described in SystemVerilog, functionally as well as timing-verified in ModelSim, and synthesised with Cadence Genus.
3. **Key performance evaluation**
   Focus on system throughput, single image inference latency, and overall chip area as quantitative benchmarks for subsequent hardware optimization and iteration.

This methodology delivers a quantifiable hardware acceleration framework that preserves the original software algorithms while meeting practical application requirements.

3.3. Evaluation Method

This section will describe the evaluation scheme and the experimental conditions of this study. First, we define key metrics, including classification accuracy, inference latency, throughput, energy consumption per sample, and ASIC resource utilization. Next, we introduce the partitioning and pre-processing methods used for the MNIST and Sequential-MNIST datasets, and provide comparison baselines such as HTM-TM, FCNN, GRU, and existing FPGA accelerators. Finally, we provide a detailed description of the hardware environment, the testing process, and the performance testing methods employed.

3.3.1. Performance Metrics

This work evaluates the proposed accelerator for four key metrics, summarised below. Each metric is formally define,d and the corresponding measurement procedure is described.

1. **Classification Accuracy** (Acc)

$$\text{Acc} = \frac{\text{Number of correct predictions}}{\text{Total number of test samples}}. \tag{3.1}$$

   Computed on the 10 000-image MNIST test set.

2. **Latency per Inference** (Lat)

$$\text{Lat} = \frac{t_{\text{end}} - t_{\text{start}}}{N_{\text{samples}}}, \tag{3.2}$$

   where t_{start} is asserted when the first pixel enters the accelerator and t_{end} when the soft-max result becomes valid.

3. **Throughput** (FPS)

$$\mathrm{FPS} = \frac{1}{\mathrm{Lat}} \left[\mathrm{frames\,s^{-1}}\right], \tag{3.3}$$

measured over a batch of 5,000 images to amortise I/O overhead.

4. **PPA: Power, Performance, Area**

- *Power* (P) Total on-chip power, including both dynamic (P_{dyn}) and leakage (P_{leak}) components reported by Cadence Genus.
- *Performance* ($\mathcal{P}$) Classification accuracy and latency per inference in hardware.
- *Area* (A) Standard-cell footprint after place-and-route, reported in mm^2 for a 40 nm technology, excluding I/O pads:

3.3.2. Experimental Dataset

This study uses the MNIST data, which is widely used in the field of artificial intelligence. The MNIST dataset (Modified National Institute of Standards and Technology database) represents one of the most classic and widely adopted benchmark datasets in the fields of machine learning and deep learning. This dataset comprises 70,000 grayscale images of handwritten digits, each with dimensions of 28×28 pixels, encompassing ten digit classes from 0 to 9. The dataset is partitioned into 60,000 training images and 10,000 test images. Each image has undergone standardized preprocessing: digits are centered within the image frame, and pixel values range from 0 to 255, where 0 represents white and 255 represents black. The MNIST dataset originates from authentic handwritten digit samples collected by the National Institute of Standards and Technology, which have been subjected to size normalization and centering procedures to ensure data consistency and comparability [77].

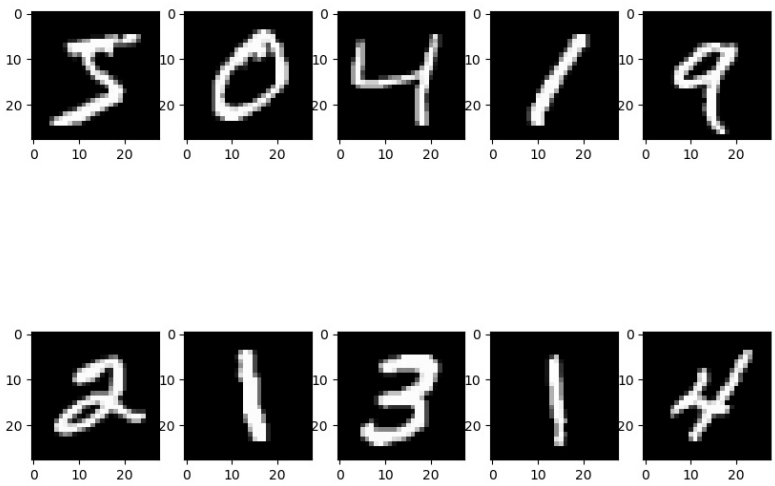


Figure 3.2: Handwriting Images in the MNIST Dataset.

Due to its moderate scale and simple structure, the MNIST dataset has established itself as an ideal testing platform for validating the effectiveness of novel algorithms [78]. Another significant characteristic of this dataset is its relatively low computational complexity, enabling rapid model training and evaluation even on standard hardware configurations. This accessibility allows researchers and

practitioners to focus on algorithmic development without concerns regarding computational resource constraints.

The study relies on the MNIST handwritten-digit benchmark in two complementary forms. For pipelines operating on a single static frame namely the SP + FCNN baseline we adopt the standard MNIST split (60,000 training images and 10,000 test images, each measuring 28 × 28 pixels). Each image is converted into a binary representation and encoded by the SP, and the resulting SDR is then sent directly to the fully connected network for classification.

To evaluate sequence-aware models, we use a sequential MNIST variant in which each image is partitioned into 28 raster rows and streamed to the network one row at a time [79], Sequential MNIST is a variant of the MNIST dataset in which the 28×28 image is split into 28 rows, yielding a 28-step binary sequence that preserves spatial locality while introducing temporal structure. These rows are then fed into the model sequentially, followed by the image label, allowing the model to learn from the ordered data [79]. This sequential input is fed to the complete HTM (SP + TM) so that the temporal memory predicts the digit label only after processing the final row. It is also fed to the hybrid SP + RNN back-ends (GRU and minGRU), which replace TM. Using identical training and testing splits for the static and sequential forms ensures that comparisons of accuracy across architectures remain fair and directly interpretable.

3.3.3. Baselines

To quantify the performance gains of the accelerator in this study, we established two benchmarks, one for the software and one for the hardware. At the software level, we adopted the 'SP + SVM' scheme proposed by Zyarah as the baseline. First, the SP generates a 2% sparse SDR, which is then classified by a linear SVM. This approach achieved over 91% MNIST accuracy in its FPGA experiments and can serve as an algorithmic reference [19]. On the hardware side, we implemented the same algorithm in pure C on the Ibex RISC-V processor and used performance counters to measure the number of clock cycles for the entire inference process. We will then directly compare the cycle overhead of the custom accelerator with that of the Ibex software baseline using the same dataset and clock frequency conditions, in order to evaluate the acceleration ratio and the improvements in energy efficiency achieved under area- and power-constrained conditions.

4

Software Implementation

This chapter provides detailed information on the software-level implementation involved in this research. First, Section 4.1 provides an overview of the concrete implementation of the HTM algorithm and analyzes its key parameters. Through experimentation, certain deficiencies of the HTM algorithm were identified; hence, Section 4.2 and Section 4.3 explore how the SP component can be combined with FCNN and GRU models, respectively. Based on the shortcomings of the GRU model observed during implementation, Section 4.4 introduces the development of minGRU and analyzes the advantages and disadvantages of this improvement through experimental data. Section 4.5 then focuses on the application of the attention mechanism in neural networks, highlighting its advantage in enabling temporal models to capture long-term dependencies, which in turn optimizes the performance of the minGRU model. Finally, Section 4.6 presents a preliminary comparison of hardware deployments for each implemented model, offering insights for future hardware-based implementations. The objective of the discussions in these sections is to establish a solid software foundation for the subsequent experiments and analysis.

4.1. Implementation of HTM

Based on Section 2.3, we have established that the HTM algorithm is primarily targeted at problems involving sequential inputs. Once trained, it can predict upcoming inputs by using the current input in conjunction with previously observed information. However, a review of the relevant literature suggests that most existing research focuses on classifying the standard MNIST dataset [19] or predicting relatively simple time series data [80]. Therefore, this study aims to investigate how well the HTM algorithm performs in classifying the Sequential MNIST dataset, thus fully exploiting its ability to learn and predict from sequential input.

Given the nature of the sequential MNIST shown in Section 3.3.2, our goal is for the HTM algorithm, after training, to be able to accurately predict the correct label after receiving all 28 sequential rows, thereby demonstrating its ability to learn from and adapt to sequential inputs. As demonstrated in Figure 4.1, the HTM algorithm's inference process involves several key steps. Initially, the binarized MNIST data is fed into the SP module. The SP then generates a corresponding SDR, the length of which is indicative of the number of columns in the TM module. For instance, an SDR length of 1024 implies that there are 1024 columns in the TM. Within the SDR yielded by the SP, the presence of a 1 at any given time step indicates the necessity for an active cell within the corresponding TM column. If a column does not contain any active cells, it enters a bursting state, meaning that every cell in that column becomes active. Within the TM module, the active cells at a given time step generate predictive cells based on the synaptic weights between cells. As illustrated in the accompanying figure, the green cells represent the predictive cells produced by the active cells, which will become active in the subsequent time step. The prediction is deemed correct when the active cells are generated in the appropriate columns. Based on the existing open source Python implementation of the HTM algorithm [63], we modified its parameters according to the requirements of the Sequential MNIST dataset. The detailed configuration of the SP and TM, including all parameter settings and their corresponding explanations, has been relocated to Appendix for clarity and completeness. Section A.1 summarises the parameters for the SP, and Section A.2 shows the parameters used for the TM part.

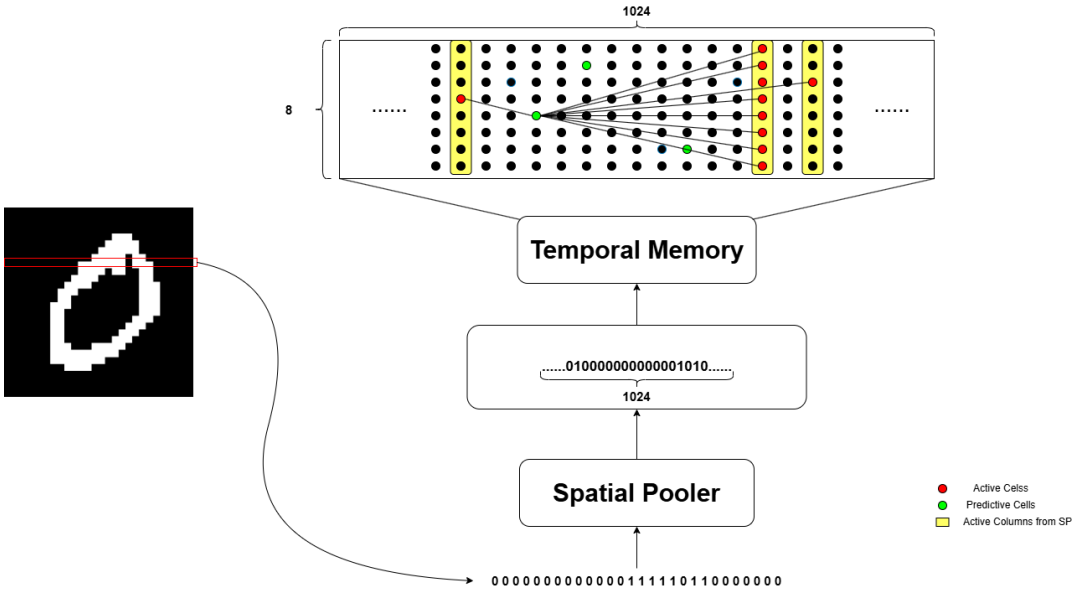


Figure 4.1: The diagram illustrates the inference process of the HTM algorithm. First, the binarized MNIST data is fed into the SP module. As shown, the SP generates an SDR of a given length, where each bit corresponds to a column in the TM module. When a bit in the SP output is set to 1, it indicates that the corresponding column in the TM should contain active cells at that particular time step; if no active cells are present, the column enters a bursting state. In addition, the active cells in the TM at the current time step evolve from the predictive cells generated at the previous time step. If the active columns match the predicted columns, the prediction is considered correct. Finally, active cells are used as the output of this time step.

In addition, the MNIST images must be binarized before they are fed into the HTM algorithm. Since each MNIST image is in grayscale, with pixel values ranging from 0 to 255, a threshold of 128 is sufficient: pixels below this threshold are set to 0. In contrast, those above are set to 1. Figure 4.2 shows an example of an MNIST image before and after this binarisation process.



Figure 4.2: Comparison of the MNIST image before and after binarisation. On the left is the original image and on the right is the image after binarisation. It can be seen that the image retains all the original information.

In the beginning, the HTM algorithm is first trained and validated on the input of a single image to test whether the algorithm can correctly predict the corresponding label of the image. Figure 4.3 shows the prediction results at each time step for the Sequential MNIST dataset using the HTM algorithm implemented in Python. The accuracy is computed as follows: after training, the SP part generates an output for each time step of the Sequential MNIST image. Then, the prediction results from the TM part are compared with the corresponding SP outputs to compute the accuracy. In this experiment,

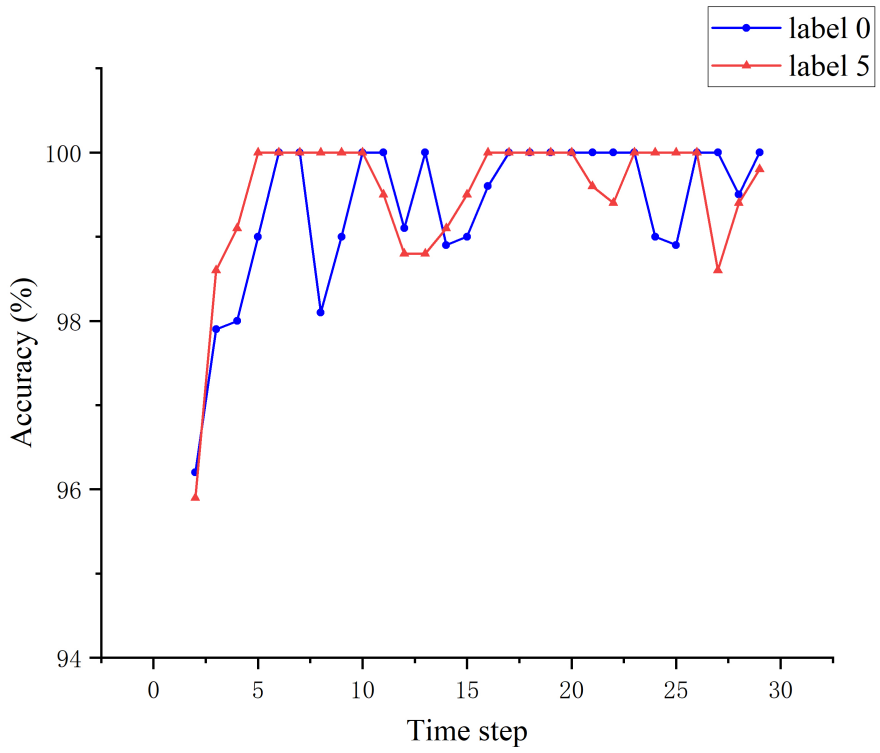


Figure 4.3: Prediction accuracy of HTM over time when processing Sequential MNIST images. The blue curve corresponds to predictions made on a digit 0 image across 30 time steps, while the red curve shows results on a digit 5 image. Accuracy is computed at each step based on overlap with the expected SDR for that digit.

we evaluated the HTMs temporal behavior by sequentially feeding binarized MNIST images from all digit classes (0 - 9) over 30 time steps. For illustration, we present the prediction accuracy curves for digit 0 and digit 5 as representative examples to analyze stability and convergence. As shown in the figure, accuracy is reported from the second step onwards because the HTM algorithm starts to generate predictions for the second step and beyond based on the information from the first step. Initially, the prediction accuracy is the lowest due to the limited information available from only the first time step. However, as more timesteps are provided, the accuracy increases and eventually stabilizes at around 100%. This demonstrates that the HTM algorithm has a strong ability to learn sequential inputs, effectively integrating previous outputs to predict the subsequent input at each timestep accurately.

Based on the above experiment, it is clear that for a single image, the HTM model can effectively learn from sequential inputs and successfully predict the next input for each time step. Consequently, proceeded to input the entire Sequential MNIST dataset into the HTM model to see if it could accurately predict the label for each image. After training the HTM model on the complete Sequential MNIST training set, the overall accuracy is less than 40%. Indeed, when the full dataset is used, the model has to accommodate a diverse set of input patterns, which can lead to interference between different sequences and greater sensitivity to noise.

his indicates that the HTM model has good learning and prediction capabilities when dealing with a single input image. However, as the dataset size increasesspecifically after the model has learned the entire Sequential MNIST dataset, its prediction performance degrades, and it can no longer make accurate predictions on the overall data. After reviewing the HTM algorithm and consulting related information, it is found that the memory capacity of the existing HTM models is limited, and the research on the HTM model for time series data is limited to relatively simple waveform data as well as a single sequential MNIST image.

This result can be attributed to several factors. First, the HTM algorithm requires the input to be encoded as high-dimensional sparse binary vectors, typically retaining about 2% active bits [81].

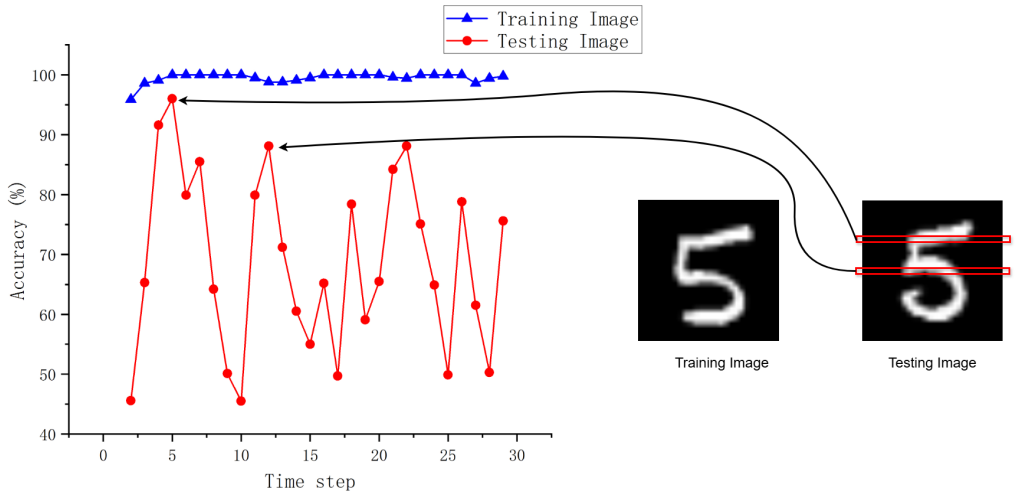


Figure 4.4: For digit 5, the accuracy of the training image and one test image. For each similar but slightly different time step, relatively high accuracy can be achieved at the current time step, but these differences cause the new columns to enter a bursting state, which causes a significant drop in the accuracy of subsequent predictions. Additionally, the accuracy curve does not commence at time step 0. This is because no data is included in the first two rows of the MNIST image and no corresponding output is produced.

Suppose the original data is not naturally sparse. In that case, it becomes necessary to increase the coding dimensionality to achieve the required sparsity, which in turn greatly increases the model size and computational load. On the other hand, if the data contains a large number of rare patterns that occur only a few times, the HTM's local Hebbian learning mechanism may struggle to adequately learn these infrequent patterns because synapses require multiple repeated co-activations to cross the threshold and form stable connections [82]. As a result, frequent patterns dominate the learning process, while predictions for rare patterns tend to be inaccurate, limiting the model's overall adaptability and generalization across the data distribution.

Second, as the volume and diversity of data increase, the HTM model often encounters novel sequences. Without established predictive links, it initially responds with a full-column bursting, leading to inaccurate predictions. As shown in the Figure 4.4, the continuous influx of new sequences dilutes the impact of previously learned patterns, keeping the overall error rate high until repeated exposure gradually reinforces these patterns [83]. In extremely large, evolving datasets, the model may be constantly adapting to new inputs, causing the error rate to remain high or even increase. In addition, increased noise and subtle variations in such datasets further challenge the limited generalization capability of HTM, as even small changes are treated as entirely new inputs.

In summary, these limitations highlight that while the HTM effectively captures frequently occurring local temporal patterns through sparse, high-dimensional representations, it still struggles to maintain accurate predictions in the face of non-sparse data, rare patterns, and data-heavy sequences. This suggests the need for further improvements. One approach is to combine deep learning methods with SP, using the global optimization capabilities of deep networks to complement the strengths of SP in local sequence learning and sparse representation [84]. The following sections present the implementation process for combining SP with FCNN and RNN models.

4.2. Combination of SP and FCNN

Based on the above analysis, we can conclude that HTM algorithms still face challenges when beyond toy scripts. Reviewing the related literature, we found that most existing studies focus on combining SP with deep learning algorithms to improve overall performance [85] [74]. In this architecture, SP can be used as an encoder to transform the input data and generate sparse representations. Since the output generated by SP is not only highly sparse but also has good stability and interpretability, it can be used as an effective input to deep learning models to improve the learning capability and generalization performance. This combination approach has shown good application potential in several studies.

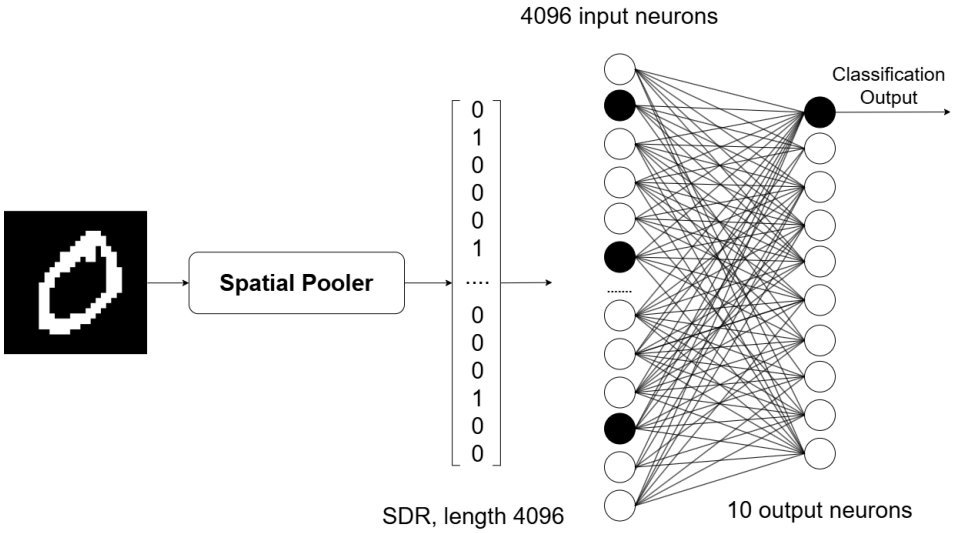


Figure 4.5: The diagram illustrates the combination of SP and FCNN. In the first instance, the MNIST image is binarized and input into SP, which is a departure from the previous method. Rather than dividing the MNIST image into 28 rows and inputting them sequentially, it is inputted into SP at one time. Subsequently, the SDR data generated by SP is input into FCNN. It is evident from the figure that the size of the FCNN input dimensions is equivalent to the length of the SDR data. Finally, the classification output is generated by the FCNN.

Therefore, based on the existing work in this section, I will further try to combine SP with FCNN and explore its applicability and optimization strategies in different data scenarios to provide new ideas for integrating HTM and deep learning.

As demonstrated in Figure 4.5, the integrated model architecture combines the SP with an FCNN. As opposed to the conventional approach, which involves segmenting the MNIST image into 28 rows for sequential processing, this model directly inputs the entire binarized image into the SP module in a single step. Upon receiving the complete image, the SP module extracts global features and generates a corresponding SDR. The length of the SDR directly determines the input dimension of the subsequent FCNN, ensuring that the two modules are perfectly aligned, as depicted in the diagram. Following the encoding process by the SP, the FCNN performs deep feature extraction and pattern recognition on the SDR data, ultimately producing accurate classification results. This integrated approach leverages the SP's strength in capturing comprehensive image information and the FCNN's capability in processing high-dimensional data.

During the training process, the input data is first passed to the SP module so that it can learn how to generate SDRs. Subsequently, these SDR data are passed to the FCNN for further training, with both modules jointly learning and optimizing their respective parameters throughout the process. In the actual implementation, it can be observed that the dimension of the SDR data generated by the SP has a key impact on the accuracy of the whole model. Higher-dimensional SDRs can contain more information, thus providing richer input features for FCNN and facilitating model learning. Moreover, since SDR is the input to FCNN, the increase of its dimension will also directly increase the input scale of FCNN, thus increasing the number of parameters and computational complexity of the model. From the experimental results shown in Figure 4.6, it can be seen that the accuracy of the model reaches its maximum when the SDR dimension reaches 128 at 97% accuracy. Before this, the model accuracy exhibited a linear increase with the SDR dimension. This is due to the relatively low complexity of the MNIST dataset, meaning that the model does not require extra bits to represent additional information once the SDR size is increased within a certain range.

4.3. Combination of SP and GRU

In the previous section, we observed that integrating SP with deep learning algorithms can achieve promising performance. However, the SDR produced by the SP typically contains only 2%-10% active bits. Directly feeding the entire SDR, which contains a large proportion of zero bits, into an FCNN yields minimal computational benefit but imposes additional hardware resource utilization and power

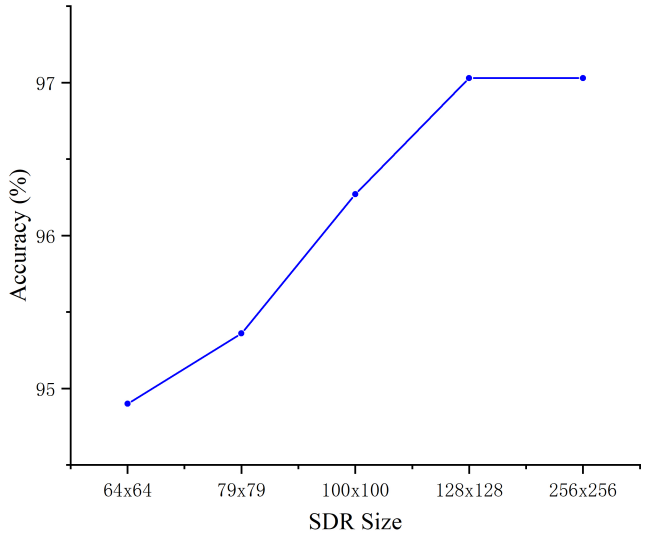


Figure 4.6: The relationship between the accuracy of the model integrating SP with FCNN and the size of the SDR produced by SP is demonstrated, where 79×79 is the baseline given in the source code [63]. It is evident that as the SDR size increases, the accuracy concomitantly improves. However, once the SDR size attains 128, no further enhancement in accuracy is discernible.

consumption. Consequently, we seek to further exploit the sparsity of the SDR by extracting only the positions of its active bits and passing this information to subsequent deep-learning modules.

More specifically, we first extract the indices of every 1 bit in each SDR and then pass these indices to the subsequent module. Since the number of active bits in each SDR varies depending on the input, as shown in Figure 4.7, the length of the index sequence also becomes variable. To handle variable-length sequence data, we can use a RNN, of which LSTM and GRU as previously described in Section 2.2. Considering our application requirements and ease of hardware deployment, we chose the simpler GRU model, which requires less computation, to process the extracted 1-bit indices. In doing so, we not only exploit the SDR's ability to preserve essential features through its inherent sparsity but also minimize the transfer of superfluous data, thereby reducing both hardware resource usage and power consumption.

In RNNs, such as the GRU, the hidden state is continuously updated at each time step to integrate and refine the network's internal representation of the sequential input data. By the time the last time step is reached, this hidden state has effectively accumulated and encapsulated all relevant historical information from the beginning of the sequence, thus serving as a comprehensive or global encoding of the entire input sequence. In other words, the hidden state at the final time step can be viewed as a compressed representation that encompasses the entire sequence context. Using this hidden state as input to a subsequent FCNN greatly simplifies the model architecture and reduces computational complexity, as it eliminates the need to concatenate outputs from individual time steps.

More importantly, the extraction of the hidden state at the final time step ensures that the model captures the full temporal context of the sequence, which, to some extent, improves classification accuracy, as shown in the Figure 4.8. Subsequently, the fully connected layer maps this global representation to the target class space via linear or non-linear transformations, ultimately producing the classification result. This approach thus effectively exploits the rich contextual information embedded in the final hidden state, improving classification performance while maintaining computational efficiency. As demonstrated in Figure 4.10, the GRU model exhibits a reduced parameter count relative to the FCNN model and reaches a comparable level of accuracy.

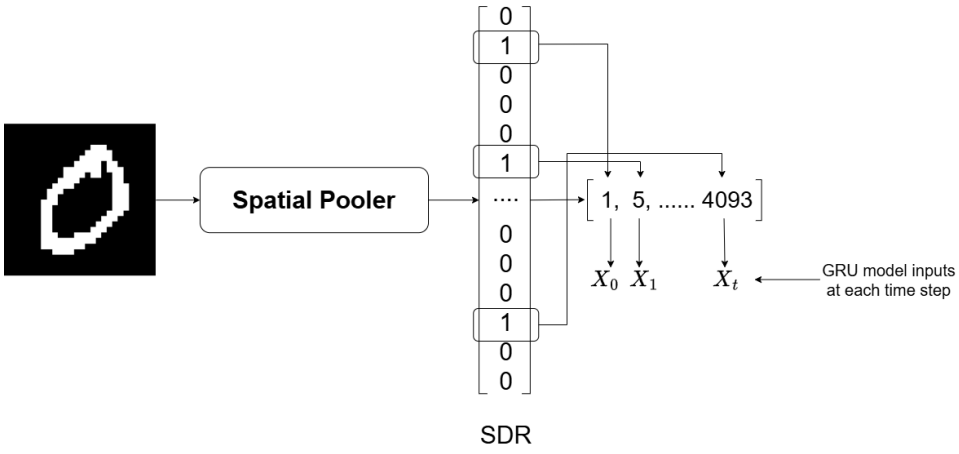


Figure 4.7: Illustration of how the output of the SP is converted into the input for the GRU. First, the binarized MNIST image is fed into the SP, which produces the corresponding SDR. Then, the indices of the active bits (1 bits) in the SDR are extracted and fed into the GRU model, with each index serving as the input for a single time step.

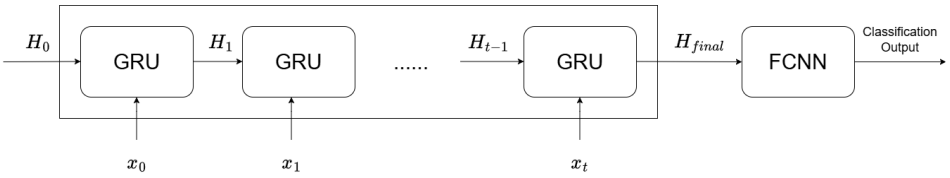


Figure 4.8: The diagram illustrates the process of generating classification results using GRU. Initially, the inputs are arranged in time steps, with the GRU module, in each time step, accepting the hidden state of the inputs from the previous time step, in addition to the inputs from the current time step. Ultimately, the hidden state of the final moment is output to the FCNN to generate the final classification results.

4.4. Implementation of minGRU

The GRU model reduces the number of parameters by employing a gating mechanism. Such a reduction in parameters is particularly advantageous for hardware implementations, as it substantially alleviates resource constraints in hardware-limited scenarios. However, as mentioned in Section 2.2.2, it can also be observed that the computation process within the standard GRU architecture is not fully parallelized. Specifically, as shown in Equation 2.15 and Equation 2.16, the computations of the update gate (z_t) and reset gate (r_t) must be completed before calculating the candidate hidden state ($\tilde{h}_t$), which in turn must be finalized prior to generating the current hidden state output (h_t).

In order to address the aforementioned limitation, the minGRU model, as introduced in Section 2.2.3, was implemented. This novel model represents an optimized variant of the conventional GRU architecture, in which the reset gate (r_t) is completely removed and the associated computational complexity is reduced. The elimination of the reset gate not only simplifies the internal structure of the GRU model but also facilitates parallel computations within the gating mechanism. Consequently, minGRU achieves a substantial reduction in model parameters, while exhibiting only a marginal decrease in classification accuracy.

Apart from these modifications, the minGRU implementation follows the same computational procedure as the standard GRU. Specifically, at each time step, the model receives an input along with the hidden state generated from the previous time step. The hidden state produced at the final time step is then fed into an FCNN to obtain the final classification output. The effectiveness of minGRU is evaluated through experimental results summarised in Table 4.1. Notably, for a model configuration with an SDR size of 4096, the minGRU model demonstrates an 18.5% reduction in the number of parameters compared to the original GRU, with only a 1.5% reduction in accuracy. Furthermore, the inference runtime measured in Python indicates that minGRU outperforms GRU by approximately 50%, highlighting its significant advantage in computational efficiency and suitability for hardware acceleration.

Table 4.1: Comparison of GRU and minGRU Models on MNIST Dataset

Model	Parameters	Accuracy	Training Time per Epoch
GRU	34,138	94.5%	1:35:09
minGRU	27,994	92.9%	0:47:32

Note: Training time refers to the duration of a single training epoch, each epoch containing 6000 images from the MNIST dataset.

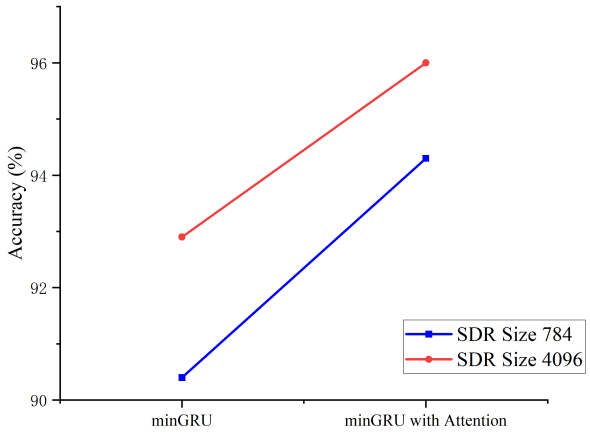


Figure 4.9: This figure demonstrates the change in the accuracy of the minGRU model before and after using the attention mechanism. The SDR size refers to the size of the SDR data that the minGRU model receives as output from the SP. It is evident that the accuracy of the minGRU model with SDR sizes of 784 and 4096 is enhanced through the integration of the attention mechanism, thereby substantiating the efficacy of the attention mechanism in conjunction with the minGRU model.

4.5. Applying the Attention Mechanism

The minGRU model achieves parallelization by simplifying the complexity inherent in the standard GRU architecture. This modification effectively addresses the limitations of the GRU in terms of computational parallelism while reducing the overall computational complexity of the model. However, as shown in Table 4.1, this simplification results in a noticeable drop in classification accuracy, from 94.5% for the original GRU to approximately 92.9% for the minGRU model. Given that the FCNN initially achieved an accuracy of around 95%, there is considerable motivation to improve the minGRU architecture to match the performance of the FCNN, while retaining the benefits of parallel computing.

Inspired by previous work presented in [86], we propose to integrate the general form of attention mechanism introduced in Equation 2.22 in Section 2.2.4 into the minGRU model. The attention mechanism has been widely demonstrated in previous research to improve the classification capabilities of RNN-based architectures [87]. Since minGRU essentially belongs to the family of RNN architectures, combining it with the attention mechanism is expected to improve the overall classification accuracy.

Figure 4.9 shows a performance comparison between the minGRU models with and without the attention mechanism. It can be seen that the accuracy of the minGRU model improves with the inclusion of the attention mechanism, regardless of whether the SDR size is set to 4096 or 784. This improvement is mainly because the attention mechanism effectively allows the network to selectively focus on the most informative parts of the input sequence. By adaptively weighting different time-step representations according to their relevance to the classification task, the attention layer helps to extract and emphasize critical features that would otherwise be missed or underrepresented. As a result, this targeted feature enhancement directly leads to an increase in discriminative power and, ultimately, a significant improvement in the classification accuracy of the minGRU model.

The integration of these modifications resulted in the successful development of a minGRU model combined with the attention mechanism, thereby yielding a neural architecture that exhibited both

high computational parallelism and enhanced classification accuracy. However, it should be noted that the incorporation of the attention mechanism introduces additional computational complexity into the model, particularly due to the inclusion of operations such as the softmax function and layer normalization.

These two functions present challenges in terms of efficient hardware implementation. For instance, the softmax operation involves exponentiation and division, which are computationally intensive and typically require high-precision arithmetic to maintain numerical stability [88], thus complicating hardware design and resource allocation. A similar issue arises with layer normalization, which involves the dynamic computation of mean and variance. This additional data dependency hinders the potential for parallelization in hardware [89]. The impact of these operations on pipeline latency and resource consumption is a key area of concern, as they can lead to significant increases in power consumption and hardware resource demands.

Therefore, in order to identify the most appropriate model for hardware deployment, a comprehensive comparative analysis of all previously discussed architectures will be conducted in the following section. The objective of this analysis is to evaluate the trade-offs between computational complexity, accuracy, parallelization capability, and hardware implementation feasibility. The objective of this evaluation is to guide the selection of the optimal hybrid model combining the HTM model with deep learning algorithms that are best suited for efficient and practical hardware realization.

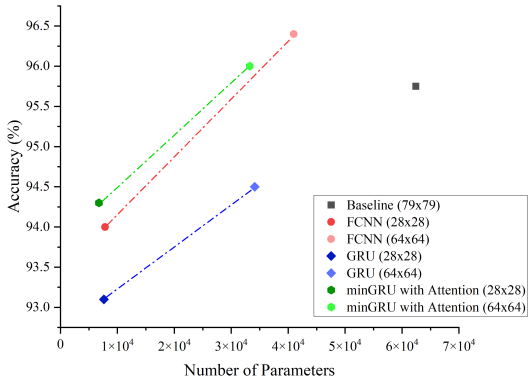
4.6. Comparison for Hardware Implementation

In the previous sections, we introduced how the SP part of the HTM algorithm can be combined with the FCNN, GRU, and minGRU models. This section provides a comprehensive analysis of the advantages and disadvantages of each model, and then identifies the most suitable candidate for subsequent hardware implementation, taking into account hardware implementation considerations.

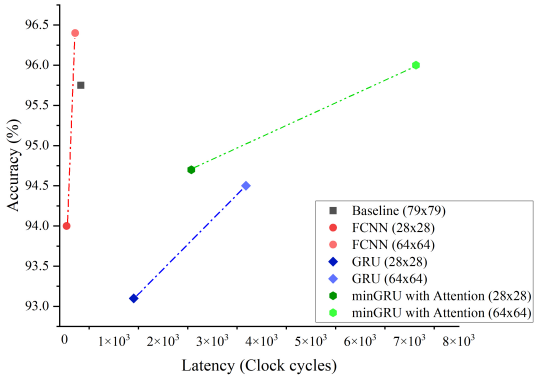
In the context of hardware implementation, the primary considerations revolve around PPA (performance, power, and area). Figure 4.10a illustrates the relationship between the total number of parameters of a model and its classification accuracy. The number of parameters not only reflects the complexity of the model itself but also the hardware resources, like memory and chip area, required during implementation. As shown in the figure, for the same model architecture, increasing the input dimensionality directly leads to a larger number of parameters, which generally correlates with higher accuracy. The baseline model shown here is adapted from the original work in [63]. It integrates the SP component of the HTM model with an MLP, resulting in an FCNN input dimension of 79×79. Although this baseline achieves an accuracy of around 95%, it has the largest number of parameters of all the models examined and does not offer the highest overall accuracy. In contrast, the FCNN (64 × 64) model reduces the number of parameters compared to the baseline, while achieving the highest accuracy of all the models tested. Meanwhile, the minGRU (64 × 64) model achieves slightly lower accuracy (by about 1%) than the FCNN (64 × 64) model, but further reduces the total number of parameters. Based on the results of the above analyses, the minGRU (64 × 64) model appears to be the best choice at this time, as the number of parameters in the model decreases by 0.5×10^4 with only a 1% decrease in model accuracy. The reduction in the number of parameters in the model has been achieved with a small change in the accuracy of the model.

Figure 4.10b illustrates the trade-off between inference latency and model accuracy. Here, inference latency is defined as the time elapsed from the moment an input is fed into the model to the moment the final output is produced. To estimate the inference latency of each model, we assume that all models are running on a hardware platform that can support 1024 multiply-accumulate (MAC) operations and has a memory read speed of 1024 bytes per cycle. Figure 4.10b shows that all FCNN models have relatively short inference latencies. This result can be attributed to the fact that these models consist of a single fully connected layer, which allows for highly parallel computation. Consequently, the dominant source of latency in FCNN models is typically data transfer (read/write) rather than computation. The Baseline model, which is essentially an FCNN with an identical structure to the FCNN, has a comparable latency profile for the same reason. In contrast, the GRU and minGRU models have longer inference latencies because they process inputs sequentially. Although these recurrent models can exploit the sparsity of the SP output by reading only about 10% to 20% of the activated columns, they still need to process a large SP output dimension. Specifically, for an SP output size of 28 × 28, GRU and minGRU need to perform 156 sequential input steps; for 64 × 64, they need to process 409 steps. Among these

sequential architectures, minGRU benefits from better parallelization than GRU, resulting in slightly lower latency. In addition, the use of the attention mechanism in the minGRU model, which includes the softmax and layernorm functions, requires multiple traversals of the input sequences, resulting in a longer inference time for minGRU than for GRU. Overall, the FC models have a significant advantage over both GRU and minGRU in terms of inference latency. Combined with the relatively high accuracy of FC-based architectures, these observations suggest that FC models may be the most favorable choice when balancing accuracy and latency requirements.



(a) Comparison of model accuracy (%) vs the number of parameters.



(b) Estimated latency (in clock cycles) vs accuracy (%).

Figure 4.10: Figure (a) and Figure (b) show the accuracy (%) versus model parameter sizes, and the accuracy versus inference latency for different models (Baseline, FCNN, GRU, and minGRU), respectively. For the calculation of inference latency, the implementation hardware is assumed to be capable of performing 1024 MAC parallel computations per cycle with a memory bandwidth of 1024 bytes/cycle. Values in parentheses (e.g., 28×28) indicate the size of the input dimension used by the model.

In addition to the relationships between accuracy, model size, and inference latency shown in Figure 4.10, Table 4.2 summarizes the hardware implementation trade-offs of each model. FCNNs have a simple structure with fewer layers, facilitating easy software and hardware implementation, and leveraging parallel MAC operations for high throughput. However, larger input dimensions cause parameter explosion, increasing memory and computational overhead.

GRU and minGRU simplify high-dimensional input processing by sequentially handling a single integer per time step, capturing temporal dependencies but limiting parallel computation and increasing latency, especially with large input dimensions. Compared to GRU, minGRU further reduces computational complexity and MAC operations through simplified gating mechanisms and enhanced parallelism. However, minGRU introduces an attention mechanism, including softmax and layer normalization operations, adding computational overhead and complexity. Additionally, as the minGRU model is a recent proposal, limited Python library support makes quantization procedures challenging. Despite inherent sequential constraints, minGRUs improved parallelization partially mitigates latency concerns.

Overall, the analyses in this section show that from a hardware implementation perspective, the FCNN model is better suited to be combined with the SP model to achieve the classification task for MNIST datasets due to its streamlined computational process and highly parallelized features. In contrast, GRU-based models effectively handle high-dimensional inputs by processing sequential data step by step, thereby capturing temporal dependencies. However, their reliance on time-step iterations results in higher inference latency, especially for large SP output dimensions. The minGRU architecture mitigates some of these limitations by reducing the computational load through simplified gating and partial parallelization. However, the incorporation of attention mechanisms and the lack of readily available quantization libraries introduce additional design complexity. Ultimately, the best choice for hardware implementation depends on the specific application constraints and performance requirements, underscoring the importance of balancing parameters, accuracy, latency, and ease of implementation.

Table 4.2: Comparison of Different Models

Models	Strengths	Limitations
Fully Connected	Easy to implement High throughput Easy to quantify	Input size is excessive
GRU	Only one integer per input	Limited parallelization Long time steps
minGRU with Attention	Only one integer per input More parallel than GRU Highest accuracy Fewest MAC operations	Complex to implement Hard to quantify Long time steps

5

Hardware Implementation

This chapter provides a detailed overview of the three custom hardware accelerator architectures proposed in this thesis and the system framework implementation process. During the initial design phase, a function-level cycle analysis was conducted on the open-source Ibex RISC-V processor to identify the core modules within the overall algorithm that were the most computationally intensive. The most computationally intensive modules were found to be the overlap calculation of SP in HTM (52.9%), the multiply-accumulate operations in fully connected neural networks (25.5%), and the k-WTA sorting logic used in sparse activation (20.2%). Consequently, this chapter focuses on the dedicated hardware design of these three computationally intensive components, as well as their integration and verification on the CROC SoC platform.

First, in Section 5.2.1, we introduce the SP accelerator design based on binary vector dot products. This design adopts a 25-way parallel processing unit (PE) structure, combining input/weight blocking, pipeline scheduling, and addition tree aggregation to achieve high-speed inference with a fixed latency of 25 clock cycles per column.

Secondly, Section 5.2.2 details the hardware implementation of the min-heap sorting accelerator. This involves constructing a hierarchical heap structure, using dual-port SRAM for node storage and combining sorting nodes for parent-child data transfer and heap operations. This approach can efficiently complete the k-WTA selection task, reducing the comparison and jump overhead required for sorting logic.

Next, Section 5.2.3 provides a brief description of the design principles of the MAC-NN accelerator. This accelerator supports 16 $\times$ 16 multiply-accumulate operations in FCNN, thereby reducing the latency of neural network inference.

The latter half of this chapter covers system integration and verification in Section 5.3 and hardware synthesis and resource utilisation analysis in Section 5.4. It explains how to connect the accelerator to the system on a chip (SoC) using standard memory-mapped I/O (MMIO) methods, and how to verify its functional correctness through simulation waveforms. Additionally, the entire system is synthesised using a 40 nm process, and the area, power consumption and timing performance are analysed. The accelerator is then compared with the software platform for acceleration to verify that it achieves a performance improvement of approximately 26.4$\times$ that of the Ibex CPU when performing inference process for a single MNIST image.

5.1. System overview

Before designing the accelerator, we need to identify the parts of the hardware system that require the most acceleration. As shown in Figure 5.1, cycle-level profiling of the pure C reference implementation was performed on the Ibex CPU. The built-in performance counter was used to count the total cycles for a single MNIST image input, which were then categorised by function. The results show that SP-overlapping multiply-accumulate operations consume the most cycles (52.9%), followed by weight-multiply-accumulate operations in FCNN inference (25.5%), k-Winner-Take-All sorting (20.2%), and the Boost coefficient multiplied by the overlap value (1.4%). Clearly, over 98% of the runtime is concentrated in three computationally intensive cores: SP multiply-accumulate, FCNN multiply-accumulate,

and sorting competition. Based on this analysis of the profile, the design isolates these three logic components from the processor and incorporates dedicated hardware accelerators to achieve computational acceleration. The remaining less than 2% of control and boost multiplication operations are handled by the Ibex CPU. This division enables targeted hardware acceleration of the most time-consuming processes while ensuring compliance with area and power consumption budgets.

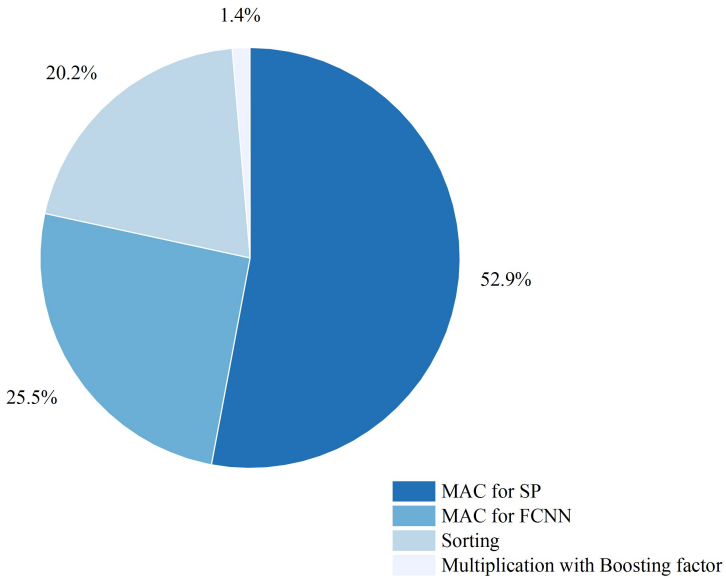


Figure 5.1: Execution Cycles of SP-FCNN Algorithm on Ibex Core. It can be observed that the MAC operations in the SP component consume the largest number of cycles. This arises because the SP portion involves 4096 × 784 MAC operations, which cannot be executed in parallel on the Ibex core and therefore must be processed sequentially within a for loop, leading to significant overhead. Similarly, the MAC operations in the FCNN component account for the second-largest number of cycles. Consequently, our subsequent optimization efforts focus primarily on parallelizing these MAC operations.

Figure 5.2 shows the overall framework of the hardware architecture proposed in this thesis. This research project plans to use the Ibex Core, which is based on the RISC-V instruction set, as the central control processor for the CROC open-source system-on-chip (SoC) [90]. The core interconnection structure will be a standard open bus interface (OBI) crossbar switch [91]. This connects the processor, storage units and multiple dedicated hardware accelerator modules, achieving efficient transmission of data and instructions.

Specifically, the Ibex processor is encapsulated in the Ibex wrapper and connected to the OBI crossbar via independent instruction and data interfaces. The system includes separate storage for instructions (Instruction SRAM with a capacity of 512 KB) and data (Data SRAM with a capacity of 1 MB), which provides ample memory for executing programs and accessing data. The processor's debugging functionality is provided via a standard JTAG interface and a debug module, supporting real-time debugging and program loading. In terms of SoC peripherals, the OBI Demux module connects the SoC registers, universal asynchronous receiver-transmitter (UART), general-purpose input/output (GPIO) and timer modules. This provides the necessary external communication and clock management functions.In terms of accelerators, this study involved the design of dedicated hardware units, including multiply-accumulate (MAC) arrays for SP and NN computations, as well as a sorting accelerator that is also used for the SP portion. Two MAC arrays perform vector computations in the SP and NN sections, respectively, and are equipped with preloaded dedicated SRAM for storing weight data. This reduces data loading latency and power consumption. The sorting algorithm accelerator performs efficient column competition and k-winner-take-all operations. The aforementioned accelerator modules are interconnected with the main crossbar bus via separate OBI Mux and OBI Demux interfaces. This allows the main processor to access the control registers and data storage areas of each accelerator directly via specific address mapping. The components of the hardware system are shown in the Table 5.1.

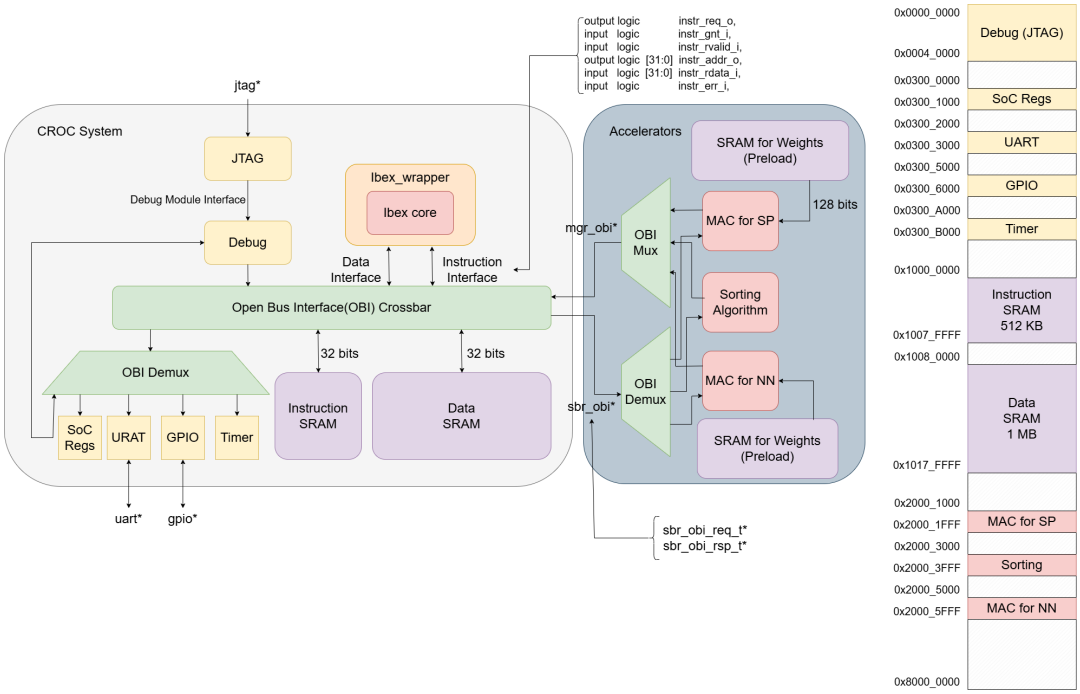


Figure 5.2: This diagram illustrates the hardware system architecture and memory address mapping designed in this study. Based on the open-source RISC-V Ibex Core, the system uses an OBI crossbar to efficiently interconnect the instruction memory (512 KB), data memory (1 MB) and peripheral modules (UART, GPIO, timers, etc.). The blue area on the right represents the dedicated hardware accelerator, comprising MAC units and sorting modules for the SP and NN, as well as dedicated weight SRAM. Memory address space allocation for each module is shown on the far right, enabling the central processor to schedule and access them uniformly.

5.2. Detailed Design of Accelerator Blocks

5.2.1. Architecture of the SP Accelerator

To speed up the inference process of SPs in the HTM network, we proposed using dedicated hardware for the 'input-weight product' stage. While the synaptic weights of SPs are updated in floating-point form during training, during inference it is only necessary to determine whether the weights exceed the threshold. Values above the threshold are recorded as 1 and values below as 0, meaning that weights can be treated as binary vectors, similarly to inputs. For binarized and flattened MNIST images, each input vector has a length of 784 bits. SP column operations are essentially reduced to the dot product of two 784-bit binary vectors. Taking into account constraints relating to chip area and power consumption, the project initially focuses on accelerators for single-column computations, implementing an efficient parallel computation core for binary vector dot products and reducing the computational burden on the CPU.

After introducing the requirements for binary SP column operations, we need to consider how to convert the specific algorithm into a circuit. First, we will formalise the 784-bit binary vector dot product as a synthesizable logic expression. Then, we will formulate a data chunking scheme based on on-chip bandwidth, storage width and clock constraints. Finally, we will derive parallel and pipeline designs. Next, we will introduce the mathematical model of SP operations.

During inference, every input pixel $\hat{x}_k$ and the corresponding synaptic weight $\hat{w}_k$ are binarized to the set $\{0, 1\}$. We first define their bitwise conjunction

$$y_k = \hat{x}_k \wedge \hat{w}_k, \qquad k \in \{0, \ldots, 783\}. \tag{5.1}$$

Collecting all bits yields two column vectors

$$\mathbf{x} = \left(\hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{783}\right)^\top, \qquad \mathbf{w} = \left(\hat{w}_0, \hat{w}_1, \ldots, \hat{w}_{783}\right)^\top,$$

each of length 784 bit.

Table 5.1: Key modules of the hardware system and their specifications

Module	Key parameters	Function / role
Instruction SRAM	512 kB, single-port, 1-cycle	Stores Ibex firmware and bootloader code
Data SRAM	1 MB, single-port, 2-cycle	Caches input images, SDRs, intermediate results
SoC Peripherals	UART, GPIO, Timer, SoC Regs	External communication and system control
MAC-SP Array	784 × 784 bits AND Operation, 64 kB weight SRAM	Handles SP overlap computation
Sorting Accelerator	Heap sort for 410 numbers	Executes k-winner-take-all column selection
MAC-NN Array	128 × 128 INT8 MAC, 64 kB weight SRAM	Accelerates FCNN / GRU inference
OBI Crossbar	32-bit data path, single-cycle	Interconnect for CPU, memories and accelerators
OBI Mux / Demux	Address decoding, handshake logic	Bridges CPU and accelerators via memory-mapped registers

The activation score of one SP column is the Hamming overlap, that is, the population count of the bitwise AND of the two vectors:

$$s = \sum_{k=0}^{783} y_k = \text{popcount}(y) = \text{popcount}(\mathbf{x} \wedge \mathbf{w}). \tag{5.2}$$

Because $s \in [0, 784]$, an unsigned 10-bit word is sufficient, although our hardware keeps a margin and outputs 16 bits.

Next, we need to consider how the accelerator obtains the required computational data. The specific data chunking details are shown in Table 5.2. Since input data can only be transferred to the SP accelerator via the CPU and due to OBI limitations within the SoC, it can only be read in 32-bit chunks. Therefore, a total of 25 reads are required to read 784 bits of data completely. Regarding weight information, as the entire algorithm has already been pre-trained in the software portion, the trained weight results can be stored in SRAM in advance. The SRAM module can be connected directly to the SP accelerator, providing a 128-bit port width. Only 7 reads are required to retrieve the weight portion.

Table 5.2: Chunking parameters for the SP-column accelerator

Data stream	Port width	Chunks / vector	Reason
Input vector **x**	32 bit	25	Fixed by the 32-bit OBI bus width
Weight vector **w**	128 bit	7	The wider 128-bit on-chip SRAM port reduces the number of read cycles

Figure 5.3 shows the specific details of the SP accelerator. A fully parallel data pipeline runs from the input to the output: the input on the left is 784-bit flattened binary MNIST image data, read in 32-bit chunks each time. This data is stored in the FIFO and, whenever the core is ready, a 32-bit block is sent via the ready/valid handshake. The on-chip SRAM in the top right corner stores the weights, which the control logic reads out in a single 128-bit operation using column indexing. The weight data first enters the weight FIFO on the right and is split into four 32-bit sub-blocks over the next three clock cycles, aligning with the A-chunk input in the core. The core accelerator region contains 25 parallel processing units (PEs), each performing a 32-bit bit-wise AND and pop-count operation and outputting

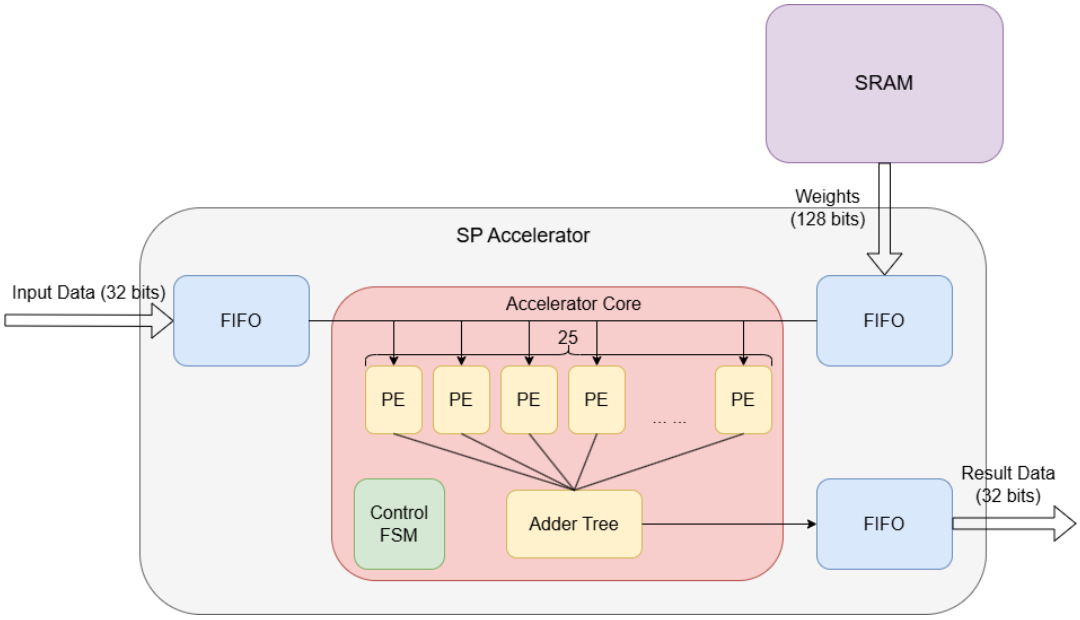


Figure 5.3: This is a block diagram of the SP column accelerator. Data is supplied to 25 parallel PEs from a 32-bit input FIFO and a 128-bit weight FIFO. Each PE performs bitwise AND operations and pop count, and its 6-bit output is reduced to a 16-bit dot product through an adder tree and cached in the output FIFO. A control state machine coordinates four pipeline stages: the simultaneous fetching of instructions A and B; the fetching of instruction B; decomposition; and the calculation of the output.

a 6-bit count. The results from all the PEs are fed into the bottom addition tree synchronously, where a five-stage prefix addition operation aggregates the 25×6 bits into a 16-bit dot product value within one clock cycle. This result is written into the output FIFO at the bottom right, awaiting readout by the SoC bus.

As shown in Figure 5.4, the SP accelerator's finite state machine (FSM) consists of seven explicit states and implements a three-cycle data fetch pipeline. Upon startup, the controller transitions from IDLE to Fetch Inputs and Weights. In this state, it fetches one 32-bit input block per cycle and issues one 128-bit weight read request to the local SRAM. Once the read request has been issued, the process immediately jumps to 'Wait Weights from SRAM', where it waits for one cycle to receive the returned 128-bit data. It then enters 'Decompose Weights Vector', where it splits the 128-bit weights into four 32-bit sub-blocks and writes them into the B-chunk buffer. Once decomposition is complete, the process returns to 'Fetch Inputs and Weights' to begin the next round of data retrieval. This three-step loop ('Fetch → Wait → Decompose') is executed seven times to read all seven weight blocks, and during this time, the input stream is also written synchronously into the first seven A-chunks. Even when `weights_done` is set, the FSM may remain in the 'Fetch Inputs and Weights' state. However, since no further reads are sent to SRAM at this point, it effectively only executes the input fetch loop. Once `inputs_done` is also set, indicating that all 25 input blocks are ready, the state machine jumps directly to Process and Output via the thick arrow in the diagram. In this state, the 25 PE-aligned inputs and weight blocks perform bit-wise AND and popcount operations in a single clock cycle. The addition tree then aggregates these into a 16-bit dot product and writes it to the output FIFO. After successful handshaking, if another column is to be processed, the state machine returns to 'fetch inputs and weights' to start a new round. Otherwise, it enters 'Finish', generates a `done_tick` pulse, and ultimately returns to `IDLE`. Thanks to the three-cycle loop that hides the latency of the SRAM, combined with the subsequent 18 cycles of pure input fetching and four cycles of output computation, the latency remains fixed at 25 clock cycles per column. This enables one column of results to be output per cycle once the pipeline is filled.

Figure 5.5 shows the waveform diagram of the SP accelerator simulation results. The data fetch pipeline rhythm of the first 12 clock cycles after the accelerator starts up can clearly be seen from this waveform. The start signal is pulled high for one cycle of around 30 ns, causing the FSM to transition immediately from `IDLE` to `FETCH_A_B`. At the same time, the busy signal is set to 1, indicating that the

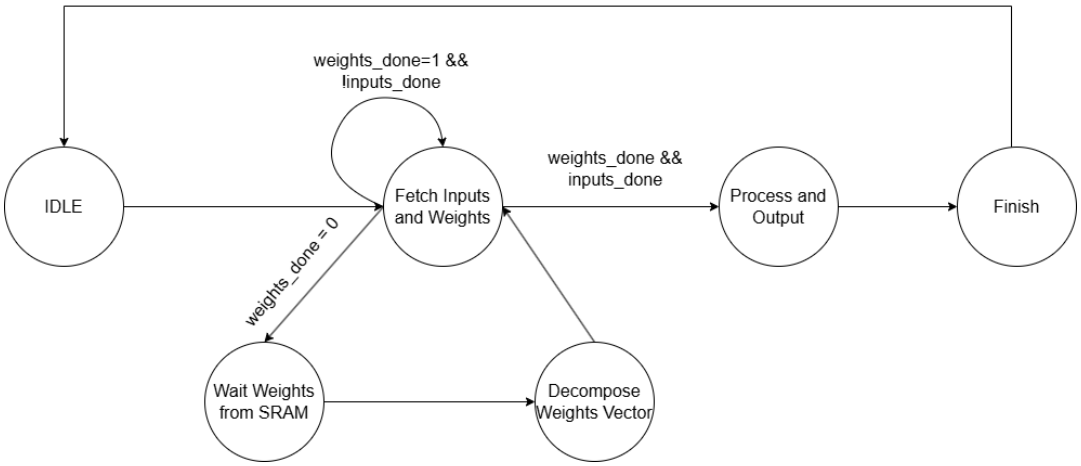


Figure 5.4: The six-state FSM of the SP accelerator. After the IDLE state, the controller enters the 'Fetch Inputs and Weights' state, initiating a three-cycle micro-loop. First, it issues a 128-bit SRAM read. Then, it moves to the 'Wait Weights from SRAM' state for the one-cycle latency. Finally, it reaches the 'Decompose Weights Vector' state, where it splits the returned 128-bit word into four 32-bit chunks. It then returns to the 'Fetch' state. This loop repeats until `weights_done` equals 1. The FSM then remains in the 'Fetch Inputs and Weights' state until `inputs_done` is also asserted, acquiring inputs only. Once both vectors are ready, the machine jumps to 'Process and Output', where 25 parallel PEs and an adder tree produce the dot product. Finally, it transitions through 'Finish' back to 'IDLE' to start the next column.

core has begun to operate. The subsequent three clock cycles form a fixed microcycle:

1. **`FETCH_A_B` k^{th} cycle:** This stage means fetch inputs and weights. The valid signal of input vector is pulled high and sends the kth 32-bit input block; `current_state` displays `FETCH_A_B`, and the `input_vector_idx` counter below increments from 00; Within the same clock cycle, the controller issues a 128-bit weight read request to the SRAM, while `weights_vector_idx` retains the previous value.

2. **`FETCH_B_WAIT` $(k+1)^{th}$ cycle** The FSM switches to "FETCH_B_WAIT", and the core waits for one clock cycle for the SRAM to return; when the data returns, `weights_vector_idx` is latched to 0, 1, 2..., corresponding to the transition indicated by the arrow.

3. **`DECOMPOSE_B` $(k+2)^{th}$ cycle**: This stage means decomposing the weights vector, where the newly received 128-bit weights are split into four 32-bit segments and written to the weight buffer. Once the splitting process is complete, immediately return to `FETCH_A_B` to begin fetching the next input segment and trigger the next weight reading operation.

As shown in the figure, the input vector index increments by one with each `FETCH_A_B`, while the weight vector index only changes during the `FETCH_B_WAIT` phase. This indicates that the SRAM read latency is fully hidden within the three-phase ring pipeline. After seven cycles, with the weight count reaching six, all seven weight blocks will have been loaded into the FIFO, and the FSM will no longer jump to `FETCH_B_WAIT/DECOMPOSE_B`, remaining at `FETCH_A_B` instead to fetch the remaining 18 input blocks individually. This process will be demonstrated in the second waveform.

Figure 5.6 shows the simulation waveform of the second half of the SP accelerator. Once the weight vector index reaches 6, the weight-side data is fully buffered, and the index is frozen at 7. Meanwhile, `FETCH_A_B` remains the current state, during which only input fetching occurs and no further read requests are sent to SRAM. The index of the input vector increases monotonically from 0x08 to 0x18, writing the remaining 18 32-bit input blocks into the A-chunk buffer exactly 18 times. Throughout this process, it can be seen that the input vector's valid signal remains active, and the ready signal never deactivates. This indicates that the wait introduced by weight retrieval has been eliminated, allowing the input stream to flow continuously without interruption. When the input vector index jumps to 0x18 for the final time (i.e., when the 25th input block is filled), the control logic immediately recognizes that the inputs and weights are complete, and the current state enters the processing and output stages. The next clock cycle transitions to 'Output', followed by the valid output signal being pulled high for

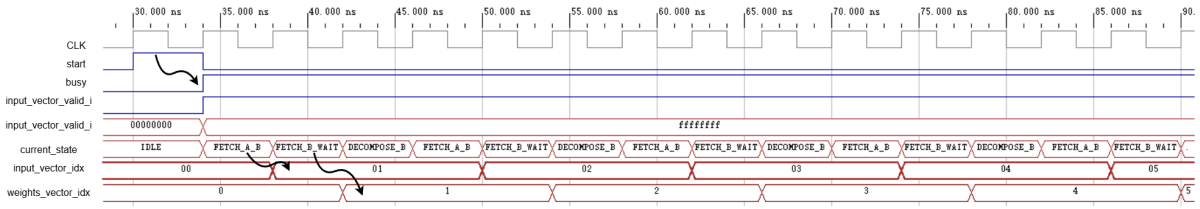


Figure 5.5: This is the waveform during the first half of the SP accelerator's operation after startup. After the start trigger, the state machine transitions from IDLE to a three-cycle pipelined flow comprising the 'fetch inputs and weights', 'fetch weights wait', and 'Decompose weights' stages. The index of the input vector increments in each 'Fetch inputs and weights' section, while the index of the weight vector is only updated in the 'fetch weights wait' section. This indicates that the 1-cycle delay of the 128-bit weight read is fully hidden within the fetch pipeline.

one cycle this is the first 784-bit dot product result. The FSM then enters the 'Finish' state to generate a completion flag, and 'busy' is pulled low on the next clock edge, returning to 'IDLE' and indicating the end of the current column operation. If there are subsequent columns, 'start' is pulled high again to re-enter the three-cycle ring pipeline at the same rhythm. This demonstrates that the weight reading delay is hidden by the first three clock cycles and that the remaining input blocks are filled within 18 clock cycles. The computation and output then take an additional four clock cycles, resulting in a fixed delay of exactly 25 cycles per column.

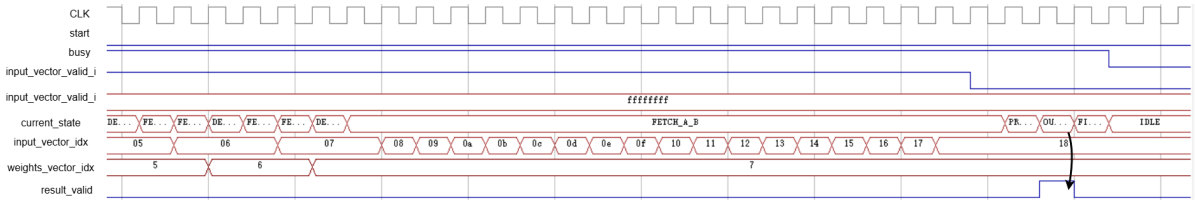


Figure 5.6: The waveform is produced after the SP accelerator has completed the data retrieval process. Once the weight index count is fixed at 6 (i.e., once 7 blocks have been retrieved), the state machine remains in the 'fetch input and weights' state and continues to retrieve the remaining 18 input blocks. When the index of the input vector reaches 18 and the `inputs_done` and `weights_done` conditions are satisfied, the FSM sequentially transitions to Process, then Output, and finally Finish. The result signal is pulled high at the first output point, followed by the busy signal being set to 0 and the state machine returning to IDLE. This verifies that the fixed delay for a single column is 25 clock cycles.

5.2.2. Architecture of the Heap Sort Accelerator

In the SP stage, we must complete the dot product between the input vector and the column weights and select a small number of the columns with the highest scores as 'activated columns', based on the sparsity ρ (typically 2% - 10%) of the obtained SDR. All other columns must be suppressed. This k-winner-take-all process is achieved using a min-heap selection algorithm rather than sorting all columns completely [92]. The specific approach is as follows:

- The accelerator maintains a min-heap with a fixed capacity of $k = \rho N$ (where N is the total number of columns) while calculating the overlap score.
- The top of the heap always stores the column with the smallest overlap in the current active set.
- When a new overlap score arrives, it is compared with the top of the heap: if the score is higher, the top is replaced and a sink operation is performed; otherwise, it is discarded directly.

This minimum heap strategy only requires the input data to be compared with the top of the heap. This makes it highly compatible with hardware implementation: a sorting accelerator only needs to configure registers or SRAM with enough capacity to hold 2 - 10% of the number of columns to complete k-winner-take-all, without having to store the entire SDR vector [93].

As shown in Figure 5.7, this illustrates the detailed execution process of the heap sort algorithm [92]. First, the unordered sequence is converted into a binary tree that satisfies the 'maximum heap' property, meaning that the value of each parent node is greater than or equal to that of its child nodes (as shown in (a) of the figure, where the root node contains the maximum element, 16). Next, the root

node (the largest element at the top of the heap) is swapped with the last element of the heap (as shown in (b)), and this process is repeated until only one element remains. The largest element is now fixed at the end of the sequence and no longer participates in subsequent heap adjustments. Next, a 'sift-down' operation is performed on the new root node. This involves comparing the new root node with its child nodes one by one, swapping positions as necessary to ensure that the heap once again satisfies the 'maximum heap' property. For example, in Figures (b) and (c), element 1 is first swapped with 14, then with 10, sinking to the appropriate position each time. This process is repeated continuously during the sorting process. Each round fixes the largest element at the far right of the sorted region, gradually narrowing the heap's scope until all elements are sorted, as shown in Figure (j). This ultimately yields a fully sorted sequence in ascending order, as shown in Figure (k) [94].

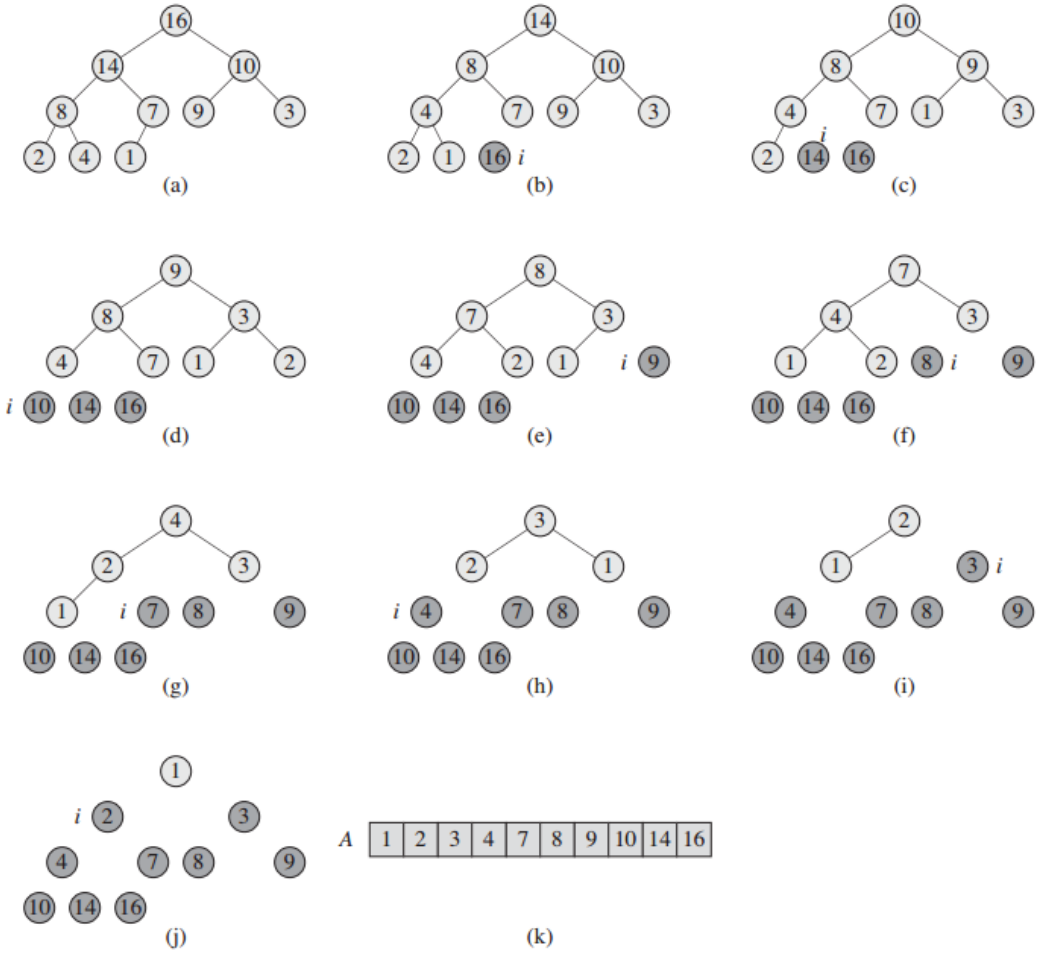


Figure 5.7: This is a schematic diagram showing the execution process of the heap sort algorithm [92]. The figure shows the various stages of the maximum heap sort process. Starting with the initial construction of the maximum heap (a), the top element is exchanged with the last element and sunk to maintain the maximum heap structure (bj). Each step moves the current maximum value to the end of the final sorted array. Figure (k) shows the final sorted result. Grey nodes represent elements that have already been sorted, while italicised nodes indicate elements currently involved in swapping or sinking operations.

Next, we will analyse the time complexity of minimum heap sorting. Based on the structure of minimum heap sorting described above, we can see that the height of the minimum heap is:

$$h = \lfloor \log_2 n \rfloor, \tag{5.3}$$

where n is the number of elements in the heap. Using the bottom-up method to construct a min-heap requires at most O(n) time:

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^{h+1}} \cdot h = O(n), \tag{5.4}$$

where $\frac{n}{2^{h+1}}$ is the number of nodes in each layer, and each node can sink a maximum of h steps. When a new element is added to the bottom of the heap, it moves upwards until it satisfies the minimum heap property, up to the maximum height of the root node:

$$T_{\text{insert}} = O(\log n) \tag{5.5}$$

Move the tail element to the top of the heap, then perform a sink operation, sinking to the leaf layer at most:

$$T_{\text{delete min}} = O(\log n) \tag{5.6}$$

For a min-heap of size k, traversing n elements takes:

- Insert the first k elements into the heap: $k \cdot O(logk)$.
- For the remaining n − k elements, compare each one with the top of the heap. If it is greater, replace it and perform a heapify, which takes at most $(n-k) \times O(logk)$.

Total time complexity:

$$T(n,k) = O(k \log k) + O((n-k) \log k) = O(n \log k) \tag{5.7}$$

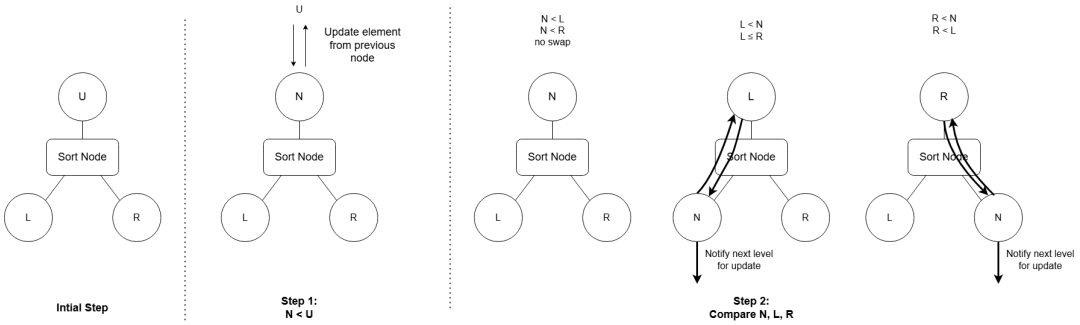


Figure 5.8: This diagram illustrates the process of data comparison and exchange within the Sort Node. It shows the specific steps involved in data exchange by the Sort Node during heap sorting in detail. In the initial stage, the current node (N) receives updated data from the upper node (U). The comparison stage then begins, determining the size relationship between the current node (N) and its left and right child nodes (L and R) to establish whether an exchange is necessary. Based on the results of the exchange, the next layer of nodes is notified to perform further data updates.

The following section provides a detailed introduction to the process of implementing a hardware accelerator for the min-heap algorithm. Figure 5.8 first shows the working process of the sort node of the min-heap algorithm. The processing flow of the sort node is divided into three main steps:

- **Initial step** The sort node contains data from three other nodes. The parent node (U) includes data from the upper-level node, the left child node (L), and the right child node (R) contain data from nodes stored in the SRAM of this layer.
- **Update the parent node data (Step 1) :** The sort node receives updated data from the upper-level node (as shown in the figure, from node U to node N). The sort node then stores this new data in its internal register (N), which indicates that the current value of the node has been updated. At this point, the internal state of the sort node changes to the 'pending comparison' state in preparation for the next comparison and exchange step.
- **Compare and swap step (Step 2):** The node compares the data sizes of the following three items: the current node data (N); the left child node data (L); and the right child node data (R). Based on the min-heap rule, the sort node selects the smallest data among the three and promotes it to the new value of the current node (node N), while the replaced data is promoted to the child nodes. Simultaneously, it notifies the right node at the next level to update, as indicated by the arrows in the figure.

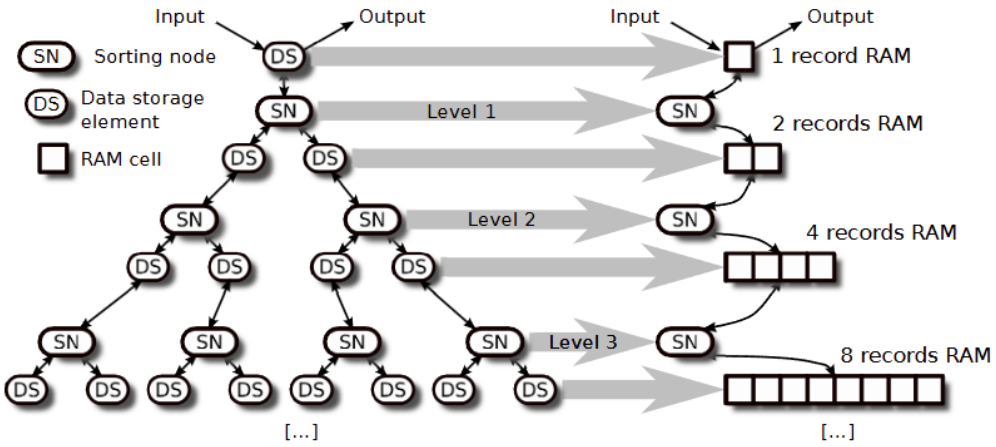


Figure 5.9: The hardware structure of a heap sorter is implemented to illustrate the transformation from independent storage elements to a RAM-based hierarchy. The top-level sorting node manages both the input and output data streams uniquely [95].

In the implementation of heap sorting, a suitable structure for hardware implementation was proposed in [95]. As illustrated in Figure 5.9, a complete heap binary tree is formed by combining sorting nodes (SN) and data storage elements (DS). Each node in the tree contains logic for data comparison and exchange (SN), as well as temporary data storage (DS). Input data flows downwards through each level, undergoing comparison and exchange operations to ensure that the minimum or maximum value moves towards the root node. The right-hand side illustrates the corresponding hardware implementation process. Each layer requires only one sorting node to perform comparison functions. Corresponding RAM units are used to store the sorted data records; the lower layers have larger RAM capacities to match the number of heap nodes. This structure combines hardware parallel computing with the fast read/write advantages of dual-port RAM, enabling the efficient mapping of the sorting algorithm to hardware by implementing only one sorting node per layer and reducing the required hardware resources.

Figure 5.10 shows the overall architecture of the minimum heap sorting hardware implementation described in [95]. Each level corresponds to a layer of nodes in the heap, and the structure's modular design makes it easy to expand and reuse. Each layer comprises a sorting node and a double-port SRAM (DP SRAM) unit. The sorting node compares and updates the data of the current node and its two child nodes to ensure that the heap order is maintained. The sorting node and the SRAM communicate via address control logic, which manages precise data read/write operations. The control logic schedules data blocks from the DP SRAM based on access requests from the current level, transmitting the data of the corresponding nodes that the sorting node needs to compare. The 'Update Information' signal path between the sorting nodes shown in the figure propagates updated information upwards after data updates. This enables rapid propagation of local changes throughout the heap, accelerating the sorting process. Level 0, the topmost level, also handles the heap's input and output ports, serving as a bridge between the top node of the heap and external interfaces.

Figure 5.11 clearly shows the process of inserting data and gradually establishing the heap structure at the beginning of a heap sorter based on a min-heap. The blue waveform in the figure represents the LEVEL 0 sort node, and the purple waveform represents the LEVEL 1 sort node. As shown in the diagram, the sorting nodes (`sort_node_0` and `sort_node_1`) at each level are activated sequentially as input data is fed in. The `pl_update` signal in the waveform indicates whether each level is processing input data or data passed down from the level above. In the first black box area on the left, once `data_in = 0x00000209` has been inputted, `sort_node_0` receives the data first and immediately writes it to the left RAM (`lm_we = 1`). Based on a comparison with the current heap top value of 0, it is determined that 209 should sink. Consequently, an update signal is sent to the next layer (Level 1). The `sort_node_1` is then activated and repeats the same process: it receives 209, compares it, and writes it to the left-side child node RAM. The behaviour in the waveform is consistent with the logic in Figure 5.8: each sorting node compares the current value (N) with its left (L) and right (R) child nodes. If $N \leq L$ and $N \leq R$, it remains unchanged; if L or R is smaller, it swaps and sends an update

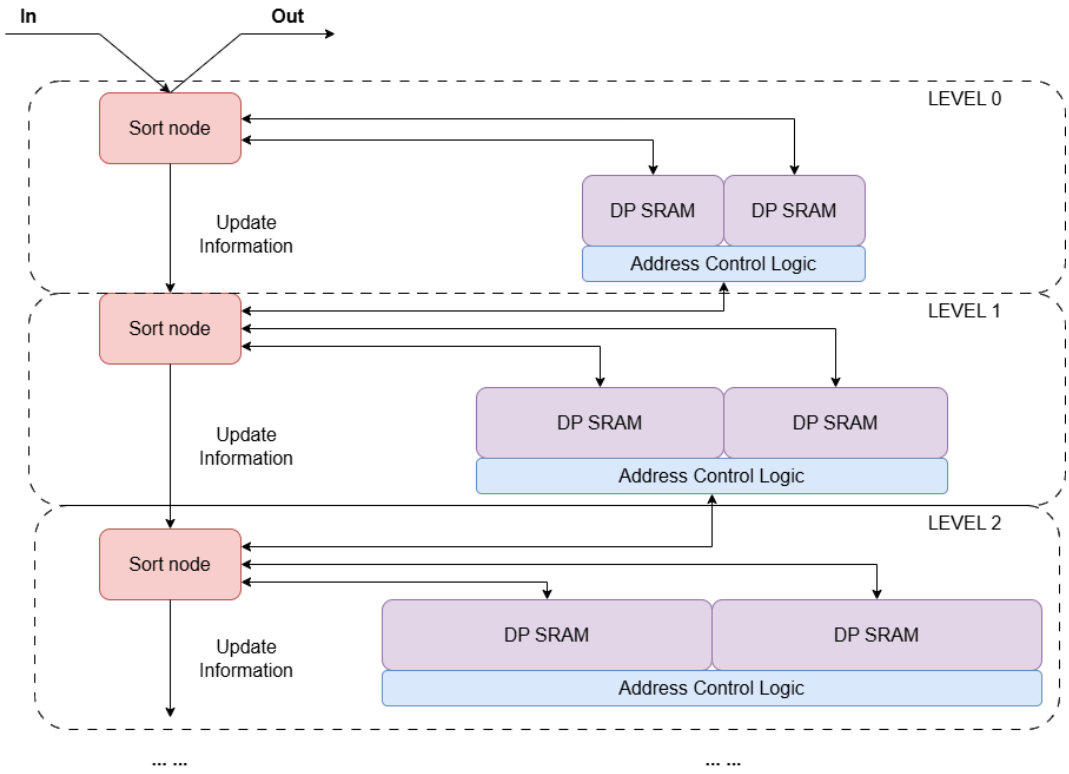


Figure 5.10: Hardware architecture of a hierarchical minimum heap sorter implementation. Each level consists of a sort node that is responsible for comparing and updating heap elements, as well as dual-port SRAM blocks that enable parallel data access and address control logic that manages memory transactions. Update information is propagated upwards through the levels to dynamically maintain heap properties. The topmost level (Level 0) interfaces directly with external input and output, coordinating heap insertion and extraction operations.

instruction to the next level.

Figure 5.12 illustrates the detailed operational process of the minimum heap sorter hardware during the flush phase. By combining waveforms with structural diagrams, it clearly shows how the maximum value is used in heap reconstruction to produce ordered output data. Once the flush signal has been raised, the sorter sequentially outputs the top heap elements and pushes `0xFFFFFFFF` as a placeholder value to the top of the heap, thereby triggering the heap reconstruction mechanism. During the first flush operation, once `0xFFFFFFFF` has been inserted, the sort node at Level 0 first activates the `left_memory_we signal`. This is followed by the sort node at Level 1 activating the `right_memory_we` signal. This indicates that the maximum value propagates down the heap path layer by layer, ultimately being written into the left RAM of the right child node at Level 1. This process is marked in the structure diagram below, where arrows indicate the transmission path of the maximum value from top to bottom, while smaller node values in the current heap float up to replace it. Using the output data as an example, after the top value of 12 is outputted, the value of 25 floats up to fill the top position. It is then outputted and replaced by the value of 53. Subsequently, data such as 58, 77, and 104 successively become the top value and are popped out. Each step is accompanied by the sinking of `0xFFFFFFFF` and the floating up of valid data. Throughout the flush process, `flush_cnt` gradually increases, and the `output_valid` signal is raised. The `output_data` then sequentially outputs the sorted values, ultimately achieving stable output of all data in the min-heap in ascending order. Rather than deleting the maximum value, the hardware structure continuously sinks it, avoiding complex pointer management or data movement. This achieves the hardware implementation of sorting logic through local comparisons and RAM writes, resulting in an efficient and clear design approach.

5.2.3. Architecture of the MAC-NN Accelerator

In Section 5.2.1 and Section 5.2.2, we designed hardware accelerators for the dot production and sorting algorithm of SP, thereby boosting performance for the SP stage. To accelerate the entire inference

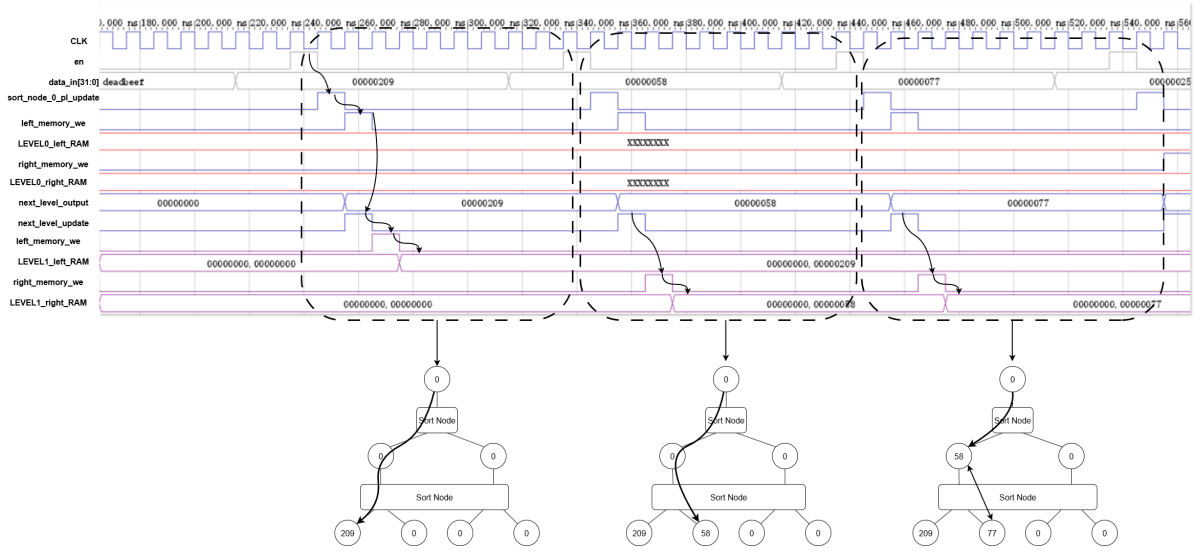


Figure 5.11: Simulation waveforms and state diagrams of a 2-level minimum heap sorter during the initial stage of constructing the heap structure. As mentioned earlier, we only have one sort node on each layer. The upper figure shows how the sorting nodes behave when performing sequential comparisons and writing to the double-port RAM with input data of 0x209, 0x58, and 0x77. The `left_memory_we` and `right_memory_we` signals at Levels 0 and 1 perform valid write operations as the data sinks, while the `next_level_update` signal transmits the sorting results. The lower figure shows the corresponding heap structure diagram at that moment, illustrating the sorting process whereby larger values sink down and smaller values float up, in line with the heap construction logic shown in the diagram: If the child node is smaller than the current node, swap and update the next level. To save hardware resources, it is important to note that the implemented min-heap structure only requires the root node to be less than or equal to the left and right child nodes, without enforcing the size relationship between them.

pipeline, we must also target FCNNs. As analyzed in Section 2.1, the dominant workload in an FCNN is the MAC operation between input vectors and weight matrices. Because individual input samples are independent, this workload exhibits a high degree of parallelism. Accordingly, this section focuses on the FCNNs MAC core and adopts the widely used systolic array architecture for parallel acceleration [96]. By arranging PEs in a regular spatial grid and streaming data through them in a time-skewed fashion, a systolic array maximizes data reuse while simplifying interconnect, and has been extensively validated as an efficient solution for largescale MAC tasks.

A systolic array is a highly parallel hardware architecture that arranges a large number of PEs in a grid and lets data pulse through the array in lockstep, performing computations every cycle [97]. This organization is ideal for multiply-accumulate workloads because such operations exhibit substantial data reuse and map naturally onto a spacetime pattern. In matrix-vector or matrix-matrix multiplication, the same weights and inputs are consumed many times [98]. By injecting inputs and weights into the array from orthogonal directions, each PE can repeatedly reuse locally stored data instead of making costly, highlatency accesses to large memories [99]. As shown in Figure 5.13, inputs are streamed from the left on-chip memory and propagate rightward each cycle, while weights enter from the top (or side) memory and flow downward. Inside every PE, the two operands are multiplied and added to the running partial sum, which then passes to the PE below until the bottom memory bank collects the final results. Throughout this process, data moves only between neighboring PEs via extremely short, high-speed wires, eliminating frequent large-memory reads/writes, reducing path delay, and lowering energy consumption [100]. Because I/O occurs solely at the arrays periphery, the interior retains a clean Manhattan interconnect [101], simplifying place-and-route and enabling straightforward scaling to larger arrays.

Figure 5.14 illustrates the timing of a 3×3 systolic array executing a matrix-matrix multiplication. At the initial cycle ($t = 0$), the first-row input vector (a_{13}, a_{12}, a_{11}) is injected from the left and aligns with the firstcolumn weights (w_{11}, w_{12}, w_{13}) inside the corresponding processing elements (PEs), completing the first set of multiplyaccumulate (MAC) operations. One cycle later ($t = 1$), these inputs shift right by one PE while the weights move down by one PE; simultaneously, the second-row inputs and second-column weights enter the array. The toprow PEs add the new products from the left to their stored

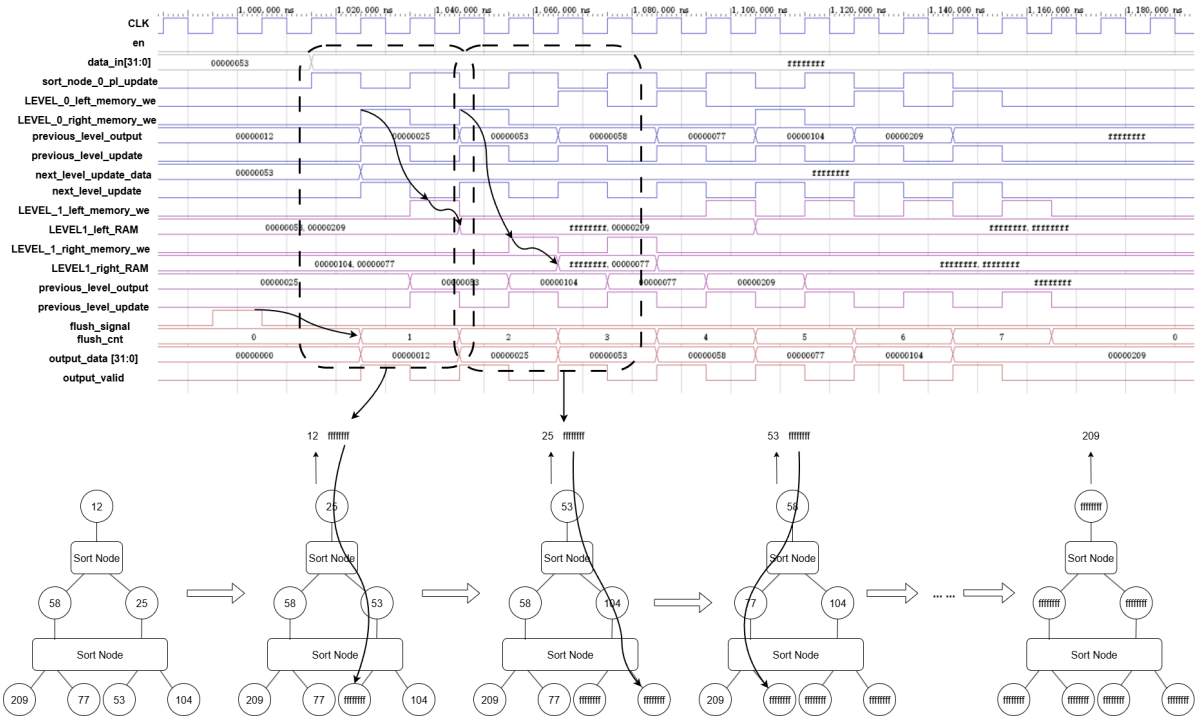


Figure 5.12: The simulation waveform and schematic diagram of the heap structure changes during the flush phase of the min-heap sorter. The upper half of the figure shows the simulation waveform, illustrating the complete process: after the `flush_signal` is pulled high, the top elements of the heap are popped out one by one; the maximum value, `0xFFFFFFFF`, is pushed into the top of the heap; and then it propagates downward along the path. The `flush_cnt` corresponds to the count of each output; `output_data` outputs the sorted data; and `output_valid` indicates the validity of the output. The lower half of the figure shows the heap structure changes that correspond to each step of the flush process. Arrows indicate the propagation paths of smaller values rising and the maximum value sinking during the heapification process. Using the first flush as an example, once the top element (12) has been output, the maximum value propagates downwards and eventually settles in the left subtree RAM of the right child node at level 1. Subsequently, smaller values (such as 25, 53, and 58) float up to the top of the heap in sequence.

partial sums, whereas the row below begins its first MAC. As the clock advances, inputs propagate rightward, row by row, and weights drift downward, column by column; each PE keeps its partial sum in local registers and continuously adds the new product. This strict spacetime skew ensures that, at any cycle k, a PE requires only the current input from its left neighbor and the current weight from the PE above, eliminating expensive accesses to distant memory. Once the data wavefront fully fills the array (at $t = 4$ in the figure), every subsequent cycle produces a fresh output that flows out of the bottomright PE to the lower memory bank, yielding sustained throughput.

Figure 5.15 shows a MAC-NN systolic array accelerator composed of a controller (FSM), input buffer, weight buffer, and result buffer, and a 16 × 16 PE array. External inputs and weight addresses are supplied with a 32bit width. Based on these addresses, the controller fetches 32bit weights in batches from the Weight SRAM on the right and writes them into the weight buffer, while simultaneously directing the input buffer to stream activations into the left side of the array. Weights are injected from the top of the array and inputs from the left; each PE performs a multiplyaccumulate operation and passes the partial sum downward. The final 32bit results exit from the bottom of the array into the result buffer and are then delivered to the external system. Figure 5.16 depicts the FSM of the MAC-NN accelerator. Starting from IDLE, the FSM proceeds through weights read (`READW`), inputs read (`READI`), compute (`CALCULATE`), and results output (`OUTPUT`) before reaching `FINISH` and returning to IDLE. Each stage is governed by its own counter: in `READW` the machine remains in the same state until `WEIGHT_CNT` reaches 16, then moves to `READI`; `READI` behaves similarly, advancing to `CALCULATE` once `INPUT_CNT` hits 16. During `CALCULATE`, the controller checks `CAL_CNT`. If all MAC operations are not yet finished, it loops within the state and stores the results into the result buffer; otherwise, it switches to `OUTPUT`. The OUTPUT phase writes the result from the buffer to the CPU, the FSM enters `FINISH` and resets to IDLE to await the next launch. This sequence guarantees that weights and activations

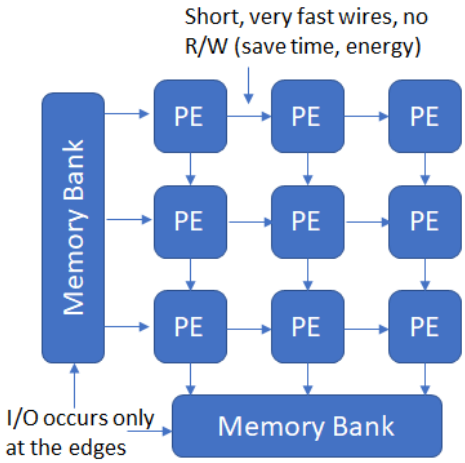


Figure 5.13: 3 × 3 systolic array: the left onchip memory bank feeds input data row by row, which shift rightward across the PE grid every cycle; weights are injected at the top and stream downward column by column, multiplying with the current inputs in each PE and accumulating into partial sums that continue flowing downward until the bottom memory bank collects the final results [102].

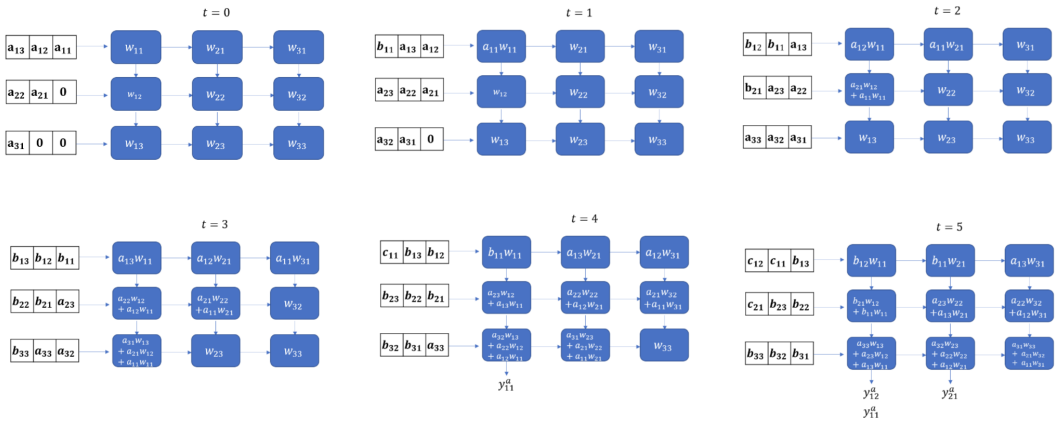


Figure 5.14: Timing diagram of a 3 × 3 systolic array multiplying two matrices: at $t = 0$, the first-row input vector is injected from the left and aligns with the first-column weights in the upper-left PEs to perform the initial MACs. Each subsequent cycle shifts the entire input wave one column to the right and the weight wave one row downward; the partial sums stored in each PE are updated with the new product from the left and passed to the PE below. At $t = 4$, the data wavefront has filled the array, after which a new final result emerges from the bottom-right PE every clock cycle [102].

are loaded in blocks, computation begins only after the data are fully prepared, and the accelerator terminates cleanly after the results have been stored.

5.3. System Integration and Functional Verification

After introducing the design details of the accelerators, we will now provide a detailed explanation of the system integration component and the associated functional verification process. Based on the description in Section 5.1, we have selected the open-source CROC SoC system as the base platform and integrated the three accelerators from the aforementioned design into this system, namely: the MAC SP accelerator, the minimum heap sort accelerator, and the MAC NN accelerator. The overall system structure is shown in Figure 5.2. For clarity, we will first use the MAC SP accelerator as an example to illustrate its integration method and verification process within the SoC.

In this design, the MAC-SP accelerator receives control instructions and data inputs from the CPU via a set of MMIO registers located in memory space. The CPU can configure and control the accelerator using standard LOAD and STORE instructions, thus avoiding the need for any dedicated instructions or bus protocols.

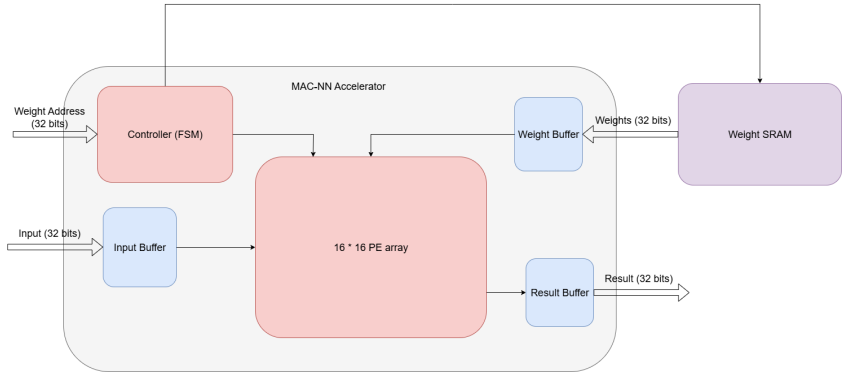


Figure 5.15: Block diagram of the MAC-NN systolic array accelerator. A controller (FSM) fetches 32bit weights from the external Weight SRAM into the Weight Buffer and streams 32bit inputs into the array via the Input Buffer; weights enter from the top, inputs from the left, and each PE performs MAC operations while propagating partial sums downward to the Result Buffer for output.

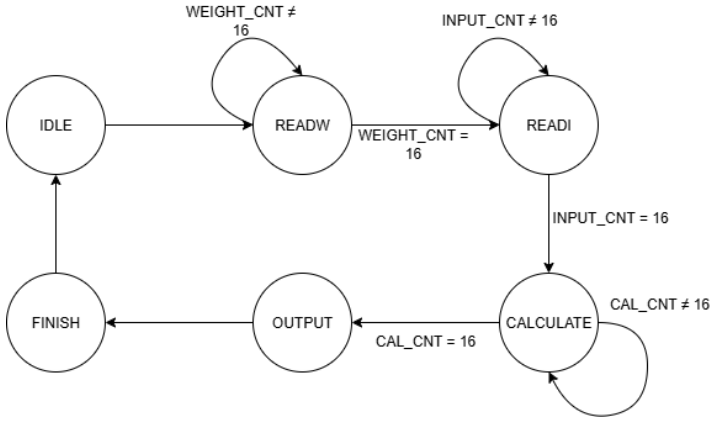


Figure 5.16: Illustration of the FSM of the MAC-NN accelerator. In `READW`, weights are fetched from the SRAM and stored in the weight buffer; in `READI`, the inputs are written into the corresponding input buffer. During CALCULATE, if `CAL_CNT` has not yet reached sixteen, the computation continues and intermediate results are stored in the result buffer. Finally, once computation is complete, all results are output sequentially from the result buffer.

The interaction process begins with the CPU using the STORE instruction to write the base address of vector B (where weights are stored in local SRAM) into the `ADDR_VEC_B` register and the number of vectors to be processed into the `ADDR_NUM_VEC` register, as shown in Figure 5.17. Subsequently, the CPU continuously writes the data of vector A into the `ADDR_VEC_A` register, thereby implementing data stream input. Once the configuration is complete, the CPU writes the value 1 to the `ADDR_CTRL` control register to issue a start pulse. Once the accelerator begins computation, it sets the busy flag to a high level, as shown in Figure 5.18. Upon completion, it generates a single-cycle done flag. The CPU can then poll the `ADDR_CTRL` register to determine whether the computation is complete, or read the `ADDR_RESULT` register to obtain the final dot product result. Table 5.3 shows the specific information of the MAC-SP accelerator's MMIO registers. Controlling the entire accelerator using standard read/write operations simplifies the software driver logic and makes it easier to expand other computational modules in the SoC system in the future.

Table 5.3: MMIO Register Mapping of the MAC-SP Accelerator

Address (offset)	Name	Description
0x00	ADDR_CTRL	Control register. Writing '1' triggers a one-cycle start pulse. Reading returns the status bits {done, busy}, indicating whether the accelerator is currently processing or has completed.
0x04	ADDR_VEC_B	Base address of Vector B in the local SRAM. The address is word-aligned (128-bit per word).
0x08	ADDR_NUM_VEC	Specifies the number of vectors to be processed (i.e., the number of dot-product operations to perform).
0x0C	ADDR_VEC_A	Stream-in port for Vector A data. Each 'store' operation sends a 32-bit word, sequentially forming the input vector.
0x10	ADDR_RESULT	Output port for dot-product results. Each 'load' returns a 16-bit result left-aligned within the 32-bit OBI read data.

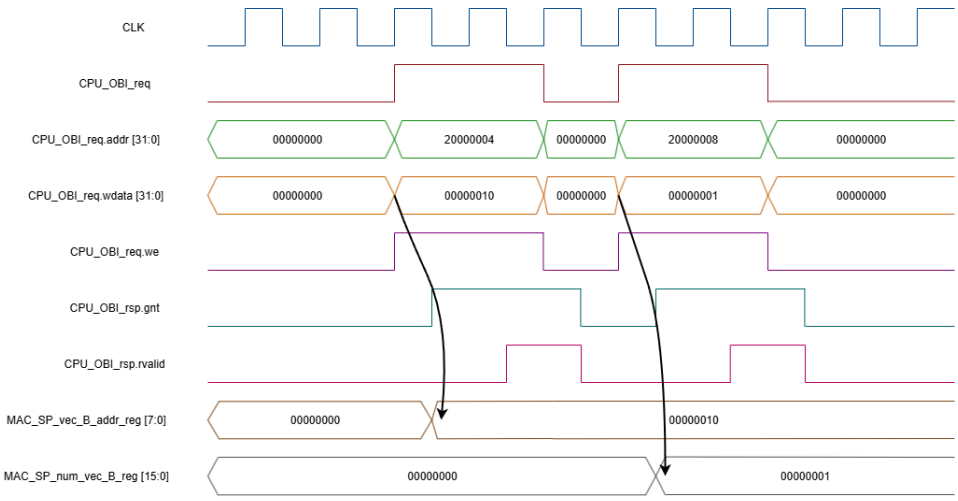


Figure 5.17: Illustration how the CPU configures the registers of the MAC-SP accelerator via the OBI bus. First, the CPU writes the value 0x10 to address 0x04 (ADDR_VEC_B), setting the base address of vector B. Then, it writes the value 0x01 to address 0x08 (ADDR_NUM_VEC), specifying that one vector is to be processed. Each write operation is completed via the gnt handshake signal, and the corresponding values are latched in the internal registers vec_B_addr_reg and num_vec_B_reg. This indicates that the CPU can initialise and control the accelerator using standard STORE instructions alone.

In summary, this implementation successfully integrates the proposed MAC-SP accelerator into the open-source CROC SoC platform, achieving interconnection with the system CPU via the standard OBI bus protocol. The accelerator is mounted as an MMIO slave device within the address space of the user domain. The CPU's LOAD and STORE instructions are used for the entire control and data interaction process, eliminating the need for special peripheral drivers and enhancing versatility and ease of use. For functional verification, two sets of critical simulation waveform diagrams detail how the CPU controls the MAC-SP accelerator to perform the entire process, from configuration to result output. The first set of waveforms illustrates the register write process and verifies the correct configuration of parameters. The second set of waveforms shows the accelerator's start-up and operational status (busy signal), as well as changes in the result validity signal (result_valid). This confirms the successful retrieval of calculation results via ADDR_RESULT and validates the functional correctness of the entire hardware path and MMIO interface. This modular design approach enables the accelerator to be easily integrated into any SoC platform supporting the OBI protocol, offering excellent reusability and scalability. Furthermore, the adoption of FIFO buffering and streaming processing mechanisms enables the accelerator to maintain high throughput when processing large-scale inputs, paving the way for future multi-core collaborative processing and optimization.

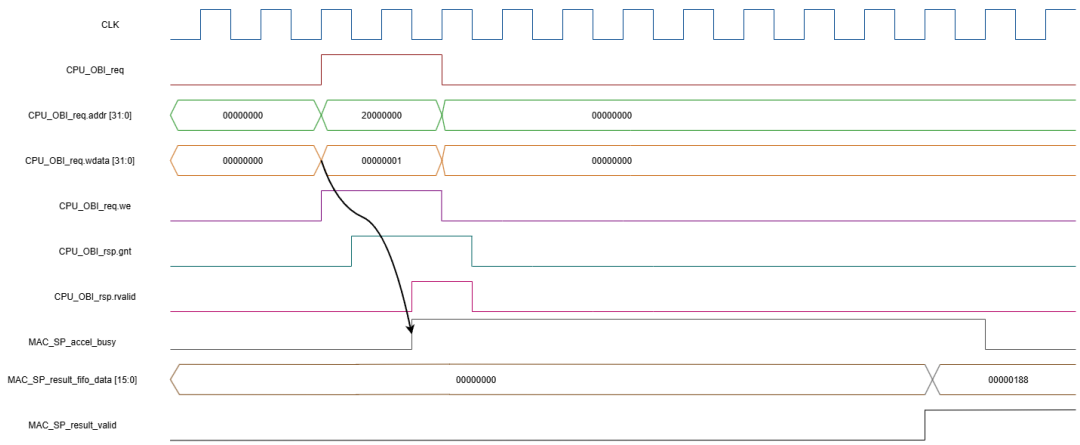


Figure 5.18: This diagram illustrates the accelerator's entire start-up process, from the CPU writing the start command to address `0x00` (`ADDR_CTRL`) to outputting results. The CPU uses the STORE instruction to write the value 0x1 to the control register, which triggers the start pulse. The `MAC_SP_accel_busy` signal is then pulled high, indicating that the accelerator has started processing. Once the calculation is complete, the `MAC_SP_result_valid` signal is pulled high and the final dot-product result (`0x0188` in this case) is output via the FIFO.

5.4. Hardware Synthesis and Resource Utilisation Analysis

This study used Cadence Genus as the synthesis tool and TSMC 40nm as the process node. The results obtained after synthesis are shown in Table 5.4. The table summarises the comprehensive results of the CROC SoC following the integration of the MAC-SP accelerator. It includes analyses of three key dimensions: area, timing, and power consumption. The area section lists the cell and net areas obtained from synthesis, and estimates the total chip area after fabrication at a typical routing density of 70%. The timing section provides the critical path delay, which is used to evaluate the system's maximum achievable operating frequency. The power consumption analysis is divided into three categories: static leakage power, internal dynamic power, and switching power. This provides a reference for the system's static and dynamic power characteristics.

Table 5.4: CROC SoC Synthesis Summary

Category	**Metric**	**Description**
Area Analysis	Total area 0.112 mm^2	The total area obtained after synthesis is 78,155 μm^2. Assuming a 70% placement density, the estimated post-layout area is approximately $\frac{78155}{10^6 \times 0.7} \approx 0.112\ mm^2$.
Timing Analysis	Maximum clock frequency 400 MHz	The longest path lies in the `ibex_if_stage` module, spanning from a register output through several logic gates to the instruction readback register.
Power Analysis	Total power 1.078 mW	This includes static leakage power of approximately 64.66 μW, internal dynamic power of 830.12 μW, and switching power of 183.46 μW.

Note: Power values are estimated post-synthesis using switching activity annotated from gate-level simulation (VCD-based). These results reflect realistic usage conditions under typical workloads.

Table 5.5 shows the acceleration effect of the custom hardware accelerator in this design, compared to the Ibex RISC-V CPU pure software implementation (the baseline platform). As can be seen in the table, the MAC-SP dot product accelerator achieves an acceleration ratio of approximately 28.7 when performing large-scale binary vector multiplication tasks. This is primarily due to its parallel data path design and high-bandwidth access to local SRAM, which reduces data movement and redundant computation overhead effectively. The minimum heap sort accelerator achieves a performance improvement of around 23.5 times compared to the CPU's computation when performing selection-based sorting operations. Its core advantage lies in its use of a structured hardware pipeline to perform heap ad-

justment operations, thereby avoiding a large number of conditional checks and jump delays. For the FCNN accelerator, using a 16 × 16 systolic array achieves a 38.1 × speedup. This is primarily because the array can perform 256 MAC operations per clock cycle while reducing data movement. However, computation cannot begin until both inputs and weights have been fully loaded, introducing an initial delay. In addition, after the compute phase, the system must wait for the remaining partial results to flush out of the array, and the 32-bit width of the OBI bus can cause stalls during data transfers. Most importantly, there is loop overhead: each computation handles only a 16 × 16 block, whereas the FCNN performs a 4096 × 4096 weight matrix multiplied by a vector of 4096 inputs. Consequently, the operation must be partitioned into many tiles, and each tile requires reloading data and writing results back. In summary, these results demonstrate that customised hardware modules can accelerate computation in specific task scenarios within resource-constrained embedded SoC systems.

Table 5.5: Speedup of Hardware Accelerators Compared to Ibex CPU

Accelerator Name	**Baseline Platform**	**Speedup Factor**
MAC-SP Dot Product Accelerator	Ibex CPU (software implementation)	28.7 ×
Heap Sort Accelerator	Ibex CPU (software implementation)	23.5 ×
MAC-NN Accelerator	Ibex CPU (software implementation)	38.1 ×

Table 5.6 compares the clock-cycle counts for a single-image MNIST inference on two platforms. A software implementation base on C program running entirely on the Ibex CPU consumes roughly 3.78×10^7 cycles, whereas the same workload executed on the CROC SoC augmented with the MAC-SP, Heap-Sort, and MAC-NN acceleratorsfinishes in only 1.43×10^6 cycles, corresponding to an overall speed-up of 26.4×.

This gain is slightly lower than the average of module level speedup shown in Table 5.5. This is because: First, all image data, weights, and biases must traverse a 32-bit OBI bus before inference can start; the limited bus width and arbitration delays throttle the otherwise highly parallel hardware. Second, the MAC-NN systolic array incurs a fixed warm-up and flush latency of a few dozen cycles to fill the array and drain the remaining partial sums; because the benchmark processes only one image, this overhead cannot be amortised over multiple inputs. Finally, the 4096-element vector is handled in 16 × 16 tiles, and each tile insertion introduces control bubbles between load, compute, and writeback phases. Together, these factors reduce the measured speed-up relative to the theoretical upper bound.

Table 5.6: Clockcycle count for a single MNIST inference, before and after adding hardware accelerators

Implementation	**Clock cycles**	**Speedup**
Ibex CPU (C, software-only)	37,755,951	1×
CROC SoC + Accelerators	1,430,150	26.4×

5.5. Design Comparison with Prior HTM Work

Table 5.7 contrasts the key metrics of the CROC SoC proposed in this thesis with the digital HTM-SP core reported by Zyarah *et al.* [19]. The latter, implemented on a Zynq-7000 FPGA, combines reconfigurable logic with an on-chip MCU and completes a full inference cycle (spatial pooling across 100 columns followed by SVM classification) in just 5.75 μ s at 100 MHz. This speed, however, comes at the cost of heavy reliance on LUT and BRAM configuration cells, whose static leakage and long programmable interconnect raise the sustained power to 13.9 mW and make migration to an area- and energy-efficient ASIC difficult. By contrast, the CROC SoC is fabricated in a 40 nm ASIC process: it tightly couples an Ibex CPU with three custom accelerators and, through gate-level optimization, packs a 4 096-column SP, a 16 × 16 systolic MAC array, and a sorting unit into just 0.112 mm^2. With low-voltage cell libraries and clock gating, total power drops to 1.078 mW while the core operates at 400 MHz. Although the larger column count and CPU-driven control raise single-image MNIST latency to 3.58 ms, the design still surpasses the FPGA solution in energy efficiency and attains 96.5% accuracy.

From an algorithmic-stack perspective, the FPGA design adopts a fixed SVM classifier, simple but limited in accuracy, achieving 91.16% accuracy on MNIST [19]. Our SoC replaces TM with lightweight

FCNN classifiers and enlarges the SP column count, forming an SP + learned discriminator path that improves accuracy to 96.5%. This indicates that, while remaining hardware-friendly, introducing a modest learnable component can significantly enhance generalization. The difference in task complexity also explains the higher end-to-end latency of our design: it executes a broader, heavier operator sequence, yet offers advantages in energy efficiency, scalability, and ASIC portability.

The two systems also diverge in memory hierarchy and data-flow organization. Relying on LUT/BRAM, the FPGA offers flexible routing but suffers from long programmable interconnects and non-negligible static leakage. In contrast, our SoC centers on SRAM macros and standard-cell logic, coordinated through an OBI bus and a unified MMIO register map so that the SP overlap computation, k-WTA sorting algorithm, and NN MAC operation can be stitched together in a streaming/blocked manner, reducing on-chip transfers and write-backs. On the SP side, column-wise partitioning and index compression shrink interface width; on the MAC-NN side, the systolic array exploits local reuse, improving compute utilization under the same bandwidth.

In terms of programmability, the CROC SoC employs the open-source RISC-V Ibex processor as a unified control plane. Driver software uses MMIO reads/writes to orchestrate accelerator start/stop, data I/O, and status querying, which eases both operator-level/system-level profiling and the integration of more complex pre-processing and post-processing pipelines on the same platform. By comparison, the FPGA is ideal for rapidly exploring architectural variants during development, but for deployment, the standardized HW/SW interface and broader software-ecosystem integration of a SoC platform are more complete.

Table 5.7: Comparison between the digital HTM–SP core by Zyarah *et al.* [19] and the CROC SoC proposed in this thesis

Design	Tech.	Accuracy	SP columns	Clock	Power	Latency	Area
Zyarah *et al.*	FPGA Zynq7000	91.16 %	100	100 MHz	13.9 mW	5.75 μs	44.39% FPGA resources
This work	40nm ASIC	96.5 %	4 096	400 MHz	1.078 mW	3.58 ms	0.112 mm^2

In summary, this chapter introduces the design, implementation, and integration processes of three types of specialized hardware accelerators systematically, with a focus on performance bottlenecks in sparse representation and neural network inference. Firstly, a function-level cycle analysis of the Ibex CPU was performed to identify the three most computationally intensive components in the system: SP overlap calculation, k-Winner-Take-All sorting selection, and FCNN/GRU multiply-accumulate operations. To address these three core computational tasks, we designed a MAC-SP binary dot product accelerator, a sorting accelerator based on a min-heap structure, and a MAC-NN neural network inference accelerator. This effectively alleviates the serial bottlenecks in the original software implementation.

The chapter also introduces the integration methods for each accelerator within the CROC SoC platform. A unified MMIO interface and OBI bus protocol enable the processor to flexibly control the start/stop of the accelerator, data input/output, and status queries while maintaining the simplicity of software programming and system scalability. Functional simulation and waveform verification confirmed the correctness and collaborative working mechanisms of each module. Additionally, synthesis analysis was completed using TSMC 40 nm process technology to evaluate area, power consumption, and timing performance. Finally, compared with the Ibex software baseline implementation, the SP , sorting, and FCNN accelerators achieved performance improvements of approximately 28.7×, 23.5×, and 38.1× respectively. With all three accelerators integrated into the CROC SoC, end-to-end inference on a single MNIST image runs 26.4× faster than the Ibex software baseline.

6

Conclusion and Future Work

6.1. Conclusion

This thesis addresses the core issue of how to efficiently implement HTM on low-power digital circuits through a systematic exploration ranging from the algorithmic to the hardware level. First, a complete software implementation and performance analysis of HTM were conducted. This revealed that the TM component of HTM is extremely sensitive to input perturbations, resulting in poor generalization capabilities in complex sequence tasks. To address this issue, we propose an alternative solution of using SP as a sparse encoder combined with a deep neural network: SP+FCNN is used for static scenes, while SP+RNN is used for temporal scenes. An attention mechanism is also introduced. Validation on the MNIST dataset shows that the improved SP+FCNN architecture increases classification accuracy from 86% to 96%, while maintaining SDR sparsity robustness and efficiency.

Subsequently, a function-level cycle analysis of the Ibex RISC-V core revealed that 98% of runtime was concentrated in three types of computation: SP overlap dot product (52.9%), FCNN/GRU multiply-accumulate (25.5%), and k-Winner-Take-All sorting (20.2%). To address these three bottlenecks, the following accelerators were proposed:

- a 25-way parallel MAC-SP binary dot product accelerator with fixed 25-cycle latency;
- a Heap-Sort sorting accelerator with a hierarchical min-heap structure supporting online k-WTA;
- a MAC-NN neural network multiply-accumulate array supporting 128 $\times$ 128 INT8 MAC operations.

These accelerators are mounted on the open-source CROC SoC via a unified MMIO interface and are interconnected with the Ibex core via the OBI bus. They support pipeline data flow and FIFO buffering. TSMC 40 nm process synthesis results show that the fully integrated SoC layout area is approximately 0.112 mm^2, the clock cycle is 750 ps, and the total power consumption is 1.078 mW. Compared to the Ibex pure software implementation, the MAC-SP, heap-sort accelerators, and FCNN accelerators achieve speedups of 28.7×, 23.5×, and 38.1× respectively, significantly enhancing inference throughput and energy efficiency.

6.2. Future Work

This thesis verifies the feasibility and efficiency of the 'SP-SDR + deep backend + dedicated triple accelerator' scheme on a 40 nm SoC. However, there are still areas that warrant further exploration, summarized below.

For software part, first is the SDR transmission encoding compression. In distributed systems, that transmitting large amounts of SDR between cores slows down overall throughput. One possible solution is information entropy-driven adaptive coding. By analysing the statistical distribution of active SDR bit positions, shorter codewords can be assigned to frequently occurring bit indices using ShannonFano or Huffman coding, thereby reducing the total number of bits without loss of information. This reduces final transmission power consumption at the hardware level.

Second, current experiments show that the original TM is highly sensitive to input perturbations, resulting in poor generalization in complex or noisy sequences. Inspiration could be drawn from deep learning techniques such as dropout, layernorm, or noise injection mechanisms by incorporating random perturbations or normalization terms during cell potential updates and synaptic threshold adjustments. This would smooth activation distributions and suppress overfitting. Another approach would be to integrate HTM column-cell structure with gated units or lightweight self-attention layers, which would allow for the dynamic regulation of synaptic connection strengths and enhance the modeling capability for long-range temporal dependencies.

For hardware part, first is to complete PNR and silicon verification. results in this thesis are based on logic-level synthesis and are still lacking in terms of complete placement and routing, DRC/LVS, and sign-off. In the future, we could consider completing the place-and-route process to observe the actual layout and routing, IR drop, clock tree, power consumption, and other conditions. We could also perform a tape-out to test power consumption and performance on an actual platform, comparing the RTL estimates with the actual measurements.

Second is optimization of the storage subsystem of the MAC-SP accelerator. The MAC-SP accelerator uses a single-port SRAM with a 128-bit bus. This hides a single read latency, but it may be limited by bandwidth when scaling to high parallelism. Consider replacing it with dual-port SRAM. Design separate clock domains for the read and write ports to enable the reading of one frame's weights and the writing of the next frame's weights to be completed in the same cycle. This will further improve throughput and reduce the need for arbitration logic.

Third is on-chip networking and scalability. As the number of columns and the scale of neural networks continue to grow, the single-core CPU plus single accelerator model will become constrained by interconnect and memory access bottlenecks. A suitable Network on Chip (NoC), such as Token-Ring or Mesh-Lite, can then be designed to provide scalable interconnects for multiple SP-MAC arrays, sorting engines, and NN arrays.

A

Appendix

A.1. Spatial Pooler Parameters

Table A.1: Parameters of Spatial Pooler.

Parameter	Value
Potential radius	6
Boost strength	30.0
Duty cycle period	1000
Local area density	0.1
Minimum overlap duty cycle percentage	0.1
Potential percentage	0.1
Stimulus threshold	2
Synaptic permanence increment p^+	0.1
Connection threshold for synaptic permanence	0.5
Synaptic permanence decrement p^-	0.02

- `potentialRadius`: defines the "receptive field" radius for each column in the SP, specifying how many input bits a column can potentially connect to within that radius.
- `potentialPct`: determines the percentage of input bits within the potential radius that will be connected to each column. For instance, if `potentialPct` = 0.5, the column randomly selects 50% of the bits in that radius. This ensures diversity among columns potential synapse pools, preventing identical learning patterns across all columns.
- `boostStrength`: controls the level of column "boosting". Boosting encourages infrequently winning columns to become more active, ensuring a more balanced use of resources. When `boostStrength` = 0.0, boosting is disabled; higher values make underactive columns more susceptible to activation, although too high values can lead to jittery behavior and instability.
- `dutyCyclePeriod`: refers to the length of the time window used to calculate the moving average of the column activation and overlap duty cycles. A longer duty cycle period smooths out fluctuations, making the system less sensitive to sudden changes but more stable, whereas a shorter period makes the system more sensitive to fluctuations at the cost of potential instability.
- `localAreaDensity`: is the proportion of columns that are intended to be active within a local inhibition region. During the inhibition phase of the SP, no more than `localAreaDensity` * N columns (where N is the total number of columns in the region) are allowed to remain active, thus enforcing a degree of sparsity.

- `stimulusThreshold`: is the minimum number of overlapping (i.e., active and connected) synapses required for a column to be considered in the subsequent inhibition phase. If a column's overlap with the current input falls below this threshold, the column's overlap is treated as zero, preventing activation due to noise.

- `synPermActiveInc`: is the permanent increase for synapses that are both active and belong to a winning column during learning iterations. For example, a default value of 0.05 increases permanence by 5% of the maximum possible value. A higher value means that active connections are strengthened more quickly.

- `synPermInactiveDec`: is the permanence decrease for synapses that belong to a winning column but are not active during learning iterations. For example, a default value of 0.008 reduces the permanence of these synapses by 0.8% per iteration. Higher values result in faster "forgetting" of unused connections and faster adaptation to new patterns.

- `synPermConnected`: is the threshold that determines whether a synapse is considered "connected". Only synapses with permanence values at or above this threshold contribute to the overlap count. If a synapse's permanence falls below this threshold, it is considered "unconnected". Typically, potential synapses are initialized near this threshold, so small increases or decreases in permanence will shift them between connected and unconnected states.

A.2. Temporal Memory Parameters

Table A.2: Parameters of Temporal Memory.

Parameter	Value
Cells per column	8
Activation threshold	5
Initial synaptic permanence	0.21
Connection threshold for synaptic permanence	0.3
Minimum segment activity threshold	5
Maximum number of new synapses per segment	30
Synaptic permanence increment p^+	0.1
Synaptic permanence decrement p^-	0.1

- `cellsPerColumn`: specifies the number of cells in each column. In HTM, multiple cells within the same column allow the system to represent different temporal contexts for the same spatial input. Larger values enable the model to encode higher-order sequences but also increase the number of potential predictive branches.

- `activationThreshold`: the threshold above which a dendritic segment is deemed active, determined by how many connected synapses are both active and attached to that segment. If the number of currently active, connected synapses on a segment reaches or exceeds the activation threshold, the segment is considered active, and the corresponding cell enters a predictive state.

- `initialPermanence`: the initial permanence value assigned to any newly formed synapse during the learning process. Permanence ranges from 0.0 to 1.0, reflecting the connections strength.

- `connectedPermanence`: The permanence threshold above which a synapse is deemed connected. Only connected synapses contribute to a segments overlap or activation count.

- `minThreshold`: The minimum overlap threshold for classifying a dendritic segment as matching. If a segments active synapses reach minThreshold, it is considered to partially match the current input, even if not enough to generate a prediction. During an HTM bursting event, a cell with the best-matching segment is chosen to learn or reinforce connections. If no segment in the column meets minThreshold, a new segment is formed on a previously unused cell.

- `maxNewSynapseCount`: The maximum number of new synapses allowed on a learning segment per iteration. When a segment is chosen for learning, new synapses may be added to connect previously active cells from the prior time step.

Bibliography

[1] James H Fetzer. "What is artificial intelligence?" In: *Artificial intelligence: Its scope and limits*. Springer, 1990, pp. 3–27.

[2] Samantha Hayman. "The mcculloch-pitts model". In: *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No. 99CH36339)*. Vol. 6. IEEE. 1999, pp. 4438–4439.

[3] Sun-Chong Wang. "Artificial neural network". In: *Interdisciplinary computing in java programming*. Springer, 2003, pp. 81–100.

[4] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[5] Keiron O'shea and Ryan Nash. "An introduction to convolutional neural networks". In: *arXiv preprint arXiv:1511.08458* (2015).

[6] Larry Medsker and Lakhmi C Jain. *Recurrent neural networks: design and applications*. CRC press, 1999.

[7] Vivienne Sze et al. "Efficient processing of deep neural networks: A tutorial and survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[8] Fei-Yue Wang et al. "Where does AlphaGo go: From church-turing thesis to AlphaGo thesis and beyond". In: *IEEE/CAA Journal of Automatica Sinica* 3.2 (2016), pp. 113–120.

[9] Andreas S Panayides et al. "AI in medical imaging informatics: current challenges and future directions". In: *IEEE journal of biomedical and health informatics* 24.7 (2020), pp. 1837–1857.

[10] Éloi Zablocki et al. "Explainability of deep vision-based autonomous driving systems: Review and challenges". In: *International Journal of Computer Vision* 130.10 (2022), pp. 2425–2452.

[11] JE (Hans) Korteling et al. "Human-versus artificial intelligence". In: *Frontiers in artificial intelligence* 4 (2021), p. 622364.

[12] Francisco Pedro. "Theoretical Analysis of the Brain and Artificial Intelligence". In: *Journal of Robotics Spectrum* 1 (2023), pp. 024–035.

[13] Goldman Sachs Research. *AI Is Poised to Drive 160% Increase in Data Center Power Demand*. Accessed 14 July 2025. May 2024. URL: https://www.goldmansachs.com/insights/articles/AI-poised-to-drive-160-increase-in-power-demand (visited on 07/14/2025).

[14] Xi Chen, Wei Wang, and Wei Li. "An overview of Hierarchical Temporal Memory: A new neocortex algorithm". In: *2012 Proceedings of International Conference on Modelling, Identification and Control*. IEEE. 2012, pp. 1004–1010.

[15] Subutai Ahmad and Jeff Hawkins. "Properties of sparse distributed representations and their application to hierarchical temporal memory". In: *arXiv preprint arXiv:1503.07469* (2015).

[16] Ryan William Price. *Hierarchical temporal memory cortical learning algorithm for pattern recognition on multi-core architectures*. Portland State University, 2011.

[17] Jia Wu, Weiru Zeng, and Fei Yan. "Hierarchical temporal memory method for time-series-based anomaly detection". In: *Neurocomputing* 273 (2018), pp. 535–546.

[18] Olga Krestinskaya, Irina Dolzhikova, and Alex Pappachen James. "Hierarchical Temporal Memory Using Memristor Networks: A Survey". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 2.5 (2018), pp. 380–395. DOI: 10.1109/TETCI.2018.2838124.

[19] Abdullah M Zyarah and Dhireesha Kudithipudi. "Neuromorphic architecture for the hierarchical temporal memory". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 3.1 (2019), pp. 4–14.

[20] Son Ngoc Truong, Khoa Van Pham, and Kyeong-Sik Min. "Spatial-pooling memristor crossbar converting sensory information to sparse distributed representation of cortical neurons". In: *IEEE Transactions on Nanotechnology* 17.3 (2018), pp. 482–491.

[21] Natalia Caporale and Yang Dan. "Spike timing–dependent plasticity: a Hebbian learning rule". In: *Annu. Rev. Neurosci.* 31.1 (2008), pp. 25–46.

[22] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[23] Ling Zhang and Bo Zhang. "A geometrical representation of McCulloch-Pitts neural model and its applications". In: *IEEE transactions on neural networks* 10.4 (1999), pp. 925–929.

[24] Xu Ma et al. "Development of a Robotic Catheter Manipulation System Based on BP Neural Network PID Controller". In: *Applied Bionics and Biomechanics* 2020 (Dec. 2020), pp. 1–11. DOI: 10.1155/2020/8870106.

[25] Stephen I Gallant et al. "Perceptron-based learning algorithms". In: *IEEE Transactions on neural networks* 1.2 (1990), pp. 179–191.

[26] Marvin L Minsky and Seymour A Papert. *Perceptrons: expanded edition*. 1988.

[27] Marius-Constantin Popescu et al. "Multilayer perceptron and neural networks". In: *WSEAS Transactions on Circuits and Systems* 8.7 (2009), pp. 579–588.

[28] David S Touretzky and Dean A Pomerleau. "Whats hidden in the hidden layers". In: *Byte* 14.8 (1989), pp. 227–233.

[29] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. "Activation functions in neural networks". In: *Towards Data Sci* 6.12 (2017), pp. 310–316.

[30] Jun Han and Claudio Moraga. "The influence of the sigmoid function parameters on the speed of backpropagation learning". In: *International workshop on artificial neural networks*. Springer. 1995, pp. 195–201.

[31] Juncai He et al. "ReLU deep neural networks and linear finite elements". In: *arXiv preprint arXiv:1807.03973* (2018).

[32] Daniele Grattarola. "Deep Feature Extraction for Sample-Efficient Reinforcement Learning". In: *Unpublished. https://doi. org/10.13140/RG* 2.30267.31527 (2017).

[33] K-Y Hsu, H-Y Li, and Demetri Psaltis. "Holographic implementation of a fully connected neural network". In: *Proceedings of the IEEE* 78.10 (1990), pp. 1637–1645.

[34] Timothy O Hodson. "Root mean square error (RMSE) or mean absolute error (MAE): When to use them or not". In: *Geoscientific Model Development Discussions* 2022 (2022), pp. 1–10.

[35] David M Allen. "Mean square error of prediction as a criterion for selecting variables". In: *Technometrics* 13.3 (1971), pp. 469–475.

[36] Zhilu Zhang and Mert Sabuncu. "Generalized cross entropy loss for training deep neural networks with noisy labels". In: *Advances in neural information processing systems* 31 (2018).

[37] Yaoshiang Ho and Samuel Wookey. "The real-world-weight cross-entropy loss function: Modeling the costs of mislabeling". In: *IEEE access* 8 (2019), pp. 4806–4813.

[38] Zachary C Lipton, John Berkowitz, and Charles Elkan. "A critical review of recurrent neural networks for sequence learning". In: *arXiv preprint arXiv:1506.00019* (2015).

[39] Stephen Grossberg. "Recurrent neural networks". In: *Scholarpedia* 8.2 (2013), p. 1888.

[40] Paul J Werbos. "Generalization of backpropagation with application to a recurrent gas market model". In: *Neural networks* 1.4 (1988), pp. 339–356.

[41] Klaus Greff et al. "LSTM: A search space odyssey". In: *IEEE transactions on neural networks and learning systems* 28.10 (2016), pp. 2222–2232.

[42] Rahul Dey and Fathi M Salem. "Gate-variants of gated recurrent unit (GRU) neural networks". In: *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*. IEEE. 2017, pp. 1597–1600.

[43] Aston Zhang et al. *Dive into Deep Learning*. https://D2L.ai. Cambridge University Press, 2023.

[44] Shudong Yang, Xueying Yu, and Ying Zhou. "Lstm and gru neural network performance comparison study: Taking yelp review dataset as an example". In: *2020 International workshop on electronic communication and artificial intelligence (IWECAI)*. IEEE. 2020, pp. 98–101.

[45] Leo Feng et al. "Were rnns all we needed?" In: *arXiv preprint arXiv:2410.01201* (2024).

[46] Richard Sproat and Navdeep Jaitly. "RNN approaches to text normalization: A challenge". In: *arXiv preprint arXiv:1611.00068* (2016).

[47] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[48] Gianni Brauwers and Flavius Frasincar. "A general survey on attention mechanisms in deep learning". In: *IEEE transactions on knowledge and data engineering* 35.4 (2021), pp. 3279–3298.

[49] Hiroshi Fukui et al. "Attention branch network: Learning of attention mechanism for visual explanation". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 10705–10714.

[50] Shashank. *Understanding Attention Mechanism*. 2020. URL: https://shashank7-iitd.medium.com/understanding-attention-mechanism-35ff53fc328e.

[51] Meng-Hao Guo et al. "Attention mechanisms in computer vision: A survey". In: *Computational visual media* 8.3 (2022), pp. 331–368.

[52] Noha O El-Ganainy et al. "On the performance of hierarchical temporal memory predictions of medical streams in real time". In: *2019 13th International Symposium on Medical Information and Communication Technology (ISMICT)*. IEEE. 2019, pp. 1–6.

[53] Olga Krestinskaya, Irina Dolzhikova, and Alex Pappachen James. "Hierarchical Temporal Memory Using Memristor Networks: A Survey". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 2 (5 2018). ISSN: 2471285X. DOI: 10.1109/TETCI.2018.2838124.

[54] Peer Neubert, Subutai Ahmad, and Peter Protzel. "A sequence-based neuronal model for mobile robot localization". In: vol. 11117 LNAI. 2018. DOI: 10.1007/978-3-030-00111-7_11.

[55] Dmitri A Rachkovskij and Ernst M Kussul. "Binding and normalization of binary sparse distributed representations by context-dependent thinning". In: *Neural Computation* 13.2 (2001), pp. 411–452.

[56] Scott Purdy. "Encoding data for HTM systems". In: *arXiv preprint arXiv:1602.05925* (2016).

[57] Subutai Ahmad and Jeff Hawkins. "How do neurons operate on sparse distributed representations? A mathematical theory of sparsity, neurons and active dendrites". In: *arXiv preprint arXiv:1601.00720* (2016).

[58] Denis Kleyko, Evgeny Osipov, and Dmitri A Rachkovskij. "Modification of holographic graph neuron using sparse distributed representations". In: *Procedia Computer Science* 88 (2016), pp. 39–45.

[59] J. Hawkins et al. "Biological and Machine Intelligence (BAMI)". Initial online release 0.4. 2016. URL: https://numenta.com/resources/biological-and-machine-intelligence/.

[60] Zheng Zhang et al. "A survey of sparse representation: algorithms and applications". In: *IEEE access* 3 (2015), pp. 490–530.

[61] Subutai Ahmad and Luiz Scheinkman. "How can we be so dense? The benefits of using highly sparse representations". In: *arXiv preprint arXiv:1903.11257* (2019).

[62] Sai Kiran Reddy Malikireddy, Bipinkumarreddy Algubelli, and Snigdha Tadanki. "Hierarchical temporal memory for real-time pattern recognition in distributed data streams". In: *European Journal of Advances in Engineering and Technology* 5.12 (2018), pp. 1156–1167.

[63] M. Otahal, D. Keeney, D. McDougall, et al. *HTM.core implementation of Hierarchical Temporal Memory*. https://github.com/htm-community/htm.core/. 2019.

[64] Jeff Hawkins and Subutai Ahmad. "Why neurons have thousands of synapses, a theory of sequence memory in neocortex". In: *Frontiers in neural circuits* 10 (2016), p. 23.

[65] Lulu Ge and Keshab K Parhi. “Classification using hyperdimensional computing: A review”. In: *IEEE Circuits and Systems Magazine* 20.2 (2020), pp. 30–47.

[66] Anthony Thomas, Sanjoy Dasgupta, and Tajana Rosing. “A theoretical perspective on hyperdimensional computing”. In: *Journal of Artificial Intelligence Research* 72 (2021), pp. 215–249.

[67] Peer Neubert, Stefan Schubert, and Peter Protzel. “An introduction to hyperdimensional computing for robotics”. In: *KI-Künstliche Intelligenz* 33.4 (2019), pp. 319–330.

[68] Dmitri B Chklovskii, BW Mel, and K Svoboda. “Cortical rewiring and information storage”. In: *Nature* 431.7010 (2004), pp. 782–788.

[69] Eman Hassan et al. “Hyper-dimensional computing challenges and opportunities for AI applications”. In: *IEEE Access* 10 (2021), pp. 97651–97664.

[70] Olga Krestinskaya, Timur Ibrayev, and Alex Pappachen James. “Hierarchical temporal memory features with memristor logic circuits for pattern recognition”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.6 (2017), pp. 1143–1156.

[71] Yuwei Cui, Subutai Ahmad, and Jeff Hawkins. “The HTM spatial poolerA neocortical algorithm for online sparse distributed coding”. In: *Frontiers in computational neuroscience* 11 (2017), p. 111.

[72] Nelson Spruston. “Pyramidal neurons: dendritic structure and synaptic integration”. In: *Nature Reviews Neuroscience* 9.3 (2008), pp. 206–221.

[73] Timur Ibrayev et al. “A design of HTM spatial pooler for face recognition using memristor-CMOS hybrid circuits”. In: *2016 IEEE international symposium on circuits and systems (ISCAS)*. IEEE. 2016, pp. 1254–1257.

[74] James Mnatzaganian, Ernest Fokoué, and Dhireesha Kudithipudi. “A mathematical formalization of hierarchical temporal memorys spatial pooler”. In: *Frontiers in Robotics and AI* 3 (2017), p. 81.

[75] Alex Waibel. “Modular construction of time-delay neural networks for speech recognition”. In: *Neural computation* 1.1 (1989), pp. 39–46.

[76] Kevan AC Martin and Sylvia Schröder. “Functional heterogeneity in neighboring neurons of cat primary visual cortex in response to both artificial and natural stimuli”. In: *Journal of Neuroscience* 33.17 (2013), pp. 7325–7344.

[77] Li Deng. “The mnist database of handwritten digit images for machine learning research [best of the web]”. In: *IEEE signal processing magazine* 29.6 (2012), pp. 141–142.

[78] Alejandro Baldominos, Yago Saez, and Pedro Isasi. “A survey of handwritten character recognition with mnist and emnist”. In: *Applied Sciences* 9.15 (2019), p. 3169.

[79] Alex M Lamb et al. “Professor forcing: A new algorithm for training recurrent networks”. In: *Advances in neural information processing systems* 29 (2016).

[80] Lei Song, Haoran Liang, and Taisheng Zheng. “Real-time anomaly detection method for space imager streaming data based on HTM algorithm”. In: *2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE. 2019, pp. 33–38.

[81] Yanzhe Chen et al. “Fast and general distributed transactions using RDMA and HTM”. In: *Proceedings of the Eleventh European Conference on Computer Systems*. 2016, pp. 1–17.

[82] Chundong Wang et al. “A distributed anomaly detection system for in-vehicle network using HTM”. In: *IEEE Access* 6 (2018), pp. 9091–9098.

[83] Jonathan Mackenzie, John F Roddick, and Rocco Zito. “An evaluation of HTM and LSTM for short-term arterial traffic flow prediction”. In: *IEEE Transactions on Intelligent Transportation Systems* 20.5 (2018), pp. 1847–1857.

[84] Mohamed AbdElhamid Abbas. “Improving deep learning performance using random forest HTM cortical learning algorithm”. In: *2018 first international workshop on deep and representation learning (IWDRL)*. IEEE. 2018, pp. 13–18.

[85] Abdullah M. Zyarah and Dhireesha Kudithipudi. "Neuromorphic Architecture for the Hierarchical Temporal Memory". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 3.1 (2019), pp. 4–14. DOI: 10.1109/TETCI.2018.2850314.

[86] Hengshuang Zhao, Jiaya Jia, and Vladlen Koltun. "Exploring self-attention for image recognition". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 10076–10085.

[87] Feng Wang and David MJ Tax. "Survey on the attention based RNN model and its applications in computer vision". In: *arXiv preprint arXiv:1601.06823* (2016).

[88] Jacob R Stevens et al. "Softermax: Hardware/software co-design of an efficient softmax for transformers". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2021, pp. 469–474.

[89] Jingjing Xu et al. "Understanding and improving layer normalization". In: *Advances in neural information processing systems* 32 (2019).

[90] Phillippe Sauter et al. "Croc: An End-to-End Open-Source Extensible RISC-V MCU Platform to Democratize Silicon". In: *arXiv preprint arXiv:2502.05090* (2025).

[91] PULP Platform. *OBI (Open Bus Interface)*. https://github.com/pulp-platform/obi. Accessed: 2025-07-31. 2022.

[92] John William Joseph Williams. "Algorithm 232: heapsort". In: *Communications of the ACM* 7.6 (1964), pp. 347–348.

[93] Russel Schaffer and Robert Sedgewick. "The analysis of heapsort". In: *Journal of Algorithms* 15.1 (1993), pp. 76–100.

[94] Marcellino Marcellino et al. "Comparative of advanced sorting algorithms (quick sort, heap sort, merge sort, intro sort, radix sort) based on time and memory usage". In: *2021 1st international conference on computer science and artificial intelligence (ICCSAI)*. Vol. 1. IEEE. 2021, pp. 154–160.

[95] Wojciech M Zabootny. "Dual port memory based heapsort implementation for fpga". In: *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2011*. Vol. 8008. SPIE. 2011, pp. 117–125.

[96] Hsiang Tsung Kung and Charles E Leiserson. "Systolic arrays (for VLSI)". In: *Sparse Matrix Proceedings 1978*. Vol. 1. Society for industrial and applied mathematics Philadelphia, PA, USA. 1979, pp. 256–282.

[97] Hsiang-Tsung Kung. *Why systolic architecture?* Design Research Center, Carnegie-Mellon University Pittsburgh, PA, USA, 1982.

[98] W Morven Gentleman and HT Kung. "Matrix triangularization by systolic arrays". In: *Real-time signal processing IV*. Vol. 298. SPIE. 1982, pp. 19–26.

[99] Rui Xu et al. "A survey of design and optimization for systolic array-based DNN accelerators". In: *ACM Computing Surveys* 56.1 (2023), pp. 1–37.

[100] Javier Mora and Eduardo de la Torre. "Accelerating the evolution of a systolic array-based evolvable hardware system". In: *Microprocessors and Microsystems* 56 (2018), pp. 144–156.

[101] Cheng-Kok Koh and Patrick H Madden. "Manhattan or non-Manhattan? A study of alternative VLSI routing architectures". In: *Proceedings of the 10th Great Lakes symposium on VLSI*. 2000, pp. 47–52.

[102] Ankur Mohan. *Understanding Matrix Multiplication on a WeightStationary Systolic Architecture*. Blog post on the Telesens website. July 2018. URL: https://www.telesens.co/2018/07/30/systolic-architectures/ (visited on 07/22/2025).