Journal of
*Low Power Electronics and Applications*

MDPI

*Article*

# Reinforcement-Learning-Based Synthesis of Custom Approximate Parallel Prefix Adders

**Apostolos Stefanidis and Giorgos Dimitrakopoulos ***

Electrical and Computer Engineering, Democritus University of Thrace, 67100 Xanthi, Greece; apstefan@ee.duth.gr
* Correspondence: dimitrak@ee.duth.gr

**Abstract:** Approximate hardware units compute a sufficiently accurate result rather than a fully accurate one using fewer transistors or equivalently logic gates than their accurate counterparts. This approach significantly saves energy while maintaining acceptable application-level quality. In this work, we focus on synthesizing custom approximate parallel prefix adders tailored to specific applications. The introduced reinforcement learning (RL) framework co-optimizes application performance and hardware complexity. An RL agent learns approximate addition strategies by exploring the entire design space and receiving feedback from hardware synthesis and application performance. Experimental results demonstrate that the synthesized adders can reduce area and power consumption by 12% and 10%, respectively, on average without compromising the error behavior of the application, or can significantly improve the application's error metrics without degrading area and power consumption for a variety of practical applications.

**Keywords:** approximate adders; reinforcement learning; synthesis; energy efficiency



**Citation:** Stefanidis, A.; Dimitrakopoulos, G. Reinforcement-Learning-Based Synthesis of Custom Approximate Parallel Prefix Adders. *J. Low Power Electron. Appl.* **2024**, *14*, 57. https://doi.org/10.3390/jlpea14040057

Academic Editor: Mansun Chan

Received: 23 October 2024
Revised: 30 November 2024
Accepted: 4 December 2024
Published: 6 December 2024



## 1. Introduction

Approximate computing that sacrifices accuracy in computations for energy savings has emerged as a promising approach for addressing the growing energy demands of modern computing systems [1]. By intentionally introducing errors into hardware components, approximate computing techniques aim to reduce power consumption without significantly impacting overall system performance. A variety of error-resilient applications make use of approximate computing for area and power reduction, such as image processing [2], FFT hardware components [3], clock generators [4] and neural networks [5].

One of the key challenges in designing approximate hardware lies in identifying the optimal trade-offs between accuracy and energy efficiency. In this paper, we explore the design space of approximate adders. Instead of focusing on a specific adder architecture, we design approximate parallel prefix adders that effectively represent a wide range of adder architectures that span from serial ripple-carry adders to fast carry-lookahead adders [6]. Optimizing the structure of parallel prefix adders is a complex task that has been extensively studied in the past for accurate parallel prefix adders [7–10].

By automating the synthesis of approximate parallel prefix adders tailored to specific applications, we aim to provide a flexible and efficient approach to harnessing the benefits of approximate computing.

Previous research on approximate parallel prefix adders has focused on various techniques to reduce their complexity and energy consumption. These approaches include shortening the carry chains [11,12], pruning lower levels of the adder [13], and removing logic gates from specific cells [14]. While these methods have yielded promising results, they often rely on predefined architectures, limiting the exploration of the entire design space. Recent work [15] has sought to address this limitation by proposing a more comprehensive approach to the design of approximate parallel prefix adders.

The work in [15] enumerates all possible approximate parallel prefix adders that satisfy the designer's constraints, such as carry chain length and the number of prefix levels. This exhaustive enumeration led to the discovery of highly efficient approximate adder designs. However, even if we can identify all possible approximate adders for a given set of constraints, there is no clear way to determine which one is the best for a specific application. Not all approximate adder architectures work well for every problem. Therefore, to approach optimal results, approximate adders should be customized for each application.

In this work, approximate parallel prefix adder customization is performed using reinforcement learning (RL) [16,17]. RL optimizes the hardware's area and, at the same time, minimizes the approximation error at the application level by incorporating appropriate feedback during learning. The evaluation of solutions during learning does not rely on heuristics or abstract performance prediction metrics but uses a neural network that learns strategies purely through exploration of the unexplored design space and direct feedback from logic synthesis and application-level performance [17]. In this way, for each examined application, we can automatically synthesize bespoke approximate parallel prefix adders without the need to rely on generic architectures that may perform poorly on certain occasions. Overall, the novelty of the proposed approach can be summarized as follows:

- The introduced RL agent learns how to synthesize adders on a per-application basis, satisfying certain application-level quality metrics and hardware complexity constraints. Effectively, the RL framework identifies the specific approximation strategy that maps to energy-efficient hardware and offers the quality needed by the application and the associated user constraints.
- The RL agent is able to adapt its approximation strategy to the input data that the application uses, taking into consideration both signedness and data range, resulting in even more efficient adder architectures.
- The proposed approach can generate various approximate adders that outperform existing designs. These adders have been rigorously tested using both synthetic data and real-world applications. In all cases, they are superior or at least equally efficient to state-of-the-art parallel prefix adders by either improving the area and power consumption by 12% and 10% on average, respectively, without degrading the application performance, or by significantly improving the application level error behavior without increasing the area and power consumption.

The rest of this paper is organized as follows: Background on accurate and approximate parallel prefix adders and state-of-the-art approaches is presented in Section 2. The generic strategy for approximate addition followed by the proposed approach is described in Section 3. The RL framework that optimizes approximate parallel prefix adder design for each application is described in Section 4. Experimental results are presented in Section 5. Additional related work on approximate addition is analyzed in Section 6. Finally, conclusions are drawn in the last section.

## 2. Background on Accurate and Approximate Parallel Prefix Adders

Adding two $N$-bit binary numbers $A = A_{N-1}A_{N-2}\dots A_0$ and $B = B_{N-1}B_{N-2}\dots B_0$, the sum bit $S_i$ at the $i$-th bit position is computed via $A_i$, $B_i$ and the carry $C_{i-1}$ computed in the previous bit position:

$$S_i = A_i \oplus B_i \oplus C_{i-1}. \tag{1}$$

The carry out of the $i$-th bit position $C_i$ is computed using the local carry generate and propagate bits $G_i = A_i B_i$ (AND) and $P_i = A_i + B_i$ (OR) and the recursive carry propagation formula

$$C_i = G_i + P_i C_{i-1}. \tag{2}$$

The carry-in signal is assumed to be equal to $C_{-1}$.

Carry computation is transformed to a prefix problem [18] by using the associative operator ∘, which associates pairs of generate and propagate bits as follows:

$$(G, P) \circ (G', P') = (G + P\,G', P\,P').$$

$(G_{k:j}, P_{k:j})$ denotes the group generate and propagate term produced out of bits $k, k-1, \ldots, j$ using a series of associations with the ∘ operator:

$$(G_{k:j}, P_{k:j}) = (G_k, P_k) \circ (G_{k-1}, P_{k-1}) \circ \ldots \circ (G_j, P_j). \quad (3)$$

When excluding the carry-in signal, each carry $C_i$ corresponds to $G_{i:0}$. This condition for carry $C_i$ makes the adder *accurate*. When carry-in is included, an *accurate* $C_i$ should correspond to the group generate term $G_{i:-1}$. Effectively, adding a carry-in signal is equivalent to expanding the bit positions of the adder by an extra least-significant bit [6].

The carry calculation of a parallel prefix adder can be pictured as a graph of prefix associative operators. For example, on the adder of Figure 1, we can see a 16-bit prefix adder, where each black node represents an associative operator ∘. The adder is fully accurate, so each bit position has a carry chain looking back until bit position 0.

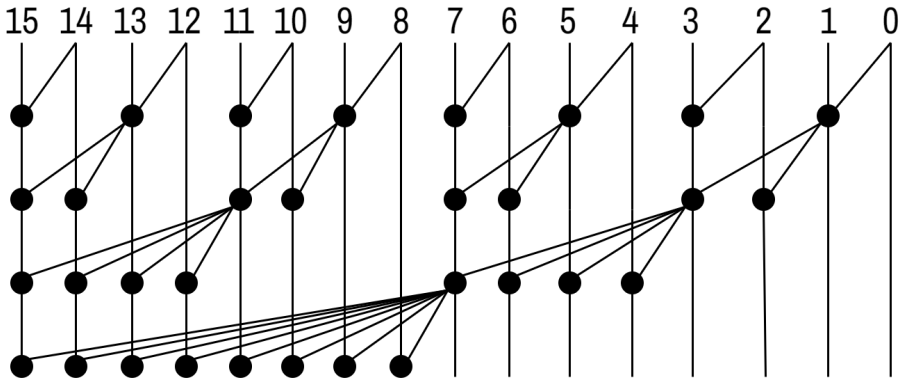



**Figure 1.** An example of an accurate 16-bit parallel prefix adder. Each black dot represents a prefix operator ∘.

In parallel prefix adders, approximation can take many different forms, such as replacing some complex logic gates with simpler ones, removing operators from the lower significance bits [19], pruning entire logic levels [12], or even performing transistor-level pruning [20,21]. For the purpose of this study, approximate carry computation allows parallel prefix adders to use group generate terms of shorter length, such as $G_{i:m}$, with $m > 0$, instead of the full group generate term $G_{i:0}$.

Following [12], approximate parallel prefix adders can be designed by removing prefix levels from equal-bit-width accurate parallel prefix adders and modifying the remaining ones. The choice of prefix level removal is carried out based on the desired minimum and maximum carry chain lengths, as well as the length of each approximate carry propagation segment. For example, an adder resulting from the pruning of levels of a Han Carlson adder is shown in Figure 2a, with minimum and maximum carry chains of 8 and 9 bits, respectively. This approximate 16-bit adder requires four fewer ∘ operators than the accurate adder of Figure 1.

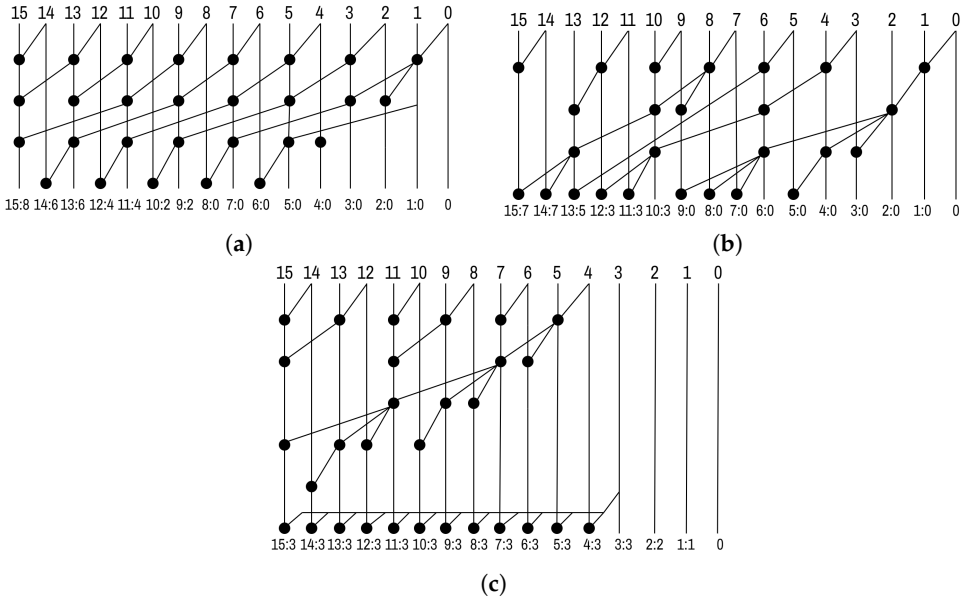



**Figure 2.** Examples of state-of-the-art approximate parallel prefix adders. (**a**) Approximate adder proposed in [12]. (**b**) Approximate adder proposed in [15]. (**c**) Approximate adder proposed in [19].

The work in [15] enumerates all possible approximate parallel prefix adders given a set of input design constraints, such as the desired logic level and the maximum fan-out, as well as the minimum allowed carry chain for all output bits. For instance, the adder in Figure 2b can be generated using the following constraints: 4 maximum prefix levels, a maximum fan-out of 3 and a minimum carry chain of 8 bit positions.

In [19], the designers split the adder into two parts, as can be seen in Figure 2c. In the lower-significance part, all operators are removed. The higher-significance part is implemented as a Ladner–Fischer adder, where all carry chains look back to the bit where the splitting happened. There is a final row of operators that extend the carry chain of the higher-significance part by 1. In the example of Figure 2c, the adder is split on bit 4.

The method presented in [15] serves as a versatile tool capable of generating any required approximate parallel prefix structure. In this study, we leverage the approximate adder generator from [15] to create custom-designed approximate parallel prefix adders tailored to the specific demands of our applications. This approach departs from the abstract hardware-centric constraints, such as prefix levels and fan-out, employed in [15], which may not be tightly correlated with actual application performance.

## 3. Generic Architecture for Approximate Parallel Prefix Adders

To be able to automatically synthesize the approximate parallel prefix adders that best fit the application domains, we need to first define a unified representation for the carry computation approximation implemented by the adder. The representation should be versatile enough to cover a wide range of approximations and simple enough so that it can be used in an RL-based optimization framework. Actually, the chosen representation for the approximation used in each adder is a mix of the architecture of generic configurable adders [15] and the adders of [12].

### *3.1. Representation of Carry Computation Approximation*

The output bits of a parallel prefix adder are separated into $B + 1$ segments. The first segment *always* begins from bit position 0 and involves the $L$ least significant bits of the adder. The rest of the $B$ segments split the $N - L$ most significant output bits of the adder into equal size groups of $b = \frac{N-L}{B}$ bits each. When $N - L$ and $B$ are not divided without a remainder, the most significant segment may involve fewer than $b$ bits.

An example of the correspondence of output segments to a 16-bit parallel prefix adder is shown graphically in Figure 3. Each adder of Figure 3 is split into four segments, i.e., $B = 3$. In both cases, the $L = 7$ least significant bits form the first segment, and the other 9 most significant bits are separated to $B = 3$ segments of $b = 3$ bits each.

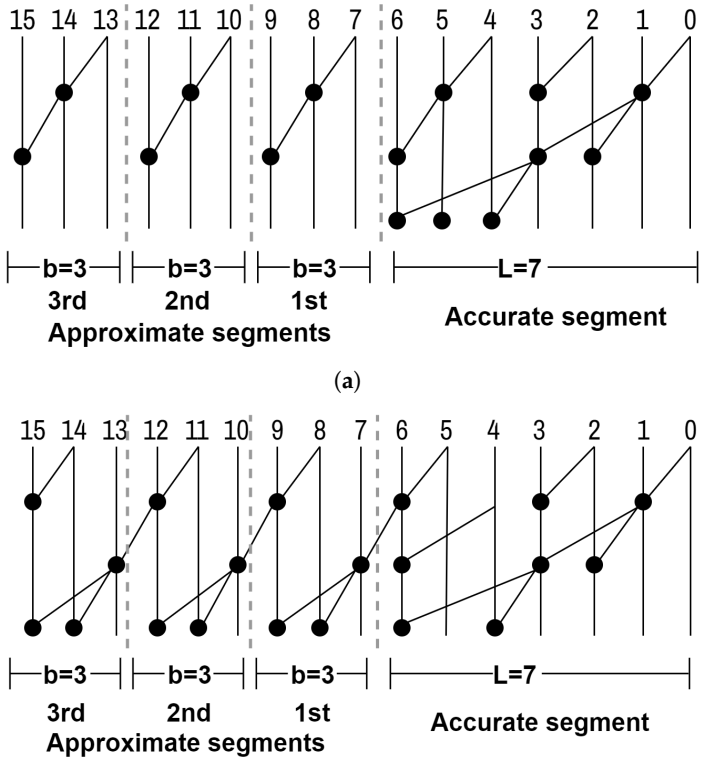



**Figure 3.** Two examples of the proposed generic carry-computation approximation. Both adders have an accurate segment of $L = 7$ bits, and three approximate segments of $B = 3$ bits each. In case (**a**), there is no overlap across approximate segments, whereas in case (**b**), there is an overlap of two bit positions.

The segment of the $L$ least significant bits is called the accurate part of the adder, since, in this part, the output carries and, in effect, the sum bits are computed accurately. The rest of the $B$ segments involve approximate carry computation. For the bits that belong to approximate segments, carry propagation does not reach bit position 0 but stops at a higher bit position.

Carry propagation inside the introduced segments may overlap or not. In Figure 3a, the four defined segments are fully isolated and carry propagation of one segment begins after the bit position that the previous segment finishes. On the contrary, in Figure 3b, carry propagation across segments overlaps. For instance, the carry computed at position 11 that belongs to the second approximate segment goes back to bits of the first approximate segment as well. This feature is controlled by the third variable of the unified approximate representation used in this work and is called overlap. For the example shown in Figure 3b, overlap = 2. Please note that overlap allows for the extension of the carry propagation of an approximate segment inside the fully accurate one. The adder of Figure 3a exhibits zero overlap in its approximate segments.

Formally, the start $s_i$ and end $e_i$ bit position of the $i$th approximate segment with $B > i > 0$ and $s_i \geq e_i$ are calculated as follows:

$$s_i = i\,b + L - 1 \tag{4}$$

$$e_i = (i-1)b + L \tag{5}$$

Carry computation at the $i$th approximate segment involves the bits from $s_i$ to $e_i - overlap$. This means that each bit in the segment will have a carry chain until the end of its segment, plus the overlap bits.

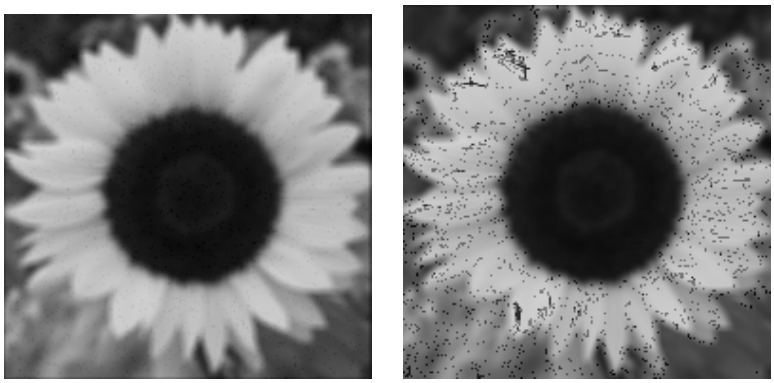
This unified representation covers also already-known approximate adders that are shown in Figure 2. The adder of Figure 2a is built using the set of constraints $(L = 9, B = 4, overlap = 7)$, while the adder of Figure 2b is characterized by the following set of parameters $(L = 10, B = 3, overlap = 7)$. The enumerative approach of [15] can even produce approximate adders with unequal approximate segments or even unequal overlap regions. However, the overall hardware complexity of the derived adders is indistinguishable to the ones produced by the proposed representation that utilizes uniform approximate segments.

### 3.2. Approximation Structure and Application Performance

Deciding which set of parameters $(L, B, overlap)$ satisfies the error quality metrics for each application and still leads to a lower hardware area and power relative to accurate adders is not an easy task.

To give a first example, in Figure 4, we compare the application-level performance of two different approximate 16-bit adders for mean filtering. The first adder is built according to the set of parameters $(L = 7, B = 2, overlap = 6)$, while the second uses $(L = 10, B = 1, overlap = 6)$.

As shown in Figure 4, the first approximate adder greatly outperforms the second one for mean filtering with respect to peak signal-to-noise ratio (PSNR).



(**a**) PSNR = 41.30 (**b**) PSNR = 22.07

**Figure 4.** The performance of two approximate adder architectures for mean filtering. The adder used in case (**a**) performs much better than the approximate adder used in case (**b**).

However, if we change the application to Laplacian filtering as performed in Figure 5, the outcome is reversed. The PSNR of the images after Laplacian filtering shown in Figure 5 proves that the second adder is more efficient than the first one.

These two contradictory examples show that customizing approximate addition based on the application characteristics is the only way to yield optimal results. Moreover, it shows the difficulty of predicting this effect: the two adders have the same carry overlap, with the second one having a lengthier accurate segment and one less approximate segment, making it hard to select one of the two as the predominantly best adder.

(**a**) PSNR = 25.73 (**b**) PSNR = 32.97

**Figure 5.** The performance of the same adders used for mean filtering in Laplacian filtering. The second adder that lags in mean filtering performs better in Laplacian filtering as shown by the PSNR of case (**b**) relative to case (**a**).

## 4. Reinforcement Learning Framework for the Synthesis of Approximate Parallel Prefix Adders

Selecting the approximation structure and the way that it will be implemented is not an easy task, since it is application-dependent and small changes in this configuration can lead to a very different quality of results. For this reason, we utilize reinforcement learning as a way of exploring the state space and finding high-quality solutions.

The designer needs to implement an application in hardware, and for this reason, decides to use approximate parallel prefix adders in their implementation, with the goal of saving area and power without affecting the quality of results of the selected application. To achieve this goal, the designer sets the required constraints such as (a) the bit width of the adder, (b) the application to optimize for and (c) constraints related to the error behavior, as well as (d) the maximum delay, area and maximum fan-out of the desired adder. Then, the RL agent starts training from scratch, with the goal of providing the user with an area-efficient adder that fits those criteria.

The overall flow of the proposed approach is shown in Figure 6. The RL agent shown on the left takes actions to select the set of variables $(L, B, overlap)$ that determine the approximation architecture of each adder. As shown in the 'Environment' part of Figure 6, for each selected set of parameters $(L, B, overlap)$, the approximate adder generator of [15] is used to generate the corresponding approximate parallel prefix adder. The method of [15] returns multiple adders that satisfy the given parameters. In each step, we select the adder with the smallest number of prefix operators that performs addition to the minimum required prefix levels (i.e., $\log_2 N$). The hardware complexity of the generated approximate parallel prefix adder is quantified after logic synthesis. In parallel, the application-level error performance is also quantified using simulation with a representative set of input data. The resulting area, delay and error are then used to reward the RL agent and drive it toward optimal solutions.

Formally, the RL problem includes a state space $S$, an action space $A$ and a reward function $R$, where applying action $a \in A$ on state $s \in S$ will take the environment in the next state $s'$ with a reward of $r = R(s, a)$ [16]. The process is carried out over a set of discrete timesteps $t = 0, 1, 2\ldots$, where, for each timestep, the agent observes $s_t$ and selects $a_t$ based on some action selection policy $\pi(a_t|s_t)$, with each value representing the probability of $a_t$ being chosen for state $s_t$, until a terminal state is reached. This series of actions from the starting to the ending state is called an episode. The goal of the agent is to learn the policy that maximizes the cumulative future rewards, meaning the sum of the rewards from the starting to the ending state of an episode.

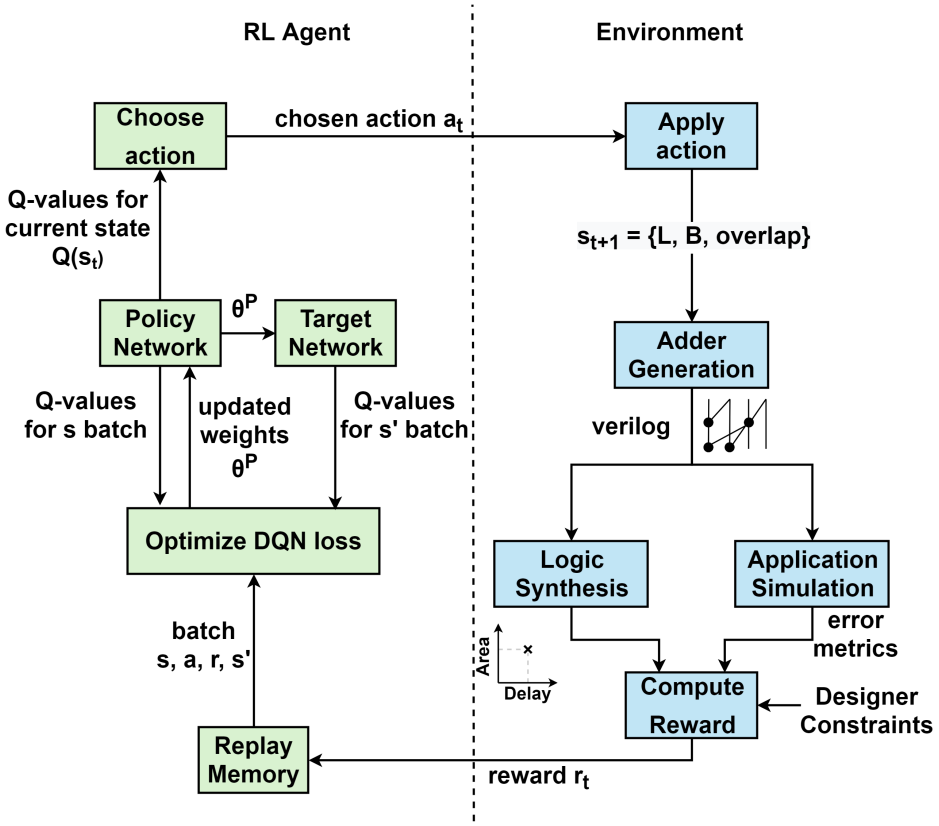



**Figure 6.** RL framework for the customization of approximate parallel prefix adder design. The RL agent chooses an action that modifies the structure of approximation during carry computation. For this structure, an approximate parallel prefix adder is synthesized and its hardware complexity and application performance is quantified. The resulting performance is transformed into a reward, according to the designer's constraints, that drives RL training.

### *4.1. State Representation and Actions*

The state $s_t$ at time step $t$ of each approximate adder is a vector of three integers $s_t = \{L, B, overlap\}$, i.e., the length $L$ of the accurate segment of the adder, the number of approximate segments $B$ and the overlap between the segments $overlap$.

For each state representing an approximate adder, there are six possible actions: increase/decrease $L$, increase/decrease $B$ and increase/decrease $overlap$. Not all of these actions are valid in each state since some of them can lead to invalid states. For example, if there are zero approximate segments, $B$ cannot be reduced since this would lead to the contradicting state of an approximate adder without approximate segments. In each step, the RL agent chooses a valid action to apply to the current state and move on to the next state.

### *4.2. Reward*

After each action is taken and its outcome is evaluated via logic synthesis and simulation as shown in Figure 6, the agent must be provided with a reward that should reflect how well the new action performed. This reward should consider how the resulting design compares to the one produced in the previous time step, while also accounting for how closely the design adheres to the original design constraints.

The reward $r_t$ after taking action $a_t$ and leading to a synthesized adder that corresponds to state $s_{t+1}$ is equal to

$$r_t = a\,\Delta_{\text{delay}} + b\,\Delta_{\text{area}} + c\,\Delta_{\text{error}}, \tag{6}$$

and its computation is illustrated in Figure 7. The differences in hardware delay and area, $\Delta_{\text{delay}} = \text{delay}_{t+1} - \text{delay}_t$ and $\Delta_{\text{area}} = \text{area}_{t+1} - \text{area}_t$, as well as application-level error

$\Delta_{\text{error}} = \text{error}_{t+1} - \text{error}_t$, correspond to the differences between the corresponding metric at the current and the previous timestep. Parameters $a$, $b$ and $c$ are used to change the focus of the optimization process, as well as to normalize the values to comparable units. In our experiments, we set them to 0.15, 0.05 and 0.8, respectively, prioritizing timing closure over area savings and giving the greatest importance to satisfying the error constraints.

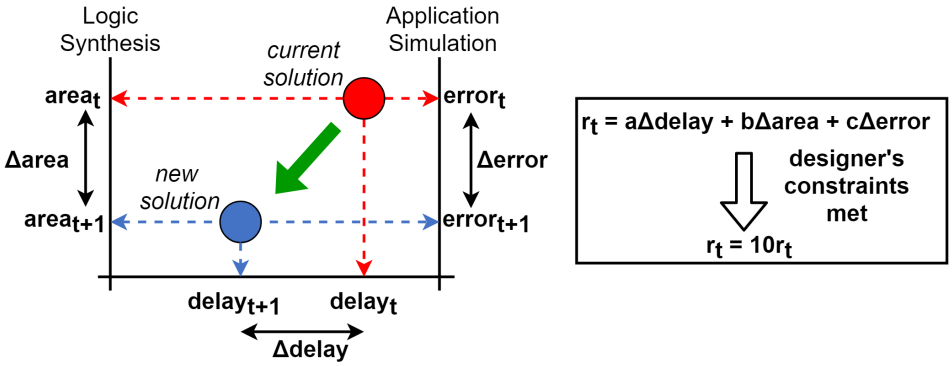



**Figure 7.** Visualization of the reward computation for transitioning from state $s_t$ (current solution) to state $s_{t+1}$ (new solution). After logic synthesis and application simulation is executed, the area, delay and error differences between the two states are used to compute the reward for this time step, which is then multiplied by a constant number if the new state is an adder that satisfies the designer's constraints.

To further ensure that the agent does not try to optimize a metric more than the constraint that the designer has set, the $\Delta$ difference of a constraint that is already satisfied is set to 0. For example, once the hardware has reached the delay target set by the designer, there will not be any additional delay reward unless the agent regresses and increases the delay past the designer's target.

Finally, the computed reward $r_t$ is multiplied by a high constant value, in our experiments, set to 10, if an adder that satisfies all the designer's constraints is reached:

$$r_t = 10r_t, \text{ if } s_{t+1} \text{ satisfies all designer's constraints} \tag{7}$$

This will assist the agent in avoiding being stuck in a local optima by only taking actions with a high immediate reward.

### *4.3. Deep Q-Network Algorithm*

In this work, we utilize an efficient and widely used reinforcement learning algorithm, the deep $Q$-network (DQN) [17]. The main idea of the DQN algorithm is training a neural network that takes the current state as input and outputs the $Q$-value of each possible action. By taking the action with the maximum predicted $Q$-value in each state at the end of training, the agent should be able to reach a high-quality final solution. The basic blocks of a DQN agent can be seen in the left part of Figure 6.

The policy network is a neural network that takes a representation of present state $s_t$ as an input and outputs the estimated $Q$-values for each possible action at this state. The $Q$-value of a state–action pair quantifies the expected cumulative reward for taking action $a_t$ at state $s_t$ and then taking the action with the highest $Q$-value until the end of the episode, and it is defined as

$$Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a)), \tag{8}$$

where $\gamma$ is a hyperparameter between 0 and 1 and is used to discount future rewards.

These $Q$-values are used by the agent when making a decision on the next action. The agent will choose either the action with the maximum $Q$-value in its current state or a

random action, based on a linearly declining variable $\epsilon$, which denotes the probability of choosing a random action.

The structure of the policy network used in this framework is shown in Figure 8. It consists of a fully connected neural network layer of size $3 \times 128$ that takes as inputs the state variables $\{L, B, overlap\}$, an RELU activation and a second fully connected layer of size $128 \times 6$, with the Q-values of the six possible actions.

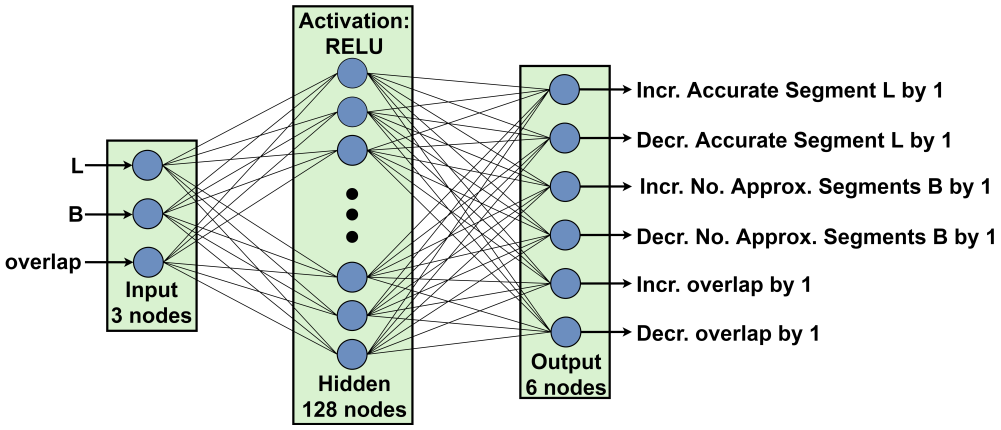



**Figure 8.** Structure of the policy and target network. It consists of an input layer of 3 neurons, a fully connected $3 \times 128$ hidden layer using an RELU activation function and a fully connected $128 \times 6$ output layer that decides the actions of the next round.

The target network is an identical copy of the policy network but with frozen weights, helping to ensure the stability of the DQN method. As seen in Equation (8), both the estimated current $Q$-value ($Q(s_t, a_t)$) and the estimated target value ($\max_a Q(s_{t+1}, a)$ are computed from the same network. This can lead to instability, since the network would be trying to converge toward a moving target, because updating its weights can potentially change both the current and the target $Q$-value. To address this, the target $Q$-value comes from the target network, which provides the policy network with a more fixed target. After a certain number of training steps, the policy network's weights are copied to the target network so that the target is up to date.

The replay memory stores the current state, action taken, next state and reward observed by applying this action to reuse at a later time for learning. This helps to improve the robustness of the algorithm and enhance learning.

### *4.4. Training Flow*

The approximate adder environment is initialized to a fully accurate $N$-bit adder $L = N, B = 0, overlap = 0$). The exploration $\epsilon$ variable is initialized to 1 (meaning that the agent will choose a random action at the start of the algorithm) and an untrained RL agent is initialized, meaning that the policy and target networks start with random weights. Also, an empty replay memory is assumed at the beginning.

At each step, the chosen action is passed to the environment that applies it, meaning that it will increase or decrease $L, B$ or $overlap$ by 1 based on the chosen action. The new state variables $L, B, overlap$ are then provided to the approximate adder generator that produces the corresponding approximate parallel prefix adder. The adders generated during the calibration of the proposed framework for various applications are stored in a database and reused when necessary to avoid executing the adder generator again for the same examined state. In this way, the adder generator is only applied to new state vectors that have not been explored so far.

The corresponding Verilog netlist is used for measuring the error performance of the application as well as the area(power)–delay characteristics of the adder. The area, delay and error metric values, combined with the designer's constraints, are then used to calculate the reward $r_t$ based on Equation (6). If either a maximum number of steps

in this episode is reached or if the adder returned satisfies the designer's constraints, the episode ends.

The next step is to optimize the policy network, which is performed by sampling a random batch of fixed size transitions from the replay memory and using it to train the policy network using Equation (8). The loss function for a transition of state $s$, action $a$, reward $r$ and next state $s'$ sampled by the replay memory can be evaluated as

$$\text{Loss}\left(Q(s,a,\theta^P), r + \gamma \max_{a'} Q(s',a',\theta^T)\right),$$

where $\theta^P$ is the weights of the policy network and $\theta^T$ is the weights of the target network. After the network is trained, $\epsilon$ decreases linearly.

After optimizing, a check is made to see if it is time for the target network to be updated, i.e., if the weights of the main network $\theta^P$ should be copied to the target network. Then, the agent receives the next state from the environment and a new iteration can start if the episode has not finished.

At the end of the episode, the average returns (discounted sum of rewards in an episode) of the last 50 episodes are compared against the average returns of the previous 50 episodes. If the average return is not increasing and the agent can synthesize an adder respecting the designer's constraints, the training loop is terminated.

After training finishes, the agent can be used to generate an approximate parallel prefix adder that can satisfy the designer's constraints, or try to meet them as close as possible. When synthesizing a new approximate adder for an already used bit width and application, the weights of the first layer of the already trained neural network can be reused for speeding up the training process. In this work, all of the agents presented are trained from scratch.

## 5. Experimental Results

For the evaluation of the proposed approach, we generate approximate parallel prefix adders for four different applications, and compare them to state-of-the-art approximate parallel prefix adders [12,15,19], as described in Section 2. It should be noted that the adders of [12] also include an error correction mechanism that is not taken into account in this evaluation. Its application is orthogonal to the approximate parallel prefix adder design and can be equally applied to all adders under comparison.

The evaluation of the hardware complexity of each adder is carried out by synthesizing the Verilog RTL file describing each adder using Cadence's digital implementation flow and a commercial-grade 40 nm standard-cell library under a mixed $V_T$ configuration of 0.8 V. All synthesis runs target a clock period of 1 ns, which is relaxed enough to allow for timing closure for all adders.

To ensure fair comparison, for each application, we choose to compare against the state-of–the-art adder that has the best area–error efficiency. This is decided by carefully reconstructing all relevant state-of-the-art adders and measuring their area and power for a clock period of 1ns. Then, the area ratio of each adder is computed by dividing its hardware area with the area of the largest state-of-the-art adder. Similarly, we measure the error performance of each for all applications under examination. The error ratio of each adder is computed by diving the error of each adder with the largest error observed. In each application, we select the state-of-the-art adder that has the lowest product of area–error ratio. This means that the adders that we compare against may change per application.

For naming conventions, adders from [19] will be named *AxPPA*, followed by the bit on which the adder is split. Adders from [12] are built by approximating an existing adder architecture, so they will be named *Approx* followed by the name of the original adder architecture. Finally, in [15], the adders are built by specifying a minimum carry chain (MCC) for one of the two possible segments. Therefore, we name the adders as *MCC*, followed by the length of the minimum carry chain and the bit in which splitting happens, with 0 indicating that there is no splitting. The proposed adders are named as *RL*,

followed by the values of $L$, $B$ and $overlap$, which describe the adder's approximate carry computation structure.

The values of the hyperparameters for training the RL agent are shown in Table 1. They were chosen experimentally with the goal of providing a fast convergence without falling in local optima. For each agent trained, we provided a clock period target of 1 ns, an example maximum fan-out of 3 for 16-bit adders and 4 for 32-bit adders and varying area and error constraints based on the application and the adders that we compare against. These maximum fan-out constraints are similar to the ones of the adders presented in [12,15], and can generally achieve high-quality results.

**Table 1.** Values of the hyperparameters used to train the RL adder generation agent.

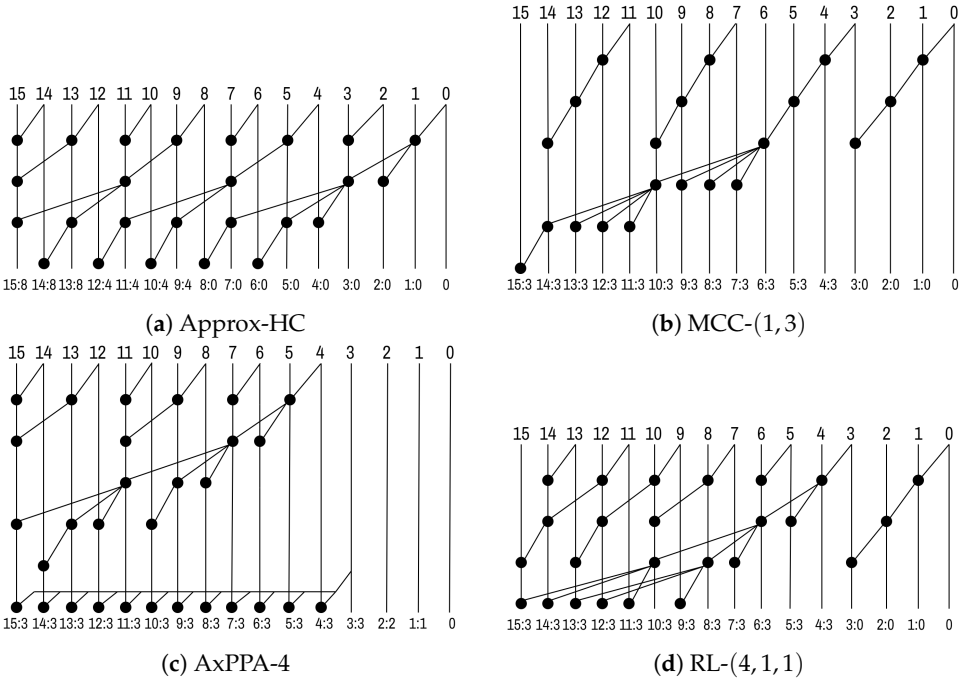
| Hyperparameter | Value |
|---|---|
| learning rate | $4 \times 10^{-5}$ |
| max. episode number | 5000 |
| max. actions in episode | 300 |
| $\epsilon$ linear decrease | $3 \times 10^{-7}$ |
| minimum $\epsilon$ value | 0.01 |
| $\gamma$ value for Q-learning equation (see Equation (8)) | 0.95 |
| batch size of experiences sampled from memory for each optimization step | 64 |
| number of steps before target network is updated | 60 |

### 5.1. Image Mean Filtering

The mean filtering application is used for smoothing 8-bit grayscale images by computing the average value of the pixels in a window around each pixel. In our implementation, an $11 \times 11$ window was used, meaning that the mean values of 121 neighbors are computed for each pixel using 16-bit approximate adders. The size of the filter ensures that the whole bit width of a 16-bit adder is used.

The error metrics used to quantify the overall performance of each approximate adder compared to the reference mean filtering result returned from an accurate adder are the peak signal-to-noise ratio (PSNR) and the mean structural similarity (MSSIM) [22]. The PSNR is a logarithmic quantity that measures the amount of noise present in the image, with a low PSNR value meaning a less accurate, noisy image and a high value meaning a high-quality image closer to the reference. The MSSIM is a value between $-1$ and 1 that quantifies the similarity of the image to the reference, with a high value showing a high similarity.

The performance of state-of-the-art adders shown in Figure 9a–c is depicted in Table 2 for five representative images. For each category of state-of-the-art adders, we chose the ones that give the best overall results for PSNR and MMSIM. An example of applying mean filtering to an image using the three state-of-the-art approximate adders shown in Figure 9a–c is depicted in Figure 10b–d, respectively. It is evident that Approx-HC has many inaccuracies, which can be seen as black pixels in places where they should be absent. The average PSNR and MSSIM of this adder are also much lower than the ones of the other two state-of-the-art adders. MCC-$(1,3)$ and AxPPA-4 exhibit very similar performance, both resulting in images that are almost indistinguishable from the reference image. AxPPA-4 has slightly higher average MSSIM but it does not make a noticeable difference in the resulting image.




(**a**) Approx-HC (**b**) MCC-$(1,3)$

(**c**) AxPPA-4 (**d**) RL-$(4,1,1)$

**Figure 9.** State-of-the-art approximate parallel prefix adders that give the best performance in each category for mean filtering: (**a**) Approx-HC [12]; (**b**) MCC-$(1,3)$ [15]; (**c**) AxPPA-4 [19]; and (**d**) the adder RL-$(4,1,1)$ synthesized by the proposed RL framework.

(**a**) Accurate (**b**) Approx-HC (**c**) MCC-$(1,3)$ (**d**) AxPPA-4 (**e**) RL-$(4,1,1)$

**Figure 10.** Examples of applying mean filtering using different adders. It can be seen that the RL-based approach is able to match the performance of the highly optimized adders from [12,19].

For the design of the proposed adder, we targeted a PSNR of 38db, essentially aiming to match the performance of MCC-$(1,3)$ and AxPPA-4. For the training of the RL agent, we used one thousand $11 \times 11$ image frames randomly extracted from a set of one thousand real images. Training the RL agent on these realistic image frames is enough for handling other images too. The RL agent is trained to optimize for the PSNR. We report also MSSIM as an additional performance metric.

The derived optimized approximated parallel prefix adder that follows the structure RL-$(4,1,1)$ is shown in Figure 9d. Its performance in terms of PSNR and MMSIM is shown in the last row of Table 2, and an example of its application can be seen in Figure 10e. The RL-driven synthesis method is able to match the performance of MCC-$(1,3)$, meaning that it also matches the PSNR of AxPPA-4 while having a slightly inferior MSSIM on average, while greatly outperforming Approx-HC. In the last columns of Table 2, we can see that our adder closely matches the area and power consumption of MCC-$(1,3)$ and AxPPA-4. Compared to Approx-HC, our adder improves on the average PSNR by a factor of 1.29 and the average MSSIM by a factor of 3.5, while also saving 8% area and 10% power.

**Table 2.** The performance of the approximate adders under comparison for a mean filtering application relative to a fully accurate result using the peak signal-to-noise ratio (PSNR) and the mean structural similarity (MSSIM) quality metrics.

| Adder | Metric | Image | | | | | | Area ($\mu m^2$) | Power (nW) |
|---|---|---|---|---|---|---|---|---|---|
| | | Apple | Flower | Cherry | Horse | House | Avg | | |
| Approx-HC | PSNR | 17.38 | 16.92 | 15.89 | 15.50 | 15.29 | 16.20 | 93 | 34 |
| | MSSIM | 0.26 | 0.37 | 0.10 | 0.10 | 0.17 | 0.20 | | |
| MCC-$(1,3)$ | PSNR | 37.11 | 37.25 | 37.11 | 37.08 | 36.91 | 37.09 | 86 | 28 |
| | MSSIM | 0.95 | 0.97 | 0.92 | 0.91 | 0.75 | 0.90 | | |
| AxPPA-4 | PSNR | 37.28 | 37.44 | 37.26 | 37.24 | 37.06 | 37.26 | 86 | 30 |
| | MSSIM | 0.98 | 0.99 | 0.95 | 0.95 | 0.79 | 0.93 | | |
| RL-$(4,1,1)$ | PSNR | 37.11 | 37.25 | 37.11 | 37.08 | 36.91 | 37.09 | 86 | 29 |
| | MSSIM | 0.95 | 0.97 | 0.92 | 0.91 | 0.75 | 0.90 | | |

For this application, the adder of MCC-$(1,3)$ is highly optimized, since the work of [15] tried different bit positions for splitting their adder and then exhaustively enumerated all possible resulting split adders on the chosen bit, with the goal of specifically optimizing the performance for mean filtering. AxPPA-4 ends up being equally efficient, since it is similar to MCC-$(1,3)$, with some structural differences and also the removal of operators in bit positions 1-2, which does not impact the result negatively in this case. Given the highly optimized profile of these two adders, it is expected that the RL agent cannot find a better design. In fact the adder that it returns *automatically without any manual tuning* as performed in [15], RL-$(4,1,1)$, has the exact same carry chains as MCC-$(1,3)$, and consequently very similar area and power behavior. This demonstrates the ability of the RL agent to match highly optimized architectures for applications where such adders exist.

### *5.2. Laplacian Filtering*

Laplacian filtering, or image edge detection, is applied for detecting edges in 8-bit grayscale images by applying the following $3 \times 3$ filter across the image:

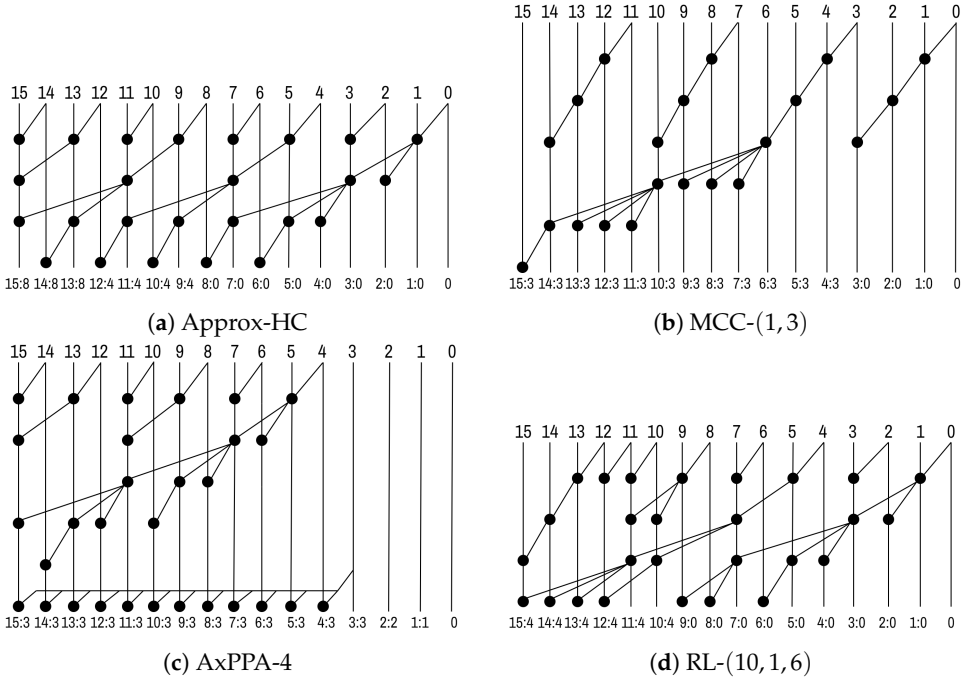
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Each pixel of the image aligned with the $3 \times 3$ window is multiplied by the respective coefficient of the filter, and the derived products are summed together. This final addition is performed by an approximate 16-bit parallel prefix adder.

The fundamental difference in this application compared to mean filtering is the presence of a subtraction, which tests if approximate adders work well for signed operands. The error metrics used to quantify the effectiveness of each approximate adder are again PSNR and MSSIM, similarly to the mean filtering. The performance of the state-of-the-art adders is shown in Table 3 for the same five representative images and their structure is depicted in Figure 11a–c. An example of applying Laplacian filtering to an image using the state-of-the-art adders is shown in Figure 12. Approx-LF has many white spots compared to the reference, meaning that it is missing many edges. AxPPA-4 and MCC-$(1,3)$ are closer to the reference, with AxPPA-4 missing edges closer to the top of the image and MCC-$(1,3)$ having an overall better performance for this specific image, but also failing to detect some edges, making the resulting image slightly sparser toward the top and center. Based on the PSNR and MSSIM metrics showed in Table 3 for all state-of-the-art adders, it can be seen that, on average, Approx-HC is the most error-prone adder of the three, with AxPPA-4 marginally winning in terms of PSNR and MCC-$(1,3)$ having the highest average MSSIM value.

**Table 3.** The performance of the approximate adders under comparison for a Laplacian filtering application relative to a fully accurate result using the peak signal-to-noise ratio (PSNR) and the mean structural similarity (MSSIM) quality metrics.

| Adder | Metric | Image | | | | | |
|---|---|---|---|---|---|---|---|
| | | Apple | Flower | Cherry | Horse | House | Avg |
| Approx-HC | PSNR | 31.24 | 25.78 | 25.25 | 23.06 | 28.96 | 26.86 |
| | MSSIM | 0.37 | 0.58 | 0.53 | 0.66 | 0.46 | 0.52 |
| MCC-(1,3) | PSNR | 34.78 | 30.86 | 30.26 | 28.53 | 31.32 | 31.15 |
| | MSSIM | 0.79 | 0.85 | 0.77 | 0.86 | 0.84 | 0.82 |
| AxPPA-4 | PSNR | 33.94 | 31.48 | 31.74 | 30.02 | 32.16 | 31.87 |
| | MSSIM | 0.62 | 0.80 | 0.64 | 0.84 | 0.71 | 0.72 |
| RL-(10,1,6) | PSNR | 38.99 | 32.97 | 38.60 | 30.27 | 38.12 | 35.79 |
| | MSSIM | 0.85 | 0.91 | 0.77 | 0.91 | 0.76 | 0.84 |




(**a**) Approx-HC (**b**) MCC-(1,3)

(**c**) AxPPA-4 (**d**) RL-(10,1,6)

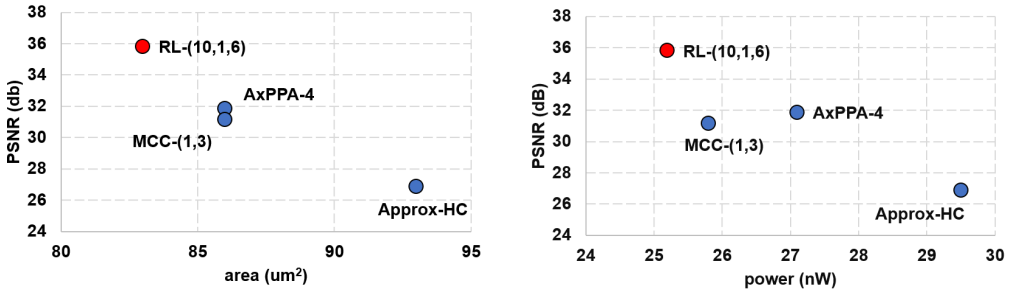
**Figure 11.** State-of-the-art approximate parallel prefix adders that give the best performance in each category for Laplacian filtering: (**a**) Approx-HC [12]; (**b**) MCC-(1,3) [15]; (**c**) AxPPA-4 [19]; and (**d**) the adder RL-(10,1,6) synthesized by the proposed RL framework.

Using the proposed automated approximate parallel prefix adder design framework, we attempted to outperform the most accurate state-of-the-art adder by targeting a PSNR of 35 dB. The RL agent is trained on one-thousand random $3 \times 3$ frames extracted from real images. The resulting approximate parallel prefix adder after the training of the RL agent is RL-(10,1,6), which is shown in Figure 11d, and its performance can be seen in the last row of Table 3. The proposed adder outperforms all other state-of-the-art adders for this application, both for the PSNR and the MSSIM error metrics. An example of applying Laplacian filtering to an example image using the RL-generated adder is seen in Figure 12e, where the proposed adder produces an image that is nearly identical to the reference image processed by the accurate adder.

(**a**) Accurate (**b**) Approx-HC (**c**) MCC-$(1,3)$ (**d**) AxPPA-4 (**e**) RL-$(10,1,6)$

**Figure 12.** Examples of applying Laplacian filtering using different adders. It can be seen that the RL-based approach is the closest to the reference.

The efficiency in terms of PSNR for each adder relative to its hardware complexity is presented in Figure 13, where it can be seen that our adder has the highest PSNR compared to all three state-of-the-art adders while having the smallest area and closely matching the power consumption of MCC-$(1,3)$. Our proposed adder RL-$(10,1,6)$ improves the average PSNR by 33% compared to Approx-HC while simultaneously reducing the area and power consumption by 11% and 17%, respectively. Compared to MCC-$(1,3)$ and AxPPA-4, our adder improves the average PSNR by 12% while also improving the area by 3% and achieving virtually the same power consumption as MCC-$(1,3)$ and 7% better power compared to AxPPA-4.




(**a**) (**b**)

**Figure 13.** Plot of average PSNR in relation to the (**a**) area and (**b**) power of the approximate adders examined for the application of Laplacian filtering.

To intuitively understand why the proposed adders performed better than the state of the art, we examine the approximation of MCC-$(1,3)$ relative to RL-$(10,1,6)$ in more detail. After applying Laplacian filtering to the examined images, we observe that MCC-$(1,3)$ makes fewer mistakes on average than RL-$(10,1,6)$, erroneously calculating 18% of the output pixels, compared to 22% for RL-$(10,1,6)$. However, the proposed adder better adapts to this specific application, making errors with smaller magnitudes. For example, the average absolute error of RL-$(10,1,6)$ is 4.42, while that of MCC-$(1,3)$ is 9.07. In general, image filtering applications can tolerate multiple small errors more effectively than fewer errors with larger magnitudes, which negatively impact image quality. For a different application where the number of mistakes is more critical than their magnitude, MCC-$(1,3)$ could have been a better choice than RL-$(10,1,6)$, further emphasizing the importance of selecting approximate adders based on the specific requirements of the target application.

### *5.3. Vector Inner Product*

The vector inner product multiplies the corresponding elements of two vectors and then sums the resulting products to a scalar value. For vectors $c$ and $x^T$, the computation is the following:

$$S = \begin{bmatrix} c_0 & c_1 & \dots & c_n \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{bmatrix} = c_0x_0 + c_1x_1 + \dots c_nx_n$$

For this example, we assume that vector $c$ contains fixed values and only $x^T$ can assume arbitrary inputs. In our implementation, we chose 16-element vectors, where each element is a 12-bit signed integer. The final result will fit in a 32-bit signed number assuming that multiplication is performed by an accurate multiplier producing full-width products. Approximate addition is again used to perform the addition of products at 32 bits. The goal of this application is to discover adders that can perform well in a general multiply–accumulate scenario, as opposed to the image processing applications where the input is limited to image frames and the filter has specific values that serve this application.

The error metric used for the evaluation of the addition results is the error frequency (EF). It measures how often the adder returns an inaccurate result and is equal to

$$EF = \frac{\text{total inaccurate samples}}{\text{total samples}}$$

An inaccurate sample is a set of inputs for which the adder returns an erroneous sum. The error is computed after the final sum $S$ is calculated, not for every individual summation. It is measured over 50.000 different sets of inputs that are sampled from a uniform random distribution.

The performance of state-of-the-art adders with the highest EF efficiency for this application is shown in Table 4. Their structure is depicted in Figure 14. The results show that the AxPPA-2 adder gives the worst performance. The adders from AxPPA [19] make many small mistakes, which can be effective in error-resilient applications such as image processing, but yield low-quality results when an application requires a low frequency of errors, regardless of the magnitude. Approx-HC performs better than AxPPA both in terms of error frequency and hardware complexity. When compared to MCC(21, 0), it offers better hardware and power but at an inferior performance in terms of EF.

**Table 4.** The performance in terms of error frequency and hardware complexity of state-of-the-art adders and two variants generated automatically by the proposed approach.

| Adder | #Nodes | Area ($\mu m^2$) | Power (nW) | EF |
|---|---|---|---|---|
| Approx-HC | 72 | 211 | 72 | 0.0109 |
| MCC-$(21, 0)$ | 85 | 220 | 75 | 0.0002 |
| AxPPA-2 | 87 | 242 | 82 | 0.6593 |
| RL-$(20, 3, 15)$ | 70 | 199 | 71 | 0.0055 |
| RL-$(27, 3, 21)$ | 85 | 218 | 73 | 0.0002 |

For this application, we designed two RL-based adders. For the first one, the constraint for the RL agent was to achieve an EF of 0.01 and, for the second one, the constraint was tightened to 0.0002, which is 50 times more accurate. During training the error estimation happens over 400 samples, and the values of vector $c$ are kept fixed to ensure that the training does not diverge.

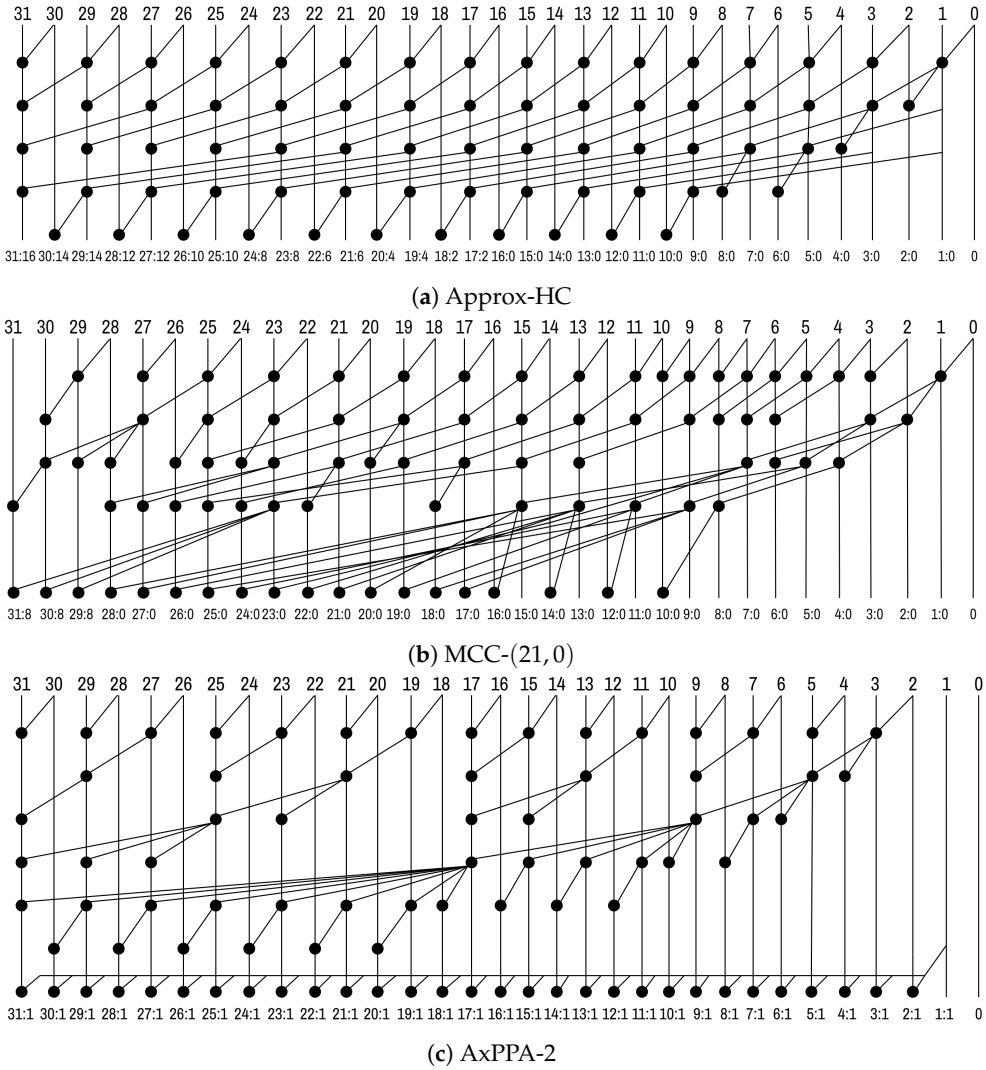
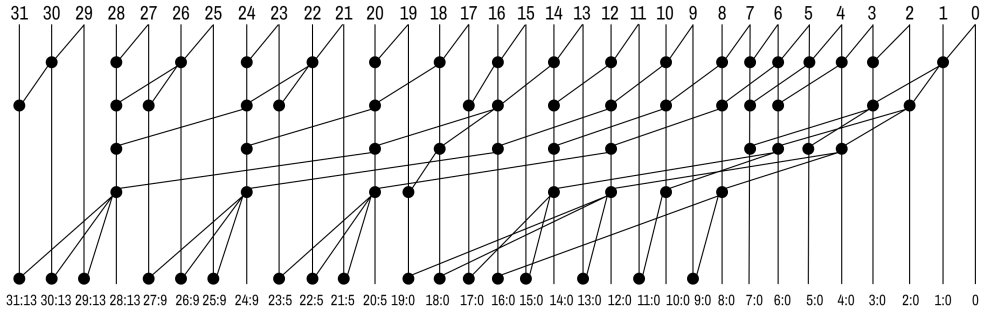


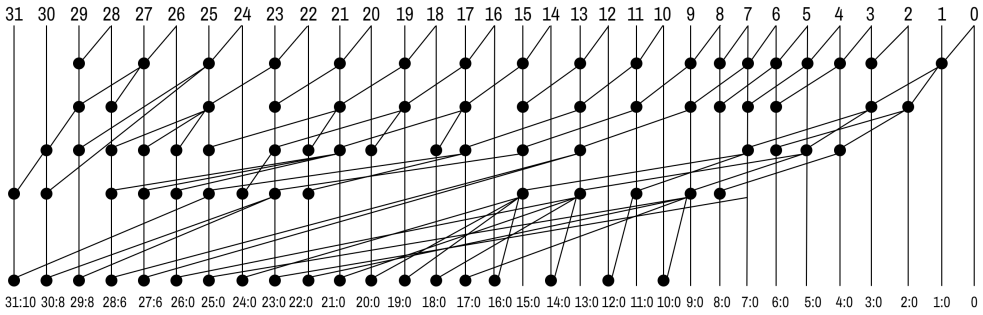

**Figure 14.** State-of-the-art adders that achieve the lowest EF achievable for their respective family of adders when computing vector inner products.

The first RL-generated adder is RL-$(20, 3, 15)$, shown in Figure 15a, whereas the second is RL-$(27, 3, 21)$, shown in Figure 15b. Their performance is shown in the last two rows of Table 4. RL-$(20, 3, 15)$ should be compared against Approx-HC and AxPPA-2 since it has a higher error frequency in favor of saving area. On the contrary, the second adder RL-$(27, 3, 21)$ targets a much lower EF, so it is compared against MCC-$(21, 0)$.

RL-$(20, 3, 15)$ optimized for a relaxed error target is able to greatly outperform Approx-HC in terms of error behavior, resulting in a 50% lower EF, while also saving 6% area and 1% power. Compared to AxPPA, it saves 18% area and 13% power while significantly decreasing the error frequency. The more accurate adder RL-$(27, 3, 21)$ matches the error frequency of MCC-$(21, 0)$ while improving the area and power by 1% and 3%, respectively. It can be seen from Figure 14b that MCC-$(21, 0)$ is a fully accurate adder up until bit 28, and only introduces approximation in the last 3 bits. This imposes a very tight error constraint on the RL agent, which has very little margin for optimization and for trading off accuracy for improved QoR, making it expected that the resulting adder is not dramatically improving power and area.




(**a**) RL-$(20, 3, 15)$




(**b**) RL-$(27, 3, 21)$

**Figure 15.** RL-generated adders with efficient performance for vector multiplication application. Adder (**a**) is more area-efficient whereas adder (**b**) achieves a better EF.

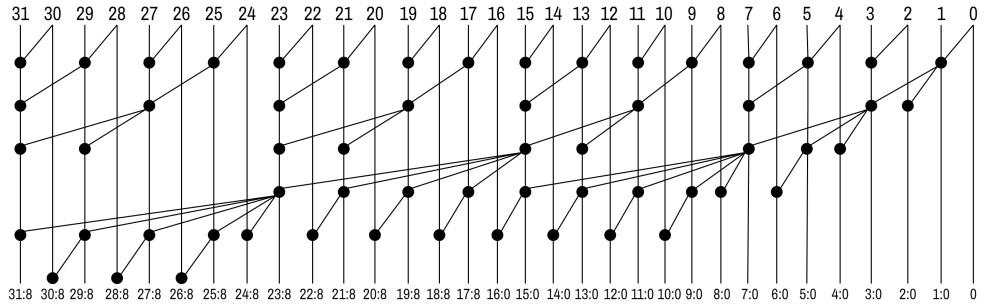
### *5.4. Neural Network*

For the final application, we designed a multi-layer perceptron (MLP) consisting of an input layer of 784 neurons, a hidden layer of 500 and an output layer of 10, which has been pre-trained for digit recognition. It was trained with PyTorch 2.0.1 [23] on the MNIST dataset [24] using 50.000 $28 \times 28$ images, with the goal of minimizing the classification error. The weights of the MLP were quantized in 8-bit values. For the inference stage, we used the remaining 10.000 images, evaluating the classification accuracy using approximate adders. Similarly to the previous application, the multiplication happens using fully accurate multipliers that produce 16-bit products. The approximate adder is a 32-bit wide adder that sums those 16-bit products. When using a fully accurate adder, the reference accuracy is equal to 96.82%.

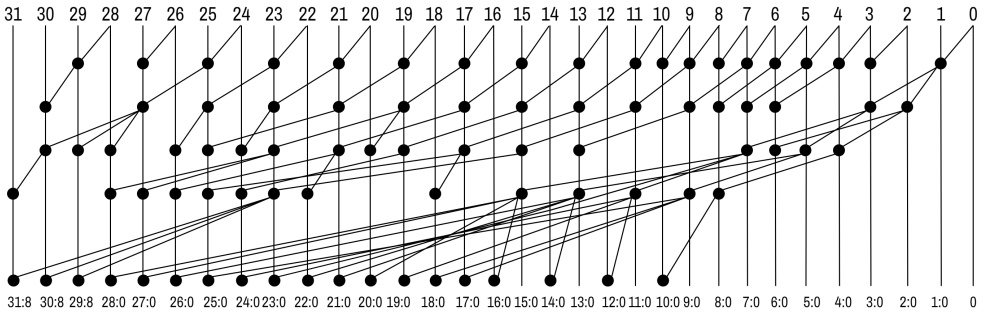
For this application, we chose the most area-efficient state-of-the-art adders that can achieve a classification error equal to that of the fully accurate adder for comparison. The adders can be seen in Figure 16, while their area and power metrics can be compared in Figure 17. It can be seen that AxPPA-6 is the most efficient adder for both area and power, followed by MCC-$(21, 0)$ and Approx-LF.

For the design of the proposed adder, the agent was presented with the constraint of achieving a classification accuracy of 96.82%. During training, inference is executed on 1.000 images instead of the full 10.000 image dataset for a faster accuracy estimation.

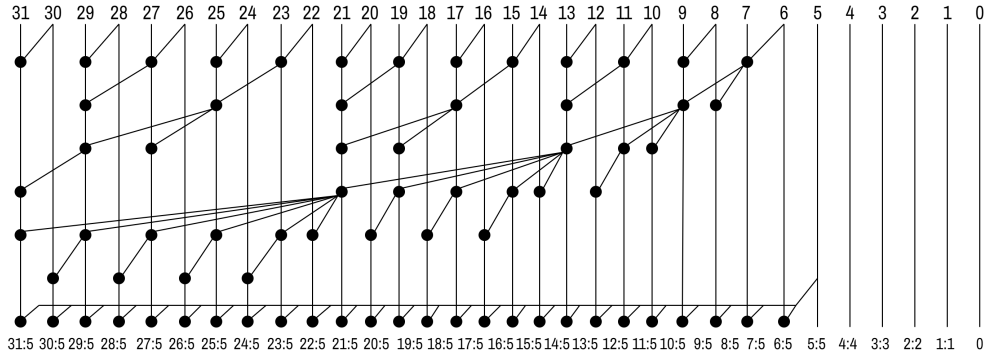
The resulting adder is RL-$20, 3, 16$, shown in Figure 16d, which is able to match the classification accuracy of the accurate adder. It can be seen in Figure 17 that our adder is able to achieve the accuracy target while lowering both area and power compared to the state-of-the-art adders. Specifically, our approach saves 13% area and 8% power compared to Approx-LF, 10% area and 7% power compared to MCC-$(21, 0)$ and, finally, 9% area and 5% power compared to AxPPA-6.



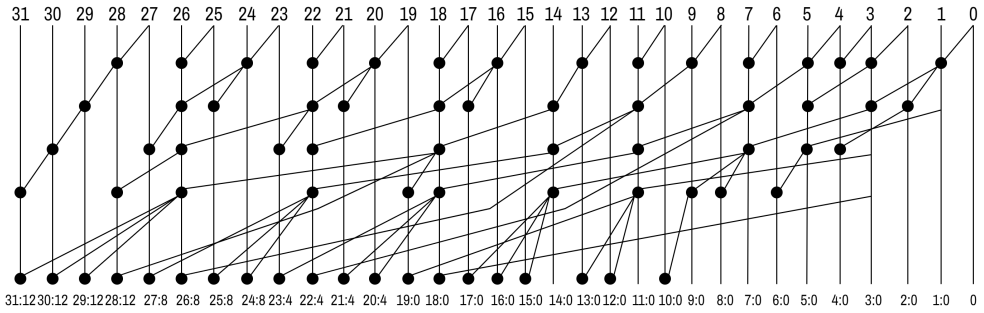

(**a**) Approx-LF




(**b**) MCC-$(21, 0)$

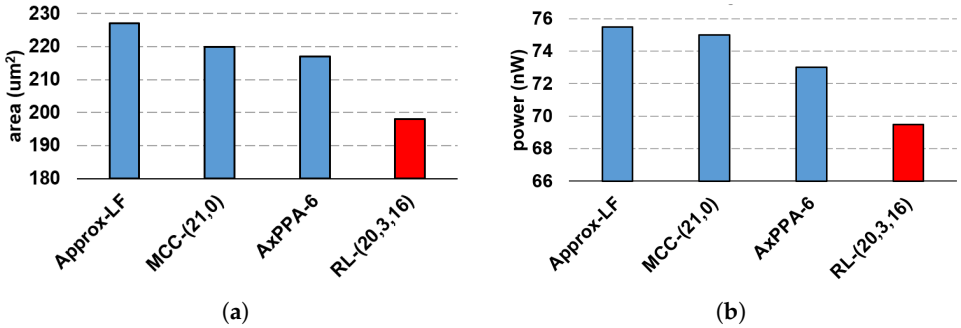



(**c**) AxPPA-6




(**d**) RL-$(20, 3, 16)$

**Figure 16.** Approximate adders used in the digit classification application. All adders achieve a classification accuracy equal to the one of an accurate adder.




(**a**) (**b**)

**Figure 17.** (**a**) Area and (**b**) power consumption of all adders under comparison for the case of neural network inference on digit classification.

### *5.5. Runtime Analysis*

The runtime analysis for the execution of the RL framework for each of the proposed adders is presented in Table 5. For each adder, we report the total time taken to train the DQN and the time necessary for adder and logic synthesis, as well as the duration of the application simulation in the training loop. These results are cumulative for running the entire framework, starting with an untrained RL agent. We also report the percentage of state space explored during the execution.

**Table 5.** The total runtime required to execute the reinforcement learning framework and generate each of the proposed adders.

| Application | Adder | Explored States (%) | Runtime (mins) | | | | |
|---|---|---|---|---|---|---|---|
| | | | Train | Adder Generation | Logic Synthesis | Application Simulation | Total |
| Mean filtering | RL-(4, 1, 1) | 84 | 6.87 | 32.49 | 4.92 | 457.84 | 502.12 |
| Laplacian filtering | RL-(10, 1, 6) | 83 | 8.73 | 32.04 | 5.06 | 33.66 | 79.49 |
| Inner product | RL-(20, 3, 15) | 19 | 6.63 | 1382.16 | 20.23 | 166.21 | 1575.23 |
| | RL-(27, 3, 21) | 18 | 10.37 | 1312.61 | 19.96 | 157.35 | 1500.29 |
| Neural network | RL-(20, 3, 16) | 20 | 8.92 | 1426.31 | 21.01 | 631.28 | 2087.52 |

As mentioned in Section 4, adders encountered when performing experiments are stored in a database to avoid re-running adder and logic synthesis for them. Roughly 90% of adders encountered by the RL agent during the framework execution for obtaining the proposed adders are retrieved from the database, and the approximate adder generation runtime reported in Table 5 results from the generation of the remaining 10% of adders who had not already been visited in previous experiments. Adder simulation is executed when an adder is encountered for the first time in a specific experiment and the error metrics are stored so that, if the adder is visited again in the same experiment, simulation is skipped and its saved error result is retrieved.

The first observation made is that the neural network training time is very short compared to the total runtime of the algorithm due to the simplicity of the structure of the network. This is true for both 16-bit and 32-bit applications since the network structure does not change with a different bit width, making our proposed RL formulation scalable to higher-order adders.

The adder generation time is a big factor of the runtime for the 16-bit applications (mean and Laplacian filtering), and takes up the biggest part of the runtime for the 32-bit applications. More specifically, approximate adder generation lasts for approximately half

an hour for 16-bit adders, with the average runtime needed to generate one 16-bit adder being 12 s (the exact value differs based on the $L$, $B$ and $overlap$ provided to the adder generator). The runtime for approximate adder generation increases to almost a day for the 32-bit applications, with the average time needed to generate a 32-bit adder being close to 8 min.

The time needed for logic synthesis is very short, both for 16- and 32-bit applications. Similarly to adder generation, if an adder is found in the database of previously explored adders, then logic synthesis is not performed, and its area and delay information are retrieved. For 16-bit applications, logic synthesis takes approximately 5 min, and, for 32 bits, it runs for slightly more than 20 min.

The application simulation time depends both on the application and the bit width of the adder. In the case of mean filtering, it takes up the lion's share of the runtime, since, for each adder, we execute mean filtering over 1.000 11 $\times$ 11 frames, resulting in a high number of 16-bit additions. On the other hand, simulating the Laplacian filtering application is much faster since the frame dimensions are lowered to 3 $\times$ 3. The behavior for the 32-bit applications is similar, with the inner product application simulation being relatively fast, while simulating the neural network application takes over 10 h due to the presence of a high number of 32-bit additions.

Overall, it can be observed that the main runtime bottleneck is the approximate adder generation, especially when the bit width increases from 16 to 32 bits. This overhead can be mitigated with a variety of techniques, including extended book-keeping of intermediate solutions and multithreading. Any improvement on the adder generator would be orthogonal to our proposed RL framework, which demonstrates its efficiency by resulting in low training times, showing that it is an efficient method of state space exploration with very small overhead.

By examining the 'Explored states' column, we observe that our framework efficiently finds high-quality solutions by exploring approximately 85% of the valid state space for 16-bit adders and roughly 20% for 32-bit adders. This efficiency translates to significant runtime savings compared to a greedy approach, which would require exploring the entire state space. While a greedy approach avoids neural network training time, this overhead is negligible compared to the overall runtime. Consequently, our framework achieves a 1.17$\times$ speedup for 16-bit adders and a 5$\times$ speedup for 32-bit adders

## 6. Related Work

Related work spans both the approximate adder design as well as the use of RL techniques for design optimization in various parts of the flow.

There are multiple ways to introduce approximation in adder designs for saving area and delay that go beyond the scope of this paper, which focuses on designing parallel prefix adders.

One design choice is using a split-accuracy configuration for the adder's design blocks. In the case of [25], the adder is divided into an accurate and approximate part. The accurate part sums the highest-significance bits and the approximate one replaces addition blocks with OR/AND gates. The work of [26] improves on this method by utilizing error correction in order to lower the error frequency of [25]. In a similar vein, another segmented architecture is presented in [27], where the approximate adder is segmented into three parts. The lowest-significance segment is implemented using simple logic gates and does not propagate carry signals, the middle segment is an approximate carry look-ahead adder and the high-significance part is an accurate adder. In [14], the authors utilize a tool for automatic gate pruning.

Another option for achieving approximation through logic gate simplification is at the transistor level. The approximate mirror adder [28] is built using addition cells with fewer transistors, and the works of [20,21] present approximate full adders by applying transformations at the transistor level.

Another option widely explored in the academia is focusing on the higher-level architecture of the approximate adder. A frequently used approach is shortening the carry chain length. In [29], the authors propose the almost correct adder (ACA), which consists of parallel shorter carry generators. The work of [30] extends this by sharing some computational blocks, leading to a reduced area, as well as by adding error detection and recovery circuits. The work of [31] makes use of error compensation.

Similarly to the gate-level approximations, there are many proposed approaches in high-level approximation techniques that segment the adder into an accurate and approximate part, such as [32,33]. This approach is extended in [34] with the addition of a selector module that determines the bit position where the adder will be split into the accurate and approximate segment. In [35], the approximate part of the adder is replaced with a constant output, leading to reduced power and area, while maintaining an acceptable error behavior. The work of [36] adds an error detection and correction module.

Approximation techniques have also been proposed for carry-select and carry-skip architectures. The carry-select adder proposed in [37] is split into $n/k$ sub-adders, with $n$ being the bit width and $k$ the carry chain length of the adder. Each sub-adder consists of two $k$-bit adders as needed by carry-select operation. A similar splitting architecture is used in [37] for the design of approximate carry-skip adders.

Another splitting technique is used by [38], which separates the adder into $k$-bit sub-adders, each connected to a multiplexer for the selection of the appropriate carry-in connection. This has been extended to include error correction in [39]. The work of [40] divides a speculative adder into sub-adders, each one including a carry predictor. A similar approach is followed by [41], but a wider set of input bits is used for the carry prediction. The work of [42] uses a feedback technique to cut the carry chain and relax the critical path. A carry propagate block is used to generate the cut signal. More configurations relying on smaller adder blocks and carry prediction are presented in [43,44].

Finally, RL has been used successfully in a plethora of design tasks, such as floorplanning [45,46], placement [47,48] and routerless networks-on-chip loop placement [49], as well as the design of accurate parallel prefix adders [10]. The topic of [10] is closely related to this work. However, there are several fundamental differences between the two approaches. First, the method of [10] does not allow for the design of approximate adders. Second, the proposed approach uses RL to optimize the approximation strategy but relies on [15] for synthesizing the selected approximate parallel prefix adder. On the contrary, RL in [10] learns a detailed representation of the parallel prefix graph. Third, the proposed approach co-optimizes hardware complexity and application performance, while the method of [10] focuses only on the hardware characteristics of the adders.

## 7. Conclusions

Approximate parallel prefix adders provide a versatile solution for designers seeking to balance addition accuracy and quality of results (QoR). However, the multitude of approximation options and varying error tolerances across applications make this a challenging task. This paper introduces an RL approach that combines a structured state representation with real-world application simulations to train an RL agent to synthesize approximate prefix adders based on designer constraints.

The introduced RL agent generates adders that, when synthesized using a logic synthesis tool, consistently achieve superior or equivalent QoR–accuracy tradeoffs compared to existing methods. By leveraging knowledge of the target application, the agent can optimize for more efficient architectures that either reduce area and power without compromising accuracy or significantly improve summation quality with minimal overhead.

Future work is planned for extending the RL-based approach to other arithmetic components such as approximate multipliers or combined multiply–add datapaths.

**Author Contributions:** Conceptualization, A.S. and G.D.; Methodology, A.S.; Software, A.S.; Validation, A.S.; Writing—original draft, A.S.; Writing—review & editing, G.D.; Supervision, G.D. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Data Availability Statement:** The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References


1. Stanley-Marbell, P.; Alaghi, A.; Carbin, M.; Darulova, E.; Dolecek, L.; Gerstlauer, A.; Gillani, G.; Jevdjic, D.; Moreau, T.; Cacciotti, M.; et al. Exploiting Errors for Efficiency: A Survey from Circuits to Applications. *Acm Comput. Surv.* **2020**, *53*, 1–39. [CrossRef]
2. Rehman, S.A.; Jeffrey, Z.; Sun, Y.; Simpson, O. SASHA: A Shift-Add Segmented Hybrid Approximated Multiplier for Image Processing. In Proceedings of the 2022 International Conference on Intelligent Systems and Computer Vision, Fez, Morocco, 18–20 May 2022; pp. 1–5.
3. Ferreira, G.; Pereira, P.T.L.; Paim, G.; Costa, E.; Bampi, S. A Power-Efficient FFT Hardware Architecture Exploiting Approximate Adders. In Proceedings of the 2021 IEEE 12th Latin America Symposium on Circuits and System, Arequipa, Peru, 21–24 February 2021; pp. 1–4.
4. De Caro, D.; Di Meo, G.; Napoli, E.; Petra, N.; Strollo, A.G.M. A 1.45 GHz All-Digital Spread Spectrum Clock Generator in 65nm CMOS for Synchronization-Free SoC Applications. *IEEE Trans. Circuits Syst. Regul. Pap.* **2020**, *67*, 3839–3852. [CrossRef]
5. Magadum, P.; Ghosh, S. Network Compression for end-to-end trainable neural networks using Approximate Computing. In Proceedings of the 2021 12th International Conference on Computing Communication and Networking Technologies, Kharagpur, India, 6–8 July 2021; pp. 1–5.
6. Zimmermann, R. Binary Adder Architectures for Cell-Based VLSI and Their Synthesis. Ph.D. Thesis, ETHZ, Zürich, Switzerland, 1998.
7. Knowles, S. A family of adders. In Proceedings of the IEEE Symposium on Computer Arithmetic, Adelaide, SA, Australia, 14–16 April 1999; pp. 30–34.
8. Roy, S.; Choudhury, M.; Puri, R.; Pan, D.Z. Towards Optimal Performance-Area Trade-Off in Adders by Synthesis of Parallel Prefix Structures. In Proceedings of the Design Automation Conference (DAC), Austin, TX, USA, 29 May–7 June 2013; pp. 1–8.
9. Ma, Y.; Roy, S.; Miao, J.; Chen, J.; Yu, B. Cross-Layer Optimization for High Speed Adders: A Pareto Driven Machine Learning Approach. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2019**, *38*, 2298–2311. [CrossRef]
10. Roy, R.; Raiman, J.; Kant, N.; Elkin, I.; Kirby, R.; Siu, M.; Oberman, S.; Godil, S.; Catanzaro, B. PrefixRL: Optimization of Parallel Prefix Circuits using Deep Reinforcement Learning. In Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 5–9 December 2021; pp. 853–858.
11. Esposito, D.; De Caro, D.; Napoli, E.; Petra, N.; Strollo, A.G.M. Variable Latency Speculative Han-Carlson Adder. *IEEE Trans. Circuits Syst. Regul. Pap.* **2015**, *62*, 1353–1361. [CrossRef]
12. Esposito, D.; De Caro, D.; Strollo, A.G.M. Variable Latency Speculative Parallel Prefix Adders for Unsigned and Signed Operands. *IEEE Trans. Circuits Syst.* **2016**, *63*, 1200–1209. [CrossRef]
13. Cilardo, A. A New Speculative Addition Architecture Suitable for Two's Complement Operations. In Proceedings of the Design, Automation Test in Europe (DATE), Nice, France, 20–24 April 2009; pp. 664–669.
14. Schlachter, J.; Camus, V.; Palem, K.V.; Enz, C. Design and Applications of Approximate Circuits by Gate-Level Pruning. *IEEE Trans. Very Large Scale Integr. (Vlsi) Syst.* **2017**, *25*, 1694–1702. [CrossRef]
15. Stefanidis, A.; Zoumpoulidou, I.; Filippas, D.; Dimitrakopoulos, G.; Sirakoulis, G.C. Synthesis of Approximate Parallel-Prefix Adders. *IEEE Trans. Vlsi Syst.* **2023**, *31*, 1686–1699. [CrossRef]
16. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*; MIT Press: Cambridge, MA, USA , 2018.
17. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.A.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-level control through deep reinforcement learning. *Nature* **2015**, *518*, 529–533. [CrossRef]
18. Brent, R.P.; Kung, H.T. A Regular Layout for Parallel Adders. *IEEE Trans. Comput.* **1982**, *31*, 260–264. [CrossRef]
19. da Rosa, M.M.A.; Paim, G.; da Costa, P.U.L.; da Costa, E.A.C.; Soares, R.; Bampi, S. AxPPA: Approximate Parallel Prefix Adders. *IEEE Trans. Very Large Scale Integr. Syst.* **2023**, *31*, 17–28. [CrossRef]
20. Yang, Z.; Jain, A.; Liang, J.; Han, J.; Lombardi, F. Approximate XOR/XNOR-based Adders for Inexact Computing. In Proceedings of the International Conference on Nanotechnology (IEEE-NANO), Beijing, China, 5–8 August 2013; pp. 690–693.
21. Nanu, D.; Roshini, P.; Sowkarthiga, D.; Ameen, K.S.A. Approximate Adder Design Using CPL Logic for Image Compression. *Int. J. Innov. Res. Dev.* **2014**, *3*, 362–370 .
22. Liu, C.; Han, J.; Lombardi, F. An analytical framework for evaluating the error characteristics of approximate adders. *IEEE Trans. Comput.* **2014**, *64*, 1268–1281. [CrossRef]
23. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. Pytorch: An imperative style, high-performance deep learning library. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 8026–8037 .
24. Deng, L. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Process. Mag.* **2012**, *29*, 141–142. [CrossRef]
25. Mahdiani, H.R.; Ahmadi, A.; Fakhraie, S.M.; Lucas, C. Bio-Inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-Computing Applications. *IEEE Trans. Circuits Syst.* **2010**, *57*, 850–862. [CrossRef]

26. Seo, H.; Yang, Y.; Kim, Y. Design and Analysis of an Approximate Adder with Hybrid Error Reduction. *Electronics* **2020**, *9*, 471. [CrossRef]
27. Ban, T.; Wang, B.; Naviner, L. Design, Synthesis and Application of A Novel Approximate Adder. In Proceedings of the International Midwest Symposium on Circuits and Systems (MWSCAS), Windsor, ON, Canada, 5–8 August 2018; pp. 488–491.
28. Gupta, V.; Mohapatra, D.; Raghunathan, A.; Roy, K. Low-Power Digital Signal Processing Using Approximate Adders. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2013**, *32*, 124–137. [CrossRef]
29. Lu, S.L. Speeding Up Processing with Approximation Circuits. *Computer* **2004**, *37*, 67–73.
30. Verma, A.K.; Brisk, P.; Ienne, P. Variable Latency Speculative Addition: A New Paradigm for Arithmetic Circuit Design. In Proceedings of the Design, Automation and Test in Europe (DATE), Munich, Germany, 10–14 March 2008; pp. 1250–1255.
31. Mohapatra, D.; Chippa, V.K.; Raghunathan, A.; Roy, K. Design of Voltage-Scalable Meta-Functions for Approximate Computing. In Proceedings of the Design, Automation and Test in Europe (DATE), Grenoble, France, 14–18 March 2011; pp. 1–6.
32. Zhu, N.; Goh, W.L.; Zhang, W.; Yeo, K.S.; Kong, Z.H. Design of Low-Power High-Speed Truncation-Error-Tolerant Adder and Its Application in Digital Signal Processing. *IEEE Trans. Very Large Scale Integr. (Vlsi) Syst.* **2010**, *18*, 1225–1229.
33. Zhu, N.; Goh, W.; Yeo, K.S. An Enhanced Low-Power High-Speed Adder For Error-Tolerant Application. In Proceedings of the International Symposium on Integrated Circuits (ISIC), Incheon, Republic of Korea, 22–23 November 2010; pp. 69–72.
34. Zhu, N.; Goh, W.L.; Yeo, K.S. Ultra Low-Power High-Speed Flexible Probabilistic Adder for Error-Tolerant Applications. In Proceedings of the Intern. SoC Design Conference, Jeju, Republic of Korea, 17–18 November 2011; pp. 393–396.
35. Seo, H.; Yang, Y.S.; Kim, Y. An Energy-Efficient Imprecise Adder with a Lower-part Constant Approximation. In Proceedings of the International SoC Design Conference (ISOCC), Yeosu, Republic of Korea, 21–24 October 2020; pp. 143–144.
36. Kahng, A.B.; Kang, S. Accuracy-Configurable Adder for Approximate Arithmetic Designs. In Proceedings of the Design Automation Conference (DAC), San Francisco, CA, USA, 3–7 June 2012; pp. 820–825.
37. Kim, Y.; Zhang, Y.; Li, P. An Energy Efficient Approximate Adder with Carry Skip for Error Resilient Neuromorphic VLSI Systems. In Proceedings of the Internet Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, 23 December 2013; pp. 130–137.
38. Ye, R.; Wang, T.; Yuan, F.; Kumar, R.; Xu, Q. On Reconfiguration-Oriented Approximate Adder Design and Its Application. In Proceedings of the Internet Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, 18–21 November 2013; pp. 48–54.
39. Shafique, M.; Ahmad, W.; Hafiz, R.; Henkel, J. A low latency generic accuracy configurable adder. In Proceedings of the Design Automation Conference (DAC), San Francisco, CA, USA, 7–11 June 2015; pp. 1–6.
40. Lin, I.C.; Yang, Y.M.; Lin, C.C. High-Performance Low-Power Carry Speculative Addition With Variable Latency. *IEEE Trans. Very Large Scale Integr. (Vlsi) Syst.* **2015**, *23*, 1591–1603. [CrossRef]
41. Li, L.; Zhou, H. On Error Modeling and Analysis of Approximate Adders. In Proceedings of the Internet Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, 2–6 November 2014; pp. 511–518.
42. Camus, V.; Schlachter, J.; Enz, C. A Low-Power Carry Cut-Back Approximate Adder with Fixed-Point Implementation and Floating-Point Precision. In Proceedings of the Design Automation Conference (DAC), Austin, TX, USA, 5–9 June 2016; pp. 1–6.
43. Xu, W.; Sapatnekar, S.S.; Hu, J. A Simple Yet Efficient Accuracy-Configurable Adder Design. *IEEE Trans. Very Large Scale Integr. (Vlsi) Syst.* **2018**, *26*, 1112–1125. [CrossRef]
44. Ebrahimi-Azandaryani, F.; Akbari, O.; Kamal, M.; Afzali-Kusha, A.; Pedram, M. Block-Based Carry Speculative Approximate Adder for Energy-Efficient Applications. *IEEE Trans. Circuits Syst.* **2020**, *67*, 137–141. [CrossRef]
45. Yan, B.; Xu, L.; Yu, Z.; Xie, M.; Ran, W.; Gao, J.; Fu, Y.; Liu, T. Learning to Floorplan like Human Experts via Reinforcement Learning. In Proceedings of the 2024 Design, Automation and Test in Europe Conference and Exhibition (DATE), Valencia, Spain, 25–27 March 2024; pp. 1–2.
46. He, Z.; Ma, Y.; Zhang, L.; Liao, P.; Wong, N.; Yu, B.; Wong, M.D. Learn to Floorplan through Acquisition of Effective Local Search Heuristics. In Proceedings of the 2020 IEEE 38th International Conference on Computer Design (ICCD), Hartford, CT, USA, 18–21 October 2020; pp. 324–331.
47. Zhao, D.; Yuan, S.; Sun, Y.; Tu, S.; Xu, L. DeepTH: Chip Placement with Deep Reinforcement Learning Using a Three-Head Policy Network. In Proceedings of the 2023 Design, Automation and Test in Europe Conference and Exhibition (DATE), Antwerp, Belgium, 17–19 April 2023; pp. 1–2.
48. Murray, K.E.; Betz, V. Adaptive FPGA Placement Optimization via Reinforcement Learning. In Proceedings of the 2019 ACM/IEEE 1st Workshop on Machine Learning for CAD (MLCAD), Canmore, AB, Canada, 3–4 September 2019; pp. 1–6.
49. Lin, T.R.; Penney, D.; Pedram, M.; Chen, L. A Deep Reinforcement Learning Framework for Architectural Exploration: A Routerless NoC Case Study. In Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), San Diego, CA, USA, 22–26 February 2020; pp. 99–110.