Low-cost multi-core on-chip learning schemes

by

Zhaofeng Shen

to obtain the degree of Master of Science at the Delft University of Technology,
to be defended publicly on September 08, 2023.

Student number:	5404568	
Project duration:	November, 2022 - September, 2023	
Thesis committee:	Dr. Charlotte Frenkel	TU Delft, supervisor
	Prof. Dr. Kofi Makinwa	TU Delft
	Dr. Chang Gao	TU Delft

TU Delft

Abstract

Nowadays, to reduce the dependence of devices on cloud servers, machine learning workloads are required to process data on the edge. Furthermore, to improve adaptability to uncontrolled environments, there is a growing need for on-chip learning. Limitations in power and area for edge devices have increased interest in low-cost neural network learning algorithms. However, as edge platforms are increasingly multi-core, new techniques are required to deploy learning algorithms on multi-core designs.

In this report, the performance of a low-cost multi-core on-chip learning platform with the local error learning (LEL) algorithm is evaluated. First, we reviewed state-of-art learning algorithms designed to solve the challenges of efficient neural network learning. We analyze these algorithms from the point of view of performance, hardware overhead, scalability, and the possibility of multi-core implementation. We propose a spatio-temporal learning framework for the combined use of LEL and e-prop. As a first proof of concept, we aim first at demonstrating multi-core LEL learning for image classification. Next, we constructed a software model suitable for multi-core on-chip driven by hardware requirements. With the software model, we then implemented the corresponding hardware and deployed it on a system-on-chip field programmable gate array (SoC FPGA) board to evaluate the performance. Results based on the CIFAR-10 image classification dataset show that the hardware design can fully reproduce the software runtime results. With a classification accuracy of 59.57% after batch-size-1 on-chip learning, our design forms a stepping stone for the development of low-cost multi-core hardware that can adapt online to its environment.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
ANN	Artificial neural network
BP	Backpropagation
BPTT	Backpropagation through time
CNN	Convolutional neural network
Conv	Convolutional
DDTP	Direct difference target propagation
DFA	Direct feedback alignment
DNI	Decoupled neural interface
DRAM	Dynamic random-access memory
DRTP	Direct random target projection
DTP	Difference target propagation
DUT	Device under test
EP	Equilibrium propagation
FA	Feedback alignment
FC	Fully-connected
FIFO	First-in-first-out
FPGA	Field programmable gate array
FPTT	Forward-propagation-through-time
IP	Intellectual properties
LEL	Local error learning
LFSR	Linear feedback shift register
LIF	Leaky integrate-and-fire
LUT	Lookup table
FSM	Finite state machine
MAC	Multiply–accumulate
MSE	Mean squared error
NAS	Neural architecture search
OSTL	Online spatio-temporal learning
PDF	Probability density function
PL	Programming logic
PRNG	Pseudorandom number generator
PS	Processing sytem
PTQ	Post training quantization
QAT	Quantization aware training
ReLU	Rectified linear unit
RNN	Recurrent neural network
RTRL	Real-time recurrent learning
SELU	Scaled exponential linear unit
SG	Synthetic gradient
SGD	Stochastic gradient descent
SNN	Spiking neural network
SoC	System-on-chip
SRM	Spike response model
STE	Straight through estimator
TP	Target propagation

Abbreviation	Definition
TTFS	Time-to-first-spike

1
Introduction

This chapter provides an overview of the thesis project. First, the background of the project is presented. Then the challenges arising from this background are presented. The objectives of the project to address these challenges are then listed, as well as the contributions of this project. Finally, an outline of this report is presented.

1.1. Background

Neural networks are powerful machine learning tools, which can have competitive or even better performance than humans on tasks like image classification [1] and the game of Go [2, 3]. However, this requires a huge amount of computing power to achieve such performance. For example, to outperform humans in the game of Go, AlphaGo Zero has improved from its several predecessors and still requires about 1kW of power consumption (Figure 1.1) while, on the other side, the human brain only requires 20W on average.

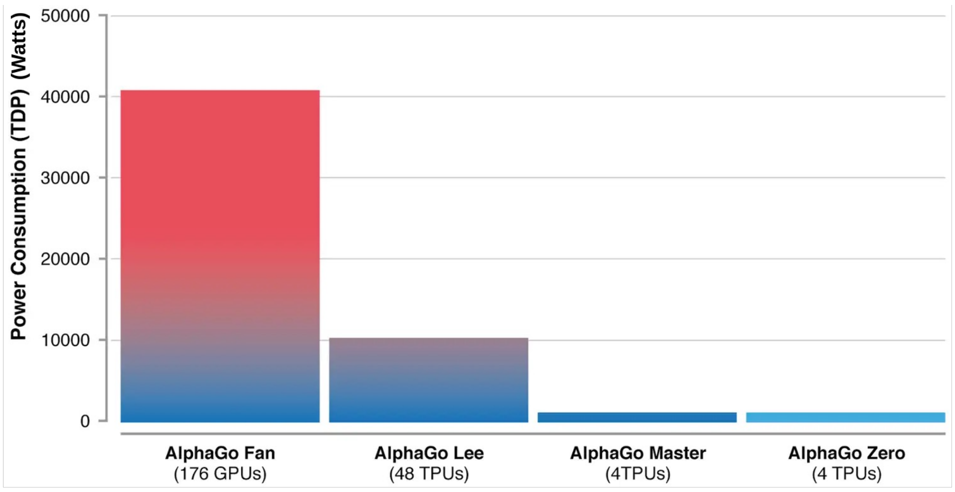

Figure 1.1: Changes in AlphaGo power consumption as hardware as well as algorithms evolve [4]: Redder and darker colors represent higher power consumption, bluer and lighter colors represent lower power consumption.

Moreover, deploying learning on edge devices is needed for use cases dealing with uncontrolled environments without a cloud link, requiring online user adaptation, or where privacy is critical [5]. Because of tight power and area budgets, when migrating neural networks to edge devices, more energy-efficiency-oriented neural network models and hardware are preferred. For example, neuromorphic hardware based on emerging bio-plausible learning algorithms offers new avenues for low-cost learning at the edge [6, 7]. At the same time, this demand has brought several new challenges in both learning algorithms and hardware.

1.2. Challenges

In this section, we will describe how the energy efficiency of multi-core on-chip learning is limited by the learning algorithm, and the main sources of overhead in hardware.

1.2.1. Learning algorithms

Here, we will refer to the error in the network as a credit [8] and discuss it in two categories. One is the problem of spatial credit assignment for multilayer networks. This problem is generally solved by error backpropagation (BP), due to its ability to propagate the error to each parameter in each layer of the network through the application of the chain rule. In Figure 1.2, x is the data input, W_i are the weights

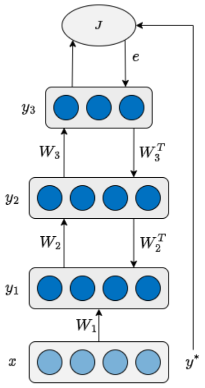

Figure 1.2: Error backpropagation (BP). Adapted from [9].

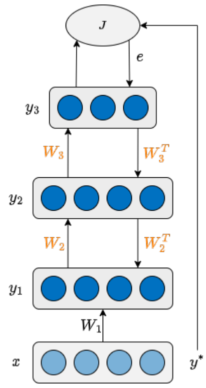

Figure 1.3: Weight symmetry. Adapted from [10].

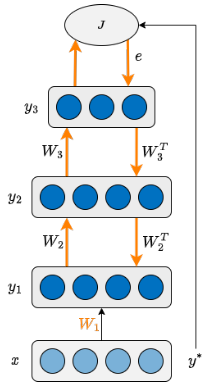

Figure 1.4: Update locking. Adapted from [11].

of the different layers, y_i are the outputs of the different layers, y^* is the target, J is the loss funtion, e is the error and i is the layer index. During learning, BP is able to propagate the credits from the loss function to the parameters of all downstream layers. Its calculation to obtain the updated value of the weight W_1 can also be expressed mathematically as:

$$\frac{\partial J}{\partial W_1} = \frac{\partial J}{\partial y_3}\frac{\partial y_3}{\partial y_2}\frac{\partial y_2}{\partial y_1}\frac{\partial y_1}{\partial W_1}$$
$$= eW_3^T W_2^T x$$

While BP can provide effective learning, its efficiency and bio-plausibility are limited by two main reasons [5]:

- **Weight symmetry:** The issue that both forward and backward connections require to access the same weights is called the "weight transport" [10] or "weight symmetry" [12], as highlighted in figure 1.3. In hardware, this can lead to the use of complex memory access patterns and architectures [5]. From a different perspective, this is also not bio-plausible for backpropagation in the brain, as this would imply that synapses are bidirectional [8].
- **Update locking:** As the gradients of each layer are derived from the final loss by using the chain rule, all the layers can only be updated after the forward and backward passes have been fully completed. This problem is called "update locking" [11]. In Figure 1.4, if we want to update W_1, we need to wait for all the calculations involved in the highlighted path to be completed. This makes BP have two phases which require saving all layer activation values before the learning phase is completed. And in a multilayer network, this coupling between layers can make the learning process slow. This two-phase learning is also not bio-plausible.

As summarized in [14], for recurrent neural networks (RNNs), there is also the temporal credit assignment. In Figure 1.5, we expand the RNN on a timeline and we can now propagate information forward (right) or backward (left) in this dimension. Backpropagation through time (BPTT) propagates credit backward in the temporal dimension by unrolling the network in time. As demonstrated in Figure 1.6, credits can be passed backwards through the highlighted path to the start of the time series to

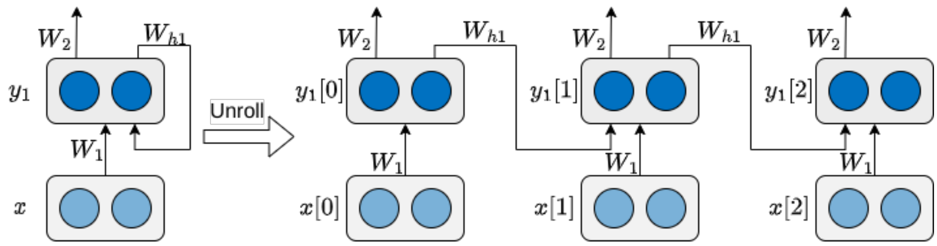

Figure 1.5: Unrolled recurrent neural network: x is the input sequence, y is the output sequence, W_i are the forward weight matrices and W_{hi} are the recurrent weight matrices. Adapted from [13].

update W_1. This corresponds to a *backward* method [14] and has a memory complexity of $O(Tn)$ [15], where T stands for the number of time steps and n for the number of neurons in one layer.

Real-time recurrent learning (RTRL) [16], on the other hand, enables online credit assignment by propagating the necessary information forward, which corresponds to a *forward* method [14]. As shown in Figure 1.7, if we want to update the weight W_1 at time point 2, RTRL can bring the relevant information from the previous time point to the current time point, thus completing the forward propagation of credits. Only this method enables online learning [13], whose computational costs do not scale with the number of time steps but will grow rapidly with the number of neurons. For example, RTRL has a memory complexity of $O(n^3)$ [15]. BPTT thus scales poorly with long temporal depths, while RTRL scales poorly with large network sizes.

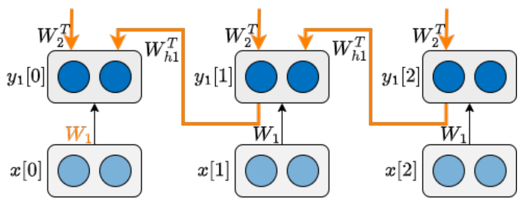

Figure 1.6: Backpropagation through time. Adapted from [13].

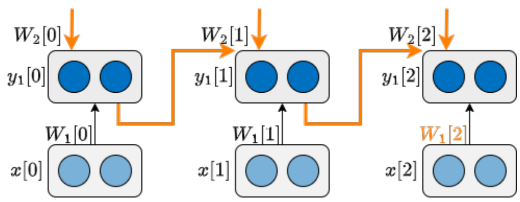

Figure 1.7: Real-time recurrent learning. Adapted from [13].

1.2.2. Hardware

Edge devices have a tight power budget, especially for battery-powered devices [17]. As for the area utilization and power consumption of the learning algorithms on the hardware, the main impact comes from the following two aspects [18]:

- **Memory overhead:** In general, memory takes up most of the area on the chip, and memory accesses introduce most of the power overhead [19, 20]. On-chip learning algorithms might induce buffering overhead or additional parameters compared to the inference-only network, which needs to be stored on-chip. Therefore, optimization of memory overhead is critical to edge devices.
- **Computational overhead:** On-chip learning also requires more hardware computing power than inference-only networks because of additional computation that will increase the power footprint.

1.2.3. Multi-core on-chip learning

In this project, we aim to implement a multi-core on-chip learning scheme in hardware, and in addition to the challenges described above, there are other challenges in multi-core design. Here are two key multicore designs that have their own strengths and weaknesses:

- **Independent multi-core setup:** The Figure 1.8 illustrates a multi-core design where each core has an independent neural network, and to obtain classification results from the entire design, we merge their respective outputs. This approach involves minimal data communication beyond the core, and it can increase the throughput of the whole system. However, due to the limited hardware resources, this design can be apparented to training multiple weak neural networks at the same time, compared to a single strong model.

- **Layer-wise multi-core setup:** Another multi-core design is illustrated in Figure 1.9 where a single layer of computation is assigned to each core. By maintaining the original network size and structure, this approach achieves optimal performance. Despite the benefits of this method, it generates significant inter-core communication, including transferring layer activation values and back-propagation errors. Furthermore, this approach is not useful for improving the system throughput.

Therefore, in order to balance data communication and performance, we need an algorithm that allows each core to focus on training locally without a significant degradation of performance.

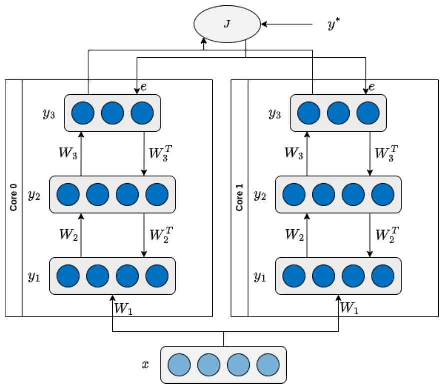

Figure 1.8: Independent multi-core setup

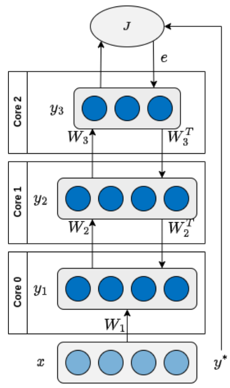

Figure 1.9: Layer-wise multi-core setup

1.3. Contributions

Based on the challenges of multi-core on-chip learning mentioned earlier, we decided to start with learning algorithm research and formed the following objectives of the thesis project:

- Compare a variety of state-of-the-art learning algorithms for artificial neural networks (ANNs), analyse their advantages over traditional BP in terms of challenges mentioned above, and select one of them as the main research object for the project.
- Use machine learning frameworks to build a model suitable for performance evaluation and optimize it for subsequent hardware implementation.
- Implement the model with multi-core hardware design, and deploy to the PYNQ-Z1 (XC7Z020) field programmable gate array (FPGA) board for practical performance evaluation.

At the end of the project we were able to:

- Verify the feasibility of the chosen learning algorithm for learning with low-cost multi-core hardware design.
- Propose improvements for the currently encountered problems and lay the foundation for future work.

1.4. Thesis outline

Chapter 2 will demonstrate the literatrue review about state-of-the-art low-cost learning algorithms. The first part will introduce algorithms designed for ANNs. The second part will compare these algorithms and select the ones for further research based on our project goals. The third part will introduce some learning algorithms that are applicable to spiking neural networks and discuss the new issues that arise in this context. Finally, the conclusion for the literature review will be provided.

Chapter 3 will illustrate the process of constructing the required hardware model in software firstly. The first part will begin by examining the employed quantization scheme. The second part will outline the layers and general structure of the network. The third part will conduct a neural architecture search to identify the network structure and quantization setup. Finally, the design of the neural network model in hardware will be finalized.

Chapter 4 will present the process of implementing and verifying the multi-core on-chip learning hardware. First, the design of the main submodules of the hardware implementation and the corresponding behavioral simulations are presented. Subsequently, the design of cores obtained using the integration of these sub-modules and the corresponding behavioral simulations are presented. Then, the design of a multi-core on-chip learning platform that instantiates multiple cores and the verification methodology are presented. Finally, a performance evaluation of the design deployed to the FPGA is performed.

Chapter 5 will summarize the work.

2

Literature review

The main content of the literature review chapter is the comparison and evaluation of state-of-the-art low-cost learning algorithms. The ANN learning algorithms being evaluated will be presented first. These learning algorithms are then compared and an algorithm is selected for subsequent research, depending on the project objectives. In addition to ANNs, we will also present algorithms related to spiking neural networks (SNNs). This is due to the fact that local learning algorithms are also more bio-plausible, and many of neuromorphic hardware using bio-plausible learning algorithms implement SNNs.

2.1. Learning in artificial neural networks

Several different learning schemes will be introduced in this section, but in general, there are two main categories. The first category can solve the spatial credit assignment problem. Most of these algorithms are designed for feedforward fully-connected networks and can solve at least one of the weight symmetry and update locking problems. The second category can solve the temporal credit assignment problem. Most of these algorithms are designed for recurrent neural networks and can help reduce memory complexity and time complexity for online learning compared to BPTT [15].

2.1.1. Spatial learning algorithms

The baseline algorithm in this category is BP. As mentioned in Section 1, this algorithm offers the highest accuracy, but it is neither efficient nor bio-plausible due to the update locking and weight transport problems.

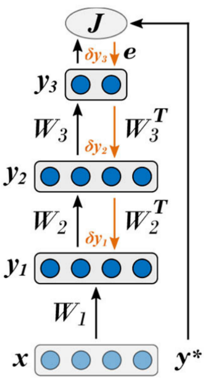

Figure 2.1: Error backpropagation: W is the weight matrix, x is the input, y is the layer output, y^* is the target and δ is the gradient [21].

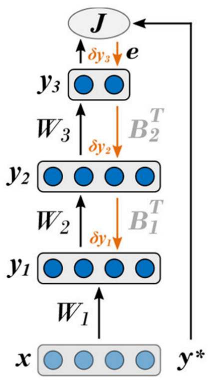

Figure 2.2: Feedback alignment: B is a random and fixed matrix [21].

Feedback alignment

To solve the weight symmetry problem, a learning algorithm called feedback alignment (FA) is introduced in [22]. In the BP algorithm, shown in Figure 2.1, we have the weight symmetry problem as we need to access the transpose of the weight matrix in the learning phase. In FA, they are replaced by fixed random connectivity matrices B (Figure 2.2). In the learning phase, the information of B will flow into W, change W and align it with B^T [22]. This helps to resolve the weight symmetry problem with no computational overhead compared to BP, at the expense of a resonable accuracy degradation on small tasks [21].

Direct feedback alignment

In order to have a more bio-plausible learning algorithm, [23] proposes direct feedback alignment (DFA) based on random feedback and non-reciprocal connections within the network. In Figure 2.3, gradients in the hidden layers are obtained by directly providing the random projection of the output error to all layers. Its accuracy is slightly worse than FA, but more importantly, it shows that learning is still possible for small problems with a simplified feedback path.

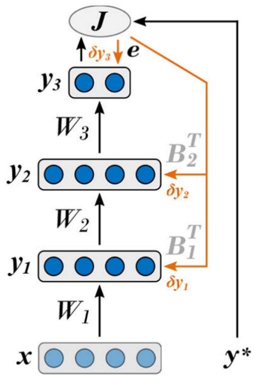

Figure 2.3: Direct feedback alignment: B is a random and fixed matrix [21].

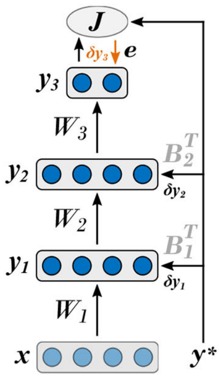

Figure 2.4: Direct random target projection: B is a random and fixed matrix [21].

Direct random target projection

Direct random target projection (DRTP) [21] is developed from DFA, which further simplifies the gradient calculation. As shown in Figure 2.4, except that the last layer will receive gradients from the loss function, hidden layers will receive a local gradient calculated directly from the fixed random matrix and the target vector which is a one-hot encoding of the classification labels.

This gives this algorithm a advantage on hardware in terms of both memory and computational overhead, because it can solve both weight symmetry and update locking problems with only a layer-wise random number selection. Accuracy is degraded compared to DFA, but still acceptable for small problems.

Synthetic gradients

To solve the update locking problem, [11] proposed a concept called decoupled neural interfaces (DNI). The idea is that each layer can get synthetic gradients (SG) from a layer-wise model, which can be trained with error backpropagation, and the use of SG results in DNIs [24].

The network structure is shown in Figure 2.5. The updating rule of this learning algorithm is [11]:

$$W_i \leftarrow W_i - \alpha \hat{\delta}_i \frac{\partial y_i}{\partial W_i}$$

where W is the weight matrix, α is the learning rate, i is the layer index, $\hat{\delta}$ is the synthetic error gradient generated with $\hat{\delta} = M_{i+1}(y_i)$. The layer output y is fed to the module M to compute the SGs in parallel, and global BP is used to adjust the module so that the local SG approximates the true gradient [24].

This algorithm releases weight symmetry and update locking, although the local models still nned to be trained by BP. But at the same time, the layer-wise loss calculation causes extra memory and computational overhead. Another study [26] shows that DNI faces convergence issues and therefore its performance cannot be guaranteed.

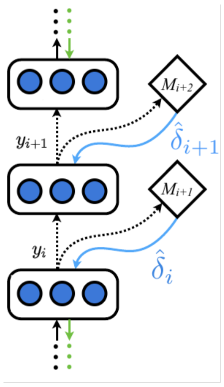

Figure 2.5: Synthetic gradients: M is the synthetic gradients model. Adapted from [11].

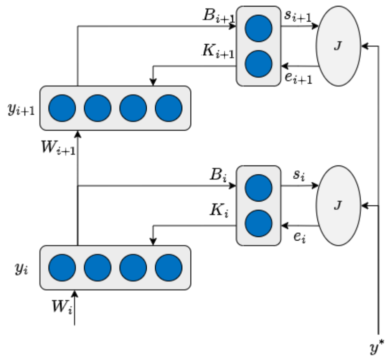

Figure 2.6: Local error learning: M and K are two matrices that can be fixed and random or learnable. Adapted from [25].

Local error learning

Local error learning (LEL) [25] is another local learning algorithm, whose local errors are generated with layer-wise local classifiers.

As shown in Figure 2.6, the class score will be generated with $s^i = B^i y^i$ [25], where i is the layer index, s is the score vector, B is a layer-wise classifier weight matrix and y is the layer output. Then the local error will be generated with the class score and the true label of the input. The matrix K will convert the local errors to gradients. The contents of the K matrix varies depending on the settings of the B matrix. Their relationships are listed as follows:

- B is fixed and random, and $K = B^T$. This is also the default setup for LEL. In this setup, there is still the weight symmetry problem for the local classifier, but the local classifier does not have to learn, so the computational and memory overheads remain low.
- B is fixed and random, but $sign(K) = sign(B^T)$ and $K \neq B^T$. In this setup, K is a sign-concordant [12] weight matrix corresponding to B. It alleviates the problem of weight symmetry for local classifiers and is also more bio-plausible. However, the K matrix introduces additional memory overhead for storage and the network performance is worse than the first setup.
- B is trainable, and $K = B^T$. Here the local classifiers are trainable and hence provide the best performance. However it has a more severe weight symmetry problem, more memory footprint and higher computational overhead than the previous two.

Similarly to SG, this method can also reduce the memory footprint. In terms of extra memory and computational overhead, LEL is more efficient than SG. This is because in the default configuration, the local classifiers do not need to be trained, whereas the layer-wise models in SG still need to be learned through global BP.

Direct difference target propagation

Target propagation (TP) (Figure 2.7) and difference target propagation (DTP) (Figure 2.8) are two learning algorithms that backpropagate targets, rather than errors. TP requires the network to find the inverse of each layer. In this way, each layer can be seen as an autoencoder, where the forward path represents the encoding part and the backward path represents the decoding part. But to accommodate non-invertible networks, DTP is proposed, which uses difference correction to obtain the estimated inverse [29], so that the reconstructed target will be propagated. Direct difference target propagation (DDTP) [29] is built upon DTP and transmits the target from the output layer to each hidden layer through one estimated inverse function, which is the matrix Q in Figure 2.9.

The weight symmetry is resolved by the estimated inverse function, but the network itself is still update-locked. Therefore, this algorithm is not as fast and low-buffering-overhead as layer-decoupled networks. Meanwhile, the learnable backward weight matrix introduces additional memory overhead.

2.1.2. Temporal learning algorithms

Two fundamental learning algorithms to solve the temporal credit assignment are BPTT and RTRL. RTRL is an online learning algorithm, whose high memory and computational complexity are prohibitive

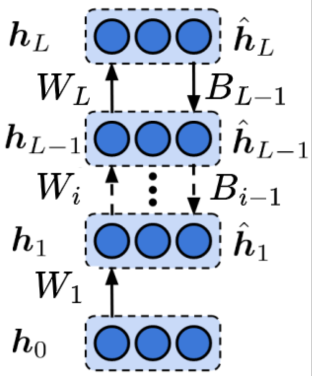

Figure 2.7: Target propagation: B is a learnable matrix, h is the layer output and $\hat{h}$ is the propagated target. Adapted from [27].

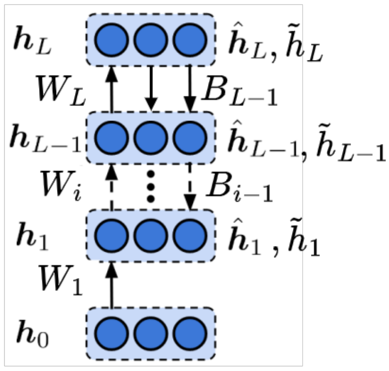

Figure 2.8: Difference target propagation: $\tilde{h}$ is the reconstructed target. Adapted from [28].

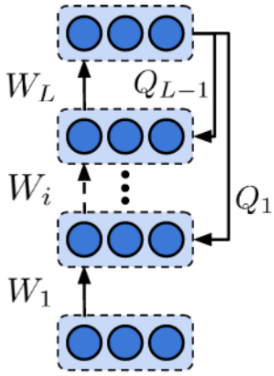

Figure 2.9: Direct difference target propagation: Q is a learnable matrix [29].

[30]. In order to be efficient and bio-plausible, the next few learning algorithms presented allow for online learning, but with improvements compared to RTRL.

E-prop

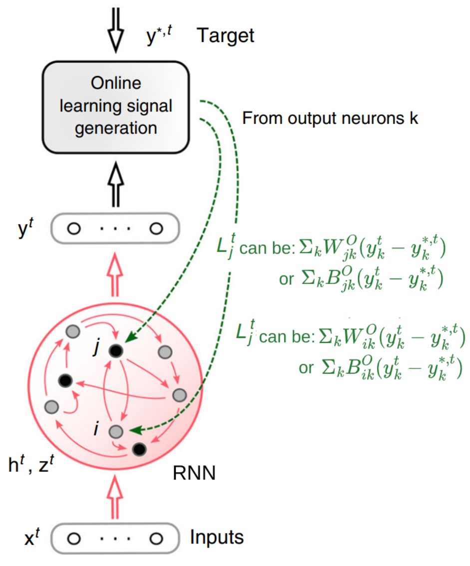

Figure 2.10: e-prop: L_j^t is the learning signal, W_{jk}^O is the output layer weight matrix, B_{jk}^O is a fixed and random matrix [30].

The learning rule and the loss calculation of e-prop for a recurrent network (Figure 2.10) are [30]:

$$W_{ji}^R \leftarrow W_{ji}^R - \eta \sum_t (L_j^t e_{ji}^t)$$

$$L_j^t = \sum_k W_{jk}^O (y_k^t - y_k^{*,t})$$

where t is the time step, W_{ji}^R is the synaptic weight from neuron i to neuron j in the recurrent layer, L_j^t is the top-down learning signal, k is the output neuron index, e_{ji}^t is the eligibility trace, $y_k^{*,t}$ is the target and the W_{jk}^O is a weight matrix from output layer.

Eligibility traces and learning signals are two concepts that make this algorithm to be more bio-plausible. Eligibility traces show that synapses maintain traces of recent activity, which can induce synaptic plasticity if it is closely followed by a top-down learning signal [31, 32]. And top-down learning signals can be found in the brain under different forms which can be supported by neurotransmitters such as dopamine and acetylcholine [33, 34, 35].

The eligibility trace can be computed fully in a forward manner, while the learning signal is approximated and available locally in time. The learning signal in e-prop is also designed for the situation where every neuron is connected to the output, shown in Figure 2.10, which also means that this algorithm is only suitable for single-layer recurrent neural networks.

The local-in-time learning signal makes the memory and computational complexity quite low. In symmetric e-prop, the W_{jk}^{O} is the transpose of the weight matrix for the output layer for higher accuracy. In a more bio-plausible random e-prop, the W_{jk}^{O} will be replaced by a fixed random matrix B_{jk}^{O}, and the weight symmetry problem can be solved.

Online spatio-temporal learning

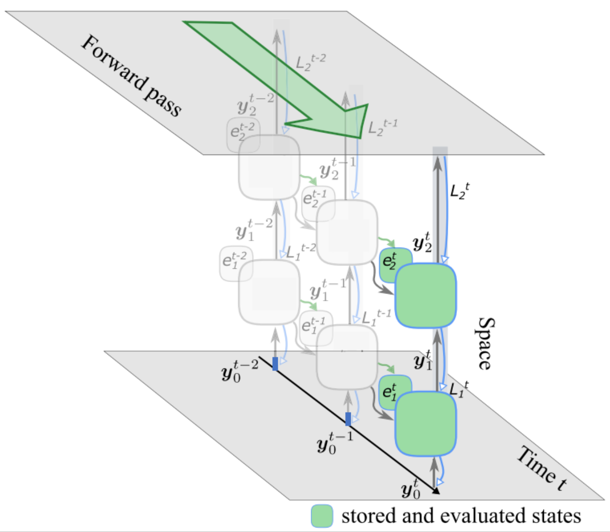

Figure 2.11: Online spatio-temporal learning: e_l^t is the eligibility trace and L_l^t is the learning signal [15].

Online spatio-temporal learning (OSTL) can be seen as a version of e-prop generalized to multilayer learning. The gradient calculation used by OSTL is[15]:

$$\frac{\mathrm{d}J}{\mathrm{d}W_l} = \sum_t (L_l^t e_l^{t,W_l} + \mathbf{R})$$

where W_l is the weight matrix, l is the layer number, e_l^{t,W_l} is the eligibility trace, L_l^t is the learning signal and **R** is a residual term that contains non-local spatialtemporal information for different layers and time steps, which is usually neglected. In Figure 2.11, each layer has an eligibility trace matrix for each synapse and the learning signals will propagate from the output to the input.

Forward-propagation-through-time

The loss function of forward-propagation-through-time (FPTT) is [37]:

$$l(W) = \underbrace{l_t(W)}_{intermediate\ loss} + \underbrace{\frac{\alpha}{2}||W - \bar{W}_t - \frac{1}{2\alpha}\nabla l_{t-1}(W_t)||^2}_{regularizer}$$

where l_t is the intermediate loss of step t, α is a hyper-parameter used to set the constraint of the regularizer, W is the weight matrix and $\bar{W}$ is the running average of previous W matrices, which can be calculated by: $\bar{W}_{t+1} = \frac{1}{2}(\bar{W}_t + W_{t+1}) - \frac{1}{2\alpha}\nabla l_t(W_{t+1})$. To minimize the loss on existing data, the regularizer term $\frac{\alpha}{2}||W - \bar{W}_t - \frac{1}{2\alpha}\nabla l_{t-1}(W_t)||^2$ is introduced. The dynamic loss function l can generate loss according to the current intermediate loss and information from the past. By recording the mean value of past weights, the regularizer can mitigate the impact of the variance of the current input on the final

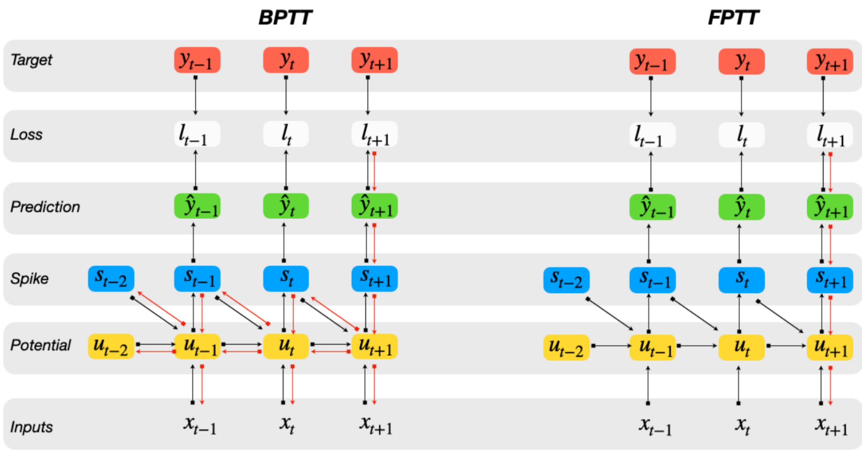

Figure 2.12: Backpropagation through time (left) and Forward-propagation through time (right) [36].

error and accelerate the learning speed. Just as Figure 2.12 shows, no gradients will backpropagate through the temporal direction, and the $\bar{W}$ is the information propagated forward over time.

With this design, although we need to keep an extra matrix for the average of past weights, the memory complexity has still been reduced as an online learning algorithm. However, this also increases computational overhead, since there are more parameter updates in each step.

2.2. Comparison

The comparison of algorithms that can solve spatial credit assignment problems is shown in Table 2.1. For these algorithms, we want to focus on their ability to solve the weight symmetry and update locking issues, and the resulting additional overhead in memory and computation. Due to DDTP having high computational overhead, and SG suffering from a non-convergence problem, we exclude them. FA and DFA are also not selected because they are update-locked. Solving the update locking problem helps reduce layer-wise overheads and will also be beneficial to reduce inter-core communication.

Finally, only LEL and DRTP are left as they can solve both weight symmetry and update locking issues. In terms of test results, LEL can provide better performance. If only sign-concordant layer-wise matrices are applied, the accuracy will be 77.9% on CIFAR-10 [38] with a convolutional neural network (CNN) that has 2 hidden convolutional layers and 1 hidden fully-connected layer. On the other hand, the hardware implementation of DRTP will be much simpler, but it can only achieve 67.35% accuracy on CIFAR-10 with a CNN that has 1 hidden convolutional layer and 2 fully-connected layers. DRTP is more power efficient, but its performance is limited for convolutional layers due to the bottleneck effect [5, 21]. We decided to choose LEL because it has better scalability.

The comparison of algorithms that can solve the temporal credit assignment problem is shown in Table 2.2. Among them, we think e-prop is a better choice: compared to OSTL and FPTT, the accuracy of e-prop is comparable, but with less memory and computational overhead.

Although e-prop has the disadvantage that it can only be applied to single-layer networks, we can avoid this problem by combining it with LEL, and this also allows LEL to gain the ability to solve the temporal credit assignment problem. Figure 2.13 illustrates a simple design where we can use the output of RNN_0 as the input of RNN_1, resulting in a multi-layer architecture. Furthermore, the learning signal in this design is applied only to the current RNN, thereby ensuring that the error generation remains local. In each layer, the learning algorithm is still based on e-prop, while the entire learning framework is more similar to LEL.

Table 2.1: Comparison table for spatial credit assignment algorithms

	Backpropagation of error						
Categories	BP	FA [22]	DFA [23]	DRTP [21]	SG [11]	LEL [25]	DDTP [29]
Weight symmetry resolved	✗	✓	✓	✓	✓	✓	✓
Update locking resolved	✗	✗	✗	✓	✓	✓	✗
Memory overhead	-	Additional layer-wise fixed random matrix	Additional layer-wise fixed random matrix	Additional layer-wise fixed random matrix	Additional layer-wise error module	Additional layer-wise classifier	Additional layer-wise learnable matrix
Computational overhead	-	-	-	-	Additional layer-wise loss calculation and matrix multiplications.	Additional layer-wise loss calculation and two low-dimensional matrix multiplications	Overhead in feedback weight training[39]
Scalability	-	10-layer CNN on CIFAR-10 has 51.3% accuracy[25]	Transformer on WikiText-103 has 52.0 validation perplexity[40]	5-layer CNN on CIFAR-10 has 32.65% error	ResNet-8 with 4 decoupled points on CIFAR-10 has about 20% accuracy[26]	10-layer CNN with trainable layer classifier on CIFAR-10 has 82.9% accuracy	LeNet on CIFAR-10 has 76.33% accuracy

Table 2.2: Comparison table for temporal credit assignment algorithms

Categories	BPTT	RTRL[16]	E-prop[30]	OSTL[15]	FPTT[37]
Method for temporal credit assignment problem[1]	Backward	Forward	Forward	Forward	Forward
Memory complexity[2]	Tn	n^3	n^2	n^2	N/A
Time complexity[2]	Tn^2	n^4	n^2	n^2	N/A
Scalability	-	-	3-layer LSTM on TIMIT has 21.24% error rate[15]	3-layer LSTM on TIMIT has 19.7% error rate	Single layer LSTM on Sequential CIFAR-10 has 71.03% accuracy

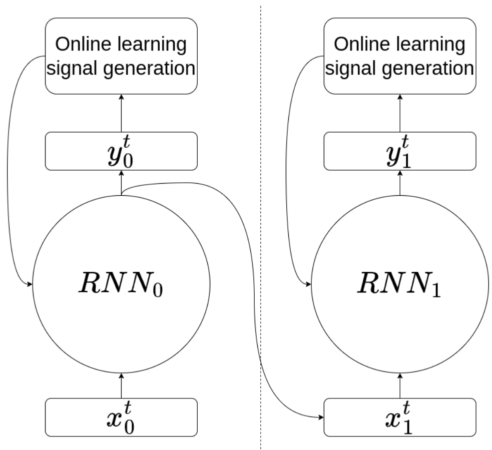

Figure 2.13: Combination of LEL and e-prop

2.3. Learning with spikes

The algorithms just surveyed, are not only more efficient but also more bio-plausible, so they are typically implemented in neuromorphic hardware, based on SNNs. However, using a spiking representation introduces additional challenges:

Non-differentiability: There are many different neuron models [41, 42, 43, 44, 45]. In this section, we will discuss the leaky integrate-and-fire (LIF) [41] and the spike response model (SRM) [43]. SRM provides a general model for the neuron state, can be expressed as [46]:

$$u(t) = \underbrace{\sum\nolimits_f \eta(t - t^{(f)})}_{refactory\ response} + \underbrace{\int_0^\infty \kappa(t - \hat{t}, s) I^{ext}(t - s) ds}_{synaptic\ response} + u_{rest} \tag{2.1}$$

where u is the membrane potential, t is the time step, $t^{(f)}$ is the firing time step, $\eta(t)$ is the refactory kernel, s is the start time step of a current pulse, $\kappa(t, s)$ is the spike response kernel, $I^{ext}(t)$ is the input current and u_{rest} is the rest membrane potential. Here, the neuron state is equivalent to the membrane

[1]Definitions of the forward and backward methods come from [14]
[2]Derived for single-layer network with n neurons, T is the length of sequence, data from [15]

potential. The refactory core defines the behavior after output a pulse and the spike response core defines the behavior after receiving an input pulse.The LIF is a special case of the SRM with the Heaviside step function as the synaptic response kernel [43]:

$$\kappa(t,s) = \frac{\Theta(s)}{\tau} exp(-\frac{s}{\tau})\Theta(t-s) \tag{2.2}$$

where $\Theta(t)$ is the Heaviside step function, τ is the time constant. With η is set to generate burst, an example voltage respone to a step current is shown in Figure 2.14. The challenge posed by the use of spiking neurons is their non-differentiability. For example, in a LIF neuron, the derivative of the Heaviside step function is required, but its value is non-zero only when the membrane potential is equal to the firing threshold. This blocks gradients propagation, making the network unable to learn effectively [14]. Therefore, there are a range of methods known as *surrogate gradients* to evade the non-linearity problem, which are able to approximate the true gradient while retaining the properties for numerical optimization.

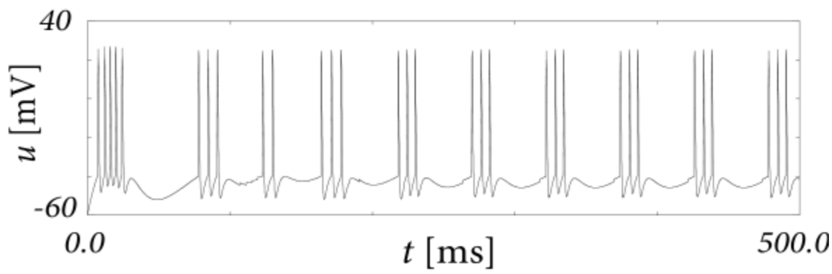

Figure 2.14: The voltage response to a step current [46].

Spike coding: [47] demonstrates several coding schemes for spikes. Among them, the rate code (Figure 2.15(a)) is commonly used to map from ANNs to SNNs [5]. This form of encoding is simple but less energy-efficient, as each spike only contains a marginal amount of information [47]. Other encoding schemes such as rank code (Figure 2.15(b)), time-to-first-spike (TTFS) code (Figure 2.15(c)), etc. can improve the efficiency of information transmission, but encoding complexity increases as there is no straightforward ANN-SNN mapping.

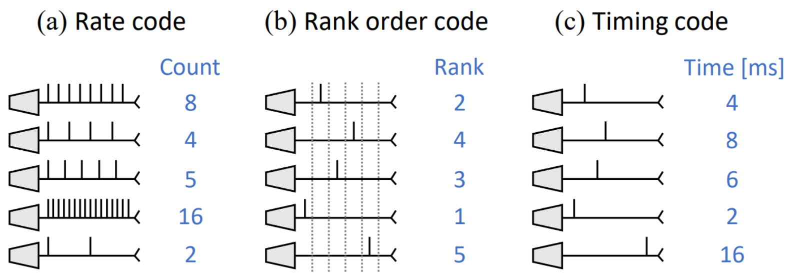

Figure 2.15: Coding examples [5].

The algorithms discussed earlier can be modified to train SNNs with rate-based coding, and some studies demonstrate the effects of applying them to SNNs, such as [48, 36]. To solve the above problem, the following algorithms are designed specifically for SNNs.

SLAYER

Spike layer error reassignment (SLAYER) [49] provides a general backpropagation mechanism that can solve the neuron non-differentiability issue. Compared to BP, the main change it makes for SNN training is to use the probability density function (PDF) of spiking state change for gradient generation. The PDF obtained from statistics after the introduction of the random perturbations to membrane potential, which shows the probability of a neuron producing a spiking state transition, i.e., changing from a spiking state to a non-spiking state, or a non-spiking state to a spiking state, as the difference between u and ϑ changes.

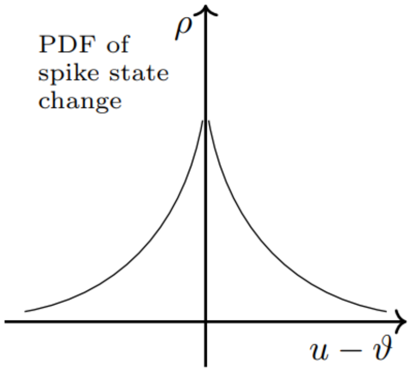

Figure 2.16: SLAYER: u is the membrane potential, ϑ is the firing threshold[49]

The PDF in Figure 2.16 helps to characterize threshold triggering while introducing bio-plausible variability [50], so an exponential function is used to calculate the surrogate gradients. This algorithm is based on BP and needs other techniques in section 2.1 for computation and memory efficiency.

DECOLLE

DECOLLE [51] is more like the LEL with fixed local classifiers, shown in Figure 2.17, and the layer-wise true targets are replaced by predefined layer-wise pseudo-targets. It also uses surrogate gradients to learn effectively in SNNs.

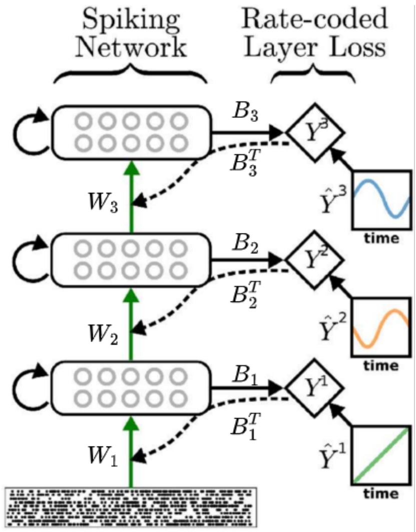

Figure 2.17: DECOLLE: B_i is the layer-wise fixed and random matrix, Y^i is the layer ouput, $\hat{Y^1}$ is the layer-wise pseudo target and surrogate gradients is used to generate the gradient flows in the dashed line [51].

Because of the dynamics of the spiking neuron itself, this algorithm can also solve the temporal credit assignment problem and, because it is derived from LEL, weight symmetry and update locking problems can be solved.

Time-to-first-spike learning

Time-to-first-spike learning [52] is designed for the TTFS coding scheme [53]. This coding scheme will let the most significant feature be input to the network first, see Figure 2.18. Compared to the rate code, TTFS has a significant advantage in information transmission efficiency because of the sparsity of spikes [54]. The authors of [52] propose a learning algorithm based on BP, with a loss function that generates errors in terms of time differences. However, encoding and decoding for TTFS are more complex than those for the rate code [54], which causes a loss of efficiency due to the coding scheme.

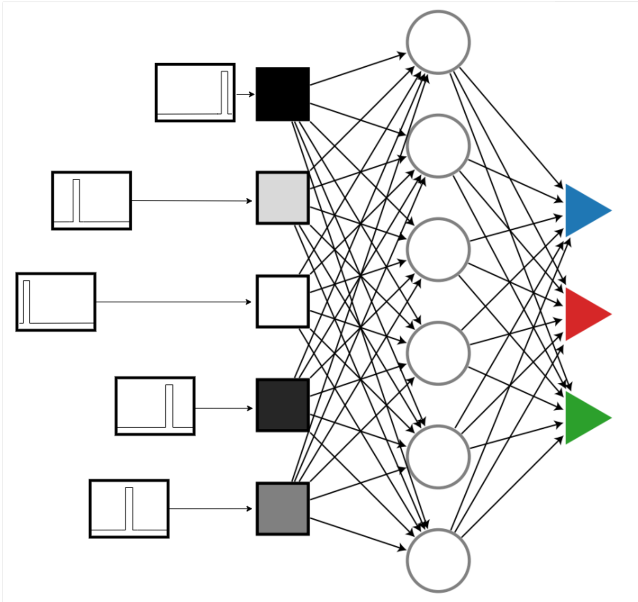

Figure 2.18: Time-to-first-spike learning: Squares are inputs, and the darker the color, the earlier the spike enters the network [52].

2.4. Conclusions

In this chapter, we investigated several state-of-art algorithms that offer low-cost learning schemes compared to BP. In order to be able to solve the problem of spatial and temporal credit assignment at the same time, we proposed a design that combines the use of e-prop and LEL. However, designing such on-chip-learning hardware relying on multiple RNN cores is beyond the scope of this project. Therefore, we prioritize the implementation and validation of the part that is applicable to multicore learning, i.e., the LEL learning algorithm. At the same time, we would like to validate its capability on complex datasets and the possibility of deploying it on edge devices, so we will use a simple CNN design with LEL for CIFAR-10 image classification.

3

PyTorch model design

This chapter explains the construction and determination of the on-chip LEL neural network model by neural architecture search (NAS). To begin, the applicable quantization scheme for the model will be formulated. Next, the basic layer building blocks and structure of the model will be determined according to the task and hardware constraints. Lastly, the final model will be determined via NAS.

3.1. Quantization scheme

FPGAs and ASICs for edge computing have limited logic and memory resources. Thus, floating-point calculations and the storage of high-precision floating-point numbers have high power and area usage costs [55]. As the model will be verified on the FPGA platform, a key modification is the application of a quantization scheme to enable training with fixed-point numbers or integers. However, many popular and mature quantization schemes nowadays are oriented towards inference-only models [56], which we will review first (Section 3.1.1). As this project requires research and design of a quantization scheme specific to low-cost on-device training, training-oriented schemes, together with the custom proposed approach, will be covered in Section 3.1.2 and 3.1.3, respectively.

3.1.1. Basic inference-oriented quantization

The PyTorch [57] machine learning framework supports two kinds of quantization schemes: post-training quantization (PTQ) (Figure 3.1) and quantization-aware training (QAT) (Figure 3.2). PTQ quantizes the 32-bit floating-point (FP32) weights and activations of a model to 8-bit integers (INT8) after training. PTQ provides a simpler way to quantize models and is appropriate when retraining is not desired. QAT, in contrast, applies fake quantization ($fake\_quant$) function to weights and activations during the training process, which can improve inference performance of the quantized model compared to PTQ [58].

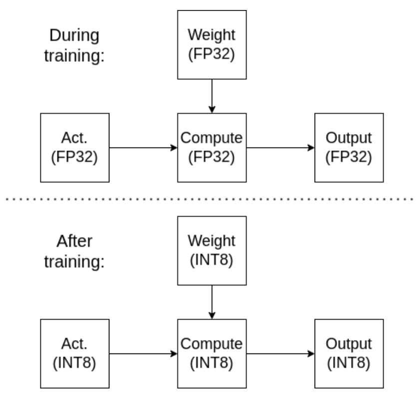

Figure 3.1: Diagram of PTQ training: Act. is the abbreviation for activation. Adapted from [59]

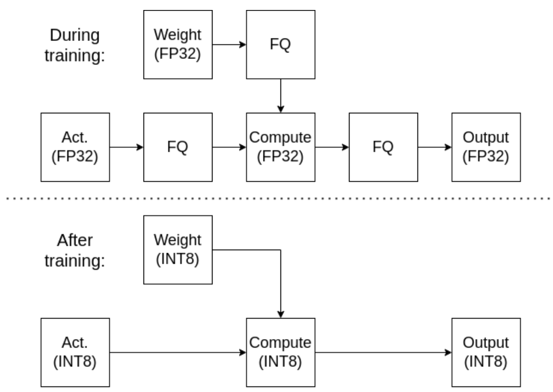

Figure 3.2: Diagram of QAT training: FQ is the $fake\_quant$ function. Adapted from [59]

The diagram of $fake\_quant$ is illustrated in Figure 3.3. During the inference phase (forward path), the input value is multiplied with the quantization scale value n, which can be expressed as $n \sim 2^{k-1}$, where k is the bit resolution, and is fine-tuned with runtime statistics in PyTorch. Following this, a rounding function is applied, and the scale value n is then removed. This process allows the input value to be transformed into a value that can be directly represented using the the chosen INT type (e.g., INT8 for 8-bit quantization). During the backpropagation phase (backward path), since the rounding function is non-differentiable, a surrogate gradient is required for it. Here, the straight-through estimator (STE) [60] is used. It can be simply expressed as $fake\_quant' \approx 1$.

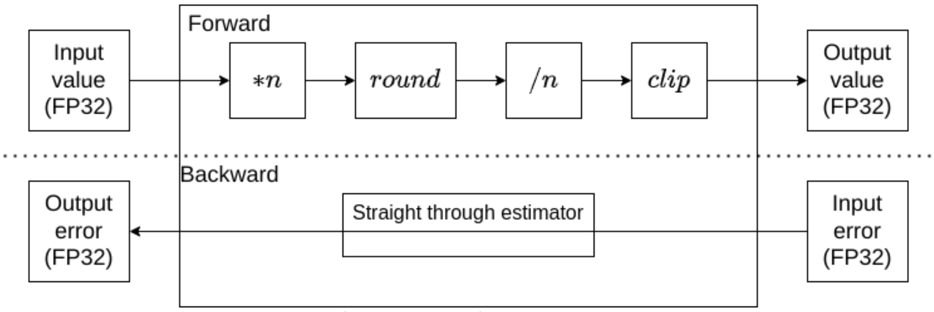

Figure 3.3: Diagram of $fake\_quant$ function: n is the quantization scale

Being applied after the training phase, PTQ is not suitable for on-chip learning. Regarding QAT, although quantization is incorporated into the training process, the error and gradient values remain unquantized to ensure accuracy, which is not adequate for on-chip learning either.

3.1.2. Training-oriented scheme: WAGE

The surveys [56, 61] list numerous quantization schemes, among which WAGE [62] goes beyond inference-only schemes by quantizing not only weights and activations, but also errors, gradients, and weight updates. As this meets our general requirements, we will further investigate it.

Quantization function: The basic quantization function of WAGE is as follows:

$$\begin{gathered}\sigma(k) = 2^{k-1}, k \in \mathbb{N}_+ \\ Q(x,k) = clip\{(\tfrac{1}{\sigma(k)} \cdot round[x \cdot \sigma(k)]), -1 + \tfrac{1}{\sigma(k)}, 1 - \tfrac{1}{\sigma(k)}\}\end{gathered} \tag{3.1}$$

where $1/\sigma$ is the uniform distance between discretized values, k is the bit resolution, x is the input value, Q is the quantized value and $clip$ is the saturation function to clamp the values to the range $[-1 + \frac{1}{\sigma(k)}, 1 - \frac{1}{\sigma(k)}]$. The STE is applied in the backward phase in this function, that is $Q'(x,k) \approx 1$. The quantization function in WAGE is similar to the $fake\_quant$ used in QAT, but here σ is used as a fixed scaling value with respect to bit resolution.

Weight initialization: The weight initialization in WAGE uses a modified Kaiming initialization method [63], which can be formulated as:

$$W \sim U(-L, +L), L = max\{\sqrt{6/n_{in}}, L_{min}\}, L_{min} = \frac{\beta}{\sigma} \tag{3.2}$$

where W is the weight matrix, U is the uniform distribution, L is the limit of the distribution, n_{in} is the input size of the layer and β is a constant larger than 1 to create overlaps between minimum step size σ and L. Figure 3.4 illustrates the case where L_{min} is not used, and the L of the distribution is smaller than $1/\sigma$. For instance, define a uniform distribution with $L < 1/\sigma$ and $1/\sigma = 0.125$. Then, in the case without quantization, sampling from the uniform distribution gives the result in Figure 3.4(a). And with quantization, one can obtain the result in Figure 3.4(b), where all values are 0. Thus the upper and lower bounds of the uniform distribution should be larger than $1/\sigma$.

This initialization method in Equation 3.2 takes into account this problem, and should therefore be used instead of the PyTorch's built-in initialization method.

Scaling factor: In addition to the fixed scaling value σ mentioned at Equation 3.1, an adjustable scaling factor α is introduced in WAGE to transform the mapping range. Depending on the use case, it takes different values.

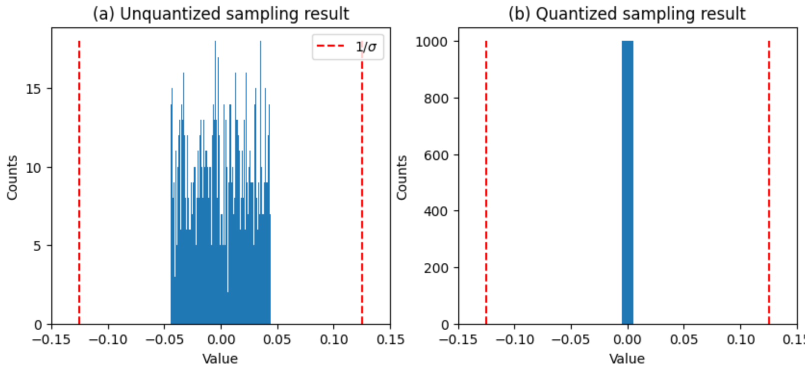

Figure 3.4: Difference in sampling from uniform distribution: Red dashed lines show the position of $-1/\sigma$ and $1/\sigma$. (a) Sampling result from uniform distribution with $L < 1/\sigma$ and values are not quantized. (b) Sampling result from the same uniform distribution and values are quantized.

When it is used to scale the activation, the quantization function can be expressed as:

$$Q_A(a) = Q(a/\alpha_A, k_A) \tag{3.3}$$

where a is the activation, k_A is the bit resolution of the activation. The value of α at this point can be calculated by:

$$Shift(x) = 2^{round(log_2 x)} \tag{3.4}$$

$$\alpha_A = max\{Shift(L_{min}/L), 1\} \tag{3.5}$$

This value can be used to mitigate the amplification problem, i.e. a larger variance of quantized weights generated during weight initialization.

When it is applied to the error, the quantization function for the error can be expressed as:

$$Q_E(e) = Q(e/Shift(max\{|e|\}), k_E) \tag{3.6}$$

where e is the error, k_E is the bit resolution of the error and $|e|$ is the absolute value of error. This method provides low-loss quantization for errors, but to obtain the maximum absolute value we still need to observe the unquantized value, which is problematic for hardware implementation.

Weight update: Finally, for quantizing the gradients g in the weight update phase, authors used the following method:

$$g_s = \eta \cdot g/Shift(max|g|) \tag{3.7}$$

$$\Delta W = Q_G(g_s) = \frac{1}{\sigma(k_G)} \cdot sgn(g_s) \cdot \{\lfloor |g_s| \rfloor + Bernoulli(|g_s| - \lfloor |g_s| \rfloor)\} \tag{3.8}$$

where η is the learning rate, k_G is the bit resolution of the gradient, sgn is the sign function, $\lfloor \rfloor$ is the floor function and $Bernoulli$ is a function that samples from the Bernoulli distribution, in order to perform a stochastic rounding of the gradient.

3.1.3. Design of the quantization scheme

For our quantisation scheme, we use WAGE's design of the quantization function and weight initialization, which are Equation 3.1 and Equation 3.2, as they are feasible on hardware. For the other parts, we will simplify the design.

We will retain the shift-based idea for the scaling factor used in the activation quantization, but we will tune the value of α during NAS for maximum accuracy. We also aim to simplify the design in WAGE for scaling factors used in error quantization. Therefore, in our quantization scheme they will be replaced by tunable hyperparameters. Due to the short single-layer path for backpropagation of error in LEL, the aforementioned amplification effect is less pronounced, so we can remove the scaling factor for layer-to-layer error quantization, and only one scaling factor will be applied at the final loss of local classifier. In gradient quantization, the scaling factor is also removed for simplicity. The design of our quantization scheme can be summarized as follows, based on the above description:

$$Q_W(w) = Q(w, k_W) \tag{3.9}$$
$$Q_A(a) = Q(a/\alpha_A, k_A) \tag{3.10}$$
$$Q_G(g) = Q(g, k_G) \tag{3.11}$$
$$Q_{E,Loss}(e) = Q(Act'(e_{Loss} * \alpha_E), k_E) \tag{3.12}$$
$$Q_E(e) = Q(e, k_E) \tag{3.13}$$

where k_W, k_A, k_G, k_E are the bit resolution for the weights, activations, gradients, and errors, respectively, α_A, α_E, α_E are the scaling factors for activations and errors, Act' is the derivative of the nonlinear activation function of the final layer, $Q_{E,Loss}$ is the quantized error from loss function and Q_E is the quantized error between layers.

This stochastic rounding method is derived from [64] and is thought to avoid local minima and overfitting [65]. However, in [58], it performs worse than the usual rounding scheme. Moreover, in hardware implementations, extra hardware is needed to implement this functionality. Therefore, we will not implement stochastic rounding in hardware. Thus, instead of stochastic rounding, stochastic weight updates [66] are used here to simplify the weight update implementation on hardware. This method can be formulated as follows:

$$\begin{aligned} p_g &= clip\{g_w \cdot lr, -1, +1\} \\ \Delta W &= \tfrac{1}{\sigma(k_W)} \cdot sgn(p_g) \cdot \{Bernoulli(|p_g|)\} \end{aligned} \tag{3.14}$$

where g_w is the gradient with respect to the weight, lr is a shift-based learning rate, p_g is the update probability. This weight update method differs from the conventional method in that the magnitude of weight update is limited to either the minimum step size $\frac{1}{\sigma(k_W)}$ or 0, with the update value taking one of three options: $\{-\frac{1}{\sigma(k_W)}, 0, +\frac{1}{\sigma(k_W)}\}$. The sign of the weight update is determined by the sign of the gradient with respect to weight, but the magnitude of the gradient is now considered as the probability of the weights being updated.

3.2. Network architecture

The project objectives state our aim to construct a network suitable for CIFAR-10 image classification by adjusting the network architecture to match the hardware conditions. This section provides a discussion of the neural network layers that will be (or not be) used in our setup. The following layers and modules required for LEL will be used to construct a neural network architecture that is ready for NAS.

3.2.1. Basic layers and modules

To perform the CIFAR-10 image classification task, it is common to employ convolutional neural networks (CNNs). This is because CNNs can utilize the known spatial structures present in natural images through convolutional layers [67].

Convolutional layer: A key layer of a CNN is the convolutional layer, which performs a convolutional operation on a two-dimensional input. The process of convolution layers can be regarded as a convolution of two two-dimensional matrices, resulting in a new two-dimensional matrix. A simplified expression of this process is demonstrated below (to simplify the implementation, the bias term is omitted):

$$\begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} \\ A_{1,0} & A_{1,1} & A_{1,2} \\ A_{2,0} & A_{2,1} & A_{2,2} \end{bmatrix} * \begin{bmatrix} K_{0,0} & K_{0,1} \\ K_{1,0} & K_{1,1} \end{bmatrix} = \begin{bmatrix} O_{0,0} & O_{0,1} \\ O_{1,0} & O_{1,1} \end{bmatrix} \tag{3.15}$$

where A is the input pixel, K is the convolution kernel weight, O is the output feature map pixel and $*$ is the convolution operator. We can express the calculation of each output of the above as:

$$O_{i,j} = \sum_a \sum_b (A_{i+a,j+b} \cdot K_{a,b}) \tag{3.16}$$

where i and j are the indices of the output feature map, a and b are the indices of the kernel weights and $\cdot$ is the multiplication operator. The equation shows two important properties of convolution s[67]:

- **Locality:** When computing an output, the convolution kernel operates on only a small area of the input image. As a result, the neural network can assign higher priority to the information contained in the local regions of the image.
- **Translation invariance:** Shifting the first two columns of Equation 3.15 one column to the right will move the result originally at position (0,0) to position (0,1) of the output feature map. This enables the neural network to detect specific objects or structures across the input image.

The prior knowledge provided by convolution enables the neural network to understand the relationship between local and global information in an image. This understanding helps the network process the image efficiently and accurately to perform various tasks, including image classification and object recognition. In addition to the above properties, in hardware, the number of parameters (i.e. kernel weights) that need to be stored is small, thus leading to low overhead of memory accesses as well as storage.

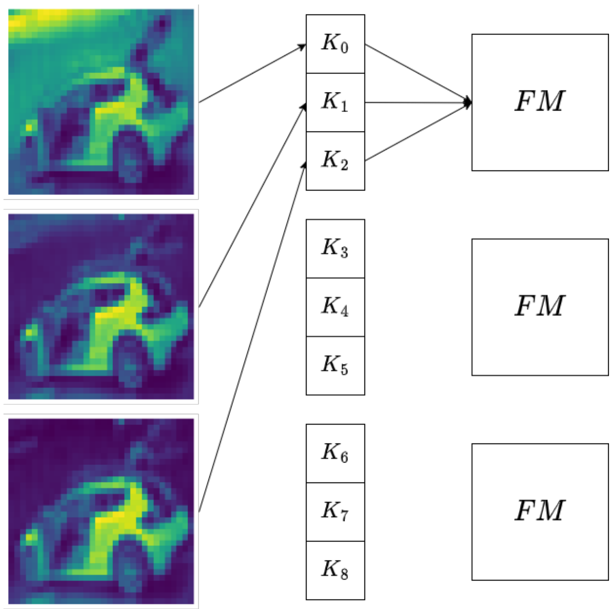

Figure 3.5: Conventional convolution: K is the convolution kernel and FM is the feature map

In PyTorch, the default convolutional layer performs the following operations as shown in Figure 3.5. The three images located on the left side of Figure 3.5 represent the three channels of the input RGB image for the convolutional layer. Each image undergoes convolution with three distinct convolutional kernels, and their resulting outputs are summed and transferred to the feature map. To minimize the required parameter storage and computation overheard on hardware and to simplify memory access logic, instead of this standard convolution method, we will use depthwise convolutions.

As shown in Figure 3.6, the input channel image is directly convolved using a single convolution kernel, and the results are go to independent feature maps. Summation of the results of different input channels is not required. For the same number of output feature map channels, the parameter storage requirements are one-third that of the previous design, and the amount of computation is similarly reduced. This approach eliminates the ability to communicate between input channels within the convolutional layer, thus compromising accuracy in exchange for lower memory and computational overhead.This tradeoff will be analyzed quantitatively during NAS in Section 3.3.

Since the convolutional layer's learning calculation is complex, the convolutional layer will not be trained on hardware. The values of the convolution kernel will either be fixed randomly or pre-trained, both of which will also be analyzed during NAS in Section 3.3

Pooling layer: A pooling layer is usually applied after a convolutional layer to reduce the feature map size, and it serves the two purposes of mitigating convolutional layers sensitivity to location and downsampling spatial representations [67]. Similar to the convolutional layer, it relies on a sliding window. However, unlike the convolutional layer, its window does not contain trainable values; it performs an operation on the input values that can be observed within this window. Max pooling and average

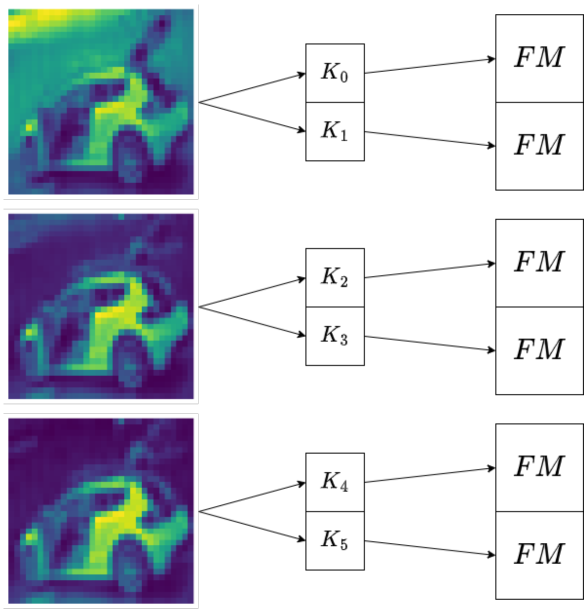

Figure 3.6: Depthwise convolution

pooling layers are commonly used pooling layers. Max pooling (shown in Figure 3.7) finds the maximum value within the window, while average pooling averages the input values within the window. The preferred pooling option will be determined during NAS in Section 3.3.

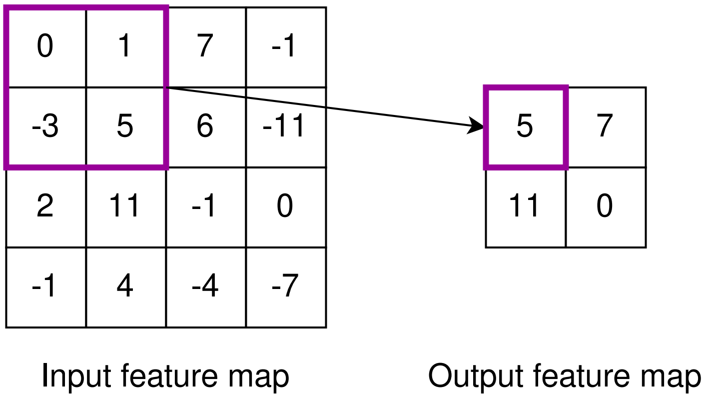

Figure 3.7: Max pooling: The maximum value in the purple box of the input feature map goes to the output feature map.

Fully-connected layer: The fully-connected layer is the most basic layer of a neural network, which can be expressed as the multiplication of two matrices (to simplify the implementation, the bias term is removed):

$$\begin{bmatrix} W_{0,0} & W_{0,1} & W_{0,2} \\ W_{1,0} & W_{1,1} & W_{1,2} \end{bmatrix} \cdot \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} \tag{3.17}$$

where W is the weight, X is the layer input and Y is the layer output. In contrast to the convolutional layer, the number of parameters in a fully-connected layer is directly correlated with the input and output size, resulting in significant hardware overhead due to memory usage. Thus, in this layer, our aim is to reduce the number of weight parameters. Reducing the input and output dimensions of the fully-connected layer will negatively impact performance. Beyond a reduction in the parameter count, reducing the output dimensions will reduce the representation ability of the attached local classifier, while reducing the input dimensions will negatively impact the sizing of the preceding convolutional layer. Therefore, to minimize the parameter count while maintaining the same input and output size, a parallel fully-connected layer structure is introduced.

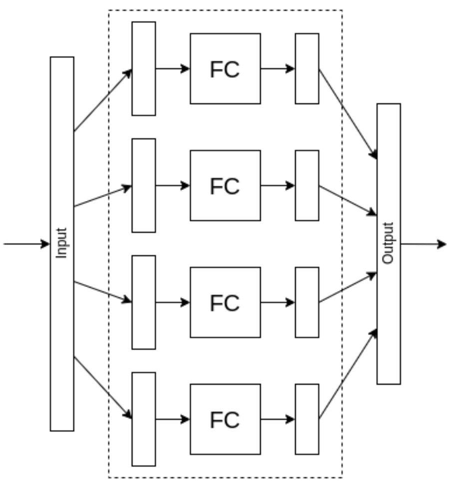

Figure 3.8: Parallel fully-connected layer: FC is the fully-connected layer

The architecture is displayed in the dashed box in Figure 3.8. Here, one input is divided into four smaller pieces, each of which is separately input to a fully-connected layer. The outputs of the four fully-connected layers are then concatenated to form a single output that is fed to the next layer. This approach significantly reduces the number of parameters (by a factor of four in this case) compared to a single fully-connected layer. Additionally, the input and output sizes remain the same. The fully-connected layer architecture will be determined in the NAS section.

Normalization layer: The batch normalization layer [68] is widely used in neural networks for image classification for its ability to effectively improve network performance [67]. The leftmost subplot of Figure 3.9 shows that the batch normalization layer calculates the mean and variance of feature maps within a channel in a batch. The input feature maps are normalized based on the calculated mean and variance. The normalization can be expressed as:

$$y_{fc} = \frac{x - \mu_x}{\sqrt{\sigma_x^2 + \epsilon}} \cdot \gamma + \beta \tag{3.18}$$

where y_{fc} is the output feature map of the fully-connected layer, x is the input feature map, μ_x is the mean of the input feature map, σ_x^2 is the variance of the input feature map, ϵ is a value for numerical stability, γ and β are two learnable parameters that can be assimilated to a weight and a bias. There are

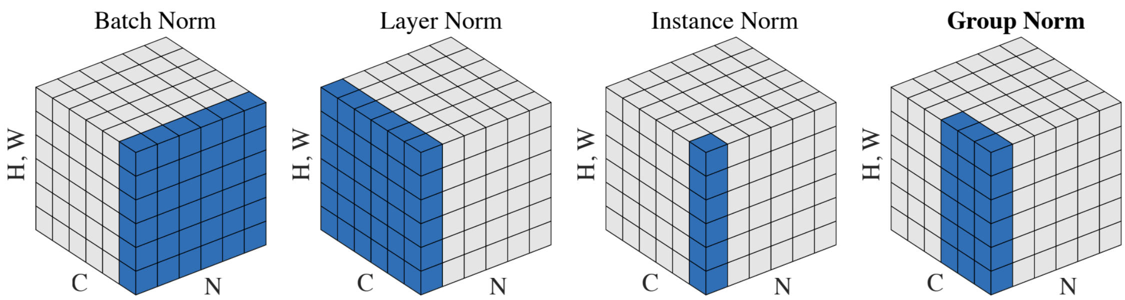

Figure 3.9: Normalization methods: Each subplot is shows a feature map tensor. Blue parts will be used to calculate the mean and variance. H and W is the feature map height and width, C is the number of channels of the feature map and N is the batch size [69].

two significant issues with using normalization layers in our on-chip learning system. The first issue is that on-chip learning will use an online learning method, i.e., stochastic gradient descent (SGD) with batch size N=1. This results in the N-axis in Figure 3.9 not existing at all, making the batch normalization layer meaningless. The second issue is the high computational complexity involved in computing the mean, variance, and gradient of the normalization layer. Thus, we must abandon the use of normalization layers in our model.

Interlayer nonlinear activation function: For nonlinear activation functions placed between different network layers, a popular choice is the rectified linear unit (ReLU) [70]. It can be expressed simply as:

$$y_{ReLU} = max(0, x) \tag{3.19}$$

where y_{ReLU} is the ReLU function output. We will select this interlayer nonlinear activation function because it is easy to implement on hardware and performs well on various tasks.

Output nonlinear activation function and loss function: In this case, the nonlinear activation function is placed after the output layer, so we need to discuss it together with the loss function. For classification tasks, a commonly used activation is softmax. Softmax is expressed as follows:

$$y_{softmax,j} = \frac{exp(x_j)}{\sum_k exp(x_k)} \tag{3.20}$$

where $y_{softmax,j}$ is the softmax function output for output neuron j. The function has the ability to convert each output value from the output neurons into a probability for each class, where the sum of the probabilities is always equal to 1. This is why it is effective for classification tasks. The cross-entropy loss function can be derived by taking the logarithm of the softmax output values and subsequently applying the negative log likelihood loss function. Although the use of the cross-entropy loss function has shown to be effective, it involves costly computations on hardware such as division and exponential operations. To avoid the computations related to logarithms and divisions, the combination of the hard sigmoid function [71] and the mean squared error (MSE) function is employed instead.

Hard sigmoid is a piece-wise linear function with the same upper and lower bounds as the sigmoid function, which can be expressed as:

$$y_{hs} = clip\{\frac{x+L}{2\cdot L}, 0, +L\} = \begin{cases} 0 & if\ x < -L, \\ 1 & if\ x > L, \\ \frac{x+L}{2\cdot L} & otherwise \end{cases} \tag{3.21}$$

where y_{hs} is the hard sigmoid function output and L is the limit. Usually, L is chosen equal to 1. However, because we will use this layer in a quantized model, the network output values are scaled compared to the floating-point values in the unquantized model. Therefore, we consider L as a tunable hyperparameter, and its value will be fine-tuned in NAS. This nonlinear activation function is similar to ReLU and easy to implement in hardware.

However, it also shares a similar problem with ReLU, which is the ”dying ReLU” problem [72]. If the absolute value of the input is greater than the predefined value of L, the derivative of the corresponding function evaluates to 0. Consequently, this input value cannot contribute to the training of the neural network. Because of this, two types of hard sigmoid functions were defined in PyTorch. The first one follows the original definition, and its derivative evaluates to 0 when the absolute value of the input is larger then the limit L. The second type has a surrogate derivative, which evaluates to $\frac{1}{2\cdot L}$. The two variants will be analyzed during NAS in Section 3.3.

The MSE function and its partial derivative to one ouput of the last nonlinear activation function can be expressed as:

$$loss_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i^*)^2 \tag{3.22}$$

$$\frac{\partial loss_{MSE}}{\partial y_i} = \frac{2}{n}(y_i - y_i^*) \tag{3.23}$$

where n is the number of class, y is the ouput of the last nonlinear activation function, y^* is the target label. Calculating the derivative of the MSE does not require any exponential operations. Additionally, the division involved in the calculation has a constant value that can be merged into the scaling factor of the loss required for quantization. Whether to include this division in the hardware design or not depends on the value of the scaling factor, which will preferably be considered to power-of-2 values in order to use shift operations.

Optimization algorithm: Minibatch stochastic gradient descent (SGD) is one of the most widely used optimization algorithms. Compared to gradient descent and online (i.e. batch-size-1) SGD, minibatch SGD is more efficient [67]. In our on-chip learning design, the batch size is limited to one, which restricts the algorithm choice to online SGD, resulting in simpler hardware design. However, when using computers or computing clusters, a batch size of one will cause significantly longer training times, as computation cannot be parallelized anymore. Therefore, in our NAS, we continue to use minibatch SGD and only switch to online SGD for verification during the hardware implementation phase. Although optimization algorithms like RMSProp [73] and Adam [74] enhance training efficiency, they both require logged parameter history and additional computation, which makes them impractical for hardware implementations.

Local classifier: The local classifier is a module that LEL requires and functions as a fully-connected layer with a loss function. We do not plan to use the parallel fully-connected layer for it because the fully-connected layer here has only 10 outputs in total, corresponding to the number of CIFAR-10 classes. As discussed in Section 2.1.1, the use of a trainable local classifier has resulted in improved performance, however, it does not address the weight symmetry issue. So, fixed and random weights will be used for this fully-connected layer in hardware.

3.2.2. General network structure

The general CNN model can be constructed by incorporating the necessary layers and modules. Figure 3.10 shows red and green layers which are specific designs required for LEL and BP, respectively. This implies that they are mutually exclusive, hence only one of them can be present in a network. Later in the test, we will compare the performance difference between LEL and BP using this setup. The layers shown in the figure are placed within separate boxes and these boxes serve as the fundamental unit for modulation of network depth. The boxes containing convolutional and pooling layers are named Convolution (Conv) blocks, while those including fully-connected layers are designated as fully-connected (FC) blocks. This method enables easy computation of the number of cores and their tasks during hardware design. The solid box represents mandatory components within the network, while the dashed box can be adjusted based on NAS results. Being a CNN, the convolution layer and pooling layer in Conv-B0, as well as the fully-connected layer in FC-B0 are compulsory.

3.3. Neural architecture search (NAS)

This section details the process of performing NAS to configure the final the neural network architecture. The network structure will be established by simple tests to determine its depth and the layer configuration shown in Figure 3.10. Subsequently, the designed quantization scheme will be applied to the network and, by sweeping, the quantization resolution and related hyperparameters will be determined.

3.3.1. Number of fully-connected blocks

It is shown in [25] that applying LEL imposes an upper limit on the number of layers in the network. Exceeding this limit will not result in performance improvements using LEL. Thus, establishing the upper limit of the number of layers is a crucial aspect of our design.

Experiment setup: The network configuration and setup are shown in Table 3.1. To evaluate the upper limit of the model's performance, the local classifiers in the model will have trainable weights, their outputs will be passed to the cross-entropy loss function, and the Nesterov momentum will be used. The results will be the average of the last three epochs of each of the three trials.

Results: Table 3.2 lists the training and test set performance of each network block. The first FC block substantially improves performance (+2.7%) while the second and third FC blocks have no significant impact on performance (+0.01% and -0.06%, respectively). In order to be able to validate multi-core training in hardware as a proof of concept, the design with two FC blocks is nevertheless chosen here.

3.3.2. Fully-connected layer setup

In the section 3.2, we explained the reasons for the parallel fully-connected layer design, and here we compare it with the conventional FC design to choose the specific implementation in the FC block.

Experiment setup: Table 3.3 shows the two configurations used for this experiment. Both configurations have an identical number of weights and thus an identical memory cost. This requires adapting

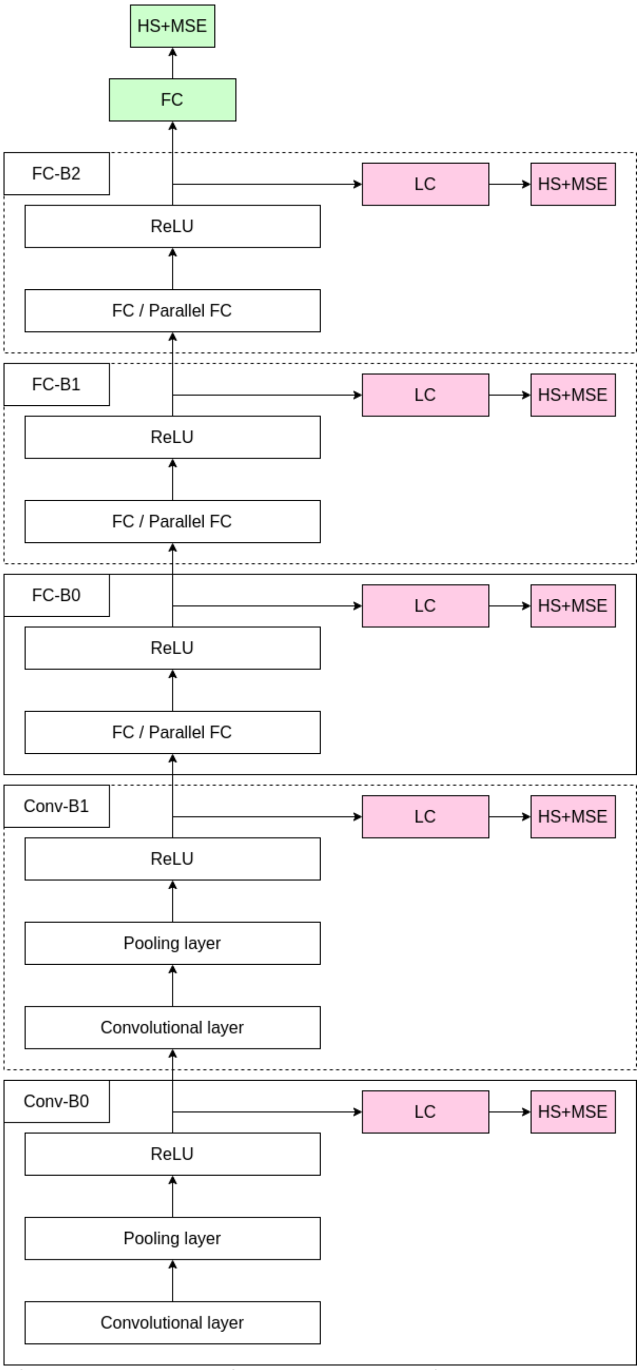

Figure 3.10: General CNN block diagram: HS is the hard sigmoid function. Red and green layers are specific designs required for LEL and BP. Solid boxes represent mandatory components within the network, the dashed boxes represent adjustable components.

the hidden layer size and thus adapting the number of output features from the Conv-B1 block.

Results: In Table 3.4, we can see that accuracies from the single FC layer setup suffer from the reduced hidden layer size. This phenomenon may be due to the degraded performance of the Conv-B1 through a smaller number of output features as well as smaller local classifiers. Using a larger hidden size can lead to better performance for the same number of weights, so our model will use parallel fully-connected layers.

Table 3.1: Setup for the number of FC blocks experiment, inspired from [25]

Configured items	Setup
Network depth	5 (2 Conv blocks, 3 FC blocks)
Hidden size between FC blocks	480
Convolutional blocks	Trainable kernels
Local classifiers	Trainable weights
Loss function	Cross-entropy loss
Optimizer	SGD with Nesterov momentum of 0.9
Batch size	100
Number of training epochs	40
Number of training trials	3

Table 3.2: Results for the number of FC blocks experiment, shown as the mean and standard deviation of accuracy over 3 trials and 3 epochs

	Conv-B0	Conv-B1	FC-B0	FC-B1	FC-B2
Train set accuracy (%)	97.93 ± 0.11	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
Test set accuracy (%)	67.03 ± 0.16	78.04 ± 0.27	80.73 ± 0.19	80.74 ± 0.18	80.69 ± 0.24

3.3.3. Convolutional blocks setup

Now that the FC blocks have been determined, we can turn our attention towards identifying a suitable setup for the Conv blocks in the hardware. As Conv blocks do not need to be trained in hardware, our focus will be on the following three aspects:

- **Computational complexity:** Different convolutional kernel size as well as step size, and different hidden layer sizes affect the amount of computation, which can also be called as computational complexity. High computational complexity leads to high power consumption as well as computation time, so in experiments, configurations should be chosen that have the lowest possible computational complexity without significant performance degradation, by counting the number of multiply–accumulate (MAC) operations.
- **Input hidden size of the first FC block:** The hidden size is directly related to the number of weights in the FC blocks, and thus affects the energy consumption and area overhead of the model on the hardware. Therefore we need to focus on the size of the feature map output by the last Conv block, which determines the input hidden size of the first FC block.
- **Implementation complexity:** This project mainly validates the LEL learning algorithm, and the

Table 3.3: Setups for the FC layer setup experiment. Other parameters are as per Table 3.1

Configured items	Setup for parallel FC layers (PARA)	Setup for single FC layer (SINGLE)
Network depth	4 (2 Conv blocks, 2 FC blocks)	4 (2 Conv blocks, 2 FC blocks)
FC layer setup	4 parallel fully-connected layers	Single fully-connected layer
Hidden size between FC blocks	480	240
Number of weights	57.6k	57.6k

Table 3.4: Results for the FC layer setup experiment, shown as the mean and standard deviation of accuracy over 3 trials and 3 epochs

	Conv-B0	Conv-B1	FC-B0	FC-B1
Train set accuracy (PARA) (%)	97.87 ± 0.10	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
Test set accuracy (PARA) (%)	66.91 ± 0.19	77.84 ± 0.23	80.1 ± 0.22	80.10 ± 0.19
Train set accuracy (SINGLE) (%)	97.85 ± 0.14	99.96 ± 0.01	100.0 ± 0.0	100.0 ± 0.0
Test set accuracy (SINGLE) (%)	66.96 ± 0.21	76.31 ± 0.34	79.79 ± 0.24	79.69 ± 0.21

Conv blocks will not be trained in hardware. Therefore the hardware implementation of the Conv blocks should be kept simple.

The aim of the experiment is to identify various configurations of convolutional and pooling layers that meet these three aspects and deliver acceptable performance.

Experiment setup: Table 3.5 shows the basic configuration used for this experiment. The variable parts in the Conv blocks are shown in Table 3.6.

Table 3.5: Setups for the Convolutional blocks setup experiment. Other parameters are as per Table 3.3

Configured items	Setup
Network depth	1 or 2 Conv blocks with 2 FC blocks
Convolutional blocks	Configurable as per Table 3.6
Number of training epochs	100
Number of training trials	1

Table 3.6: Configurable options in convolutional blocks

Variable options
Kernel size of convolutional and pooling layers
Kernel stride of convolutional and pooling layers
Number of output channels of convolutional layers
Depthwise convolution mode of convolutional layers
Type of pooling layers: max pooling or average pooling
Number of Conv blocks

In these experiments, the performance of the model is based solely on the output of the last FC block and is averaged over the last 3 epochs. The network's rough count of MAC operations and hidden size, calculated by torchinfo [75], are recorded and utilized to determine the most efficient configuration.

Results: Table 3.7 lists the configurations tested, and Figure 3.11 shows the relationship between performance and computational complexity (number of MAC operations) and hidden size corresponding to each configuration. In terms of computational complexity and hidden size, Setup4 is the best choice. However, we can find that it uses two Conv blocks. In comparison, Setup3 has a simpler design with only one convolutional layer and has a small performance gap (-0.53%), so it will be chosen for hardware implementation.

We have finalized the CNN architecture and illustrated it in Figure 3.12. The dashed boxes displayed in the figure illustrate the size of the data stream. As convolutional layers are not trainable on hardware, there is no local classifier in the Conv block.

Table 3.7: Shown experiment setups for Conv blocks setup: "[]" lists configurations in different Conv blocks

	Setup1	Setup2	Setup3	Setup4	Setup5	Setup6
Number of Conv blocks	1	1	1	2	2	1
Kernel size of convolutional layer(s)	5	3	3	[3,3]	[5,5]	3
Kernel stride of convolutional layer(s)	1	1	1	[1,1]	[1,1]	1
Number of output channels of convolutional layer(s)	12	12	24	[12,36]	[9,36]	24
Depthwise convolution mode	✓	✓	✓	✓	✓	✗
Kernel size of pooling layer(s)	2	3	3	[2,3]	[3,3]	3
Kernel stride of pooling layer(s)	2	2	3	[2,2]	[2,2]	3
Type of pooling	Max	Max	Max	Max	Max	Max
Test set accuracy (%)	65.01	66.06	68.55	69.08	70.55	69.92

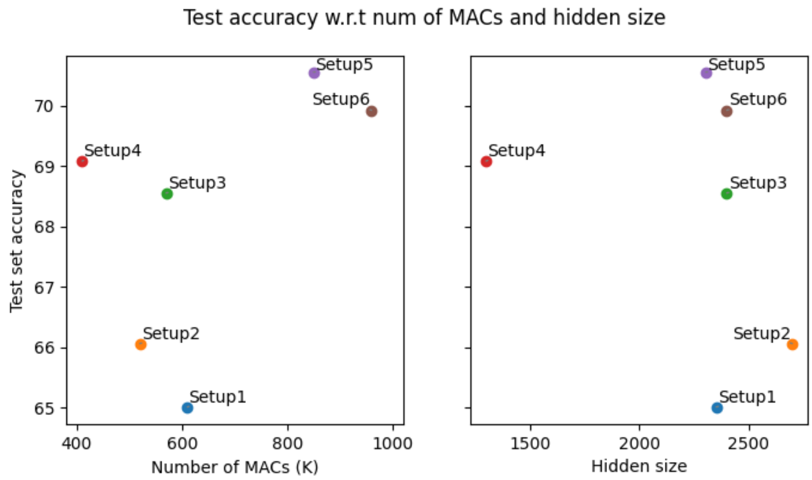

Figure 3.11: Results for Conv blocks experiment

3.3.4. Quantized network sweep

After acquiring a suitable network, we can proceed to applying the quantization scheme to the network and sweep to find the best combination of hyperparameters.

The diagram in Figure 3.13 illustrates the operations carried out during inference in the quantized untrainable Conv block, and the bit resolution of all the data streams in this block. The weights are written to memory with the quantized values during initialization, and are subsequently read out without the need to apply the quantization function again.

Figures 3.14 and 3.15 illustrate the operations carried out in the quantized FC block during inference and training, as well as the bit resolution of all the data streams within the block. As the data flow paths and operations in FC block 1 are consistent with those of FC block 0, its block diagram is omitted in Figure 3.14. The weights in the FC layers are also stored in a quantized format, which is preserved

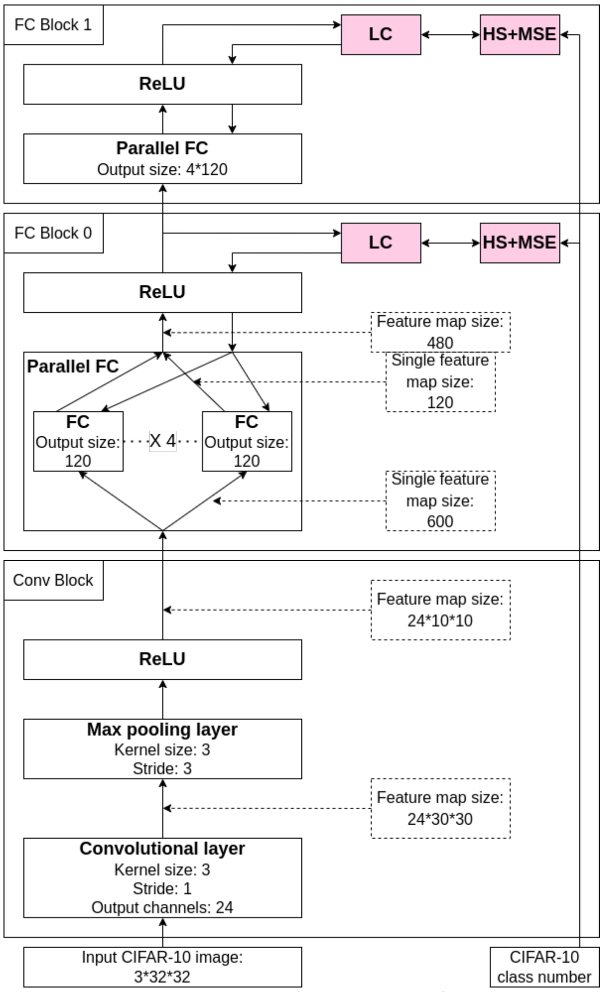

Figure 3.12: Finalized CNN architecture: The parallel FC structure in FC block 1 is omitted because it is the same as in FC block 0.

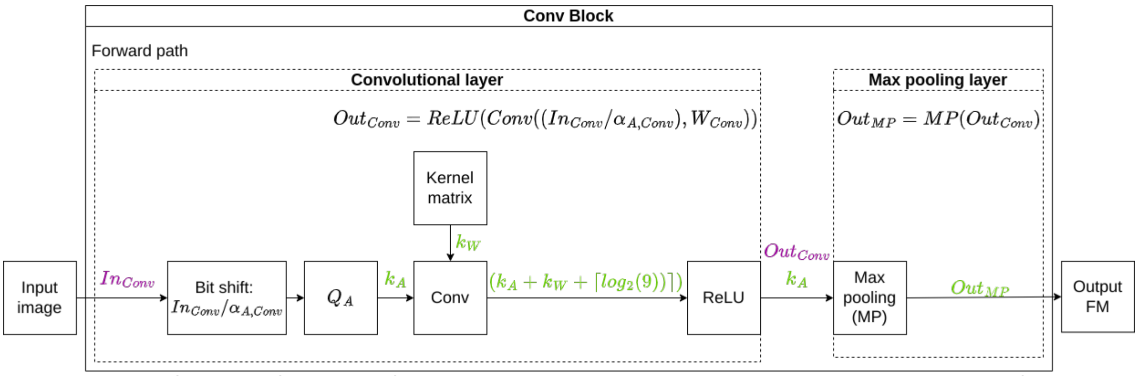

Figure 3.13: Quantized Conv block (forward path): In and Out are the input and output data streams of current dashed box. Green texts and purple texts show the bit width and contents of the data streams, respectively.

during the backward pass as the stochastic update module ensures that the minimum quantized step is used to update the weights.

We conduct a parameter sweep to determine the optimal training hyperparameters for different weight resolutions. By default, 8 bits are utilized for the resolution of activation, gradient, and error. Bit-width sweeps for activations, gradients and errors are outside the scope of this thesis.

Experiment setup: Table 3.8 shows the four configurations used for this experiment. Each config-

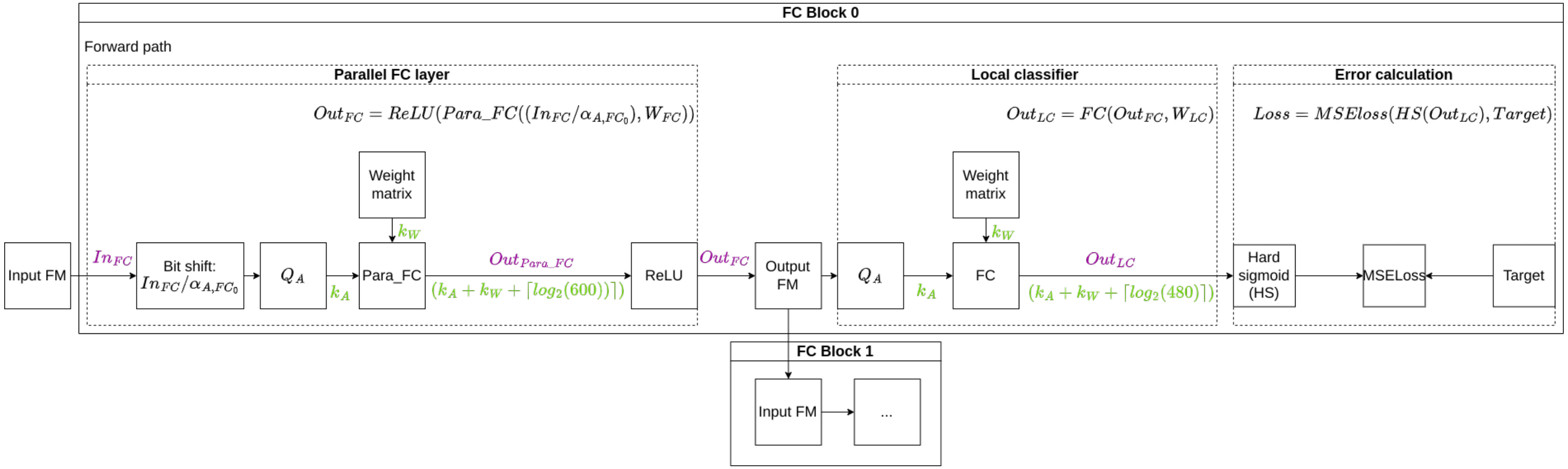

Figure 3.14: Quantized FC block (forward path): Green texts and purple texts show the bit width and contents of the data streams, respectively.

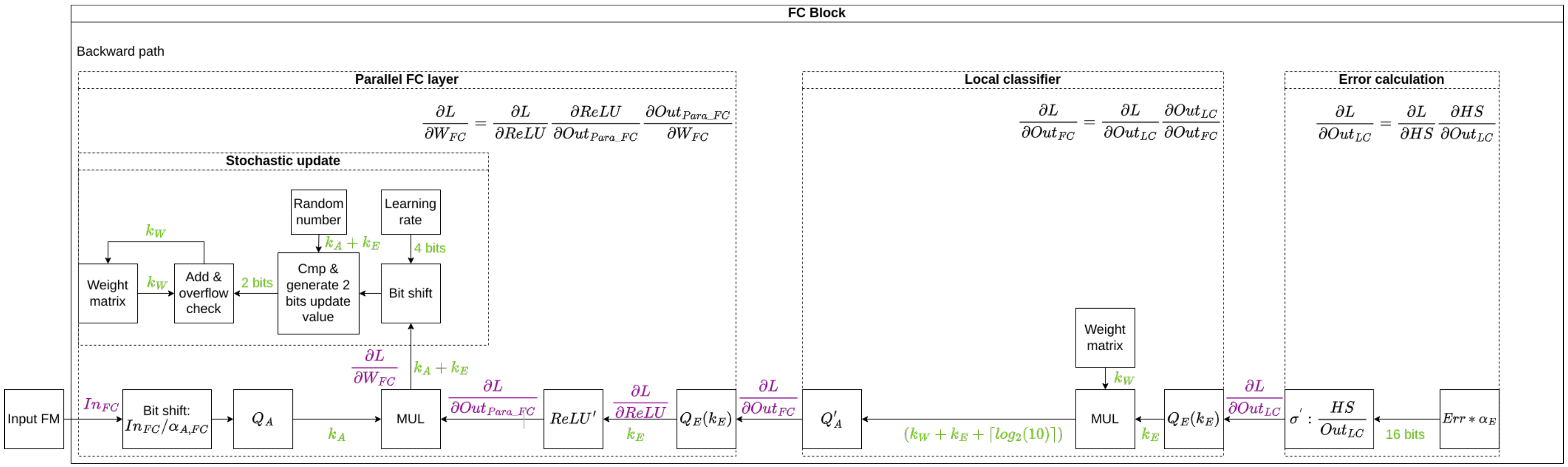

Figure 3.15: Quantized FC block (backward path): Green texts and purple texts show the bit width and contents of the data streams, respectively.

uration is evaluated for performance at 5 different weight bit resolutions.

Table 3.8: Model setup for quantized network sweep. Other parameters are as Setup3 in Table 3.7

Configured items	S1-Baseline	S2-LEL	S3-LEL	S4-LEL
Learning algorithm	BP	LEL	LEL	LEL
Convolutional layer weights	Trainable	Trainable	Fixed & pre-trained	Fixed & random
Local classifier weights	Not available	Trainable	Fixed & random	Fixed & random
Output nonlinear activation function	Hard sigmoid	Hard sigmoid	Hard sigmoid	Hard sigmoid
Loss function	MSE	MSE	MSE	MSE
Optimizer	Vanilla SGD	Vanilla SGD	Vanilla SGD	Vanilla SGD

Table 3.9: System setup for quantized network sweep

Configured items	Setup
Sweep parameters	$\alpha_{A,Conv}$, $\alpha_{A,FC}$, α_E and the hard sigmoid derivative
Sweep metric	Test set accuracy
Sweep agent	WandB [76] sweep function
Sweep method	”Bayes” method [77]
Early stopping	Enabled with ”hyperband” method [78]
Number of sweeps on each setup	Minimum 20
Number of training epochs for each sweep	100
Number of training trials for each sweep	1
Number of training epochs for final performance evaluation	100
Number of training trials for final performance evaluation	5

Table 3.9 shows the system configurations used for the parameter sweep. Because the output part of the local classifier does not undergo quantization operations, there is no need to scale the activation of the local classifier. Therefore, we do not incorporate the hypeparameter $\alpha_{A,LC}$ in the sweep. The ”Bayes” sweep method and ”hyperband” early stopping method are used to accelerate the process of the parameter sweep. However, it may also fail to find a suitable combination of parameters, so we specify a lower limit on the number of parameter sweeps on each setup, but do not include an explicit upper bound.

Once the hyperparameters have been selected by sweeping, a formal performance evaluation of the quantized network will be performed. Five trials were made to train each setup model for 100 epochs. The ultimate outcome is obtained by averaging these five final results. The most appropriate model for the hardware implementation will be chosen from among them.

Results: Figure 3.16 displays the performance of each model setup, paired with respective bit resolution weights. The blue bars indicate the performance of the unquantized model, which serves as the baseline for this experiment. It is observed that the performance of the quantized model decreases as the weight resolution is reduced. The performance drops severely with 5-bit resolutions.

Upon comparison between the models using pre-trained convolutional kernels and random ones, the model using pre-trained kernels exhibits better performance, so we will choose S3-LEL setup, especially as trainable local classifiers do not offer any significant accuray boost (see S2-LEL).

To minimize hardware footprint, we will select a resolution of 6 bits, which still offers a decent classification accuracy (64.47%) compared to its 5-bit counterpart (61.14%).

Therefore, the S3-LEL setup using 6-bit weights was selected as the model for hardware implementation. The hyperparameters and the configuration of hard sigmoid function are shown in Table 3.10.

Table 3.10: Configuration of hyperparameters and hard sigmoid function of the final model

	Value or configuration
$\alpha_{A,Conv}$	2
α_{A,FC_0}	16
α_{A,FC_1}	8
α_E	500
Hard sigmoid	L=2, non-zero surrogate derivative

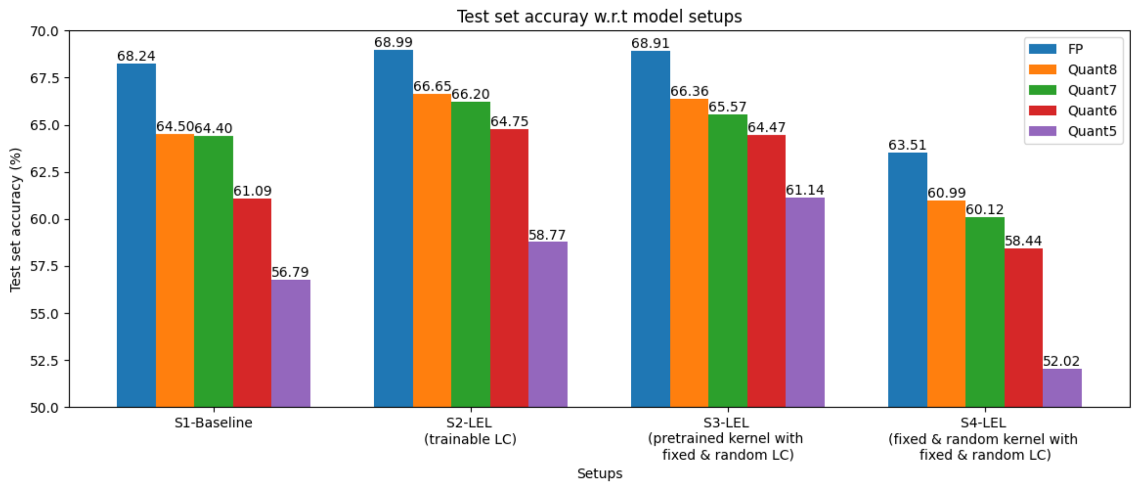

Figure 3.16: Results for quantized network sweep: FP is the floating-point model and Quant$X$ denotes on X-bit resolution model

3.4. Conclusions

This chapter provides a detailed description of how the PyTorch neural network model used in this project was designed and configured. Following performance comparisons of various designs, configurations and hardware-oriented trade-offs, we have decided to utilize the network structure depicted in Figure 3.12, the quantization scheme presented in Figures 3.13, 3.14 and 3.15, and the hyperparameter configurations outlined in Table 3.10. The network is now prepared for hardware implementation.

4 Hardware design

This chapter explains how we implemented the PyTorch model in hardware. Firstly, we will explain the implementation of the main hardware submodules. Next, we will explain the implementation of each core in our multi-core platform. Subsequently, we will use these cores to develop our on-chip multi-core learning platform. Finally, we deploy this platform to an FPGA and evaluate its performance.

In this chapter, hardware design is presented in a hierarchical bottom-up structure. Section 4.1 covers the design of submodules. Section 4.2 introduces the two cores designed by integrating the sub-modules, for the FC block and Conv block, respectively. Section 4.3 describes the design that integrates the core and applies it to the FPGA board.

4.1. Submodule design

This section presents the design of several important hardware submodules. These submodules are used to perform computational or layer functions that implement the respective parts of our neural network. Before presenting these submodules, we will introduce the environment used for behavioral simulation and the block random-access memory (BRAM) design for on-chip data storage in this section.

Simulation environment: We utilize cocotb [79] and Icarus Verilog [80] as the verification framework and simulator for behavioral simulation, respectively. Writing a testbench with cocotb only requires the use of Python. Cocotb interacts with the Icarus Verilog simulator to generate the device under test (DUT) for behavioral simulation. Consequently, waveform files and corresponding port outputs are generated. Application of this simulation environment significantly simplifies and expedites the design workflow. Additionally, software libraries such as NumPy [81] or PyTorch can be included in the Python-written testbench, which facilitates direct comparison of software and hardware implementations.

BRAM design: BRAM is a memory resource on the FPGA, so for various types of large matrices in neural networks, such as weight matrices, storing data in BRAM avoids consuming the limited lookup table (LUT) resources, which are basic FPGA building blocks that can be flexibly configured for logic or storage. BRAM instances were generated using standard Xilinx templates.

4.1.1. Matmul submodule

The parallel fully-connected layer is the most significant component of the FC block. A small fully-connected layer within a parallel fully-connected layer has a size of 600 input neurons by 480 output neurons (as shown in Figure 3.12). It is costly to directly design dedicated MAC units for each output neuron, so we use the time-division multiplexing technique, where each MAC unit will be reused multiple times for different output neurons.

Figure 4.1 depicts the block diagram of the matmul module. The ”neuron” unit is composed of a MAC unit, there are four of them in the design. The weight input port for this module has a width of 4*6 bits due to the presence of four computational units.

The purpose of each port is as follows:

- $input\ (8bits)$: Only one 8-bit input data shared across neurons will be involved in the calculation for each clock cycle.

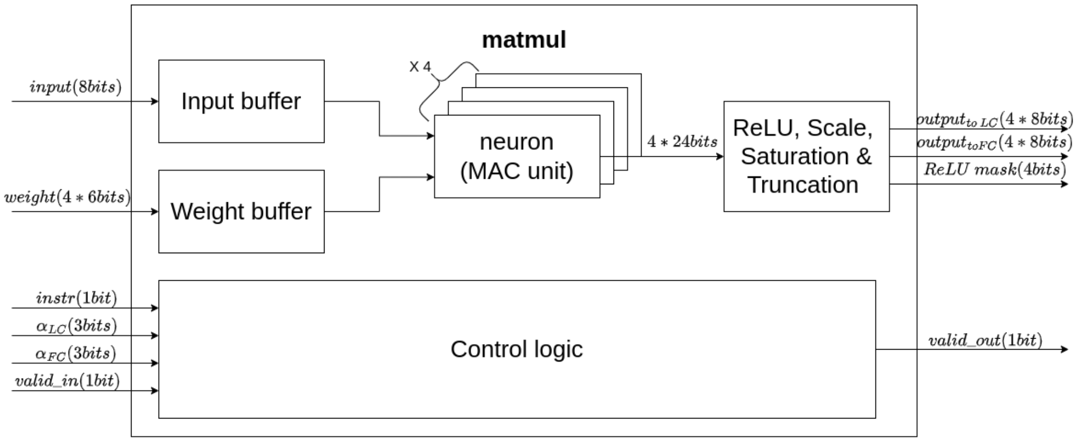

Figure 4.1: Block diagram of matmul module: clock and reset signals are ignored.

- $weight\ (4 * 6bits)$: Since there are 4 MAC units, 4 6-bit weight values are available in each clock cycle.
- $instr\ (1bit)$: Enables the current module.
- $\alpha_{FC}\ (3bits)$: This port will receive the activation scaling factor for the next FC core.
- $\alpha_{LC}\ (3bits)$: Same purpose as α_{FC}, but this activation scaling factor is for the local classifier in the current core. Reserved for possible scaling requirements.
- $valid\_in\ (1bit)$: When input data is available, this signal will be set high, then the MAC units will start computation.
- $output_{toLC}\ (4 * 8bits)$: Calculation results from 4 MAC units. The values have been scaled with α_{LC}. The output from this port goes to the local classifier.
- $output_{toFC}\ (4 * 8bits)$: Similar to the above one, but values from this port are scaled with α_{FC}, and they go to the FC core.
- $ReLUmask\ (4bits)$: It is used to indicate whether the original output value is greater than 0 and is available for recording by external logic so that the derivative value of ReLU can be determined during backpropagation. Each output value corresponds to a 1-bit value, making a total of 4 bits.
- $valid\_out\ (1bit)$: When the values of $output$ ports are available, the signal is set high by the current module.

The rounding operation in this module employs a truncation operation that rounds to the largest integer less than or equal to the input value. In contrast to the previous quantized PyTorch model, which used rounding to the nearest integer, the current design is simpler. The performance evalutaion will examine the implications of this rounding operation. In order to maintain design consistency, any other module involving rounding will also be using truncation.

Simulation setup: The steps of the simulation are as follows:

1. Use NumPy to generate a 600*4 random weight matrix and a 600*1 random input matrix. The weight matrix values range from -31 to 31 and the input matrix values range from -127 to 127.
2. Iterate through the two matrices row by row and assign the values of each row to the $weight$ port and $input$ port in each clock cycle. Enable both the $instr$ and $valid\_in$ signals.
3. Once the two matrices have been iterated through, disable the $valid\_in$ signal. Then, wait for the DUT to enable the $valid\_out$ signal.
4. Compare hardware results and the NumPy matrix multiplication results, and assert that they are equal.

After completion, repeat the above process to simulate time-division multiplexing.

Results: A one-to-one correspondence between the software implementation and the hardware implementation has been obtained. A total of 602 clock cycles are required from the start of data input to the output of a set of valid data, i.e., from the rising edge of $valid\_in$ to the rising edge of $valid\_out$.

4.1.2. Local classifer (forward path)

The subject component of the local classifier is a fully connected layer, but it only has 10 output neurons. The most straightforward implementation here is to use 10 multiply-add computation units. Figure 4.2 illustrates that the design of this part is straightforward based on the elements introduced for the matmul module (Section 4.1.1). This part features two input ports that are 4-bit wide: cur_state and nxt_state. These are employed to signal the status of the core's finite state machine, thereby determining output validity.

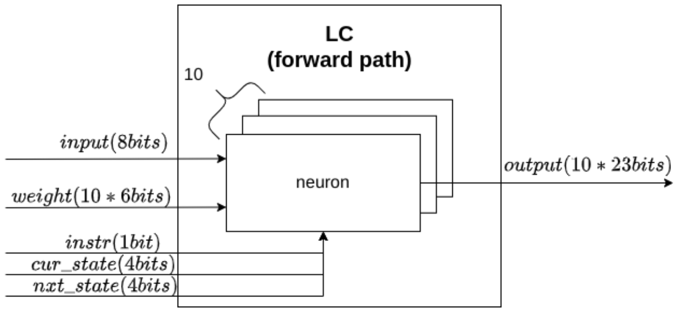

Figure 4.2: Block diagram of local classifier (forward path)

4.1.3. Local classifier (backward path)

Since the weights in the local classifier are fixed, we only need to calculate the upstream derivative of this part concerning the input during backpropagation, specifically, the $\frac{\partial L}{\partial In_{LC}}$ shown in Figure 3.15. We will obtain the upstream derivative from the loss function module, and the Q'_A using the STE, thereby making it unnecessary to compute them here. Thus, the primary concern of this implementation is the calculation of the $\frac{\partial Out_{LC}}{\partial (In_{LC}/\alpha A, LC)}$ part.

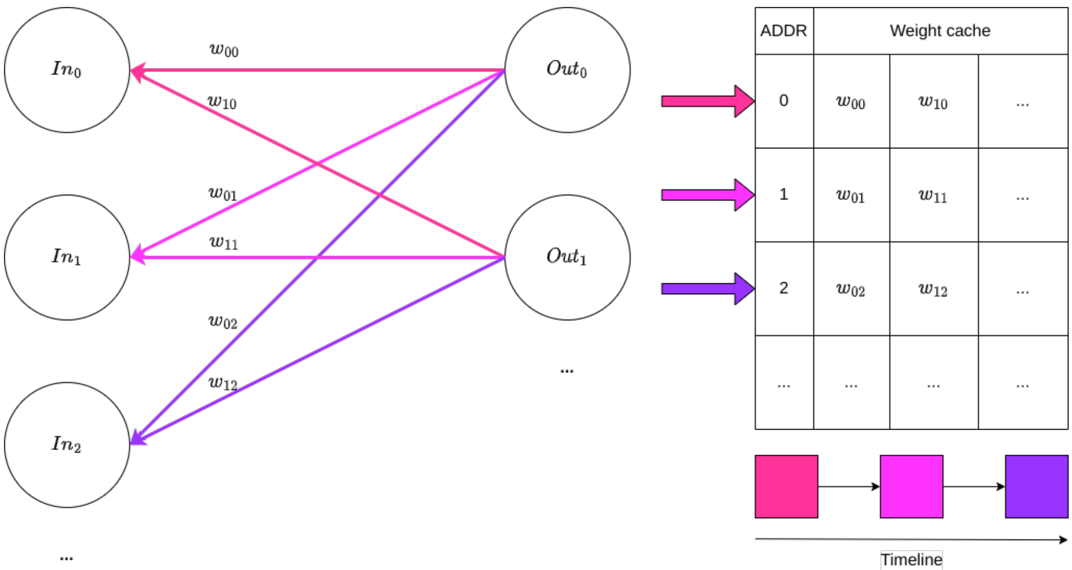

Figure 4.3: Design of local classifer (backward path): Different colors show different clock cycles

Figure 4.3 illustrates the weight access method, which is the same for the forward and backward paths. We calculate the derivative of one input neuron in each clock cycle, as expressed with $\frac{\partial Out}{\partial In_i} = \sum_{o=0}^{10} \frac{\partial L}{\partial Out_o} \cdot w_{oi}$, where i is the input neuron index and o is the output neuron index. In order to complete summation operation in 1 clock cycle, a pipelined tree adder is designed.

The purpose of each port in Figure 4.4 is as follows:

- $error\ (10 * 8bits)$: 10 error values from loss function go through this port.

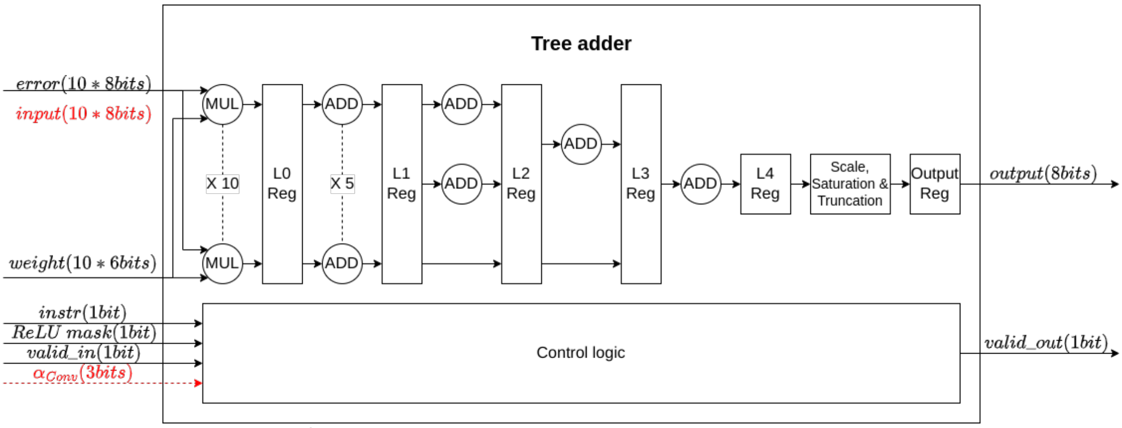

Figure 4.4: Block diagram of tree adder: MUL is the multiplication operation, ADD is the addition operation, LX Reg is the Xth pipeline register. Red names correspond to alternative port names when the tree adder is used for convolutional layers.

- $weight$ $(10 * 6bits)$: Corresponding to 10 error values, 10 LC weight values need to be provided per clock cycle.
- $instr$ $(1bit)$: Enables the current module and pipelined registers.
- $ReLU\ mask$ $(1bit)$: In the inference process, if one output value of the fully-connected layer is less than 0, then the gradient of the ReLU at the corresponding output neuron in the backpropagation is 0. Therefore, when the value from the $ReLU\ mask$ port is 0, MUL is disabled, thus skipping the unnecessary calculation process.
- $valid\_in$ $(1bit)$: When input data is available, this signal should be set high.
- $output$ $(8bits)$: Calculation result from the tree adder.
- $valid\_out$ $(1bit)$: When the value of $output$ port is available, the signal is set high by the current module.

Simulation setup: The steps of the simulation are as follows:

1. Use NumPy to generate a 1*10 random error matrix and a 480*10 random weight matrix. The error matrix values range from -127 to 127 and the weight matrix values range from -31 to 31.
2. Assign the values in the error matrix to the $error$ port. Iterate through the weight matrix row by row and assign the values of each row to the $weight$ port in each clock cycle. Enable the $instr$, $ReLU\ mask$ and $valid\_in$ signals.
3. Continuously monitor the $valid\_out$ signal and record the outputs when it is set high by the DUT.
4. Once the weight matrix have been iterated through, disable the $valid\_in$ signal. Then, wait for the DUT to disable the $valid\_out$ signal.
5. Compare hardware results and the NumPy matrix multiplication results, and assert that they are equal.

Results: A one-to-one correspondence between the software implementation and the hardware implementation has been obtained. A total of 486 clock cycles are required from the start of data input to the output of last data, i.e., from the rising edge of $valid\_in$ to the falling edge of $valid\_out$.

4.1.4. Convolutional layer

The computation performed in the convolutional layer window can be regarded as element-wise multiplication and summation between the elements of the feature map within the window and the elements of convolutional weights. The tree adder in Section 4.1.3 can be used for this purpose.

In this case, the purpose of some of the ports in Figure 4.4 has been changed to the following:

- $input$ $(10 * 8bits)$: Instead of 10 error values, the input now consists of 9 CIFAR-10 pixel values with a value of zero.

- $weight$ $(10 * 6bits)$: Instead of 10 weight values, the input now consists of 9 convolutional layer kernel weight values with a value of zero.
- $ReLU\ mask$ $(1bit)$: This port permanently receives 1.

To accommodate the use of scaling factors, the tree adder module contains an additional 3-bit wide α_{Conv} port in the current case.

4.1.5. Loss submodule

The loss submodule will integrate the hard sigmoid function and the MSE function, as shown in Figure 3.14, and compute the $\frac{\partial L}{\partial Out_{LC}}$ in Figure 3.15, which corresponds to the error. As the value of the loss itself is not important for training, only the calculation of the loss derivative will be implemented.

After fusing the hard sigmoid function, the MSE, and the error scaling factor into the same equation, the calculation in this module can be expressed as:

$$\frac{\partial L}{\partial Out_{LC,i}} = \begin{cases} y_i^* & if\ Out_{LC,i} < -2, \\ y_i^* - 1 & if\ Out_{LC,i} > 2, \\ \frac{(4 \cdot y_i^* - (Out_{LC,i} + 2))}{4} & otherwise \end{cases} \tag{4.1}$$

where i is the output neuron index of the fully-connected layer in the local classifier.

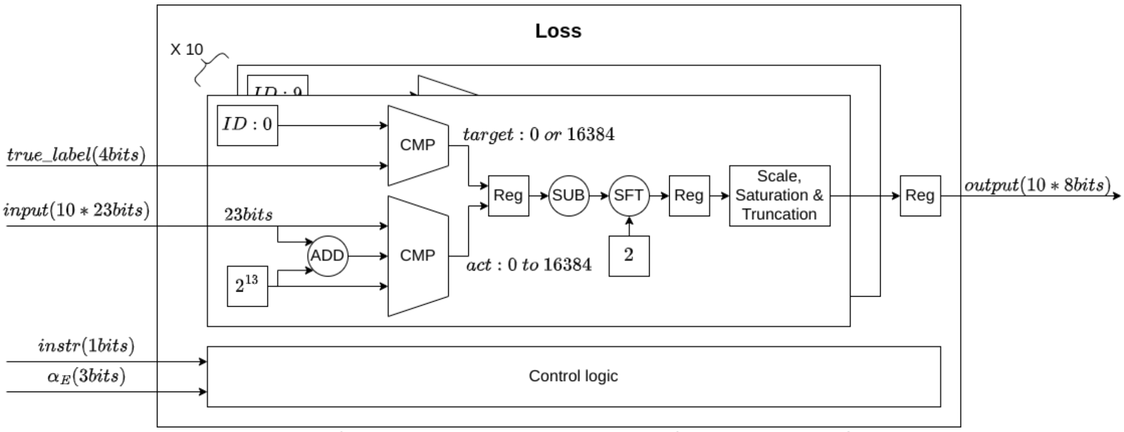

Figure 4.5: Block diagram of the loss module: ID is the index of the current part, CMP is the comparison operation, SUB is the subtraction operation, SFT is the shift operation.

The module design is presented in the Figure 4.5.
The purpose of each port is as follows:

- $true\_label$ $(4bits)$: The value of class label ranges from 0 to 9, 4 bits are needed to represent it.
- $input$ $(10*23bits)$: According to Figure 3.14, the bit width of the output values from local classifier is $8 + 6 + 9 = 23$. The local classifier forward propagation outputs ten results at once, so here it is 10*23-bit wide.
- $instr$ $(1bit)$: Used to enable the current module.
- α_E $(3bits)$: Although the scaling can be implemented directly by the hardware, the input port for the scaling factor is reserved here for hardware configurability.
- $output$ $(10 * 8bits)$: Individual 8-bit error values for each of the 10 inputs.

Simulation setup: The steps of the simulation are as follows:

1. Use PyTorch to generate a 1*10 random input matrix and a random class label value. The input matrix values range from -12,288 to 12,288 and class label value ranges from 0 to 9.
2. Assign the values in the input matrix to the $input$ port. Assign the class label value to the $true\_label$ port. Enable the $instr$ signal for 2 clock cycles and disable it afterwards.
3. After 3 clocks, get the error values from $output$ port.
4. Use PyTorch to build software model-consistent setup of hard sigmoid function and MSE function.

5. Input the input matrix and class label to this software setup and get the results.
6. Compare software results and hardware results, and assert that they are equal.

Results: A one-to-one correspondence between the software implementation and the hardware implementation has been obtained.

4.1.6. Weight update submodule

This module implements the weight stochastic update for the fully-connected layer (Figure 3.14). Additionally, this module will calculate $\frac{\partial Out_{FC}}{\partial W_{FC}}$, which is also known as the gradient. Because the batch size of the input data is 1 on our on-chip learning platform, the simplified computation of $\frac{\partial Out_{FC}}{\partial W_{FC}}$ is as follows:

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} \cdot \begin{bmatrix} E_0 & E_1 \end{bmatrix} = \begin{bmatrix} X_0 \cdot E_0 & X_0 \cdot E_1 \\ X_1 \cdot E_0 & X_1 \cdot E_1 \\ X_2 \cdot E_0 & X_2 \cdot E_1 \end{bmatrix} \tag{4.2}$$

where X is the activation, E is the error, which can be expressed as $\frac{\partial L}{\partial Out_{FC}}$. This part involves multiplication operations only, without any addition as shown in the equation above. Therefore, a set of gradient values can be produced at each clock cycle.

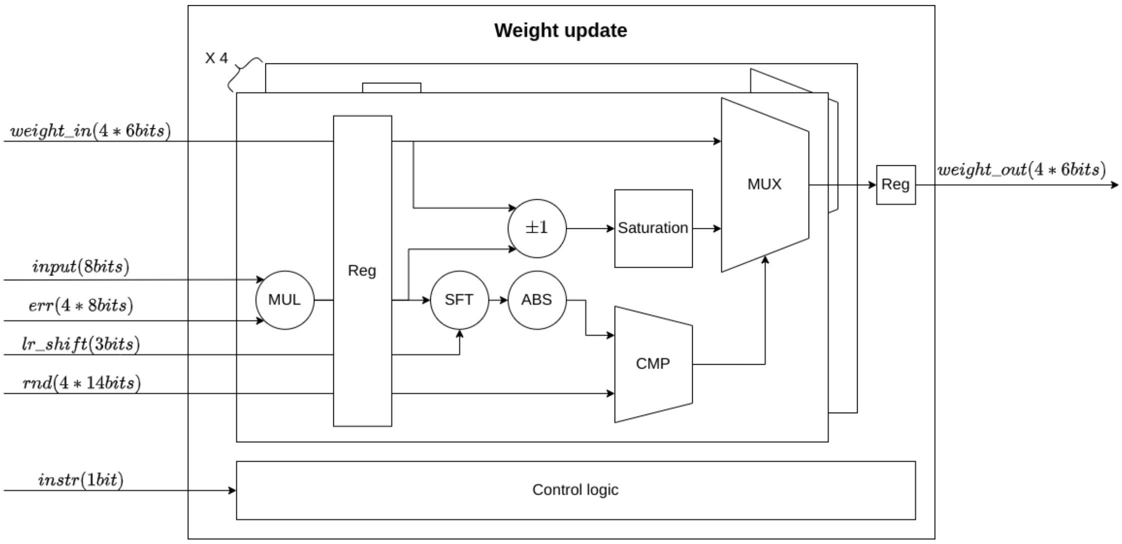

Figure 4.6: Block diagram of weight update module: ABS is the absolute value operation, MUX is the multiplexer

Figure 4.6 illustrates the design of the weight update module. In order to be able to share the data channels with the matmul module and to simplify the data cache design, this module also has four sets of computational units. The purpose of each port is as follows:

- $weight\_in$ $(4*6bits)$: This port shares the data channel with the $weight$ port of the matmul module and is therefore also 24 bits wide.
- $input$ $(8bits)$: This port shares the data channel with the $input$ port of the matmul module and is therefore also 8 bits wide.
- err $(4*8bits)$: This port receives four 8-bit error values, corresponding to the error values propagated through each output neuron in backpropagation.
- lr_shift $(3bits)$: The value goes through this port, and will determine the number of bits by which the calculated probability value will be shifted to the right.
- rnd $(4*14bits)$: Random numbers input from an external source are used to decide whether or not to update the weight values by comparing them with the calculated probability values.
- $instr$ $(1bit)$: Used to enable the current module.
- $weight\_out$ $(4*6bits)$: This port will output weight values that need to be written back to the weights cache.

Simulation setup: The steps of the simulation are as follows:

1. Use NumPy to generate a 600*4 random weight matrix, a 1*4 random error matrix, a 600*1 random activation matrix and a 600*4 random number matrix. The weight matrix values range from -31 to 31. Values in the error matrix and activation matrix range from -31 to 31. The random number matrix values range from 0 to 16383.
2. Assign the error matrix to the err port. Iterate through the matrices row by row and assign the values of each row to the $weight\_in$ port, $input$ port and rnd port in each clock cycle. Enable the $instr$ signal.
3. After two clock cycles, start monitoring the $weight\_out$ port and collect the data.
4. Once all the matrices have been iterated through, disable the $instr$ signal. Then after two clocks, stop collecting data from $weight\_out$ port.
5. Calculate the updated weights utilizing the same logic as in the software model, but replace the random part with the generated random number matrix.
6. Compare hardware results and the software results, and assert that they are equal.

Results: A one-to-one correspondence between the software implementation and the hardware implementation has been obtained.

4.1.7. Linear feedback shift register submodule

To generate random numbers for local classifier weights or random weight updates on hardware, a PRNG needs to be designed. A straightforward pseudorandom number generator (PRNG) can be implemented with a linear feedback shift register (LFSR). The LFSR can output only one bit per clock cycle, whereas our weight update module requires a total of 56 bits per clock cycle. Using parallel a LFSR structure would significantly increase the hardware overhead [82]. Therefore, this module employs the unfolded LFSR algorithm described in [83, 82]. The method operates by unfolding the data flow graph formed by the LFSR, appending multiple nodes to it, and obtaining the output value of the original LFSR after a certain number of cycles from these nodes.

The length of the LFSR is set according to [84] to avoid correlations between weights of the same output neuron, which leads to a LFSR length of 17.

Then we can take $x^{17} + x^3 + 1$ as the primitive polynomial, and the taps will be at position 3 and 17. The diagram for the basic LFSR and the corresponding data flow graph is shown in Figure 4.7.

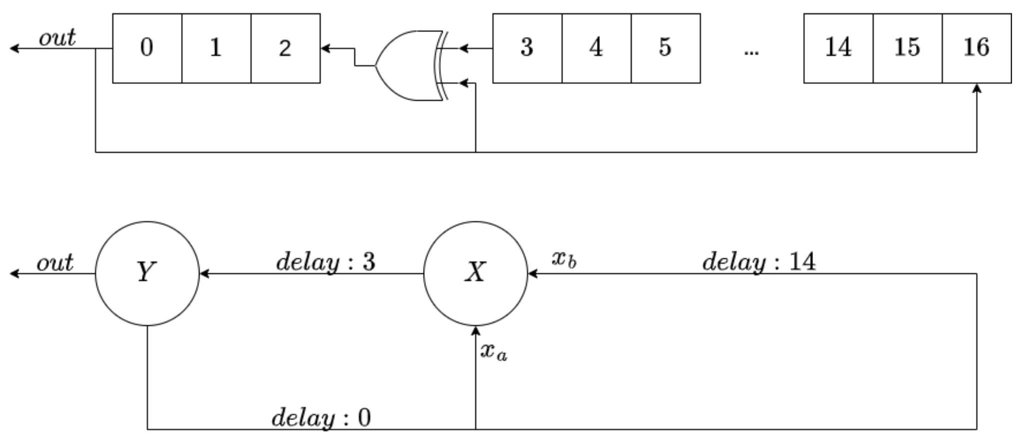

Figure 4.7: Basic LFSR diagram (above): numbers are the index of the state registers;
Data flow graph (below): Y is the output node, X is the XOR node, x_a and x_b are 2 input port of the XOR gate.
Delay units correspond to clock cycles.

The general design of the LFSR module is shown in Figure 4.8. The purpose of each port is as follows:

- $seed\ (17bits)$: The value received by this port can be used to set the initial state of the LFSR.
- $instr\ (1bit)$: Used to enable the current module.

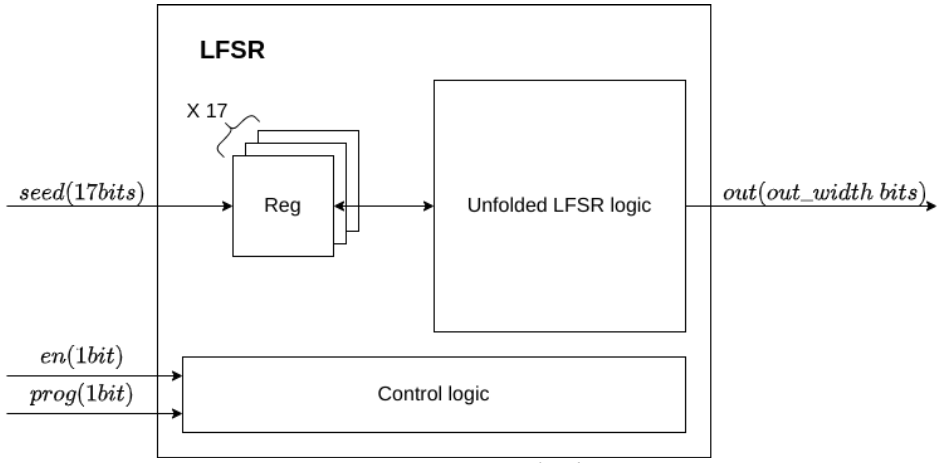

Figure 4.8: Block diagram of LFSR module

- $prog\ (1bit)$: Used to set the initial state with value from $seed$ port.
- $out\ (out\_width\ bits)$: The bit width of this port depends on the configuration at the time of instantiation. The value output by the port is all the bits that were sequentially output by the original LFSR in out_width clock cycles.

Simulation setup: In the simulation testbench for LFSR, we will use the Python library $galois$ [85], which is used to generate software reference LFSR output. The steps of the simulation are as follows:

1. Let Python generate a random value, which ranges in 0 to $2^{17} - 1$. Assign this value to the $seed$ port and set the $prog$ signal to high. Use this value initialize the software LFSR from $galois$ library, too.
2. After one clock cycle, set the $prog$ signal to low.
3. Set the en signal to high and run for one clock cycle. Get the random number from the out port.
4. Let the software LFSR genereate output bitstream for out_width steps.
5. Compare hardware results and the software results, and assert that they are equal.
6. Repeat the above three steps several times.

Results: For both 56-bit and 60-bit configurations, a one-to-one correspondence between the software implementation and the hardware implementation has been obtained.

4.1.8. Cache X4 submodule

For storing activations, weights, and upstream gradients during backpropagation, caching is required. Our design is to directly generate four small cache blocks, as there are four parallel FC layers, and access one of them when in use.

Figure 4.9 shows the overall design of the cache X4 module. The design template officially provided by Xillinx for BRAM was used directly for the design of each simple dual-port cache. As this module will be applied to three scenarios, which will store matrices with different shapes, the bit-widths of the address ports $addr_0$ and $addr_1$, and the data ports $data\_in_0$ and $data\_out_1$ are left generic in the figure. The purpose of each port is as follows:

- $addr_0$: The destination address for data writing.
- $data\_in_0$: Data will be written to the cache.
- $addr_1$: The address for data reading.
- $data\_out_1$: Data read from cache.
- $p\_sel\ (2bits)$: The specified index of the cache block.
- $wr\_en\ (1bit)$: Used to enable the data writing.

An additional register is used in the output section of the module to enhance clock-to-out timing.

Simulation setup: The steps of the simulation are as follows:

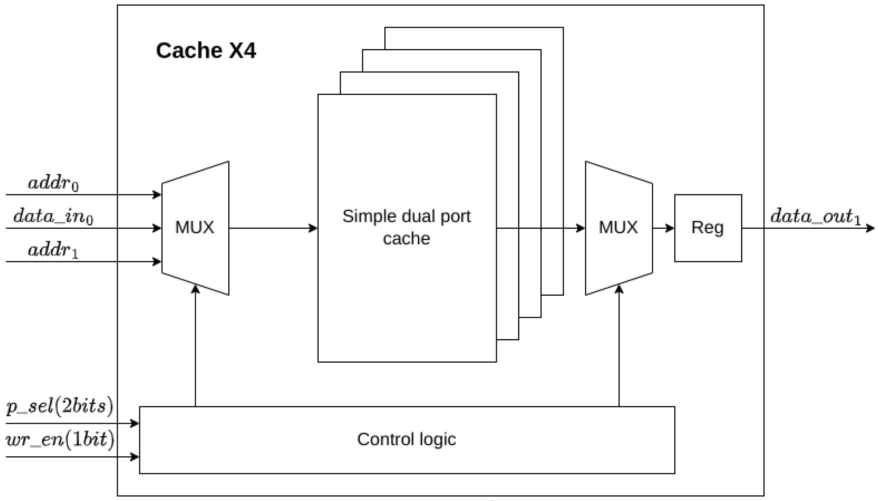

Figure 4.9: Block diagram of cache X4 module

1. A 3-dimensional matrix containing random values will be generated using NumPy. The size of the matrix is 4*$input\ depth$*$input\ size$. The $input\ size$ is the number of values written to the module within a single clock cycle, which depends on the scenario in which it is being applied.
2. Iterate through the matrix row by row and assign the values of each row to the $data\_in_0$ port in each clock cycle. Here a row refers to a 1*$input\ size$ matrix. And depending on the block in which it is selected, and the line number, specify the value of the p_sel port as well as the $addr_0$ port. Enable the wr_en signal.
3. Once the matrix has been iterated through, disable the wr_en signal. The contents of the cache are then read by specifying the values of the $addr_1$ port and the p_sel port.
4. Compare hardware readouts and values in original matrix, and assert that they are equal.

Results: For three application scenarios, a one-to-one correspondence between the software implementation and the hardware implementation has been obtained.

4.1.9. Triout cache submodule

The convolutional layer or max pooling layer's window requires to access the values of 3 columns in 3 rows of input data simultaneously. To reduce the number of cache accesses, we designed a cache module consisting of three output ports. Each port corresponds to a row of data in a CIFAR-10 image or a feature map. One port outputs the currently accessed row while the other two ports output the last two accessed rows.

The design of the triout cache module is shown in Figure 4.10. The dashed section is used to store the CIFAR-10 dataset and facilitate data selection from one of the channels of the input image, selected through the value from $cur\_channel$ port. While storing the output feature map from the convolutional layer, the design of our Conv core makes it unnecessary to store the contents of multiple channels, so the dashed section is not needed. Because this module is used in different scenarios, the $addr\_in$ port bit width is kept generic. The purpose of each port is as follows:

- $addr\_in$: The destination address for data writing.
- $data\_in$ $(8bit)$: Data will be written to the cache.
- wr_en $(1bit)$: Enables the data writing.
- clr_opt $(1bits)$: This signal should be set high when it is necessary to restore the values of three output ports to the first three rows of the feature map in the current channel.
- pop_en $(1bit)$: This signal should be set high when we want to get the next row of the feature map.
- $data\_out_0$ $(X * 8bits)$, $data\_out_1$ $(X * 8bits)$ and $data\_out_2$ $(X * 8bits)$: $data\_out_2$ will output the current row of the feature map and the remaining two ports will output the previous two rows.

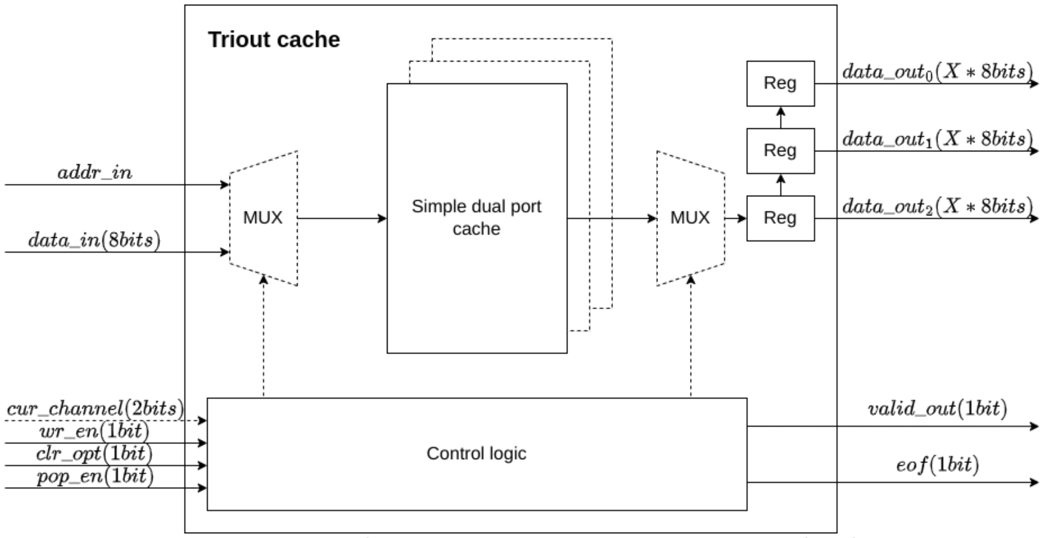

Figure 4.10: Block diagram of triout cache module: X depends on the size of the feature map

- $valid\_out$ $(1bit)$: When the values of three $data\_out$ port are available, the signal is set high by current module.
- eof $(1bit)$: This signal is set high when the feature map of the current channel has no next row to fetch.

Simulation setup: We will only test the more complex activation cache configuration in this part. In this case, the bit width of $data\_in$ port will be 4*8. The steps of the simulation are as follows:

1. Use NumPy to generate a 3*256*4 random feature map matrix. The 3 is the number of input channels. The 4 is the number of values written to the module within a single clock cycle. The matrix values range from -127 to 127. A copy of this matrix is then reshaped to be a 3*32*32 matrix, which will serve as a comparison reference for the output.
2. Iterate through the feature map matrix row by row and assign the values of each row to the $input$ port in each clock cycle. Here a row refers to a 1*4 matrix. Enable the wr_en signal.
3. Once the matrix have been iterated through, disable the wr_en signal. Enable the clr_opt signal and wait for the $valid\_out$ signal to be high.
4. Compare hardware output and the first three rows in the reference matrix, and assert that they are equal.
5. Enable pop_en signal for more than 32 clock cycles.
6. Compare hardware output and the last three rows in the reference matrix, and assert that they are equal. The eof signal should be set high by the DUT.

Results: A one-to-one correspondence between the software implementation and the hardware implementation has been obtained. 4 clock cycles are required from the start of output reset to get available output data, i.e., from the falling edge of the clr_opt signal to the rising edge of $valid\_out$ signal. 2 clock cycles are required from the start of request a new row to get available output data, i.e., from the rising edge of the pop_en signal to the rising edge of $valid\_out$ signal.

4.1.10. Tree comparator submodule

The main function of the max pooling layer in the Conv block is to identify the largest value within the current window. In our model, a pooling window size of 3*3 is used, and a simple and straightforward approach is to create a 9-input, 1-output comparator. Inspired by the tree adder in Figure 4.4, we will design a pipelined tree comparator module.

The design of the tree comparator is shown in Figure 4.11. The purpose of each port is as follows:

- $input$ $(9 * 8bits)$: The window size in the pooling layer is 3*3, so this module has nine inputs.
- $instr$ $(1bit)$: Used to enable the current module and pipelined registers.

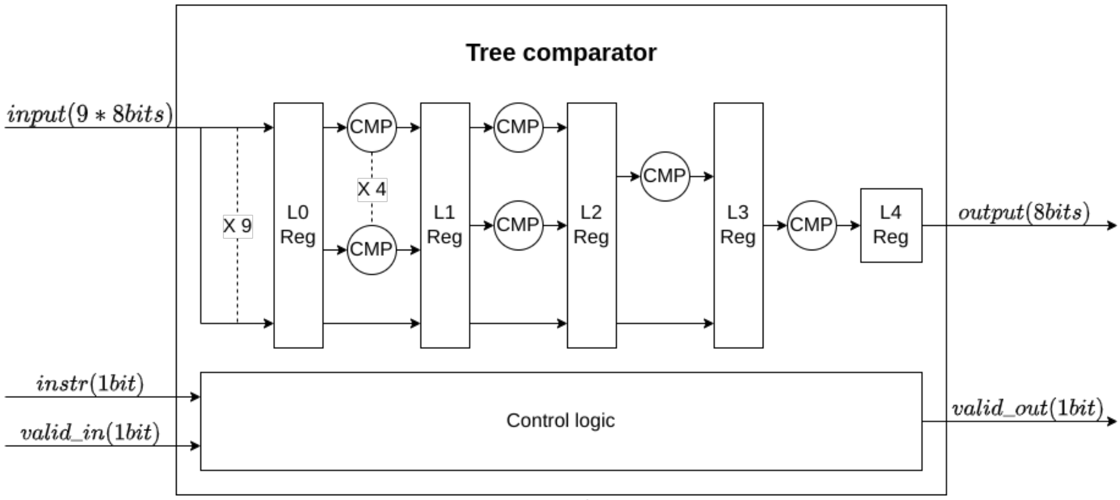

Figure 4.11: Block diagram of tree comparator module

- $valid\_in$ $(1bit)$: When input data is available, this signal should be set high.
- $output$ $(8bits)$: The output max value in nine inputs.
- $valid\_out$ $(1bit)$: When the value of $output$ port is available, the signal is set high by the current module.

Simulation setup: The steps of the simulation are as follows:

1. Use NumPy to generate a 100*9 random feature map matrix. The matrix values range from -127 to 127.
2. Iterate through the matrix row by row and assign the values of each row to the $input$ port in each clock cycle. Enable both the $instr$ and $valid\_in$ signals.
3. Continuously monitor the $valid\_out$ signal and record the outputs when it is set high by the DUT.
4. Once the matrix has been iterated through, disable the $valid\_in$ signal. Then, wait for the DUT to disable the $valid\_out$ signal.
5. Compare hardware results and the maximum value obtained within each row using NumPy, and assert that they are equal.

Results: A one-to-one correspondence between the software implementation and the hardware implementation has been obtained. A total of 105 clock cycles are required from the start of data input to the output of last data, i.e., from the rising edge of $valid\_in$ to the falling edge of $valid\_out$.

4.1.11. Classification submodule

To output the classification result, we need a module that can identify the largest value and its index from the 10 values outputted by the local classifiers in the FC core. In contrast to the high-throughput requirement of the max pooling layer in the Conv block, this module will only be used once during inference or training on a single input data, and it will not affect the other cores. As a result, this module has a simple design. The design of the classification module is shown in Figure 4.12. The purpose of each port is as follows:

- lc_out $(10*23bits)$: This port receives the output of 10 neurons from local classifiers in the FC core.
- en $(1bit)$: Enables the current module.
- max_lc_out $(23bits)$: The output max value among ten inputs.
- $class\_num$ $(4bits)$: The index of the output max value.
- $classify\_done$ $(1bit)$: When the results are available, this signal will be set high by the current module.

Simulation setup: The steps of the simulation are as follows:

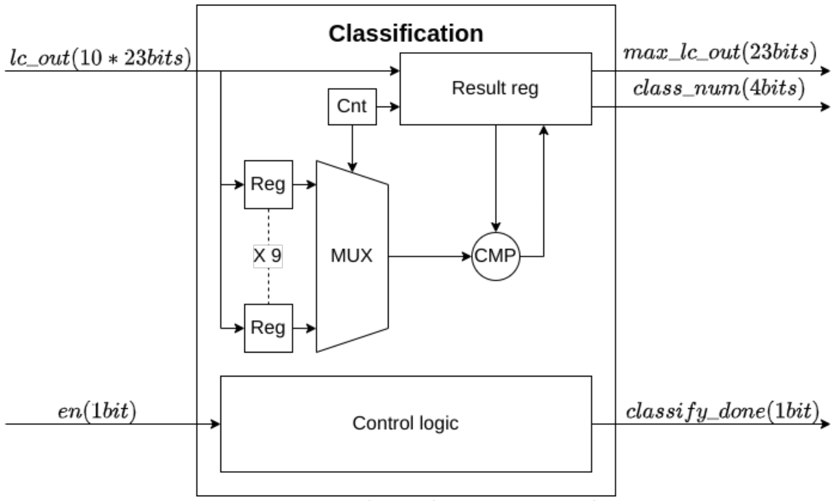

Figure 4.12: Block diagram of classification module: Cnt is the counter

1. Use NumPy to generate a 1*10 random number matrix. The matrix values range from $-2^{22}+1$ and $2^{22}-1$.
2. Assign this matrix to the lc_out port. Enable the en signal.
3. Get the max value from max_lc_out port and its corresponding index from $class\_num$ when the $classify\_done$ is set high by the DUT.
4. Compare hardware output and the max value and its corresponding index by NumPy, and assert that they are equal.

Results: A one-to-one correspondence between the software implementation and the hardware implementation has been obtained. 9 clock cycles are required from the start to get the available output, i.e., from the falling edge of the en signal and the rising edge of the $classify\_done$ signal.

4.2. Core design

We now have the necessary submodules to implement both the FC and Conv cores. First, we will present the design of the FC core, which will implement the computation within the FC block. Next, we will present the design of the Conv core, which will implement the computation within the Conv block. Before describing the two core designs in detail, we will first describe the common interface for core communication.

Core communication protocol: The final design will be implemented on Xilinx XC7Z020, which is a system-on-chip (SoC) FPGA. It includes a dual-core ARM-cpu-based processing system (PS), a programming logic (PL) and many interfaces [86]. To enhance the efficiency of CIFAR-10 dataset reading, the dataset will be stored in the on-board dynamic random-access memory (DRAM), which will then be accessed by the PS. On this SoC, an efficient approach to communicating between the PS and PL is through the use of an AXI bus [86], which follows a 2-way handshake communication protocol. Since our design requires only point-to-point transmission and no address is required, we can refer to the simplest AXI-Stream protocol [87]. This protocol does not include address channels for reading and writing. To maintain compatibility with this protocol, our own communication protocol requires just one data channel, one ready signal and one valid signal.

The timing diagram is illustrated in Figure 4.13. Once data is ready, the transmitter sets the ”Stream valid” signal high. When the receiver is ready to receive data, it sets the ”Stream ready” signal high. Once both of ”Stream valid” and ”Stream ready” signals are set high, data transmission starts. The timing of this design is identical to the AXI-Stream handshake mechanism, so that when only simple data transfers are required, one end of the AXI-Stream bus can be connected to this interface.

4.2.1. Fully-connected core

This core implements the operations illustrated in Figure 3.14 and 3.15. The design of the FC core is depicted in Figure 4.14.

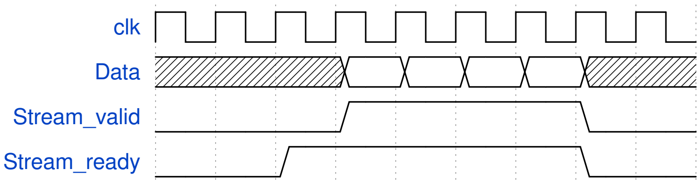

Figure 4.13: Simple core communication timing diagram

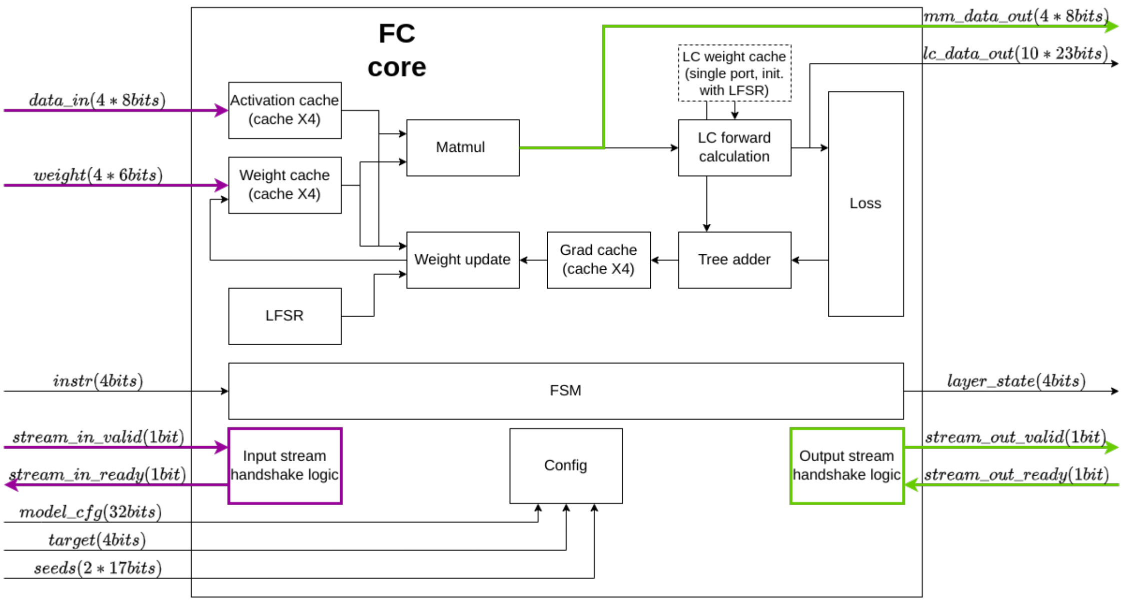

Figure 4.14: Block diagram of FC core: FSM is the finite state machine (FSM) logic. Purple parts comprise the input stream interface, while green parts comprise the output stream interface.

The FC core includes a set of stream input logic and a set of stream output logic, as follows:

- **Stream input part:** The $weight$ port receives initial weights during initialization state, while the $data\_in$ port receives feature maps during training or inference state. As these two states are mutually exclusive, the input stream handshake logic is shared.
- **Stream output part:** The mm_data_out port outputs data from the parallel FC layer. Therefore, the data on the mm_data_out port needs to be streamed, and it will be used with the output stream handshake logic.

The $model\_cfg$ port will receive a 32-bit value which contains the configuration of the scaling factor as well as the learning rate as shown in Figure 4.15. The configuration information is sent directly from the PS to the PL via the AXI interface, which is configured to use a 32-bit data signal. Using a 32-bit data signal to contain the configuration of only one core is not efficient, therefore, the configurations of all three cores are included in this 32-bit value and bit shifted appropriately before assigning to port of each core, so that each core receives the correct configuration.

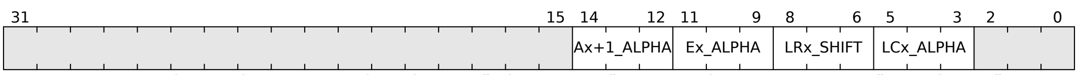

Figure 4.15: Bitfields of $model\_cfg$ for FC core: "LCx_ALPHA" is the shift-based α_{A,LC_x}, "LRx_SHIFT" is the shift-based learning rate for core x, "Ex_ALPHA" is the shift-based α_{E_x}, "Ax+1_ALPHA" is the shift-based $\alpha_{A,FC_{x+1}}$, and x is the index of the FC core.

The $target$ port will receive the 4-bit target label for the loss module. The $seeds$ port will receive two 17-bit width seeds for configuring the initial state of two LFSR modules in the core.

The core implements a FSM and will receive external instructions through the $instr$ port and output the current state through the $layer\_state$ port. The list of external instructions and their purposes are listed as follows:

- **I_IDLE:** Keep cores in the idle state.
- **I_INIT:** Let cores get into the initialization state.
- **I_TRAIN:** Let cores get into the training-related states.
- **I_INF:** Let cores get into the inference-related states.
- **I_STOP:** Force cores out of the current state and into the idle state.

Figure 4.16 shows the state transition diagram corresponding to the finite state machine in the FC core. The state corresponding to the solid circle will go to the specified next state based on external instructions from the $instr$ port, except for the **I_STOP** instruction, which in turn allows any state to be transferred to the idle state.

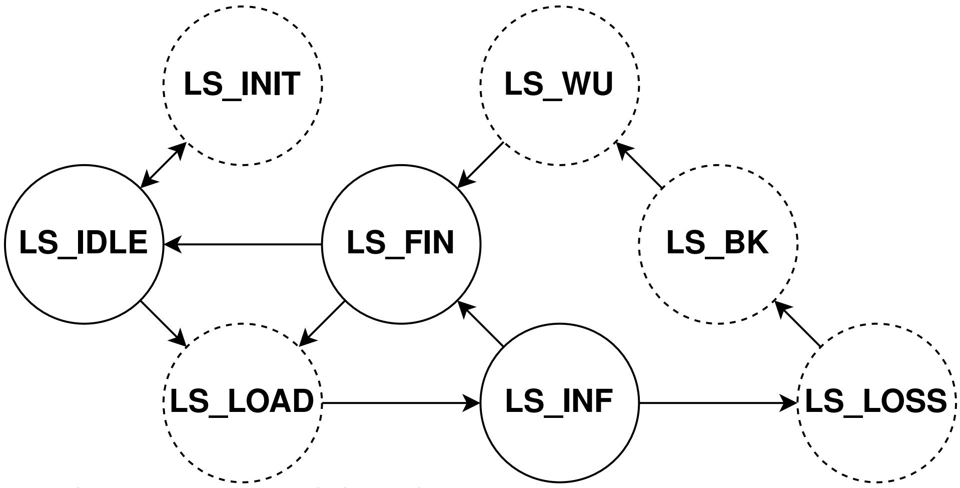

Figure 4.16: State transition diagram of FSM in FC core. For clarity, the state transition conditions are given in the main text.

The list of FC core states and corresponding behaviors are listed as follows:

- **LS_IDLE:** This is the default state of the core. In this state, no calculations nor cache accesses are performed. When the **I_INIT** instruction is given, the state is transferred to **LS_INIT**. When a **I_TRAIN** or **I_INF** instruction is given, the state is transferred to **LS_LOAD**.
- **LS_INIT:** In this state, the weight cache of parallel fully-connected layer is initialized with the values from the $weight$ port. If there is no weight cache for the local classifier, the initial state of the two LFSR modules will be initialized. Otherwise, only the initial state of the LFSR module for the weight update module will be set, and the weight cache for the local classifier will be initilized. After initialization, the state is automatically transferred to **LS_IDLE**.
- **LS_LOAD:** In this state, the activation cache is loaded with the feature map from the $data\_in$ port. Simultaneously, the model configuration from the $model\_cfg$ port, and the true label from the $target$ port are read. Once the feature map is loaded, the state is automatically shifted to **LS_INF** if the $stream\_out\_ready$ signal is high. Otherwise, the core will wait until the receiver is ready to receive the data.
- **LS_INF:** In this state, the matmul module and the computation logic for the local classifer forward path are enabled. During the computation period, the activation cache, weight cache and LC weight cache iteratively output their values. The mm_data_out port outputs data intermittently. At the end of the computation, the result from the local classifier is available at the lc_data_out port. When the **I_TRAIN** instruction is given, the state is transferred to **LS_LOSS**. When the **I_INF** instruction is given, the state is transferred to **LS_FIN**, as the loss calculation is unnecessary in our design during inference.
- **LS_LOSS:** In this state, the loss module is enabled, and $\frac{\partial L}{\partial Out_{LC}}$ will be calculated. Then, the state is automatically transferred to **LS_BK**.

- **LS_BK:** In this state, the tree adder module is enabled. During the computation period, the LC weight cache iteratively outputs the values, and the results of the tree adder is written to the gradient cache. After the computaion is done, the state is automatically transferred to **LS_WU**.
- **LS_WU:** In this state, the weight update module is enabled. During the computation period, the activation cache, weight cache, gradient cache and the LFSR module iteratively output the their values. The updated weights are written back to the weight cache. After the weights are updated completely, the state is automatically transferred to **LS_FIN**.
- **LS_FIN:** This state is similar to the **LS_IDLE** state, with the difference that it will transfer to the **LS_IDLE** state when the instruction is not **I_TRAIN** or **I_INF**. This state is designed to allow distinguishing between a run that has not started and a run that has ended.

As performing an exhaustive simulation of this standalone core is outside the scope of this work, we verified the functionality of the FSM and ensured that the correct modules are enabled with proper data access. The FC core will be validated within the full platform simulation setup in Section 4.4. The number of clock cycles for every state can be found in Table 4.1.

Table 4.1: Number of clock cycles for FC core states

FC core FSM state	Clock cycles
LS_INIT	72003
LS_LOAD	602
LS_INF	72608
LS_LOSS	1
LS_BK	489
LS_WU	72017
LS_FIN	1

4.2.2. Convolutional core

This core implements the operations illustrated in Figure 3.13. Figure 4.17 shows the design of the Conv core, which also contains a set of stream input logic and a set of stream output logic:

- **Stream input part:** The $data\_in$ port receives CIFAR-10 images during training or inference state. The input stream handshake logic will work with this port.
- **Stream output part:** The mp_data_out port will output the results from the max pooling layer. The input stream handshake logic will work with this port.

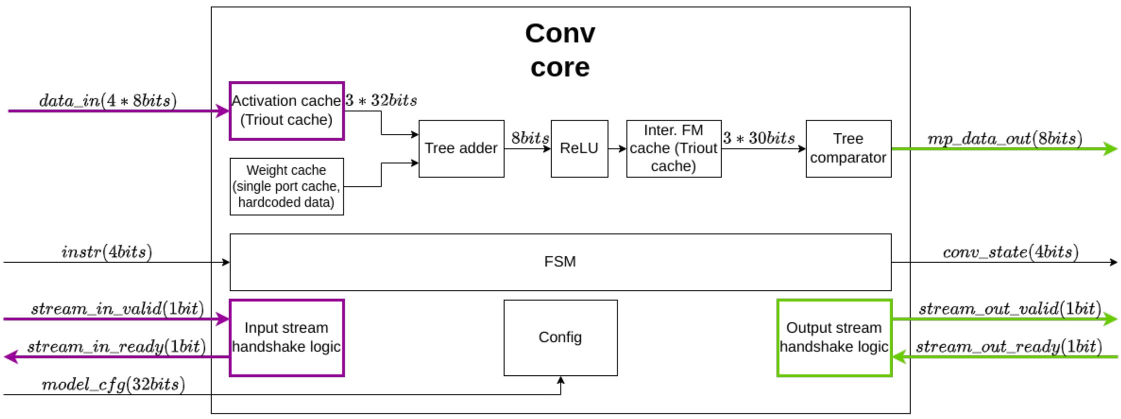

Figure 4.17: Block diagram of Conv core: Inter. FM is abbreviation for intermediate feature map. Purple parts comprise the input stream interface, while green parts comprise the output stream interface.

The $model\_cfg$ port remains 32-bit wide, as presented in Figure 4.18. However, it will only obtain one scaling factor for use in the tree adder module.

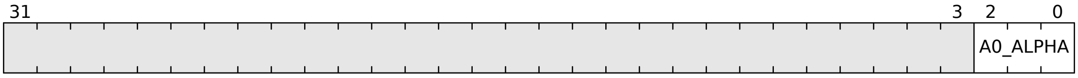

Figure 4.18: Bitfields of $model\_cfg$ for Conv core: "A_ALPHA" is the shift-based $\alpha_{A,FC0}$ for the first FC core.

Similar to the FC core, the FSM receives instructions from the $instr$ port and outputs the current state through the $conv\_state$ port. To ensure consistency in control methods for the multi-core platform, the instruction set used here is identical to that of the FC core.

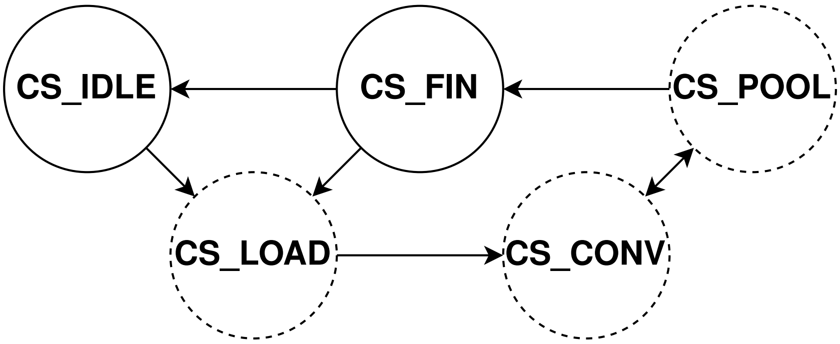

Figure 4.19: State transition diagram of FSM in Conv core. For clarity, the state transition conditions are given in the main text.

Figure 4.19 shows the state transition diagram corresponding to the finite state machine in the Conv core. Since the contents of the convolutional kernel is hard-coded, no initialization state is included in the FSM. Moreover, since the Conv core does not need to be trained, the FSM also does not contain states related to loss computation and backpropagation. The list of Conv core states and corresponding behaviors are listed as follows:

- **CS_IDLE:** This is the default state of the core. In this state, no calculations nor cache accesses are performed. When a **I_TRAIN** or **I_INF** instruction is given, the state is transferred to **CS_LOAD**.
- **CS_LOAD:** In this state, the activation cache is loaded with the feature map from the $data\_in$ port. Simultaneously, the model configuration from the $model\_cfg$ port is read. Once the feature map is loaded, the state is automatically shifted to **CS_CONV** if the $stream\_out\_ready$ signal is high. Otherwise, the core will wait until the receiver is ready to receive the data.
- **CS_CONV:** In this state, the tree adder module is enabled. During the computation period, the activation cache and weight cache output their values. The tree adder module's output will be stored in the intermediate feature map cache. To minimize the size of the intermediate feature map cache, the state is transferred automatically to the **CS_POOL** state when a complete convolution is executed within an output channel.
- **CS_POOL:** In this state, the tree comparator module is enabled. During the computation period, the intermediate feature map cache outputs the values. The ouput value will directly be transfered through the mp_data_out port. After completing the pooling operation on the current feature map, there are two possible state transitions. If the current output channel is not the last channel, then the FSM returns to the **CS_CONV** state and performs the convolution operation on the next output channel. If the current output channel is already the last channel, then the FSM goes directly to the **CS_FIN** state and ends the computations on the current input image.
- **CS_FIN:** This state is consistent with the **LS_FIN** state of the FC core and there will be no actual computational operations.

Simulation setup: The Conv core, unlike the FC core, is smaller in size and does not contain logic for training. This allows for full one-to-one hardware-to-software comparison tests. The steps of the simulation are as follows:

1. The software component calculation will be the first to run. Import an image from the PyTorch data loader into the quantized Conv block. It should not be trainable and the pre-trained convolutional kernel is loaded. Record the output matrix.
2. For the hardware simulation, the $instr$ port is first assigned with the **I_TRAIN** or **I_INF** instruction. Once the $stream\_in\_ready$ signal is high, the reshaped CIFAR-10 image is assigned to the $data\_in$ port line by line, and the $stream\_in\_valid$ signal is enabled. After running for a few clock cycles, the $stream\_out\_ready$ signal is then enabled by the testbench. Once the FSM leaves the **CS_LOAD** state, checked via the $conv\_state$ port, the image should be loaded completely.
3. At the end of the hardware simulation on current image, the $conv\_state$ signal should give **CS_FIN** signal.
4. The output of the software model will be compared with that of the DUT. If they do not match, an exception will be raised. The above steps will be repeated several times, following which the waveform will be checked.

Results: A one-to-one correspondence between the software implementation and the hardware implementation has been obtained. The number of clock cycles for all Conv core states except the **CS_IDLE** state is shown in the Table 4.2.

Table 4.2: Number of clock cycles for Conv core states

Conv core FSM state	Clock cycles
CS_LOAD	770
CS_CONV	995
CS_POOL	159
CS_FIN	1

4.3. Multi-core platform design

Now, all the cores as well as the classification module are in place, so we can build the multi-core platform for FPGA deployment. Here, we will use both the Xilinx Vivado design suite and the Vitis software platform, the former for synthesis of the hardware design, and the latter for hardware deployment, PS software programming as well as SoC platform debugging.

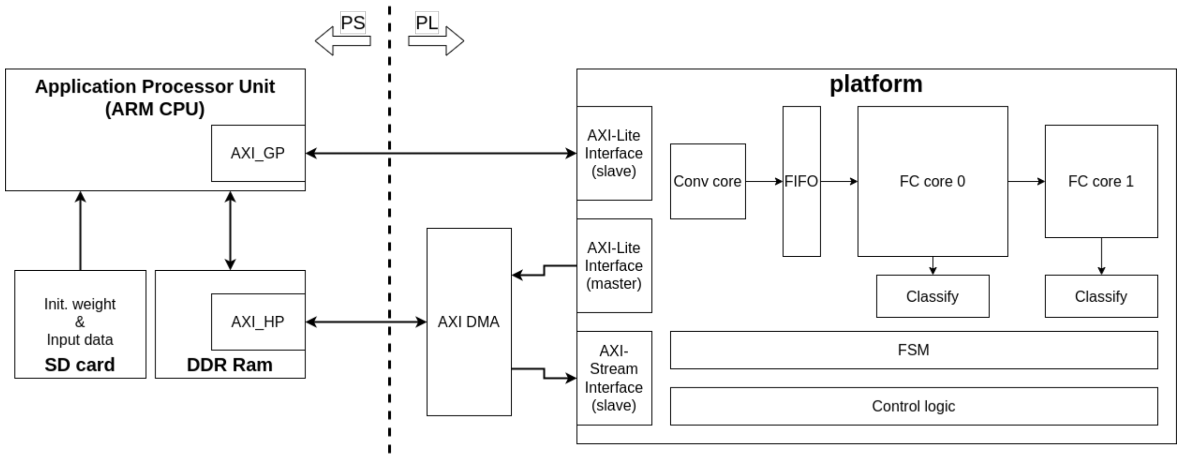

Figure 4.20: Block diagram of the multi-core platform and the SoC FPGA system design

Figure 4.20 shows the multi-core platform and the entire system design on the SoC FPGA platform. For the weight values required by the FC cores, as well as the CIFAR-10 dataset required for training, we will export them from PyTorch and store in the SD card. As the PL has direct access to the external DRAM, when the whole design is deployed on the FPGA board and booted, the ARM processor will move data from the SD card to the specified location in DRAM and subsequently start the multi-core platform.

To enable the platform to communicate with the PS, the multi-core platform is designed with three AXI interface modules, all of which are automatically generated using Vivado. Their respective uses are shown below:

- **AXI-Lite Interface (slave):** This interface corresponds to the slave side of the AXI_GP port of the PS. AXI_GP is a general-purpose AXI interface for the PS to exchange information with the PL, which means that software executing on the ARM CPU will have access to the multi-core platform through the AXI_GP port. The AXI-Lite protocol [88], a simplified version of the AXI protocol, is used here.
- **AXI-Lite Interface (master):** This interface corresponds to the master side of the AXI DMA. AXI DMA is the IP provided by Xilinx, which can help to access data in the DRAM. Compared to implementing our own logic for accessing the DRAM, we can now specify only the base address for DRAM access, and the number of bytes to access, to get the required data. This simplifies the implementation in the multi-core platform. The relevant DRAM access information will be sent to AXI DMA via the AXI-Lite protocol.
- **AXI-Stream Interface (slave):** This interface corresponds to the slave side of the AXI DMA. The data fetched from RAM by the DMA will be sent from its AXI-Stream master interface. So, in order to receive data, this AXI-Stream slave interface is included in our multi-core platform.

Figure 4.21 shows the state transition diagram corresponding to the finite state machine in the PL of the multi-core platform. The FSM will perform a state transition based on the instruction from the PS. The instruction set is the same as the one listed in Section 4.2.1. The list of the multi-core platform states and corresponding behaviors are listed as follows:

- **PL_IDLE:** Each core in this state receives the **I_IDLE** instruction. There will be no DRAM data access.
- **PL_INIT:** Each core in this state receives the **I_INIT** instruction. The platform will access the weights of FC cores stored in the DRAM and pass the values directly into the $weight$ port of FC cores. When both FC cores return to the **LS_IDLE** state, the platform state is automatically transferred to the **PL_IDLE** state.
- **PL_TRAIN:** Each core in this state receives the **I_TRAIN** instruction. The platform will access the images of the CIFAR-10 training set stored in the DRAM and pass the values to the $data\_in$ port of the Conv core. The classify module will be enabled to output classification results. When all the cores return to the **CS_IDLE** or **LS_IDLE** state, the platform state is automatically transferred to the **PL_IDLE** state.
- **PL_INF:** Each core in this state receives the **I_INF** instruction. The platform will access the images of the CIFAR-10 test set stored in the DRAM and pass the values to the $data\_in$ port of the Conv core. The rest of the operations are the same as in **PL_TRAIN**.

Just like the FSM in all cores, the **I_STOP** instruction from the PS enables the platform FSM to transfer to the **PL_IDLE** state from any state, and the FSM will send the **I_STOP** instruction to all cores at the same time.

Software design: The PS side is responsible for moving the data, controlling the multi-core platform and collecting the results. The steps to be performed by the program running on the ARM core are as follows:

1. Read the weights and the CIFAR-10 dataset from the SD card and write them to the specified address in the DRAM.
2. Send the initialization instruction to the multicore platform. Meanwhile, the program waits for the user to enter the model configuration and the number of training epochs.
3. Read the current state of the multi-core platform and wait for it to enter the **PL_IDLE** state.
4. Enter the loop. Calculate the learning rate based on the current epoch and incorporate the result into the model configuration value. Send the value along with the training instructions to the multi-core platform.

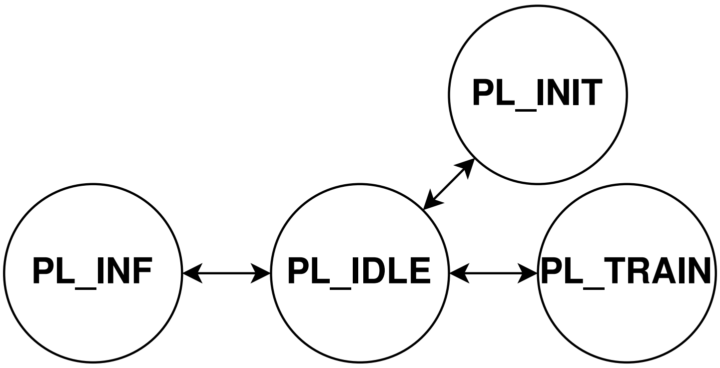

Figure 4.21: State transition diagram of FSM in multi-core platform. For clarity, the state transition conditions are given in the main text.

5. Read the platform state once per second, and read the running status after a certain time interval, or after the platform state returns to **PL_IDLE**. Print the training set results.
6. Send an inference command when the platform state is **PL_IDLE**, and wait for the test set results in the same way as in step 5.
7. If the current epoch does not reach the set value, go back to step 4 and continue training. Otherwise, the program is finished.

Similar to the simulation of the FC core part, we verified that the operationof the platform and state transitions are correct, and therefore do not perform one-to-one hardware-to-software comparison tests. The system-level operation will be verified at the implementation stage in Section 4.4.

Table 4.3 shows the number of clock cycles each core requires in a single iteration. One epoch trained on five images takes 757162 clock cycles.

Table 4.3: Number of clock cycles for each core during training

	Clock cycles
Conv core	28471
FC core 0	145718
FC core 1	30038

4.4. Performance evaluation

In this section, we will perform a complete performance evaluation of the hardware implementation. First we will compare the test set accuracy of hardware implementations and software models. We will then show some metrics of the hardware implementation on FPGA.

4.4.1. Test set accuracy

The results of the hardware will be compared with the results of the corresponding software model. In order to compare the performance of the hardware and software, the following items are set to be the same on hardware and software:

- The input order of the training set.
- The configuration of hyperparameters.
- The initial weights of all layers.

Thus, the difference between hardware and software lies in the way random numbers are generated.

Figure 4.22 shows the test set accuracy of the hardware and software for each of the 100 epochs of the training process.

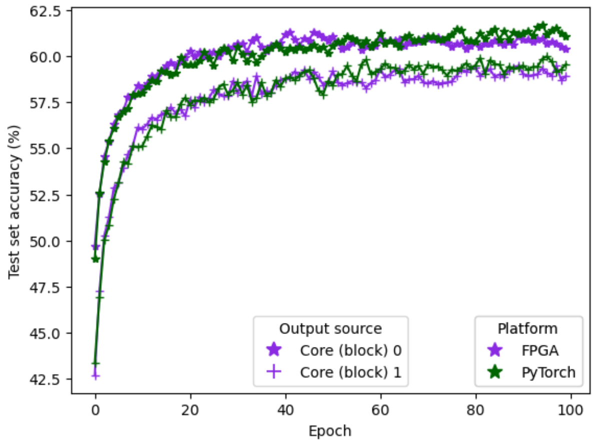

Figure 4.22: Performance comparison between FPGA and PyTorch implementations

Table 4.4: Test accuracy of different blocks in hardware and software

	Test set accuracy (hardware)	Test set accuracy (software)
FC core (block) 0	60.54%	61.29%
FC core (block) 1	59.03%	59.33%

Table 4.4 shows the final test set accuracy for FC core 0 as well as FC core 1 for both hardware and software. Data are derived from the average of the last 3 epochs.

We find that the performance of the hardware implementation is similar to that of the software implementation, however the performance of the second FC core is significantly lower than that of the first, for both hardware and software. In order to understand the possible reasons for this phenomenon, we performed an additional software model test and comparison.

The hardware-matched software model is trained using a batch size of 100. The data shown in Table 4.5 are averaged over the last three epochs of one trial. It can be seen that the performance of both FC blocks is significantly higher than the results reported for a batch size of 1 in Table 4.4, and that there is no significant difference between the performance of two blocks.

Table 4.5: Test accuracy of the FC cores of the hardware-identical model for a batch size of 100

	Test set accuracy
FC block 0	65.29%
FC block 1	65.44%

Based on results from the test above, we can find that the possible reason to affect the performance is the batch size during training.

4.4.2. Resource and timing

We used Vivado's default synthesis and implementation strategy for the hardware generation. The PL-side clock was set to a fixed 75 Mhz.

Resource overhead: Table 4.6 shows the resources of the multi-core platform. There are a total of 140 BRAM tiles on FPGA, and the table shows that 137.5 of them are already occupied, mainly due to the footprint of the weight matrix of FC core 0. Due to the fact that the BRAM resources are exhausted, part of the storage is realized directly by LUTs. Besides being used for storage, the main LUT overhead comes from the matmul module, the adder tree module, and the triout cache.

Table 4.6: Resource overhead of the main modules

Modules	Slice (Total slice resource consumption)	BRAM Tile (Total BRAM resource consumption)
Conv core	1188 (8.93%)	17 (12.14%)
FC core 0	1818 (13.67%)	99 (70.71%)
FC core 1	1500 (11.28%)	19 (13.57)
Multi-core platform	4868 (36.60%)	137.5 (98.21%)

Timing: Table 4.7 shows the timing of the entire implementation in PL. Combined with the clock cycles in Table 4.3 , it is known that a complete training on a single image will take about 2 milliseconds, and performing a complete training of one epoch will take about 1 minute and 40 seconds.

Table 4.7: Timing of all the implementations in PL

Clock signal frequency (MHz)	Worst negative slack (ns)	Worst hold slack (ns)
75	1.328	0.011

5 Conclusions

In this project, we first compared various state-of-the-art learning algorithms for ANNs and proposed an general learning framework that incorporates LEL and e-prop for spatiotemporal. However, as a first proof of concept, we decided to implement only the LEL learning algorithm on hardware and validate the possibility of multi-core learning for the CIFAR-10 image classification task. We then designed the quantization scheme as well as the building blocks suitable for the hardware and determined the software model through neural architecture search and designed the hardware blocks accordingly. We constructed our multi-core on-chip learning platform by integrating the submodules into separate cores. After deploying the platform on a FPGA board, we evaluated its performance. At the end, we draw the following conclusions about low-cost multicore on-chip learning:

1. LEL can solve the problem of update locking while mitigating the problem of weight symmetry by generating layer-wise errors locally. This makes LEL suitable for low-cost multi-core on-chip learning.
2. Through the design of the quantization scheme as well as the building blocks, we succeeded in constructing an efficient CNN network. When using 6-bit weights and a pre-trained convolutional kernel, it is able to have a test set accuracy of 64.47% with a storage overhead of about 2.07 megabits.
3. For the computation and storage requirements of the software model we have carried out targeted module design as well as simulation verification. Finally, the multi-core on-chip learning platform design deployed to a SoC FPGA can fully realize the learning function of the software model and converge to similar learning results.
4. During the performance evaluation, the multi-core on-chip learning platform is able to produce the expected performance in a complete training, demonstrating the feasibility of the LEL learning algorithm in hardware. On the other hand, the results also show that there is still room for improvements of the current design for online learning as well as low-cost design.

Overall, this work forms a key stepping stone towards low-cost multi-core spatiotemporal learning on-chip, so the natural next step is to include e-prop in the scheme.

References

[1] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[2] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[3] David Silver et al. "Mastering the game of go without human knowledge". In: *nature* 550.7676 (2017), pp. 354–359.

[4] David Silver and Demis Hassabis. *AlphaGo Zero: Starting from scratch*. Oct. 2017. URL: https://www.deepmind.com/blog/alphago-zero-starting-from-scratch.

[5] Charlotte Frenkel, David Bol, and Giacomo Indiveri. "Bottom-up and top-down neural processing systems design: Neuromorphic intelligence as the convergence of natural and artificial intelligence". In: *arXiv preprint arXiv:2106.01288* (2021).

[6] Jeongwoo Park, Juyun Lee, and Dongsuk Jeon. "A 65-nm Neuromorphic Image Classification Processor With Energy-Efficient Training Through Direct Spike-Only Feedback". In: *IEEE Journal of Solid-State Circuits* 55.1 (2020), pp. 108–119. DOI: 10.1109/JSSC.2019.2942367.

[7] Charlotte Frenkel, Jean-Didier Legat, and David Bol. "A 28-nm Convolutional Neuromorphic Processor Enabling Online Learning with Spike-Based Retinas". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9180440.

[8] Timothy P Lillicrap et al. "Backpropagation and the brain". In: *Nature Reviews Neuroscience* 21.6 (2020), pp. 335–346.

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

[10] Stephen Grossberg. "Competitive learning: From interactive activation to adaptive resonance". In: *Cognitive Science* 11.1 (1987), pp. 23–63. ISSN: 0364-0213. DOI: https://doi.org/10.1016/S0364-0213(87)80025-3. URL: https://www.sciencedirect.com/science/article/pii/S0364021387800253.

[11] Max Jaderberg et al. "Decoupled neural interfaces using synthetic gradients". In: *International conference on machine learning*. PMLR. 2017, pp. 1627–1635.

[12] Qianli Liao, Joel Leibo, and Tomaso Poggio. "How important is weight symmetry in backpropagation?" In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. 2016.

[13] Owen Marschall, Kyunghyun Cho, and Cristina Savin. "A unified framework of online learning algorithms for training recurrent neural networks". In: *Journal of machine learning research* (2020).

[14] Emre O Neftci, Hesham Mostafa, and Friedemann Zenke. "Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks". In: *IEEE Signal Processing Magazine* 36.6 (2019), pp. 51–63.

[15] Thomas Bohnstingl et al. "Online spatio-temporal learning in deep neural networks". In: *IEEE Transactions on Neural Networks and Learning Systems* (2022).

[16] Ronald J Williams and David Zipser. "A learning algorithm for continually running fully recurrent neural networks". In: *Neural computation* 1.2 (1989), pp. 270–280.

[17] Weisong Shi et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

[18] 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.

[19] Tianshi Chen et al. "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 269–284. ISBN: 9781450323055. DOI: `10.1145/2541940.2541967`. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/2541940.2541967`.

[20] Shijin Zhang et al. "Cambricon-X: An accelerator for sparse neural networks". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–12.

[21] Charlotte Frenkel, Martin Lefebvre, and David Bol. "Learning Without Feedback: Fixed Random Learning Signals Allow for Feedforward Training of Deep Neural Networks". In: *Frontiers in Neuroscience* 15 (2021). ISSN: 1662-453X. DOI: `10.3389/fnins.2021.629892`. URL: `https://www.frontiersin.org/articles/10.3389/fnins.2021.629892`.

[22] Timothy P Lillicrap et al. "Random synaptic feedback weights support error backpropagation for deep learning". In: *Nature communications* 7.1 (2016), pp. 1–10.

[23] Arild Nøkland. "Direct feedback alignment provides learning in deep neural networks". In: *Advances in neural information processing systems* 29 (2016).

[24] Wojciech Marian Czarnecki et al. "Understanding synthetic gradients and decoupled neural interfaces". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 904–912.

[25] Hesham Mostafa, Vishwajith Ramesh, and Gert Cauwenberghs. "Deep Supervised Learning Using Local Errors". In: *Frontiers in Neuroscience* 12 (2018). ISSN: 1662-453X. DOI: `10.3389/fnins.2018.00608`. URL: `https://www.frontiersin.org/articles/10.3389/fnins.2018.00608`.

[26] Zhouyuan Huo et al. "Decoupled Parallel Backpropagation with Convergence Guarantee". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 2098–2106. URL: `https://proceedings.mlr.press/v80/huo18a.html`.

[27] Yoshua Bengio. "How auto-encoders could provide credit assignment in deep networks via target propagation". In: *arXiv preprint arXiv:1407.7906* (2014).

[28] Dong-Hyun Lee et al. "Difference target propagation". In: *Joint european conference on machine learning and knowledge discovery in databases*. Springer. 2015, pp. 498–515.

[29] Alexander Meulemans et al. "A theoretical framework for target propagation". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 20024–20036.

[30] Guillaume Bellec et al. "A solution to the learning dilemma for recurrent networks of spiking neurons". In: *Nature Communications* 11.1 (July 2020), p. 3625. ISSN: 2041-1723. DOI: `10.1038/s41467-020-17236-y`. URL: `https://doi.org/10.1038/s41467-020-17236-y`.

[31] Sho Yagishita et al. "A critical time window for dopamine actions on the structural plasticity of dendritic spines". In: *Science* 345.6204 (2014), pp. 1616–1620.

[32] Wulfram Gerstner et al. "Eligibility traces and plasticity on behavioral time scales: experimental support of neohebbian three-factor learning rules". In: *Frontiers in neural circuits* 12 (2018), p. 53.

[33] Amirsaman Sajad, David C Godlove, and Jeffrey D Schall. "Cortical microcircuitry of performance monitoring". In: *Nature neuroscience* 22.2 (2019), pp. 265–274.

[34] Ben Engelhard et al. "Specialized coding of sensory, motor and cognitive variables in VTA dopamine neurons". In: *Nature* 570.7762 (2019), pp. 509–513.

[35] Jochen Roeper. "Dissecting the diversity of midbrain dopamine neurons". In: *Trends in neurosciences* 36.6 (2013), pp. 336–342.

[36] Bojian Yin, Federico Corradi, and Sander M Bohte. "Accurate online training of dynamical spiking neural networks through Forward Propagation Through Time". In: *arXiv preprint arXiv:2112.11231* (2021).

[37] Anil Kag and Venkatesh Saligrama. "Training recurrent neural networks via forward propagation through time". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 5189–5200.

[38] Alex Krizhevsky, Geoffrey Hinton, et al. “Learning multiple layers of features from tiny images”. In: (2009).

[39] Maxence M Ernoult et al. “Towards scaling difference target propagation by learning backprop targets”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 5968–5987.

[40] Julien Launay et al. “Direct feedback alignment scales to modern deep learning tasks and architectures”. In: *Advances in neural information processing systems* 33 (2020), pp. 9346–9360.

[41] L. LAPIQUE. “Recherches quantitatives sur l’ excitation electrique des nerfs traitee comme une polarization.” In: *Journal of Physiology and Patholoġy* 9 (1907), pp. 620–635.

[42] Alan L Hodgkin and Andrew F Huxley. “A quantitative description of membrane current and its application to conduction and excitation in nerve”. In: *The Journal of physiology* 117.4 (1952), p. 500.

[43] Wulfram Gerstner. “Time structure of the activity in neural network models”. In: *Physical review E* 51.1 (1995), p. 738.

[44] Eugene M Izhikevich. “Simple model of spiking neurons”. In: *IEEE Transactions on neural networks* 14.6 (2003), pp. 1569–1572.

[45] Romain Brette and Wulfram Gerstner. “Adaptive exponential integrate-and-fire model as an effective description of neuronal activity”. In: *Journal of neurophysiology* 94.5 (2005), pp. 3637–3642.

[46] Wulfram Gerstner et al. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

[47] Simon Thorpe, Arnaud Delorme, and Rufin Van Rullen. “Spike-based strategies for rapid processing”. In: *Neural networks* 14.6-7 (2001), pp. 715–725.

[48] Emre O. Neftci et al. “Event-Driven Random Back-Propagation: Enabling Neuromorphic Deep Learning Machines”. In: *Frontiers in Neuroscience* 11 (2017). ISSN: 1662-453X. DOI: `10.3389/fnins.2017.00324`. URL: `https://www.frontiersin.org/articles/10.3389/fnins.2017.00324`.

[49] Sumit B Shrestha and Garrick Orchard. “Slayer: Spike layer error reassignment in time”. In: *Advances in neural information processing systems* 31 (2018).

[50] Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002.

[51] Jacques Kaiser, Hesham Mostafa, and Emre Neftci. “Synaptic Plasticity Dynamics for Deep Continuous Local Learning (DECOLLE)”. In: *Frontiers in Neuroscience* 14 (2020). ISSN: 1662-453X. DOI: `10.3389/fnins.2020.00424`. URL: `https://www.frontiersin.org/articles/10.3389/fnins.2020.00424`.

[52] Julian Göltz et al. “Fast and energy-efficient neuromorphic deep learning with first-spike times”. In: *Nature machine intelligence* 3.9 (2021), pp. 823–835.

[53] Bodo Rueckauer and Shih-Chii Liu. “Conversion of analog to spiking neural networks using sparse temporal coding”. In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2018, pp. 1–5. DOI: `10.1109/ISCAS.2018.8351295`.

[54] Seongsik Park et al. “T2FSNN: Deep spiking neural networks with time-to-first-spike coding”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2020, pp. 1–6.

[55] Yukuan Yang et al. “Training high-performance and large-scale deep neural networks with full 8-bit integers”. In: *Neural Networks* 125 (2020), pp. 70–82.

[56] Lei Deng et al. “Model compression and hardware acceleration for neural networks: A comprehensive survey”. In: *Proceedings of the IEEE* 108.4 (2020), pp. 485–532.

[57] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[58] Raghuraman Krishnamoorthi. “Quantizing deep convolutional networks for efficient inference: A whitepaper”. In: *arXiv preprint arXiv:1806.08342* (2018).

[59] *Quantization — PyTorch 2.0 documentation*. URL: `https://pytorch.org/docs/stable/quantization.html`.

[60] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. “Estimating or propagating gradients through stochastic neurons for conditional computation”. In: *arXiv preprint arXiv:1308.3432* (2013).

[61] Tailin Liang et al. “Pruning and quantization for deep neural network acceleration: A survey”. In: *Neurocomputing* 461 (2021), pp. 370–403.

[62] Shuang Wu et al. “Training and inference with integers in deep neural networks”. In: *arXiv preprint arXiv:1802.04680* (2018).

[63] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[64] Shuchang Zhou et al. “Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients”. In: *arXiv preprint arXiv:1606.06160* (2016).

[65] Matteo Croci et al. “Stochastic rounding: implementation, error analysis and applications”. In: *Royal Society Open Science* 9.3 (2022), p. 211631.

[66] Charlotte Frenkel and Giacomo Indiveri. “ReckOn: A 28nm sub-mm2 task-agnostic spiking recurrent neural network processor enabling on-chip learning over second-long timescales”. In: *2022 IEEE International Solid-State Circuits Conference (ISSCC)*. Vol. 65. IEEE. 2022, pp. 1–3.

[67] Aston Zhang et al. “Dive into Deep Learning”. In: *arXiv preprint arXiv:2106.11342* (2021).

[68] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

[69] Yuxin Wu and Kaiming He. “Group normalization”. In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 3–19.

[70] Vinod Nair and Geoffrey E Hinton. “Rectified linear units improve restricted boltzmann machines”. In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[71] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “Binaryconnect: Training deep neural networks with binary weights during propagations”. In: *Advances in neural information processing systems* 28 (2015).

[72] Lu Lu et al. “Dying relu and initialization: Theory and numerical examples”. In: *arXiv preprint arXiv:1903.06733* (2019).

[73] Tijmen Tieleman, Geoffrey Hinton, et al. “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude”. In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[74] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[75] Tyler Yep. *torchinfo*. Mar. 2020. URL: `https://github.com/TylerYep/torchinfo`.

[76] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: `https://www.wandb.com/`.

[77] Stefan Falkner, Aaron Klein, and Frank Hutter. “BOHB: Robust and efficient hyperparameter optimization at scale”. In: *International conference on machine learning*. PMLR. 2018, pp. 1437–1446.

[78] Lisha Li et al. “Hyperband: A novel bandit-based approach to hyperparameter optimization”. In: *The journal of machine learning research* 18.1 (2017), pp. 6765–6816.

[79] Stuart Hodgson and Chris Higgs. *cocotb*. 2014. URL: `https://www.cocotb.org/`.

[80] Stephen Williams and Michael Baxter. "Icarus verilog: open-source verilog more than a year later". In: *Linux Journal* 2002.99 (2002), p. 3.

[81] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[82] "High-speed parallel CRC implementation based on unfolding, pipelining, and retiming". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 53.10 (2006), pp. 1017–1021.

[83] Keshab K Parhi. *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons, 2007.

[84] Charlotte Frenkel, Jean-Didier Legat, and David Bol. "MorphIC: A 65-nm 738k-Synapse/mm ^2 quad-core binary-weight digital neuromorphic processor with stochastic spike-driven online learning". In: *IEEE transactions on biomedical circuits and systems* 13.5 (2019), pp. 999–1010.

[85] Matt Hostetter. *Galois: A performant NumPy extension for Galois fields*. Nov. 2020. URL: https://github.com/mhostetter/galois.

[86] Xilinx. *Zynq-7000 SoC Data Sheet: Overview (DS190)*. 2018. URL: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview.

[87] Arm Limited or its affiliates. *AMBA AXI-Stream Protocol Specification*. 2021. URL: https://developer.arm.com/documentation/ihi0051/latest/.

[88] Arm Limited or its affiliates. *AMBA AXI Protocol Specification*. 2023. URL: https://developer.arm.com/documentation/ihi0022/latest.