EncoDeep: Realizing Bit-Flexible Encoding for Deep Neural Networks

MOHAMMAD SAMRAGH[*], University of California San Diego
MOJAN JAVAHERIPI[*], University of California San Diego
FARINAZ KOUSHANFAR, University of California San Diego

This paper proposes EncoDeep, an end-to-end framework that facilitates encoding, bitwidth customization, fine-tuning, and implementation of neural networks on FPGA platforms. EncoDeep incorporates nonlinear encoding to the computation flow of neural networks to save memory. The encoded features demand significantly lower storage compared to the raw full-precision activation values; therefore, the execution flow of EncoDeep hardware engine is completely performed within the FPGA using on-chip streaming buffers with no access to the off-chip DRAM. We further propose a fully-automated optimization algorithm that determines the flexible encoding bitwidths across network layers. EncoDeep full-stack framework comprises of a compiler which takes a high-level Python description of an arbitrary neural network. The compiler then instantiates the corresponding elements from EncoDeep Hardware library for FPGA implementation. Our evaluations on MNIST, SVHN, and CIFAR-10 datasets demonstrate an average of $4.65\times$ throughput improvement compared to stand-alone weight encoding. We further compare EncoDeep with six FPGA accelerators on ImageNet, showing an average of $3.6\times$ and $2.54\times$ improvement in throughput and performance-per-watt, respectively.

CCS Concepts: • **Computer systems organization → Neural networks**; **Reconfigurable computing**; • **Hardware → Reconfigurable logic and FPGAs**; **Hardware accelerators**; *Emerging tools and methodologies.*

Additional Key Words and Phrases: Resource-customized Computing, Automated Optimization, Neural Network Customization

ACM Reference Format:
Mohammad Samragh, Mojan Javaheripi, and Farinaz Koushanfar. 2020. EncoDeep: Realizing Bit-Flexible Encoding for Deep Neural Networks . *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2020), 25 pages. https://doi.org/10.1145/3391901

1 INTRODUCTION

Deep Neural Networks (DNNs) are being widely developed for various machine learning applications, many of which are required to run on embedded devices. In the realm of embedded DNNs, real-time execution under severe power limitations is hard to satisfy [20, 32]. Contemporary research has focused on the FPGA-based acceleration of DNNs [8, 10, 50, 52]. However, FPGAs are inherently limited in terms of on-chip memory capacity. Thus, the high-storage requirement of DNN models hinders an efficient and low power execution on FPGAs.

To reduce the computational complexity and memory requirement of DNNs, several pre-processing algorithms have been proposed. The existing methods generally convert conventional DNNs into compact representations that are better suited for execution on embedded devices. Examples of such compacting methods include quantization [21, 40], binarization [10, 53], tensor decomposition [29], parameter pruning [57], and compression with nonlinear encoding [13,

[*]Both authors contributed equally to this research.

Authors' addresses: Mohammad Samragh, msamragh@ucsd.edu, University of California San Diego; Mojan Javaheripi, mojan@ucsd.edu, University of California San Diego; Farinaz Koushanfar, farinaz@ucsd.edu, University of California San Diego.

1539-9087/2020/1-ART1 $15.00
https://doi.org/10.1145/3391901

45]. A higher compression rate might not always translate to better hardware performance as the platform constraints could interfere with the intended compaction methodology [58].

This paper specifically focuses on nonlinear encoding and provides solutions to tackle the challenges associated with optimizing physical performance. Encoding network parameters is rather beneficial as it reduces the memory footprint, i.e., the main source of delay and power consumption in FPGA accelerators. To devise a practical solution for implementing encoded DNNs, we simultaneously identify and address four critical issues.

DNN memory footprint is imposed by either weights or feature-maps. Figure 1 shows the relative memory requirements in several popular DNN models. As can be seen, the memory footprint of activations is notable; however, contemporary research mainly targets the (static) DNN weights for nonlinear quantization [4, 13, 45]. Firstly, developing online mechanisms for activation encoding can significantly reduce the memory footprint of DNN models. Secondly, nonlinear quantization destabilizes DNN training by adding non-differentiable elements to the model. Therefore, novel computation routines must be developed to approximate gradients for DNN fine-tuning. Thirdly, specifying the encoding bitwidth across all DNN layers by handcrafted try-and-error is exhaustive and generally sub-optimal. Hence, automated and intelligent solutions for bitwidth optimization are highly preferable. Finally, designing accelerators that are customized per application/hardware is cumbersome. Thus, easy-to-use tools are needed to ensure low, non-recurring engineering costs.

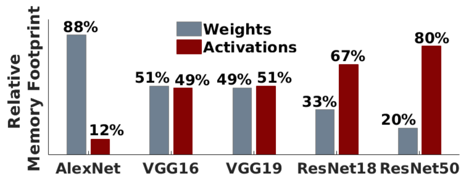

Fig. 1. Relative memory footprint of weights and activations for various DNNs, evaluated on 10 samples of ImageNet.

To tackle the aforementioned challenges, we introduce EncoDeep, a unified framework that facilitates encoding, training, bitwidth customization, and automated implementation of encoded DNNs on FPGA platforms. EncoDeep software stack allows users to automatically configure the encoding bitwidth across all DNN layers and retrain the encoded DNN. The hardware library of EncoDeep provides a set of configurable DNN layers that can be instantiated to compose a fully-functional DNN. In summary, the contributions of this paper are listed as follows:

- Introducing a novel methodology for the (online) encoding of DNN **activations**. We establish the gradient computation routines required to fine-tune encoded DNNs, enabling restoration of DNN accuracy after encoding.
- Introducing an automated algorithm for customizing per-layer encoding bitwidths. Inspired by reinforcement learning, we establish an action-reward-state system to find a bitwidth configuration that minimally affects DNN accuracy while maximally reducing memory footprint.
- Establishing a hardware library for the bit-flexible implementation of customized encoded DNN layers. Activation encoding lowers memory footprint and facilitates the use of streaming buffers for inter-layer feature transmission.
- Providing an API for fast and easy hardware implementation of encoded DNNs. Developers describe the DNN as high-level Python code which is then automatically converted to Vivado_HLS.
- Performing extensive evaluations on various datasets and DNN architectures. On MNIST, SVHN, and CIFAR-10, EncoDeep demonstrates an average of $4.65\times$ throughput improvement compared to stand-alone weight encoding. To uncover the benefits of encoding, we compare EncoDeep with six fixed-point FPGA accelerators on ImageNet, showing an average of $3.6\times$ and $2.54\times$ improvement in throughput and performance-per-watt, respectively.

2 OVERVIEW AND INSIGHTS

EncoDeep design flow is composed of an interlinked optimization scheme where algorithmic DNN compaction methods and hardware-level customization are performed in sync. In this section, we describe EncoDeep insights in high-level and look at the main components of our framework.

2.1 Streaming-based On-chip Execution

Traditional DNN accelerators store the weights and activations (features) of layers in the off-chip DRAM since commodity FPGAs are often limited in terms of on-chip memory capacity. Figure 2 (top) demonstrates the computation flow of DNNs in such settings. Alternatively, the weights and computed activations could be stored and accessed within the FPGA design using streaming buffers as depicted in the bottom of Figure 2. The benefits of the latter approach are three-fold: (i) it avoids the power-hungry and high-latency access to off-chip DRAM. (ii) The computation engines responsible for each DNN layer can be customized to comply with the pertinent layer. (iii) The streaming buffers allow pipelining for the computation engines to increase throughput. Although on-chip execution of DNNs is beneficial in many aspects, the memory requirement for weights and activations of DNN layers is often beyond the (limited) capacity of commodity FPGAs. To address this, EncoDeep employs nonlinear quantization to reduce memory footprint such that the weights/activations can be accommodated within FPGA block-RAMs.

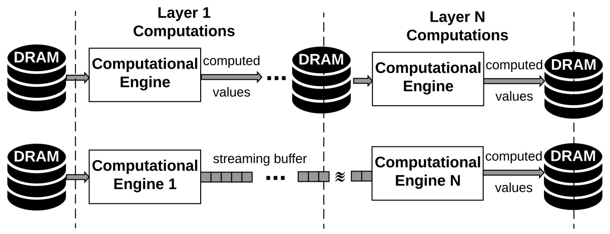

Fig. 2. The workflow of traditional vs. streaming-based DNN inference. The top diagram shows the conventional approach where all resources are allocated to one computational engine and layer input/outputs are continuously read/written from/to off-chip memory. The bottom diagram presents a streaming-based approach where each layer is allocated a different computational engine and communications with off-chip memory are limited to the first/last layer.

2.2 Memory Compression

Quantization allows a reduction in memory footprint by approximating numerical values. To perform quantization, a finite set of best representatives (a.k.a. bins) are selected and each value is approximated with the closest bin. Perhaps the most popular quantization is fixed-point approximation. Figure 3-a depicts the bins for an unsigned fixed-point quantization. In this setting, quantization bins are fixed to certain points (e.g., $\{0.00, 0.25, 0.50, 0.75\}$), regardless of data distribution. Alternatively, in nonlinear quantization, the bins are carefully selected to best represent the data as shown in Figure 3-b. In this example, both fixed-point and nonlinear quantizations require the same number of bits to represent the (approximated) data: each real-valued signal can be represented with 2 bits when there are 4 quantization bins. However, the approximation error associated with the nonlinear scheme is drastically lower.

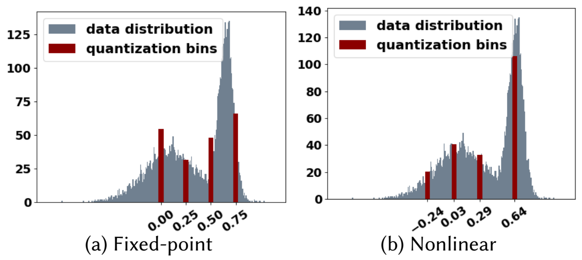

Fig. 3. Histogram of data samples and the quantization bins in fixed-point and nonlinear quantization. In this example, there are 4 quantization bins and scalars are encoded with $Log_2(4) = 2$ bits.

For a fixed nonlinear quantization bitwidth, the approximation error increases as the standard deviation σ of the data increases. Nevertheless, this error can be compensated by increasing the number of quantization bits; Figure 4 shows that for a large enough number of bits ($b > 6$ in this example), the error converges to zero. DNNs inherently have a low standard deviation due to specific measures taken during training to ensure convergence. In particular, to avoid exploding gradient values and promote a smooth convergence, contemporary DNNs comprise *batch normalization* which normalizes layer activation values. Moreover,

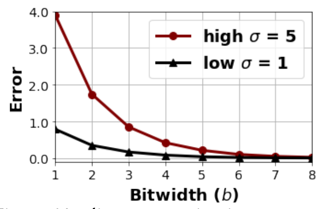

Fig. 4. Nonlinear quantization error versus bitwidth for two Gaussian data distributions with $\sigma = 1$ and $\sigma = 5$.

to prevent over-fitting and drastic neuron transitions, weight regularization is used during training to suppress large weights. We empirically demonstrate this property in Figure 5 by plotting the σ range across DNN layers for all our benchmarks. Such small σ range allows low-error estimation of DNN parameters/activations with very few bits.

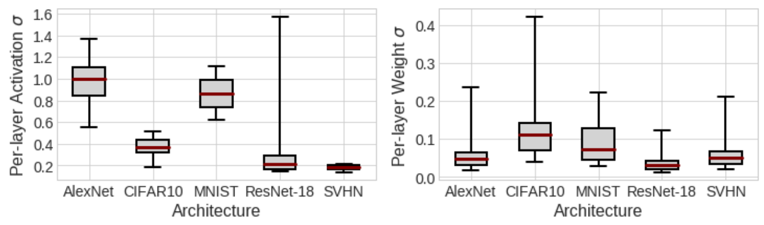

Fig. 5. Standard Deviation (σ) range for activations (left) and weights (right) across DNN layers. Here, the black dot represents the mean σ for each benchmark.

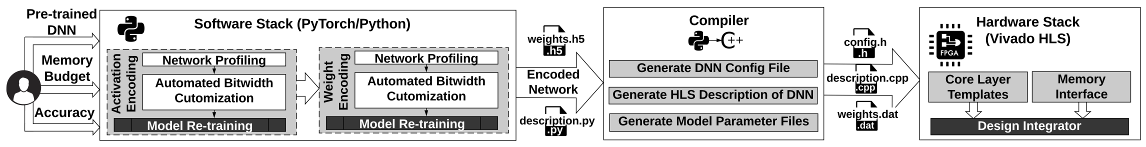

Fig. 6. The global flow of EncoDeep framework. User provides a high-level Python description of a pre-trained DNN to the software stack, which is responsible for weight/activation encoding, layer-specific bitwidth configuration, and model fine-tuning. Our compiler converts the Python code into a hardware description. The hardware stack then uses a customized library for FPGA synthesis.

2.3 Global Flow

Figure 6 depicts the global flow of EncoDeep framework. EncoDeep is composed of three interlinked design units, namely the *Software Stack* (also referred to as the *Encoding Engine*), the *Compiler*, and the *Hardware Stack*. EncoDeep aims at alleviating the complications of DNN implementation on FPGAs by incorporating an automated design stack that separates users from the details of hardware design and optimization. We implement an end-to-end automated framework that eliminates all hand-optimizations and delivers a customized accelerator implementation for various DNN architectures and FPGA platforms.

EncoDeep leverages a novel and fully automated learning algorithm to output a maximally efficient DNN architecture in terms of memory footprint while adhering to the accuracy constraints provided by the user. The key insight of EncoDeep is capturing the trade-off between the classification accuracy and memory footprint of model parameters and feature maps (activations). We use the popular neural network development API, PyTorch, to describe the DNNs in the software stack. To implement the inference engines on FPGA, we choose Vivado High-Level Synthesis (HLS) which enables faster development as well as portability. To fill the bridge between the software and the hardware stacks, we develop a compiler unit in Python. Below, we elaborate more on the incorporated design units.

Software Stack. The software stack is responsible for weight/activation encoding, layer-specific bitwidth configuration, and model fine-tuning. This step analyses the input DNN and applies nonlinear quantization (encoding) to layer weights/activations. EncoDeep encoding scheme reduces memory footprint at the cost of a small reduction in inference accuracy. We devise an automated algorithm to determine the number of encoding bins in each layer for the weights and activations such that the memory footprint is maximally reduced and/or the accuracy is minimally affected. The following steps are performed sequentially in the software stack:

- **Activation Encoding.** This step takes as input a pre-trained DNN described in Pytorch format and generates a network with encoded activations.

- **Weight Encoding.** This step takes the DNN from the activation encoding step as input and generates a network with encoded weights and activations.

Both the activation and weight encoding steps consist of three tasks: (i) network profiling where the accuracy-memory trade-off is captured by calculating the correlation between accuracy loss and memory footprint reduction (Section 3.3). (ii) Bitwidth selection where the bitwidth of encoded activations/weights is customized based on the user-defined accuracy/memory budget primitives (Section 3.3). The encoded activations/weights are then inserted in the DNN graph to replace the full-precision values (Section 3.1). (iii) Model re-training where the encoded DNN is fine-tuned to improve classification accuracy (Section 3.2). The output of the EncoDeep software stack is an encoded architecture and the trained encoded network's parameters.

Compiler. To ensure ease-of-use and design automation, we design a customized compiler. This unit takes as input the high-level DNN graph description in PyTorch format and converts it to C++ code (as used in the Vivado HLS tool). EncoDeep compiler produces a configuration file that specifies the customized encoding bitwidths for the weights and feature maps of different DNN layers. The network description in C++ together with the configuration file enable instantiation of core layer template modules. The compiler further converts the trained encoded network's parameter into a format ready to be loaded to the on-chip memory of the FPGA upon execution.

Hardware Stack. The hardware description of the DNN is rendered using Vivado_HLS, which is a standard high-level-synthesis tool that enables faster development as well as portability. EncoDeep accelerator enjoys full-precision calculations while maintaining low memory footprint using the encoded values. We provide a library of template modules that can realize different DNN functionalities. An arbitrary architecture can be described by instantiating the corresponding core layer templates in a network description file. Each template module has customized configurable primitives such as the number of input/output neurons of the layer, the bitwidth of the weights/activations, and the parallelism factors for execution. The output of the hardware stack is a bitfile that can be used to efficiently execute the desired DNN on the FPGA. We will elaborate more on the hardware in Section 4.

3 ENCODEEP SOFTWARE STACK

In this section, we elaborate on the utilized concepts for non-linear encoding of DNN parameters/activations. Section 3.1 explains EncoDeep weight/activation encoding. Our gradient computation for encoded network training is formulated in Section 3.2. Finally, our automated bitwidth selection routine is explained in Section 3.3.

3.1 Encoding Scheme

Our encoding scheme aims to estimate the parameters of a DNN layer with a subset of representatives, i.e., the *codebook*. In the rest of this section, we delineate EncoDeep encoding method for DNN weights/activations.

3.1.1 ***Weight Encoding.*** Let us denote the weight parameters in a certain DNN layer as W. In order to encode W, we first find an approximation $\widetilde{W} \approx W$ such that the elements of $\widetilde{W}$ are restricted to a finite set of real-values, $\vec{c} = \{c[1], \dots, c[K]\}$, i.e., the *codebook*. The encoded weight matrix is then constructed by replacing all elements with indices of the corresponding codebook values. We denote the encoded $\widetilde{W}$ as W_{enc}. Figure 7 illustrates this approximation for a 4×4 matrix W using a codebook of $K = 2$ elements.

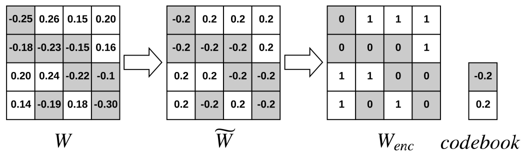

Fig. 7. Illustration of EncoDeep weight encoding. left: original matrix W, middle: approximated matrix $\widetilde{W}$, right: encoded matrix W_{enc} along with the codebook.

To approximate $\widetilde{W}$, we use the well-known K-means clustering [13]. While K-means can effectively solve the aforementioned problem for a fixed codebook size, specifying the codebook sizes in different layers of a network is a challenge yet to be solved. Specifically, different layers require different codebook sizes to capture the statistical properties of their parameters. To tackle this, EncoDeep proposes an automated bitwidth selection algorithm explained in Section 3.3. Note that weight encoding is performed only once in an offline pre-processing step. The per-layer encoded weights and codebooks are then stored in binary files to be loaded in the FPGA memory.

3.1.2 ***Activation Encoding.***

EncoDeep activation encoding is performed in two phases: (i) offline phase performed in the software stack, where the layer codebooks are generated using the K-means algorithm. (ii) Online phase performed during inference where each feature is encoded by its closest codebook value.

Offline Encoding. Algorithm 1 summarizes our methodology for computing DNN activation codebooks. First, a subsampled data set, $\{\vec{x}_n\}_{n=1}^N$, is used to generate the layer feature-maps, which we denote by $\vec{y}^{\,l}$. Next, $\vec{y}^{\,l}$ is flattened into an array, $\vec{a}^{\,l}$. For an arbitrary activation function, the K-means clustering is applied on all values of $\vec{a}^{\,l}$. For the especial case of $ReLU$ activations, since the `ReLU` non-linearity produces many 0-valued outputs, we only perform K-means on non-zero elements of $\vec{a}^{\,l}$ to reduce the K-means clustering runtime. The $(K^l - 1)$ cluster centers along with the appended 0 value form the codebook for the l^{th} layer.

Using a subsampled dataset for finding the cluster centers enables for a fast and efficient search over the space of possible codebook sizes, i.e., encoding bitwidths (see Section 3.3). To ensure that the obtained cluster values are truly compatible with the distribution of layer feature-maps, we later fine-tune the cluster center values via customized gradient operations explained in Section 3.2.

Online Encoding. Online encoding is performed during FPGA execution. The value of a feature y is compared with the elements of the corresponding layer's codebook $\vec{c}_{act}^{\,l}$ to compute the encoding as $y_{enc} = argmin(|y - \vec{c}_{act}^{\,l}|)$. This is implemented by a linear search on a small memory block containing the codebook values (Section 4.1).

Algorithm 1 Offline Activation Encoding

Inputs: input samples $\{\vec{x}_n\}_{n=1}^N$, per-layer codebook sizes $\{K^l\}_{l=1}^L$
Output: per-layer codebooks for activations $\{\vec{c}_{act}^{\,l}\}_{l=1}^L$

```
1: for $l = 1, ..., L$ do
2:     $\vec{y}^{\,l} \leftarrow DNN^l(\{\vec{x}_n\}_{n=1}^N)$
3:     $\vec{a}^{\,l} \leftarrow flatten(\vec{y}^{\,l})$
4:     $\vec{a}^{\,l} \leftarrow nonZeros(\vec{a}^{\,l})$
5:     $\vec{c}_{act}^{\,l} \leftarrow KMeans(\vec{a}^{\,l}, K^l - 1)$
6:     $\vec{c}_{act}^{\,l} \leftarrow \{0, \vec{c}_{act}^{\,l}\}$
7: end for
```

3.2 Training of Encoded Networks

Encoding weights/parameters often results in a drop in accuracy. To compensate for such accuracy loss, the codebook entries are fine-tuned after encoding using a customized back-propagation scheme. In this section, we explain the details for fine-tuning encoded neural networks via Stochastic Gradient Descent (SGD) [28]. For weights, the averaged gradient method [4, 13] is applied. For activations, we develop new gradient computation methods.

Feature encoding can be viewed as a non-linear transformation, $f(y) = y^*$, where y^* and y represent the approximated and original values, respectively. As depicted in Figure 8, the non-linear encoding function is made up of multiple *step* functions, rendering it non-differentiable. Given the gradient of the loss function with respect to the encoded values, $\nabla_{y^*} = \frac{\partial \mathcal{L}}{\partial y^*}$, we aim to compute the partial derivatives with respect to the non-encoded values ($\nabla_y = \frac{\partial \mathcal{L}}{\partial y}$) and the derivatives with respect to the codebook ($\vec{\nabla}_c = \frac{\partial \mathcal{L}}{\partial c}$).

Computing $\nabla_\mathbf{y}$. Given the partial derivative ∇_{y^*}, the gradient ∇_y can be obtained by applying the chain rule:

$$\nabla_y = \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial y^*} \times \frac{\partial y^*}{\partial y}. \tag{1}$$

This formulation, however, is not stable since the function $f(\cdot)$ is non-differentiable. To address this issue, we propose to approximate the derivative of $f(\cdot)$ as:

$$\frac{\partial f(y)}{\partial y} = \begin{cases} 1 & if\ c[1] < y < c[K] \\ 0 & otherwise \end{cases}, \tag{2}$$

where $c[1]$ and $c[K]$ are the smallest and largest codebook values, respectively. During forward propagation, y^* is computed as shown in Figure 8-left, whereas the backward propagation assumes the smooth function in Figure 8-right.
Computing $\nabla_\mathbf{c}$. Given a scalar gradient element ∇_{y^*}, the gradient with respect to $c[k]$ is computed as:

$$\vec{\nabla}_c[k] = I(c[k], y^*) \times \nabla_{y^*}, \tag{3}$$

with $I(a, b) = 1$ if $a = b$ and zero otherwise (identity operator). Given a vector of features $\vec{y}^*$ and the corresponding vector of gradients $\vec{\nabla}_{y^*}$, the derivative is:

$$\vec{\nabla}_c[k] = \sum_j I(c[k], \vec{y}^*[j]) \times \vec{\nabla}_{y^*}[j]. \tag{4}$$

Using the partial derivatives, standard back-propagation algorithms can fine-tune DNN parameters. We incorporate the customized gradient computation routines into EncoDeep software stack to support fine-tuning for encoded DNNs. As shown in the evaluations, the fine-tuning incurs negligible overhead compared to original training.

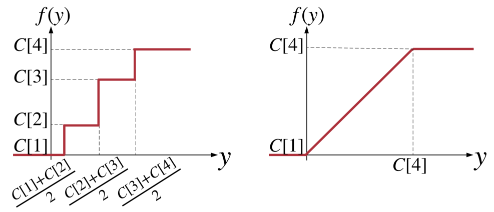

Fig. 8. Example encoding non-linearity with a codebook of $K = 4$ elements. (Left) Non-linear function applied in the forward propagation. (Right) Smooth approximation of encoding used in backward propagation for gradient computation.

3.3 Automated Bitwidth Selection

Modern DNNs are composed of many layers with high dimensional input/output parameter space. In order to successfully reduce the memory footprint of such networks while minimally affecting the classification accuracy, one is required to customize the memory compression rate on a per-layer basis. EncoDeep automated bitwidth selection aims to adjust the encoding bitwidth (determined by the codebook size) for each layer such that the network's overall memory footprint is minimized while adhering to the user-provided accuracy constraint. To this end, an efficient algorithm is desired that can search the space of possible bitwidth configurations for the optimal solution.

Recent advances in Reinforcement Learning (RL) provide a powerful automated tool for effective search. In high-level, RL approaches traverse a series of states s by taking subsequent actions $a = \pi(s)$ based on a policy function $\pi(\cdot)$. Here, $\pi(\cdot)$ corresponds to a probability distribution over actions given states. Applying an action a at state s triggers a state transition $s \longrightarrow s'$ and results in a reward $r(a, s)$ from the environment. In the training phase of RL, the goal is to find a series of actions that return the best discounted sum of future rewards. This is achieved by tuning the policy $\pi(\cdot)$ to incorporate the *long-term* return (reward) in the action-selection process. During RL training, the policy model is learned over a series of episodes $\{E_1, E_2, \dots\}$; each episode consists of all transitions form the initial state to the final state, given the policy $\pi(\cdot)$:

$$E_i : s_1 \xrightarrow{\pi_i(s_1)} s_2 \xrightarrow{\pi_i(s_2)} \dots \xrightarrow{\pi_i(s_{N-1})} s_N.$$

Training the RL policy can generally be time-consuming as it requires many training episodes and evaluations. To overcome this challenge, we propose an algorithm *inspired* by RL that does not learn probabilistic policies and relies solely on immediate rewards. Our method comprises only *one* episode where the path from the initial state to the end state is traversed *deterministically* by choosing greedy actions:

$$E_{greedy} : s_1 \longrightarrow s_2 \longrightarrow \cdots \longrightarrow s_N.$$

In this context, greedy actions are those with the maximum *immediate* reward when transitioning from state s_i to the next state s_{i+1}. By incorporating immediate rewards, EncoDeep can solve the multi-objective optimization problem at hand in a fraction of pure RL optimization time. Similar to RL, we define a state-action-reward system where the *state* is the encoding bitwidth in DNN layers at the current iteration of the algorithm. In the beginning, all layers are encoded with a maximum bitwidth (e.g, 4 bits for a 16-element codebook). The action-space for each state s, corresponds to all permissible actions that can be taken from that state. Each *action* chooses a layer l and reduces the corresponding bitwidth b_l to $b \in \{1, 2, \dots, b_l - 1\}$. The action-space thus encloses all bitwidth configurations where only a single layer's bitwidth differs from that in state s. The *reward* for each action is formulated as follows:

$$r(a, s) = \frac{mem(s) - mem(s^{'})}{acc(s) - acc(s^{'})}, \tag{5}$$

where a is the action, s is the current state, $s^{'}$ is the state after taking action a, $mem(\cdot)$ and $acc(\cdot)$ denote the total memory footprint and accuracy at a given state, respectively. The accuracy is computed by evaluating the (encoded) network on a validation dataset and the total memory for encoded weights is formulated as:

$$mem(weights) = \sum_{l=1}^{L} size(W^l) \times Log_2(K^l) + K^l \times b_{fix}, \tag{6}$$

where the $size(\cdot)$ operator returns the number of elements, K^l is the weight codebook size corresponding to the l-th layer, and b_{fix} is the fixed-point bitwidth of each codebook element. With Y^l being the output feature map of the l-th layer, the total memory footprint for the activations of the neural network is computed as:

$$mem(acts) = \sum_{l=1}^{L} size(Y^l) \times Log_2(K^l) + K^l \times b_{fix}, \tag{7}$$

Taking an action in a given step will decrease both memory footprint and accuracy. Hence, the reward function is always positive. At each state, all actions in the action-space are evaluated and the one with maximum reward is chosen. Such a greedy approach is particularly beneficial for the problem statement at hand, i.e., bitwidth configuration, as the value of each state transition can be independently evaluated without relying on the end state and the long-term return. Compared to pure RL, which includes many episodes of policy training, our greedy approach renders drastically lower computation time. Moreover, EncoDeep is able to extract the memory-accuracy Pareto curve with only one state traversal (episode). Rather, in conventional RL settings, policy training must be repeated once for each target memory, further increasing runtime.

We visualize EncoDeep search method in Figure 9. Here, we use an example 2-layer neural network that is initialized to 3-bit encodings for both layers. At state 0, there are 4 possible actions, each of which has a certain reward that can be computed using Equation 5. At this state, the second action renders the maximum reward and therefore the next state's encoding bitwidths are selected as 3 for the first layer and 1 for the second. This process continues until either the accuracy drops below the user-defined threshold or all layers are encoded with 1-bit values.

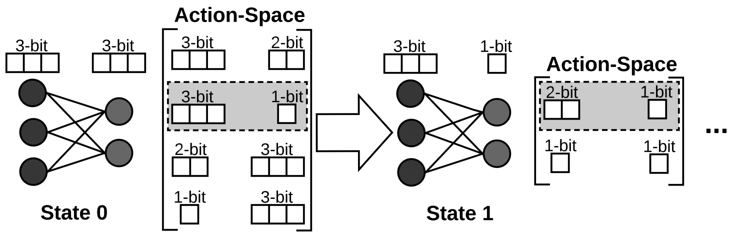

Fig. 9. Automated bitwidth selection for a 2-layer perceptron.

Note that, after choosing the optimal action at each step of the algorithm, all actions resulting in a memory footprint higher than the new state are eliminated from the search space. This enables a diminishing search cost per iteration of the bitwidth selection method. We also emphasize that the iterative bitwidth selection algorithm does not perform any re-training of the DNN in between the steps. As a result, the (offline) computational overhead of bitwidth customization is drastically smaller than that of RL techniques.

The pseudo-code for the EncoDeep automated bitwidth customization is presented in Algorithm 2. The inputs are the starting encoding bitwidth B for all layers, a minimum threshold θ for the classification accuracy, and a pre-trained DNN model D. The algorithm then gradually decreases the bitwidths, one layer at a time. The algorithm outputs a set of configurations that specify per-layer bitwidths. These configurations capture the tradeoff between memory and accuracy: the first configuration has the highest memory and accuracy whereas the last configuration has the lowest.

Algorithm 2 Automated Bitwidth Customization

Inputs: maximum bitwidth B, minimum accuracy threshold θ, DNN model D
Output: list of bitwidth configurations $\{cfg_1, cfg_2, \dots\}$ that render the optimal accuracy-memory tradeoff.

```
1:  $cfg \leftarrow \{b_1 = B, \dots, b_L = B\}$
2:  $AllConfigs \leftarrow \{cfg\}$
3:  $A = Acc(D|b_1, \dots, b_L)$
4:  $M = Mem(D|b_1, \dots, b_L)$
5:  while $A > \theta$ do
6:      for $l = 1, \dots, L$ do
7:          for $b = 1, \dots, b_l - 1$ do
8:              $A(l,b) = Acc(D|\{b_1, \dots, b_l = b, \dots, b_L\})$
9:              $M(l,b) = Mem(D|\{b_1, \dots, b_l = b, \dots, b_L\})$
10:             $reward(l,b) = \frac{M-M(l,b)}{A-A(l,b)}$
11:         end for
12:     end for
13:     $\{l,b\} \leftarrow \arg\max_{l,b} reward(l,b)$
14:     $cfg \leftarrow \{b_1, \dots, b_l = b, \dots, b_L\}$
15:     $A = Acc(D|cfg\})$
16:     $M = Mem(D|cfg\})$
17:     $AllConfigs \leftarrow \{AllConfigs, cfg\}$
18: end while
19: return $AllConfigs$
```

4 ENCODEEP HARDWARE STACK

There are several components for DNN inference on FPGA as shown in Figure 10, namely, the processing system (PS), programmable logic (PL), external memory Double Data Rate (DDR), and (on-chip) streaming buffers. The PS and PL are interconnected via an AXI4 bus. Fortunately, modern design tools such as Xilinx SDx enable automatic instantiation of memory interfaces and memory management logic, e.g., the AXI Interconnect and AXI DMA. As depicted in Figure 10, the execution flow starts when a user runs a *host code* within the host CPU. This host code may call a *kernel* function, which is run by the FPGA core, i.e., the PL. We have implemented two main kernels for EncoDeep execution: weight/bias initialization and DNN inference. The former kernel initializes the weights in specific SRAM blocks (a.k.a. BRAMs) of the FPGA. The initialization values include DNN layer parameters as well as the input/weight codebooks, which are gathered by EncoDeep compiler and stored on the DDR, along with the input data. The initialization kernel is triggered from the PS after providing the desired write values and the target memory indices via the AXI bus. For our accelerator, the weight/bias initialization is done once before DNN inference and afterward, the accelerator can run multiple inferences without parameter re-initialization.

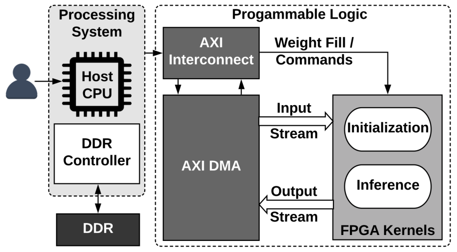

Fig. 10. Interconnections between EncoDeep accelerator and the host CPU.

EncoDeep inference kernel adopts a streaming-based architecture that facilitates pipelining and overlays the computational overhead of subsequent layers to increase overall throughput and minimize latency. Figure 11 presents such pipelined execution for a 3-layer DNN. Due to this streaming-based on-chip design, during the inference kernel execution, PS and PL merely perform two rounds of communication: the PS sends the input image to the PL and receives the output logits from the PL once the whole-network computation is finished. The input and output are transmitted in AXI4 streaming format, controlled by the AXI Direct Memory Access (DMA) module.

Our accelerator is specifically designed to accommodate low-bitwidth encoded networks while supporting full-precision computations. Figure 12 compares the computational flow of EncoDeep with conventional fixed-point accelerators. In the conventional design (top), a convolution (CONV) or Fully-Connected (FC) layer receives the inputs and weight parameters in fixed-point format. Each layer starts the computation as soon as its preceding layer starts generating output. The streaming buffers and the weight memory should thus accommodate high bitwidth full-precision values (e.g., 32 bits in Figure 12). In practice, due to the low capacity of the on-chip memory in off-the-shelf FPGA platforms and the high number of parameters/features in state-of-the-art DNNs, it is not feasible to accommodate all weights and/or streaming buffers inside the FPGA. In response to this issue, we propose the encoded DNN data flow presented in the bottom schematic of Figure 12. Here, the weights are stored in the encoded format to save memory. The computed outputs of each layer are also encoded before being sent through the streaming buffer to enable use of low-capacity buffers. The CONV and FC layers of the DNN are therefore equipped with encoder and decoder modules.

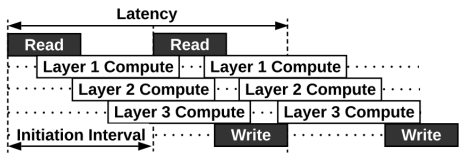

Fig. 11. Pipelined execution of layer computations in a streaming-based architecture increases throughput.

EncoDeep is equipped with a hardware library described in high-level synthesis language that allows FPGA implementation of encoded DNNs. Our hardware library consists of the essential building blocks to implement encoded DNN layers (e.g, convolution, max-pooling, etc.). Each layer-type is implemented as a template function with certain

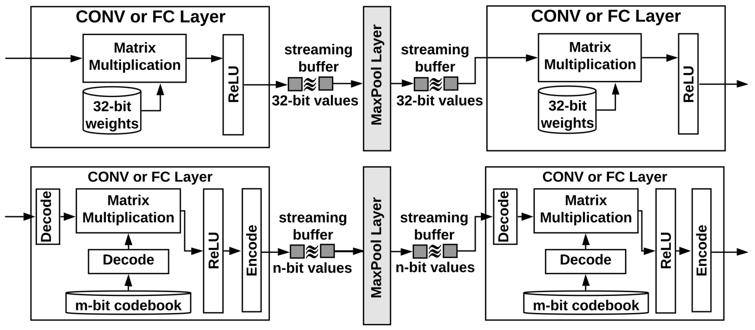

Fig. 12. Computational flow of a conventional DNN (top) and our proposed DNN with encoded weights and activations (bottom).

computing engines that are customized to the specifications of the pertinent layer such as the input/output dimensions. By means of this tailoring, EncoDeep exploits the benefits of FPGA reconfigurability and delivers a bit-flexible design.

To ensure portability and efficiency, we chose an open-source framework [53], i.e., FINN, from Xilinx as the base of our hardware accelerator. The FINN library was originally intended for the execution of binary neural networks and cannot be used for encoded DNNs as is. We thus extend the library to support customized streaming buffers that can accommodate flexible bitwidths rather than binary values across layers. We further design a data scheduling unit, dubbed the Sliding Window Unit, that reorders and populates layer input buffers in accordance with the underlying bitwidth. We implement new processing engines that support operations on encoded parameters/weights and fixed-point Multiply-Accumulate (MAC) operations, which replace the *XnorPopCount* operations required in BNNs [53]. Using our proposed encoding scheme, EncoDeep enjoys the benefits of on-chip buffering and high-precision MACs.

Figure 13 depicts the flow diagram of the EncoDeep accelerator for implementing an encoded DNN on FPGA. The Sliding Window Unit (SWU) reorders the convolution layer input feature-maps to generate appropriate streaming buffers for the Matrix-Vector-Activation Unit (`MVAU`). The `MVAU` is the core computational module of `CONV` and `FC` layers which performs the matrix-vector multiplication, activation, and batch normalization. The Max-Pooling Unit (MPU) performs max-pooling over the feature-maps. In the following, we discuss the core modules in EncoDeep Hardware.

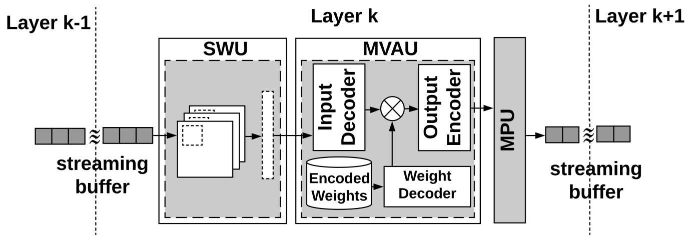

Fig. 13. EncoDeep accelerator schematic for encoded DNN inference. SWU reorders the input buffer; `MVAU` and MPU perform core computations and max-pooling, respectively.

4.1 Matrix-Vector-Activation Unit (`MVAU`)

The `MVAU` in EncoDeep hardware library is instantiated in convolution (`CONV`) and fully-connected (`FC`) layers to generate output features using the corresponding layer's specifications. Figure 14 illustrates the `MVAU` computational flow. This module performs three core tasks required in state-of-the-art DNNs, namely matrix-vector multiplication, batch normalization, and applying non-linear activation. Internally, the `MVAU` is composed of an array of Processing Engines (`PE`s) which accept a shared lane of `SIMD` inputs in parallel. In addition, EncoDeep `MVAU` has customized encoding/decoding cores for processing the outputs, inputs, and weights.

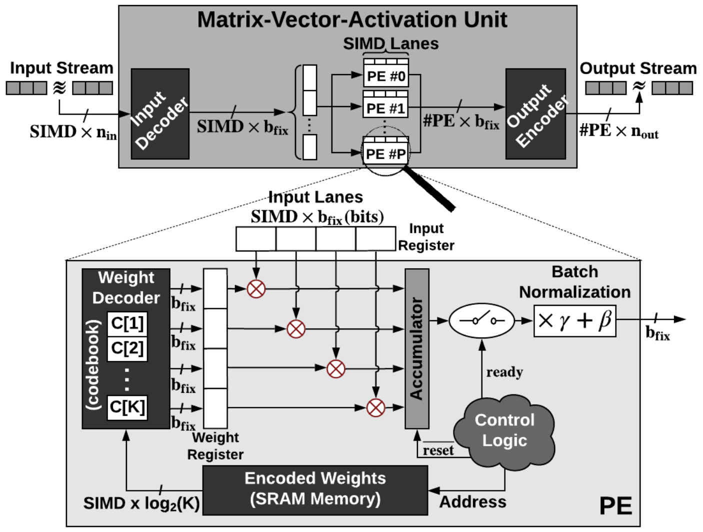

Fig. 14. **(Top)** Computational flow of EncoDeep MVAU. This unit performs matrix-vector multiplication, batch normalization, and a non-linear activation. To increase throughput, the core computations in the MVAU are distributed across parallel PEs. The MVAU is further equipped with an input decoder, an output encoder, and several weight decoders (one per-PE) to comply with EncoDeep encoded DNNs. **(Bottom)** Internal configuration of a PE. Each PE performs parallel MAC operations over SIMD operands.

Matrix-Vector Multiplication. The main operations performed in linear DNN layers can be represented as a series of matrix-vector multiplications. The matrix-vector multiplication core in EncoDeep MVAU offers two levels of parallelism to facilitate throughput control across DNN layers.

- Layer output generation is distributed among several PEs working in parallel. In this setting, each PE is responsible for generating the output of multiple feature-map channels (neurons) in a CONV (FC) layer. For instance, in a CONV layer with 64 output channels and 16 PEs, each PE is responsible for computing 4 output channels.
- Each PE operates in single-instruction-multiple-data (SIMD) mode: MACs in a PE are parallelized across SIMD lanes.

The MAC operations in EncoDeep are performed in fixed-point on decoded input/weight values, implemented using DSP slices. The per-layer encoded weight matrix is stored in the on-chip memory of the FPGA and is partitioned among all PEs within the pertinent layer. This partitioning allows all PEs to simultaneously access their share of weights. Note that the computations performed inside each PE are independent of those performed in the neighbor PEs. Thus, there is no need for inter-PE communication. We elaborate on the internal configuration of a PE in Section 4.1.1.

Decoder Modules. The encoded features/weights of EncoDeep are converted into the equivalent fixed-point format before being used in matrix-vector multiplication. Each layer in the encoded DNN contains two decoding codebooks corresponding to the inputs (activations) and weights. This functionality is implemented by a memory containing all cluster centers (codebook values) stored in fixed-point format (e.g, 32 bits). For an encoded value $y_{enc} \in \{1, \ldots, K\}$ the corresponding fixed-point approximation y^* can be obtained by feeding y_{enc} as the address of a memory block storing the cluster centers $\{c_1, \ldots, c_K\}$. Note that the cluster centers incur a negligible memory footprint since K is small.

The decoder modules are implemented via register files, rather than SRAM blocks. This design choice allows for simultaneous decoding of SIMD inputs, in parallel. To achieve maximum efficiency, the input decoder is implemented inside the MVAU. This enables us to share the decoded inputs among all PEs within one MVAU. Unlike inputs which are shared among PEs, the weights for each PE are different. Therefore, to facilitate parallelism, each PE owns a copy of the corresponding weight decoder (codebook). Upon execution of multiply-accumulate operations, the replicated codebooks can be accessed in parallel to decode weights. The replication of weight codebook across PEs incurs a negligible memory overhead which is a reasonable cost for the throughput and performance gains obtained.

Equation 8 shows the overall weight memory saving for an arbitrary DNN layer, after wight encoding. Here, N denotes the number of weights, b_{fix} is the number of bits used to represent the fixed-point values, and K is the codebook size. As shown in the experiments (Section 5), K takes a value equal to or smaller than 64 while N is in the order of 10^6. As such, $(b_{fix} \times K \times PE) \ll (b_{fix} \times N)$ and the denominator in Equation 8 remains smaller than the numerator.

$$\frac{memory(W)}{memory(\vec{c}) + memory(W_{enc})} = \frac{b_{fix} \times N}{b_{fix} \times K \times PE + Log_2(K) \times N} \tag{8}$$

Encoder Module. The encoding module compares the distance of each computed feature y to all elements of the codebook (i.e., $|y - c[1]|, \ldots, |y - c[K]|$) and outputs the index of the closest element as the encoded value:

$$y_{enc} = \underset{i}{argmin}\ |y - c[i]|\ ,\ i \in \{1, \ldots, K\} \tag{9}$$

The encoded value is then sent through the output streaming buffer to be processed by the next layer. Most contemporary DNNs use `ReLU` activation. The encoder module inherently implements `ReLU` functionality when the first codebook value is set to 0. To implement other activation functions, we keep the encoding functionality of Equation 9 and merge the activation function into the next layer's input decoder. In other words, the input decoder of the next layer stores $Act(c[i])$ rather than $c[i]$ with $Act(\cdot)$ being the desired activation function. This modification is performed offline when the codebook values are loaded to FPGA memory.

4.1.1 *Processing Element (`PE`).*

Figure 14-bottom shows the hardware architecture of a `PE` inside the `MVAU`. Each `PE` is responsible for performing `MAC` operations on `SIMD` parallel input lanes. To this end, each `PE` is equipped with `SIMD`×`MULT` units implemented using DSP slices. Each `MULT` performs one fixed-point multiplication on b_{fix}-bit values. The multiplication results are then accumulated in a register (Accumulator in Figure 14) in fixed-point format. The decoded inputs required for performing `MAC` are registered in the `MVAU` and provided to all `PE`s. Alternatively, the decoded weights are generated within each `PE`. The low-bit (encoded) weights are stored in an `SRAM` block, which is partitioned to allow `SIMD` parallel read operations via the weight decoder. To perform decoding, each `PE` comprises a weight decoder (codebook) that is implemented using a register file. Once decoded, the weights are written into `SIMD` registers for parallel `MULT`. For each `PE`, the size of the local memory is:

$$Memory_{PE} \approx \underbrace{(K + 2\texttt{SIMD}) \times b_{fix}}_{register} + \overbrace{N \times log_2(K)}^{\texttt{SRAM}\ block} \tag{10}$$

where N is the number of encoded weights in the `SRAM` block of each `PE` and K is the weight codebook size.

At each point of the computation, a control logic keeps track of the matrix-multiplication indices and generates address signals to the encoded weights memory block accordingly. Whenever computations of one neuron are finished, i.e., when one vector-dot-product is completed, the control logic resets the accumulator and activates batch normalization on the computed output. Applying batch normalization to the output of the accumulator y is equivalent to computing $y \longleftarrow \gamma y + \beta$. Here, γ and β represent the scaling factor and bias, respectively, which are constants learned during DNN training. These values are extracted by the EncoDeep compiler from the trained encoded DNN and stored (in fixed-point format) on registers within each `PE`. Note that the memory requirement of these parameters is drastically lower than that of the weight matrices; thus, EncoDeep stores these parameters in the raw (non-quantized) format. After batch normalization, the output is ready to be encoded and sent to the next DNN layer through the streaming buffer.

Table 1. Benchmarked DNNs for evaluating EncoDeep effectiveness. *Conv* layers are represented as $\langle input{-}channels\rangle \xrightarrow[stride]{\langle kernel\ size\rangle} \langle output - channels\rangle$ and *FC* layers are denoted by $\langle output - elements\rangle$FC.

	Conv+BN+ReLU	**MaxPool**	**Conv+BN+ReLU**	**MaxPool**	**Conv+BN+ReLU**	**Conv+BN+ReLU**	**Conv+BN+ReLU**	**MaxPool**	**Classifier**
LeNet [31] (MNIST)	$1 \xrightarrow[\text{stride 1}]{5\times5} 16$	2×2 stride 2	$16 \xrightarrow[\text{stride 1}]{3\times3} 32$	2×2 stride 2	-	-	-	-	256FC 10FC, softmax
VGG7 [10] (CIFAR-10 & SVHN)	$[3 \xrightarrow[\text{stride 1}]{3\times3} 32]\times2$	2×2 stride 2	$[32 \xrightarrow[\text{stride 1}]{3\times3} 64]\times2$	2×2 stride 2	$[64 \xrightarrow{3\times3} 128]\times2$	-	-	-	256FC 256FC 10FC, softmax
AlexNet [30] (ImageNet)	$3 \xrightarrow[\text{stride 4}]{11\times11} 64$	2×2 stride 2	$64 \xrightarrow[\text{stride 2}]{5\times5} 192$	2×2 stride 2	$192 \xrightarrow[\text{stride 1}]{3\times3} 384$	$384 \xrightarrow[\text{stride 1}]{3\times3} 256$	$256 \xrightarrow[\text{stride 1}]{3\times3} 256$	2×2 stride 2	2048FC 4096FC 1000FC, softmax
ResNet-18 [15] (ImageNet)	$3 \xrightarrow[\text{stride 2}]{7\times7} 64$	3×3 stride 2	$[64 \xrightarrow[\text{stride 1}]{3\times3} 64]\times4$	-	$[64 \xrightarrow{3\times3} 128]\times4$	$[128 \xrightarrow{3\times3} 256]\times4$	$[256 \xrightarrow{3\times3} 512]\times4$	7×7 average pool	1000FC, softmax

4.2 Sliding Window Unit (SWU)

The convolutional layers of a DNN compute the dot product between a window of the layer input and the `CONV` weight kernel. The window slides over the input image to produce individual elements of the output feature-map. The SWU in EncoDeep hardware simulates the sliding window operation by reordering the values in the layer input image buffer. The input image values are then grouped in chunks of SIMD words to be sent to the `MVAU` sequentially for processing.

4.3 Max-pooling Unit (MPU)

EncoDeep software stack outputs a sorted list of codebook values for the output encoding: higher values are mapped to larger encodings. This sorting is particularly useful since comparison over encoded values becomes equivalent to comparison over the original fixed-point values; therefore, EncoDeep performs the max-pooling operation on low-bitwidth encoded values rather than the full-precision cluster centers. This approach provides two benefits: (i) the memory overhead of the buffers in the MPU is considerably reduced. (ii) The logic cost of comparison between low-bitwidth encoded values is significantly smaller than the full-precision counterpart.

5 EXPERIMENTS

To evaluate EncoDeep effectiveness, we perform proof-of-concept experiments on four different classification benchmarks, namely, MNIST, CIFAR-10, SVHN, and ImageNet. Table 1 summarizes the DNN architectures used in our evaluations. EncoDeep software stack is implemented in Pytorch and the hardware stack is realized in Vivado_HLS design suite. All hardware resource utilizations are gathered after performing place-and-route via Vivado Design Suite 2017.2. Throughput values are reported from Vivado_HLS 2017.2.

5.1 EncoDeep Automated Bitwidth Selection

We showcase our bitwidth selection algorithm using the *VGG7* architecture trained on CIFAR-10 dataset. Our customization algorithm provides a set of configurations, each of which renders a certain memory footprint and accuracy. The first step of EncoDeep bitwidth customization is to encode the activations while the weights are kept at full-precision. Initially, the activations are encoded with 4 and 6 bits in `CONV` and `FC` layers, respectively. We then utilize our customization algorithm in Section 3.3 to extract the activation bitwidths. As the algorithm proceeds, both total memory footprint and DNN accuracy are decreased. The obtained accuracy/memory trade-off is shown in Figure 15-a.

We use a small portion of the training data[1], dubbed the validation set, to compute the accuracy during the iterative bitwidth customization algorithm. We empirically observed that the accuracy measured on the validation set is correlated with the accuracy measured on the entire test set. Therefore, the validation accuracy can be leveraged as a suitable, low-cost, proxy for test accuracy during bitwidth configuration. The validation set is small enough to be cached

[1] 1000 samples for all benchmarks.

into the GPU memory to ensure fast evaluations. Note that we do not retrain the model in between iterations to ensure fast customization. To illustrate the effect of retraining, we also plot the accuracy of each extracted bitwidth configuration after 1 and 10 epochs of fine-tuning in Figure 15-a. It can be seen that the accuracy is retrieved for most of the configurations even after 1 epoch, which is a fairly short post processing time compared to the original (floating-point) training which takes $\sim$ 200 epochs.

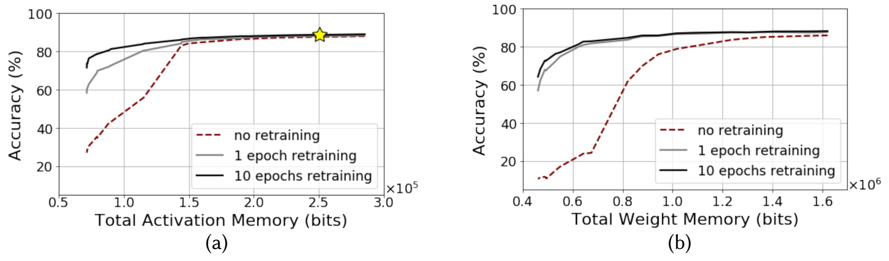

Fig. 15. Memory and accuracy trade-off for (a) activations and (b) weights of VGG-7 on CIFAR-10 dataset.

In the next step of our customization, one of the configurations for activation bitwidths (the ★ mark on Figure 15-a) is selected and fine-tuned to recover accuracy. We then proceed to the weight encoding step with initial 6-bit encoding for all layers. During this customization stage, activation bitwidths remain unchanged and only the weight bitwidths are configured. Similar to activation encoding, we obtain the accuracy/memory curves in Figure 15-b. In what follows, we evaluate EncoDeep automated bitwidth customization from two perspectives: (i) quality of the end result, i.e., the obtained bitwidth configuration. (ii) Search efficiency/performance of our heuristic algorithm.

5.1.1 ***Evaluation of EncoDeep Bitwidth Configurations.*** We apply bitwidth customization to weights and activations of various benchmarked DNNs and select several bitwidth configurations. Table 2 compares the total memory (activation+weights) and test accuracy between the original full-precision models and selected encoded DNNs. EncoDeep achieves 14.56× memory reduction with 0.026% accuracy loss for MNIST, 7.34× memory reduction with 0.37% accuracy loss for SVHN, and 6.85× memory reduction with 0.91% accuracy loss for CIFAR-10. On ImageNet, EncoDeep reduces the model size by 7.9× and 6.6× with 0.43% and 0.8% drop in top-1 accuracy for AlexNet and ResNet18, respectively.

Table 2. Comparison of full-precision networks with EncoDeep models with flexible bitwidths across layers.

		Full-Precision (FP32)	**EncoDeep Configurations**		
MNIST	Memory ($\times 10^5$)	50.67	3.48	2.17	1.89
	Test Accuracy (%)	99.28	99.02	98.69	98.31
SVHN	Memory ($\times 10^6$)	12.34	1.67	1.21	0.65
	Test Accuracy (%)	97.67	97.30	97.15	95.07
CIFAR-10	Memory ($\times 10^6$)	12.34	1.80	1.20	1.02
	Test Accuracy (%)	89.05	88.14	87.01	85.06

			Full-Precision (FP32)	**Quantized (INT8)**	**EncoDeep Configurations**	
ImageNet	AlexNet	Memory ($\times 10^8$)	11.14	2.78	1.41	0.53
		Test Accuracy (%)	56.3	55.18	55.87	53.21
	ResNet18	Memory ($\times 10^7$)	44.76	11.19	6.78	4.45
		Test Accuracy (%)	69.70	68.97	68.90	65.40

To investigate whether the solution found by the heuristic method is the absolute best, one needs to perform brute-force evaluation of all bitwidth configurations. Nevertheless, brute-force evaluation is only viable for small networks as the bitwidth search-space grows exponentially in the number of network layers. In particular, for encoding an L-layer network with weight and activation bits in the range of $[1, B_w]$ and $[1, B_a]$, respectively, a total number of $B_w^L \times B_a^L$ evaluations is required. Here, we evaluate the effectiveness of our heuristic on a small-scale bitwidth optimization problem. We perform brute-force search on the activation bitwidths of the VGG7 network for CIFAR-10 dataset and summarize the results in Figure 16. For this experiment $L = 8$ and $B_a = 4$, resulting in a total of 2^{16} evaluations (shown with blue points) which takes $\sim$ 18 hours on an NVIDIA TITAN Xp GPU. As can be seen, the obtained activation bitwidths (shown with red points), lie on the memory-accuracy Pareto front, indicating that the heuristic successfully eliminates the non-optimal solutions and finds near-optimal configurations.

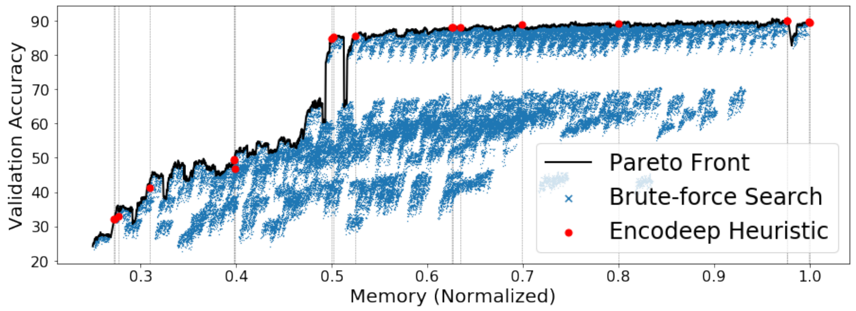

Fig. 16. Memory-accuracy Pareto curve obtained by brute-force evaluation of bitwidth configurations for VGG7 on CIFAR-10 benchmark. EncoDeep generates near-optimal bitwidths by finding configurations that lie close to the non-dominated Pareto front.

Due to the excessive runtime of brute-force search, especially for more complex benchmarks, we provide comparisons with prior art to evaluate the quality of our obtained bitwidths. Table 3 summarizes the comparison of EncoDeep configurations with prior methods for training low-bit DNNs in terms of memory, accuracy, and fine-tuning time. Each of our reported architectures and their corresponding bitwidths in Table 3 are chosen specifically to match the accuracy and/or memory footprint of the prior art. We visualize the benchmarked per-layer bitwidths for each evaluated EncoDeep DNN in Figure 17. The memory efficiency of EncoDeep can be attributed to the following main reasons:

(1) Looking at the bitwidth configurations of Figure 17, EncoDeep automatically chooses to have lower bitwidths for weights of fully-connected layers and activations of early convolutional layers. Doing so helps in minimizing the overall memory footprint since such layers have more contribution to the DNN memory.
(2) To compensate for the drop in the inference accuracy, prior work in low-bit DNN inference increases the number of neurons/channels per DNN layer [41]. In contrast, we show that by adjusting the arithmetic encoding bitwidth, one can achieve comparable accuracy with even fewer neurons per layer. For our AlexNet benchmark, for instance, we use 2× less neurons in the output of the first fully-connected layer compared to the original architecture (see Table 1). EncoDeep reduces the memory of AlexNet benchmark using this method, but preserves the accuracy by using non-linear encoding with flexible bitwidths.

It is worth mentioning that, unlike existing low-bit DNNs that train the whole network from scratch, EncoDeep extracts several near-optimal bitwidth configurations for a pre-trained DNN in one-shot execution. The benefits of this approach are three-fold: (i) EncoDeep eliminates the drastic cost of training from scratch per bitwidth configuration. Using our fine-tuning method explained in Section 3.2, model accuracy is retrieved after a few epochs, e.g., as low as 0.25 epochs for ImageNet. (ii) EncoDeep accuracy/memory to be tuned by picking different bitwidth configurations across layers. (iii) EncoDeep customization can be readily applied to publicly available pre-trained models.

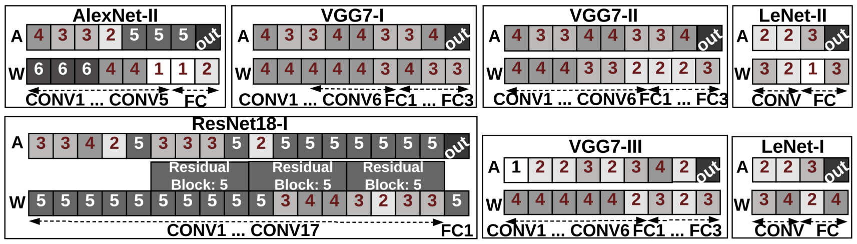

Fig. 17. Per-layer encoding bits for evaluated DNNs.

Table 3. Comparison of EncoDeep with state-of-the-art low-bit DNNs. Our per-layer bitwidths are shown in Figure 17. We normalize the memory footprint of previous works to that of EncoDeep: lower memory and higher accuracy are desirable.

	Baselines	Architecture	Test Accuracy (%)	Memory	Epochs	Bitwidth	
						Weight	Act
MNIST	QNN [21]	MLP	99.04	193×	1000	1	1
	ReBNet3 [10]		98.25	1.76×	200	1	2
	EncoDeep	LeNet-I	99.02	1.84×	$10 \times 2^*$	flexible	
	EncoDeep	LeNet-II	98.31	1	10×2	flexible	
CIFAR-10	QNN [21]†	VGG7	89.85	5.4×	500	1	1
	ReBNet3 [10]‡		86.98	1.65×	200	1	3
	EncoDeep	VGG7-I	88.14	1.32×	10×2	flexible	
	EncoDeep	VGG7-II	87.01	1	10×2	flexible	
SVHN	ReBNet3 [10]‡	VGG7	97.00	1.62×	50	1	3
	QNN [21]‡		97.2	2.15×	200	1	1
	EncoDeep	VGG7-III	97.15	1	10×2	flexible	
ImageNet	ReLeQ [5]	AlexNet	56.82	2.01×	-	flexible	32
	EncoDeep**	AlexNet-I	55.87	1	0.25×2	flexible	
	HWGQ [2]	AlexNet	52.70	1.17×	68	1	2
	HBNN [9]		52.00	2.32×	-	flexible	
	PQTS [61]		51.60	2.32×	-	2	2
	QNN [21]		51.03	1.17×	-	1	2
	DoReFaNet [60]		49.80	1.17×	45	1	2
	WRPN [41]‡		48.30	4.61×	-	1	1
	XNORNet [42]		44.20	1.15×	18	1	1
	ReBNet3 [10]		41.43	1.17×	100	1	3
	EncoDeep**	AlexNet-II	53.21	1	0.25×2	flexible	
	ABC-Net [36]	ResNet18	65.00	1.57×	-	5	5
	ABC-Net [36]		62.50	1.05×	-	3	5
	EncoDeep	ResNet18-I	65.40	1	0.25×2	flexible	

* Fine-tuning for 10 epochs post-activation and 10 epochs post-weight encoding.
† This baseline has 4× more neurons per layer than ours.
‡ This baseline has 2× more neurons per layer than ours.
** Our architecture has 2× less neurons in the output of the first fully-connected layer (see Table 1)

5.1.2 ***Evaluation of EncoDeep Search Algorithm.*** While various hyperparameter optimization methods, e.g., RL or genetic algorithms, can potentially deliver similar end results, what distinguishes these approaches is the number of evaluations required to obtain the final result. This, in turn, directly affects the algorithm runtime. In this Section, we compare EncoDeep search with existing methods in discrete combinatorial optimization in terms of the quality of end results and the search efficiency (runtime).

Comparison with Q-learning. Recall from Section 3.3 that one of the main differences between pure RL-based methods and our search algorithm lies in pursuing reward-based immediate returns in EncoDeep rather than the traditional long-term returns. Our goal in this part of the evaluation is to show how the immediate-reward optimization of EncoDeep compares to pursuing long-term rewards in pure RL. Optimizing a long-term reward can potentially lead to better end results in RL tasks. Due to the nature of our problem, however, an immediate reward is beneficial as the value of each state transition can be independently evaluated without relying on the end state. More specifically, the immediate reward can be leveraged to assess the optimality of each intermediate state without the need for traversing all states to the end. For a concrete comparison, we have implemented Q-learning [56] as an RL baseline with long-term rewards for comparison. We set the Q-learning reward for state s as follows:

$$r(s) = \begin{cases} \frac{mem(s)-mem(s_0)}{acc(s)-acc(s_0)} & if \;\; |mem(s) - \theta| < tolerance \\ 0 & otherwise \end{cases} \tag{11}$$

where θ is the target memory and s_0 is the initial state with all bitwidths set to B. An episode is finished when a state's memory drops below θ. We apply Q-learning on the activations of the VGG-7 model trained on CIFAR-10. In this experiment, we set θ to 0.5 to achieve $\sim$ 50% memory reduction compared to the initial DNN. We set the tolerance to 0.05 so that any bitwidth configurations resulting in a memory $\in [45\% - 55\%]$ is given a reward during Q-learning. Figure 18 compares EncoDeep with Q-learning. The horizontal axis shows the number of model evaluations, i.e., number of DNN inference accuracy computations. The vertical axis shows the maximum accuracy seen so far for configurations with a memory in the range [0.45-0.55]. The ★ represents the configuration with comparable memory, found by EncoDeep.

Compared to Q-Learning, EncoDeep achieves slightly higher accuracy and lower memory while requiring fewer number of DNN evaluations. This is due to the fact that EncoDeep is policy-free and only consists of one greedy state-transition episode by relying on immediate rewards. Note that for each target memory and accuracy, policy training, as in Q-learning, must be repeated from scratch with a different threshold θ (see Equation 11) to extract the optimal bitwidth configuration. In contrast, EncoDeep extracts the entire memory-accuracy Pareto curve with only one state traversal (episode). This provides adaptability by allowing users to pick their desired bitwidth configuration based on various accuracy-memory constraints, without need for re-running the entire algorithm.

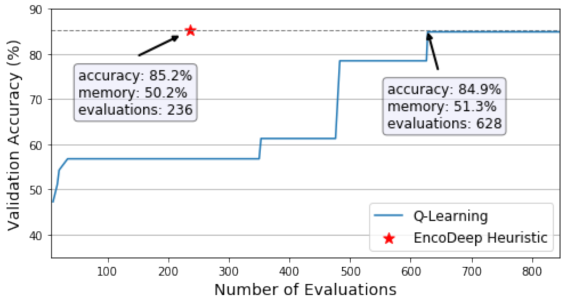

Fig. 18. Comparison of the acquired accuracy and memory as well as the number of evaluations between a Q-learning approach and EncoDeep (★), upon convergence.

Comparison with Genetic Algorithms. Another important baseline for combinatorial optimization is genetic algorithms (GA). Carefully designed GA is shown to deliver similar end results to RL-based methods in various tasks [44]. For comparison, we use the optimization framework in [25], which is a generic tool for compressing DNNs with GA, as a new baseline. Figure 19 shows the reward[2] of the genetic population across GA iterations with an accuracy threshold of 85% and a genetic population size of 50. As can be seen, GA gradually and iteratively evolves the configurations to increase the average reward in the population, i.e., lower memory and higher accuracy. The red point on Figure 19 corresponds to the best solution found by GA, which takes 1234 evaluations to find a model with 50% of the original

[2]Please refer to [25] for details about the utilized reward function for the GA.

memory and 85.6% inference accuracy. For a similar target memory and accuracy, EncoDeep achieves 50.2% memory and 85.2% accuracy via only 236 evaluations. We attribute this improvement in number of evaluations to the single-episode greedy execution of EncoDeep, which is specifically designed to find the optimal configuration with very few iterations.

In summary, EncoDeep has the following benefits compared to genetic algorithms:

(1) The success of GA relies heavily on careful design of the underlying score function used in the evaluation step. Besides, multiple design choices and hyperparameters, e.g., mutation and crossover rates, affect the optimization performance. Hand-tuning such parameters remains a standing challenge that further hinders the GA design process. EncoDeep does not include extra design hyperparameters and allows for easy automation.

(2) To obtain a trade-off between accuracy and memory, one needs to run GA multiple times with different target accuracies in the reward function. Therefore, extracting the memory-accuracy tradeoff using GA incurs a high timing overhead. In contrast, EncoDeep extracts all points lying close to the Pareto front in a single run.

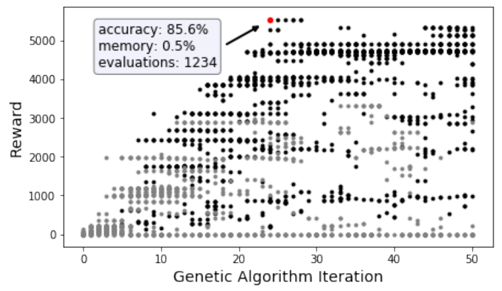

Fig. 19. Genetic evolution for bitwidth customization of VGG-7 on CIFAR-10. The reward on the vertical axis measures bitwidth optimality by combining accuracy and memory (see [25] for details). Each point on the plot represents an individual in GA. points with black color have an accuracy higher than the accuracy threshold $\theta = 85\%$.

5.1.3 ***Overhead of Customization and Re-training.*** We summarize the total runtime of EncoDeep bitwidth configuration and the break-down of different steps for all our benchmarks in Figure 20. Runtime values are gathered using a machine with a single NVIDIA Titan Xp GPU and an Intel Xeon-E5 CPU. As seen, the overhead of (offline) clustering at the pre-processing stage (EN-A and EN-W) is negligible compared to other steps. The bulk of runtime is due to running Algorithm 2 on activations (CU-A) and weights (CU-W), and retraining the customized models (RT-U and RT-W). Nevertheless, EncoDeep takes only ~ 254 minutes to customize our most complex benchmark (ResNet 18), whereas training the original ResNet-18 model on the same machine takes over a day (~ 1800 minutes). EncoDeep customization incurs only 14% of the training time even for this many-layer network. Note that the evaluations in EncoDeep customization steps (CU-A and CU-W) can be distributed across multiple GPUs to further decrease runtime.

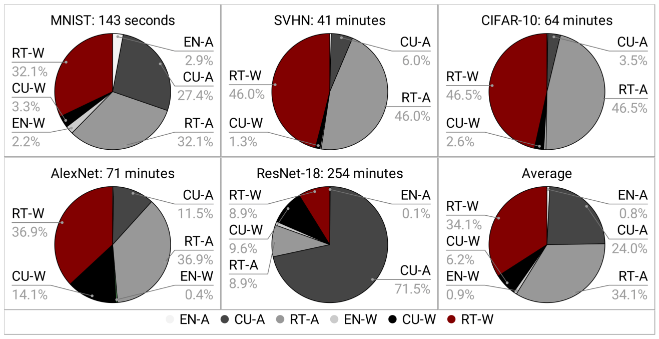

Fig. 20. Total time and break-down of EncoDeep optimization runtime for the evaluated benchmarks. Here, EN-A and EN-W represent (offline) activation and weight encoding. CU-A and CU-W denote bitwidth customization for activations and weights. RT-A and RT-W correspond to the re-training time after activation and weight encoding.

It is worth mentioning that EncoDeep customization step is much faster than pure reinforcement learning-based approaches. ReLeQ [5] is an example RL-based method that trains an LSTM model using gradient computation while our method does not include any RL model training and is thus much faster in terms of runtime (e.g., ReLeQ has 600 episodes[3] for LeNet while our method requires only a single episode with 22 iterations to achieve similar results).

[3]unknown number of evaluations per episode

5.2 EncoDeep Hardware Implementation

In this section, we evaluate EncoDeep hardware accelerator. We implement one architecture per dataset from Figure 17, namely, LeNet-I for MNIST, VGG7-I for CIFAR10, VGG7-III for SVHN, and AlexNet for ImageNet. Table 4 summarizes the evaluation platforms for each DNN architecture.

Table 4. Platform details in terms of block-RAM (BRAM), DSP, flip-flop (FF), and look-up table (LUT) resources.

Application	Platform	BRAM	DSP	FF	LUT
ImageNet	Virtex VCU108	3456	768	1075200	537600
CIFAR-10 & SVHN	Zynq ZC702	280	220	106400	53200
MNIST	Spartan XC7S50	120	150	65200	32600

Importance of Activation Encoding. We start the analysis by studying the advantages of activation encoding, from the hardware perspective, versus solely encoding the weights as proposed in [13]. Note that [13] also applies pruning and Huffman encoding which are the main contributors to the compression rate. Since these methods are orthogonal to our approach, we do not utilize them in EncoDeep to focus on the analysis of encoding itself. We compare two versions of encoded DNNs: one with encoded weights and fixed-point activations as proposed in [13], and another with both weights and activations encoded as suggested in EncoDeep. For each dataset, we separately optimize the per-layer parallelism factors `SIMD` and `PE` for both encoded and fixed-point DNNs to obtain maximum possible throughput. Table 5 summarizes resource utilization and throughput for each of the designs.

Overall, the realization of EncoDeep methodology achieves higher throughput while requiring a lower number of resources compared to a weight-only encoding approach [13]. The benefits become more prominent for architectures with higher complexity since the memory implication of activations is higher in complex networks. As seen for MNIST, CIFAR-10, and SVHN benchmarks, EncoDeep activation encoding improves the throughput by 1.1×, 6.2×, and 6.66×, respectively. Advantages of activation encoding are most significant for AlexNet: model memory with fixed-point activations is so large that it cannot fit in the FPGA block-RAM capacity, rendering the design infeasible within platform constraints. We compare EncoDeep with existing fixed-point accelerators that use off-chip memory in the following.

Table 5. Summary of hardware resource utilization and performance. *EncoDeep* presents our model with encoded weights and activations whereas DNN_{fix} denotes a network with encoded weights and (8-bit) Fixed-point activations.

		Resource Utilization				Latency
		BRAM	DSP*	FF	LUT	(ms)
MNIST	*EncoDeep*	33	53	15223	9992	0.39
	DNN_{fix}	93	53	25884	12048	0.43
	Ratio	2.82×	1×	1.7×	1.2×	**1.1**×
CIFAR-10	*EncoDeep*	197	111	53953	31632	3.58
	DNN_{fix}	181	35	68255	28433	22.21
	Ratio	0.92×	0.32×	1.26×	0.9×	**6.2**×
SVHN	*EncoDeep*	146	111	42748	28393	3.39
	DNN_{fix}	143	35	67944	27934	22.59
	Ratio	0.98×	0.32×	1.59×	0.98×	**6.66**×
ImageNet	*EncoDeep*	3336	308	159663	82791	25.05
	DNN_{fix}	Exceeds Platform Constraints				

*25 × 18 DSB array.

Comparison with Fixed-point Accelerators. We perform a comparison between EncoDeep and prior work in Table 6. Specifically, we consider AlexNet with customized encoding as in Figure 17, which corresponds to hardware results of Table 5. The reported results include performance, either in terms of throughput (frames per second) or latency. Since the existing frameworks utilize various FPGA platforms, it is crucial to take into account the instantiated computational capacity[4] and power consumption. Therefore, we compare the frameworks by means of performance-per-resource and performance-per-Watt. EncoDeep achieves higher normalized performance compared to the prior art. This is a direct result of using on-chip memory instead of the off-chip DRAM for feature transfer among DNN layers. The streaming buffers of our design allow EncoDeep to better utilize the arithmetic units by overlapping the execution of DNN layers, achieving a higher performance-per-resource. EncoDeep power advantage over existing accelerators is also rooted in the elimination of power-hungry DRAM access.

Table 6. Comparison of Alexnet implementation between EncoDeep and existing fixed-point (FXD) and floating-point (FLT) DNN accelerators. To account for platform variations, we compare the throughput (images-per-second) and $\frac{1}{\text{Latency}}$ metrics normalized by computation capacity (CAP). We also compare performance-per-Watt to reflect power efficiency.

	Criterion	[51]	[52]	[59]	[37]	[48]	[1]	EncoDeep
	Precision	FLT	FXD	FXD	FXD	FXD	FXD	Flexible
	Acc(%)	-	55.41	52.4	56.5	-	54.27	53.2
	FPGA	690T*	GSD8†	690T*	690T*	AX115‡	ZU9§	VCU108*
	Freq(MHz)	100	120	150	100	200	300	152
	DSP**	3177	1504	14400	2872	2688	442	308
Img/sec	/CAP	0.55×	1	0.88×	2.33×	1.42×	0.49×	**3.03×**
	/Watt	3.14×	1	1.82×	**5.00×**	1.36×	-	4.54×
$\frac{1}{\text{Latency}}$	/CAP	-	1	0.05×	0.15×	-	-	**3.03×**
	/Watt	-	1	0.10×	0.32×	-	-	**4.54×**

*Virtex †Stratix-V ‡Arria10 §Zynq

**DSP array size is 25 × 18 for Xilinx and 18 × 18 for Altera/Intel FPGAs.

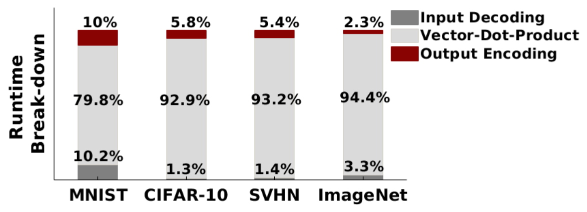

Fig. 21. Runtime breakdown of EncoDeep accelerator.

Execution Overhead of Encoding/Decoding. We study the runtime implication of online activation encoding by measuring the number of clock cycles required for different stages of EncoDeep `MVAU` engine. Figure 21 demonstrates the runtime break-down for each of the evaluated architectures. For a conventional non-encoded network, the `MVAU` would only perform Vector-Dot-Product (`VDP`) operations. As can be seen, for EncoDeep encoded models, the majority of clock cycles in MVAU execution still belong to `VDP` computation while the encoding/decoding overhead is small.

[4]the computational capacity is defined as $CAP = DSP \times Arr$, where Arr is the array size per DSP, e.g., 18 × 25 for Xilinx Virtex platforms.

6 RELATED WORK

To enable ubiquitous deployment of DNNs, several recent research efforts have focused on DNN acceleration [6, 7, 10–12, 18, 19, 22, 23, 45, 46, 49]. In a parallel track, designing efficient DNN graphs and architectural optimization has gained attraction from the community [24–26, 43, 47]. EncoDeep bridges the gap between these two research tracks by incorporating algorithm-hardware co-design. As neural networks are memory-intensive, devising methods to decrease the memory footprint can significantly enhance accelerator performance in terms of throughput and power consumption. Perhaps the most popular method for neural network compression is network pruning, where network parameters with insignificant contributions to the model's accuracy are removed. Many researchers have developed valuable work in DNN compression with pruning using either structured [16, 17, 27, 33–35, 39, 55] or non-structured pruning [14]. Hardware accelerators have also been proposed to perform fast and efficient inference using pruned neural networks, e.g., [3, 38]. The focus of our paper, however, is another attractive solution for neural network compression: inference with few-bit encoded values per weight/neuron. In practice, one might employ both pruning and encoding to achieve better compression results [13]. However, to better discuss the contributions of this paper, in our experiments we focus solely on the encoding technique.

Several methods for training DNNs with few bits have been proposed in [2, 9, 21, 36, 41, 42, 54, 60]. QNN [21] was perhaps the first work to suggest extreme neural network quantization with binary ($\in \{\pm 1\}$) parameters and activations. Following their work, many researchers have proposed low-bit DNNs with improved accuracy. Authors of XNOR-Net [42] suggest computing the average absolute value of each input vector to the convolution operation in the forward pass. They show that multiplying this average value by the corresponding XNOR-Popcount result improves the inference accuracy. WRPN [41] shows that scaling layer widths uniformly can deliver more accurate low-bit DNNs. One immediate shortcoming of this approach is the quadratic increase in the memory and computational complexity as the scaling factor grows. ABC-Nets [36] proposes multi-bit binary approximation of weights and activations of DNNs. Their method shows great accuracy improvement at the expense of training every bitwidth configuration from scratch. HBNN [9] proposes to incorporate multi-level binarization while enabling heterogeneous level selection across layers. Although this approach achieves high memory efficiency, it still has limited accuracy due to binary approximation.

Instead of using strict binary values, as proposed by the above works, our proposal uses low-bit non-linear encodings which allows high-precision arithmetics with a low memory footprint. By leveraging non-linear encoding, we can scale down layer widths (as opposed to WRPN [41]) to match the memory of a wide binary neural network while still enjoying higher accuracy. Compared to ABC-Nets' multiple rounds of training from scratch, our experimental results show that the one-time post-training approach of EncoDeep can extract better (heterogeneous) bitwidth configurations with higher accuracy. Additionally, while the above works mainly focus on developing theoretical memory improvements, EncoDeep adopts hardware-algorithm co-design to show practical performance boost and memory reduction on hardware.

To create more hardware-friendly binary DNNs, ReBNet [10] proposes co-designing a DNN with residual binary approximation and an FPGA accelerator. LUTNet [54] takes a step forward and directly incorporates hardware characteristics such as the LUT structure of FPGAs into the designed activation function. These methodologies improve the hardware efficiency of BNNs, yet their accuracy is bounded. Similar to the above work, EncoDeep incorporates DNN-hardware co-design. However, unlike the above works, EncoDeep specifically configures the bitwidths across DNN layers by finding the optimal accuracy-memory Pareto front. This customization allows EncoDeep to achieve higher accuracy by using flexible (heterogeneous) bitwidths across DNN layers. Such heterogeneity is also specifically supported by EncoDeep modular hardware design and the accompanying compiler.

DoReFa-Net [60] incorporates specific training procedures to enable fixed-point quantization for both weights and activations. HWGQ [2] mentions that low-bit approximation of activations is more difficult than weights. The authors also note that fixed-point quantization with uniformly spaced quantization bins does not deliver the minimum approximation error. Thus, during the training phase, the authors propose to approximate the distribution of activation units via a half-way Gaussian prior and find non-uniform quantization bins accordingly. In our work, we show that it is possible to create a non-uniform quantizer *after* the DNN is trained, which has several direct benefits: first, EncoDeep does not impose extra overhead on the original DNN training and therefore training convergence speed is not altered. Second, EncoDeep does not need to assume a Gaussian prior on the activation distribution and can be applied to arbitrary distributions. Third, EncoDeep can efficiently tune the number of encoding bits across layers without training every configuration from scratch.

Nonlinear encoding allows for fixed-point arithmetics accompanied by a low storage requirement. Perhaps the closest method to this paper is a stand-alone weight encoding, with no activation encoding, originally proposed in [4, 13, 45]. Weight encoding significantly reduces the memory footprint of model parameters but the activation units (especially in convolution layers) still require a large capacity of memory. To address this challenge, we extend the encoding to the activations of neural networks and introduce training routines for the corresponding encoded activations. In addition, prior work utilizes hand-crafted or rule-based heuristics to determine the encoding bitwidth. Such manual methods are generally sub-optimal and incur a drastic engineering cost. To address this issue, we propose an automated cross-layer bitwidth selection algorithm that aims to capture the accuracy/memory trade-off.

In a concurrent track, designing automated and easy-to-use tools for FPGA implementation of DNNs has been the focus of contemporary research [1, 37, 48, 51, 52, 59]. These works aim to maximize the throughput of fixed/floating-point DNN inference by distributing FPGA resources among parallel computing engines. Although accurate, fixed-point DNNs are generally memory intensive, where excessive access to off-chip memory becomes a design bottleneck. To alleviate this problem, authors of [10, 53] propose to perform inference solely using the on-chip memory and utilizing streaming buffers to realize inter-layer data transfers. These frameworks facilitate the design process of DNNs by providing configurable template functions in high-level synthesis language. However, [10, 53] are only compatible with binary DNNs and do not support fixed-point arithmetics. By incorporating activation encoding into DNN computational flow, EncoDeep hardware simultaneously enjoys the benefits of on-chip streaming buffers and high accuracy arithmetics. EncoDeep hardware stack supports flexible bitwidths, allowing the implementation of customized encoded DNNs.

7 CONCLUSION

This paper proposes a novel nonlinear quantization scheme to reduce the memory footprint of intermediate activations in convolutional neural networks' computation flow. The encoding compresses the activations and allows on-chip execution of the underlying FPGA accelerator without communicating the computed features with the off-chip DRAM. To ensure non-recurring engineering costs, an automated algorithm is proposed to configure the encoding bitwidth across layers of an arbitrary neural network. EncoDeep open-source API enables developers to convert high-level Pytorch description of a neural network into hardware modules without getting involved with the details of the design. We hope the provided API can advance research on reconfigurable DNN inference.

8 ACKNOWLEDGMENT

This work was supported in part by IARPA (2018-18022100004), ARO (W911NF1910317), and ONR (N00014-17-1-2500).

REFERENCES

[1] 2018. CHaiDNN: HLS based Deep Neural Network Accelerator Library for Xilinx Ultrascale+ MPSoCs.

[2] Z. Cai, X. He, J. Sun, and N. Vasconcelos. 2017. Deep Learning with Low Precision by Half-wave Gaussian Quantization. *arXiv:1702.00953* (2017).

[3] Shijie Cao, Chen Zhang, Zhuliang Yao, Wencong Xiao, Lanshun Nie, Dechen Zhan, Yunxin Liu, Ming Wu, and Lintao Zhang. 2019. Efficient and effective sparse LSTM on FPGA with Bank-Balanced Sparsity. In *FPGA*. ACM, 63–72.

[4] W. Chen, J.T. Wilson, S. Tyree, K.Q. Weinberger, and Y. Chen. 2015. Compressing neural networks with the hashing trick. *CoRR, abs/1504.04788* (2015).

[5] Ahmed T Elthakeb, Prannoy Pilligundla, Amir Yazdanbakhsh, Sean Kinzer, and Hadi Esmaeilzadeh. 2018. Releq: A reinforcement learning approach for deep quantization of neural networks. *arXiv preprint arXiv:1811.01704* (2018).

[6] Amir Erfan Eshratifar, Amirhossein Esmaili, and Massoud Pedram. 2019. Bottlenet: A deep learning architecture for intelligent mobile cloud computing services. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 1–6.

[7] Amir Erfan Eshratifar, Amirhossein Esmaili, and Massoud Pedram. 2019. Towards collaborative intelligence friendly architectures for deep learning. In *20th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 14–19.

[8] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 1–14.

[9] Joshua Fromm, Shwetak Patel, and Matthai Philipose. 2018. Heterogeneous bitwidth binarization in convolutional neural networks. In *Advances in Neural Information Processing Systems*. 4006–4015.

[10] M. Ghasemzadeh, M. Samragh, and F. Koushanfar. 2018. ReBNet: Residual Binarized Neural Network. In *FCCM*. IEEE, 57–64.

[11] Soroush Ghodrati, Hardik Sharma, Sean Kinzer, Amir Yazdanbakhsh, Kambiz Samadi, Nam Sung Kim, Doug Burger, and Hadi Esmaeilzadeh. 2019. Mixed-Signal Charge-Domain Acceleration of Deep Neural networks through Interleaved Bit-Partitioned Arithmetic. *arXiv preprint arXiv:1906.11915* (2019).

[12] Soroush Ghodrati, Hardik Sharma, Cliff Young, Nam Sung Kim, and Hadi Esmaeilzadeh. 2020. Bit-Parallel Vector Composability for Neural Acceleration. *arXiv preprint arXiv:2004.05333* (2020).

[13] S. Han, H. Mao, and W.J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[14] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*. 1135–1143.

[15] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep residual learning for image recognition. In *CVPR*. 770–778.

[16] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. 2018. Soft filter pruning for accelerating deep convolutional neural networks. *arXiv preprint arXiv:1808.06866* (2018).

[17] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *International Conference on Computer Vision (ICCV)*, Vol. 2.

[18] Ali HeydariGorji, Mahdi Torabzadehkashi, Siavash Rezaei, Hossein Bobarshad, Vladimir Alves, and Pai H Chou. 2020. STANNIS: Low-Power Acceleration of Deep NeuralNetwork Training Using Computational Storage. *arXiv preprint arXiv:2002.07215* (2020).

[19] Morteza Hosseini, Mark Horton, Hiren Paneliya, Uttej Kallakuri, Houman Homayoun, and Tinoosh Mohsenin. 2019. On the complexity reduction of dense layers from O (N 2) to O (NlogN) with cyclic sparsely connected layers. In *DAC*. IEEE, 1–6.

[20] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[21] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research* 18, 1 (2017), 6869–6898.

[22] Shehzeen Hussain, Mojan Javaheripi, Paarth Neekhara, Ryan Kastner, and Farinaz Koushanfar. 2019. FastWave: Accelerating Autoregressive Convolutional Neural Networks on FPGA. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[23] Mohsen Imani, Mohammad Samragh, Yeseong Kim, Saransh Gupta, Farinaz Koushanfar, and Tajana Rosing. 2018. Rapidnn: In-memory deep neural network acceleration framework. *arXiv preprint arXiv:1806.05794* (2018).

[24] Mojan Javaheripi, Bita Darvish Rouhani, and Farinaz Koushanfar. 2019. SWNet: Small-World Neural Networks and Rapid Convergence. *arXiv preprint arXiv:1904.04862* (2019).

[25] Mojan Javaheripi, Mohammad Samragh, Tara Javidi, and Farinaz Koushanfar. 2020. GeneCAI: Genetic Evolution for Acquiring Compact AI. *arXiv preprint arXiv:2004.04249* (2020).

[26] Mojan Javaheripi, Mohammad Samragh, and Farinaz Koushanfar. 2019. Peeking Into the Black Box: A Tutorial on Automated Design Optimization and Parameter Search. *IEEE Solid-State Circuits Magazine* 11, 4 (2019), 23–28.

[27] Chunhui Jiang, Guiying Li, Chao Qian, and Ke Tang. 2018. Efficient DNN Neuron Pruning by Minimizing Layer-wise Nonlinear Reconstruction Error. In *IJCAI*. 2–2.

[28] J. Kiefer, J. Wolfowitz, et al. 1952. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics* 23, 3 (1952), 462–466.

[29] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. 2015. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530* (2015).

[30] A. Krizhevsky, I. Sutskever, and G.E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS*. 1097–1105.

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

[32] D. Li, X. Wang, and D. Kong. 2017. DeepRebirth: Accelerating Deep Neural Network Execution on Mobile Devices. *arXiv preprint:1708.04728* (2017).

[33] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient convnets. *arXiv:1608.08710* (2016).

[34] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. 2017. Runtime neural pruning. In *Advances in Neural Information Processing Systems*. 2181–2191.

[35] Shaohui Lin, Rongrong Ji, Yuchao Li, Yongjian Wu, Feiyue Huang, and Baochang Zhang. 2018. Accelerating Convolutional Networks via Global & Dynamic Filter Pruning.. In *IJCAI*. 2425–2432.

[36] X. Lin, C. Zhao, and W. Pan. 2017. Towards accurate binary convolutional neural network. In *NIPS*. 345–353.

[37] Z. Liu, Y. Dou, J. Jiang, J. Xu, S. Li, Y. Zhou, and Y. Xu. 2017. Throughput-Optimized FPGA Accelerator for Deep Convolutional Neural Networks. *ACM TRETS* 10, 3 (2017), 17.

[38] Liqiang Lu, Jiaming Xie, Ruirui Huang, Jiansong Zhang, Wei Lin, and Yun Liang. 2019. An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs. In *FCCM*. IEEE, 17–25.

[39] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*. 5058–5066.

[40] N. Mellempudi, A. Kundu, D. Mudigere, D. Das, B. Kaul, and P. Dubey. 2017. Ternary Neural Networks with Fine-Grained Quantization. *arXiv preprint arXiv:1705.01462* (2017).

[41] A. Mishra, E. Nurvitadhi, J.J. Cook, and D. Marr. 2017. WRPN: wide reduced-precision networks. *arXiv preprint arXiv:1709.01134* (2017).

[42] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*. 525–542.

[43] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. 2019. {XONN}: XNOR-based Oblivious Deep Neural Network Inference. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1501–1518.

[44] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. 2017. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864* (2017).

[45] M. Samragh, , M. Ghasemzadeh, and F. Koushanfar. 2017. Customizing Neural Networks for Efficient FPGA Implementation. In *FCCM*. 85–92.

[46] M. Samragh, M. Imani, F. Koushanfar, and T. Rosing. 2017. LookNN: Neural network with no multiplication. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1775–1780.

[47] Mohammad Samragh, Mojan Javaheripi, and Farinaz Koushanfar. 2019. AutoRank: Automated rank selection for effective neural network customization. In *Proceedings of the ML-for-Systems Workshop at the 46th International Symposium on Computer Architecture (ISCA'19)*.

[48] H. Sharma, J. Park, D. Mahajan, E. Amaro, J.K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *IEEE/ACM International Symposium on Microarchitecture*. 17.

[49] Colin Shea and Tinoosh Mohsenin. 2019. Heterogeneous Scheduling of Deep Neural Networks for Low-power Real-time Designs. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 15, 4 (2019), 1–31.

[50] C. Shea, A. Page, and T. Mohsenin. 2018. SCALENet: A SCalable Low power AccELerator for Real-time Embedded Deep Neural Networks. In *2018 on Great Lakes Symposium on VLSI*. ACM, 129–134.

[51] Y. Shen, M. Ferdman, and P. Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *ISCA*. 535–547.

[52] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J. Seo, and Y. Cao. 2016. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *FPGA*. 16–25.

[53] Y. Umuroglu, N.J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. 2017. Finn: A framework for fast, scalable binarized neural network inference. In *FPGA*. IEEE, 65–74.

[54] Erwei Wang, James J Davis, Peter YK Cheung, and George A Constantinides. 2019. LUTNet: Rethinking Inference in FPGA Soft Logic. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 26–34.

[55] Huan Wang, Qiming Zhang, Yuehai Wang, and Haoji Hu. 2017. Structured probabilistic pruning for convolutional neural network acceleration. *arXiv preprint arXiv:1709.06994* (2017).

[56] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.

[57] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. 2016. Learning structured sparsity in deep neural networks. In *NIPS*. 2074–2082.

[58] T. Yang, A. Howard, B. Chen, X. Zhang, A. Go, V. Sze, and H. Adam. 2018. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. *arXiv preprint arXiv:1804.03230* (2018).

[59] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong. 2016. Energy-efficient CNN implementation on a deeply pipelined FPGA cluster. In *International Symposium on Low Power Electronics and Design*. 326–331.

[60] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. 2016. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).

[61] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. 2018. Towards effective low-bitwidth convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 7920–7928.