This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3069139, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Locality-based Encoder and Model Quantization for Efficient Hyper-Dimensional Computing

Justin Morris*†, Yilun Hao*, Roshan Fernando*, Mohsen Imani‡,
Baris Aksanli†, and Tajana Rosing*
*University of California San Diego, La Jolla, CA 92093, USA
†San Diego State University, San Diego, CA 92182, USA
‡University of California Irvine, Irvine, CA 92697, USA
{justinmorris, yih301, rdf004}@ucsd.edu, m.imani@uci.edu, baksanli@sdsu.edu, tajana@ucsd.edu

***Abstract*—Brain-inspired Hyperdimensional (HD) computing is a new computing paradigm emulating the neuron's activity in high-dimensional space. The first step in HD computing is to map each data point into high-dimensional space (e.g., 10,000), which requires the computation of thousands of operations for each element of data in the original domain. Encoding alone takes about 80% of the execution time of training. In this paper, we propose, ReHD, an entire rework of encoding, training, and inference in HD computing for a more hardware friendly implementation. ReHD includes a fully binary encoding module for HD computing for energy-efficient and high-accuracy classification. Our encoding module based on random projection with a predictable memory access pattern can be efficiently implemented in hardware. ReHD is the first HD-based approach that provides data projection with a 1:1 ratio to the original data and enables all training/inference computation to be performed using binary hypervectors. After the optimizations ReHD adds to the encoding process, retraining and inference become the energy intensive part of HD computing. To resolve this, we additionally propose model quantization. model quantization introduces a novel method of storing class hypervectors using n-bits, where n ranges from 1 to 32, rather than at full 32-bit precision, which allows for fine-grained tuning of the trade-off between energy efficiency and accuracy. To further improve ReHD efficiency, we developed an online dimension reduction approach that removes *insignificant* hypervector dimensions during training.**

***Index Terms*—Brain-inspired computing, Hyperdimensional computing, Machine learning, Energy efficiency**

I. Introduction

The emergence of the Internet of Things (IoT) has led to a copious amount of small connected embedded devices. Many of these devices need to perform classification tasks such as speech recognition, activity recognition, face detection, and medical diagnosis [1], [2]. However, these small embedded devices do not have the computing power to run sophisticated classification algorithms such as Deep Neural Networks(DNN) [3]. To resolve this, many devices send the data they collect to the cloud and the cloud performs the inference task, sending the result back to the embedded device. This leads to new problems such as network usage and user security [4]. In order to solve these new issues and still provide a way for these embedded devices to perform classification tasks, we need a light-weight classification algorithm that can achieve comparable accuracy to sophisticated resource-intensive algorithms.

Brain-inspired Hyperdimensional (HD) computing has been proposed as the alternative computing method that processes the cognitive tasks in a more light-weight way [5]. HD computing is developed based on the fact that brains compute with *patterns of neural activity* [5]. Recent research utilized high dimension vectors (e.g., more than a thousand dimensions), called *hypervectors*, to represent the neural activities and showed successful progress for many cognitive tasks such as activity recognition, object recognition, language recognition, and bio-signal classification [6], [7], [8]. HD computing offers an efficient learning strategy without overcomplex computation steps such as back propagation in neural networks. In addition, it builds upon a well-defined set of operations with random HD vectors which makes the learning model extremely robust in the possible presence of hardware failures. HD has even recently been used for more secure learning [9].

In HD computing, training data points are combined into a set of hypervectors, called *an HD model*, through light-weight computation steps. Each hypervector in the model represents a class of the target classification problem. Most of the proposed HD computing work exploits binarized hypervectors to reduce the computational/memory intensity in HD computing [10], [11]. However, the existing HD computing algorithms [10] have two main challenges: (i) the encoding is computationally expensive, as it requires the computation of thousands (e.g., 10,000) of operations to map each element of data from the original domain to high-dimensional space [8], [12]. For example, our experiments on five practical applications (described in Section VII) show that in HD computing the encoding module takes about 79% and 74% of the training and inference time. (ii) In addition, HD computing using binary encoded vectors provides significantly lower classification accuracy. In other words, HD computing requires non-binary (integer) vectors in order to provide acceptable accuracy. However, working with non-binary vectors significantly increases the memory requirement, and the computation complexity of training and inference.

Designs were previously constrained to two methodologies for performing all training and inference computation. In the first methodology, we represent the encoded training data and model class hypervectors using binary hypervectors. This process is much more efficient because we compute the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3069139, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

similarity between query hypervectors and model hypervectors via Hamming distance. However, binary representation results in significant information loss and therefore lower classification accuracy [13]. In the second methodology, we perform all training/inference computation with 32-bit integers. When using a 32-bit model, we need to utilize the more computationally expensive cosine similarity between the query hypervector and class hypervectors. However, information is preserved and we maintain high classification accuracy. In other words, there existed a trade-off between two extremes: high efficiency and low accuracy or inefficiency and high accuracy. However, many situations call for more nuanced control over the trade-off between computational efficiency and classification accuracy.

In this paper, we propose ReHD, a full rework of Brain-Inspired HD computing to make it more hardware friendly and achieve energy-efficient and high-accuracy classification. ReHD introduces a novel encoding module based on random projection with a predictable memory access pattern that can be efficiently implemented in hardware. In contrast to existing HD computing algorithms that increase the size of encoded data by 20× [10], ReHD is the first HD-based approach which provides data projection with a 1:1 ratio to the original data. In addition, ReHD encodes all data to binary hypervectors, simplifying computation in training and inference. The low memory requirement and computation cost makes ReHD a suitable candidate for embedded devices with limited resources. To address systems that need more control over the trade-off between computational efficiency and classification accuracy, we propose n-bit model quantization. With our new model quantization method, we represent hypervector elements with n-bit integers. To further improve ReHD efficiency, we improve online dimension reduction by intelligently choosing insignificant dimensions to remove.

II. Related Work & Motivation

Prior work tried to apply the idea of high-dimensional computing to different classification problems such as language recognition, speech recognition, face detection, EMG gesture detection, human-computer interaction, and sensor fusion prediction [6], [10], [8], [11], [14], [15], [16]. For example, work in [12] proposed a simple and scalable alternative to latent semantic analysis. Additionally, work in [8] proposed a new HD encoding based on random indexing for recognizing a text's language by generating and comparing text hypervectors. Work in [17] proposed an encoding method to map and classify biosignal sensory data in high dimensional space. Work in [7], [10] proposed a general encoding module that maps feature vectors into high-dimensional space while keeping most of the original data. Prior work also accelerated HD computing by binarizing the class hypervectors [18], [13], removing dimensions of the class hypervectors [19], or compressing the HD model [20]. [21] extended the idea of binarizing to instead use a ternary model to achieve higher accuracies. Work in [22] also proposed a dynamic dimensionality model to improve energy efficiency.

Prior work also tried to design hardware acceleration for HD computing by mapping its operations into hardware, e.g., in-memory architecture [23], [24], [25], [26], [27], [28], [29], [30], and tried to accelerate HD computing in hardware by binarizing the class hypervectors [18], [13] or removing dimensions of the class hypervectors [19], [31]. Work in [32] designed an FPGA implementation to accelerate HD computation in the binary domain. However, the application of these approaches is limited to simple classification problems such as language recognition [8]. To provide acceptable classification accuracy, all of these approaches have to train the model using non-binary (integer) vectors. However, using non-binary vectors requires a large memory footprint and computation cost in both training and inference.

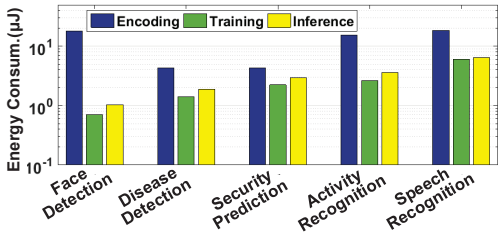

Fig. 1. Energy consumption of HD encoding, training, and inference.

Model quantization is a widely used technique in machine learning applications to improve energy efficiency. For instance, Google's TPU for performing inference on DNNs utilizes reduced bit representations [33]. Furthermore, [34] proposes a quantization method for SVMs. Model quantization has also been used to reduce the memory requirement for a more efficient hardware design [35], [36]. Work has also been done to adaptively change the precision of the model to reduce the accuracy loss online [37]. [38] proposes a method to use multiple precision levels during inference to achieve a balance between efficiency and accuracy loss. [39] tries to alleviate accuracy loss from quantization by compensating for computational errors. Other methods such as model compression have also been used to improve the energy efficiency of neural networks [40].

In this work, we observe that the existing encoding modules are algorithmically and computationally inefficient. In addition, to get high accuracy, the encoding needs to map data into vectors with integer values which significantly increases the data size [10], [11]. This large memory requirement is often not available on embedded devices with limited resources. Figure 1 shows the energy consumption of encoding, training, and inference (associative search) when running a single data point on five practical applications. Our evaluation shows that the encoding module on average takes 4.7× and 3.8× higher energy than HD training and inference. In this work, we propose a novel encoding approach that (i) significantly reduces the encoding computation cost by introducing computation locality and (ii) provides high classification accuracy while mapping data into binary vectors with much lower dimensionality than existing algorithms.

III. Encoding with ReHD

In this paper, we propose ReHD, a novel hardware friendly framework for efficient classification. ReHD consists of three

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3069139, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

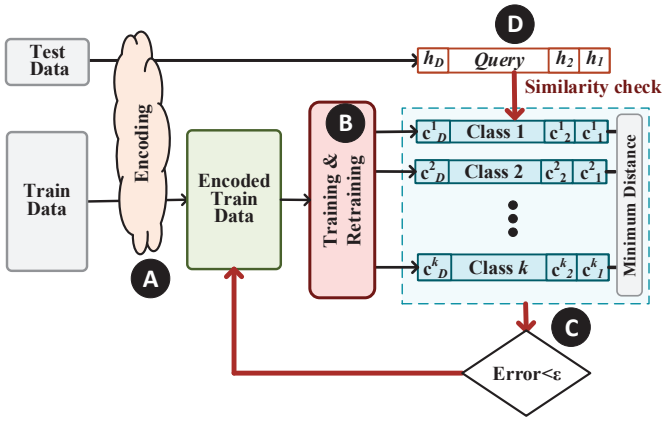

Fig. 2. Overview of how ReHD is constructed and how ReHD performs inference.

main modules shown in Figure 2: encoding, training, and inference. The encoding module maps each data point to binary high-dimensional space. Our encoding has been designed to map the maximum amount of information to high dimensional space with the minimum computation cost. ReHD accumulates every encoded binary training hypervector to create an integer model. This integer model is then used to create a quantized model. ReHD accordingly proposes a training approach that enables the values to stay quantized during training. During inference, cosine similarity has been used as the similarity metric in prior work to achieve the best accuracy in HD computing applications [28]. Quantizing the model enables ReHD inference to be supported using a more efficient n-bit cosine similarity rather than full 32-bit precision. In the following, we explain the details of ReHD functionality.

A. Baseline Encoding

Figure 2Ⓐ shows the overview of ReHD performing the classification task. Before we can work in high dimension space, we first need to encode the data to hypervectors.

Consider a feature vector $\mathbf{F} = \{f_1,\ f_2, \ldots,\ f_n\}$. The encoding module takes this n-dimensional vector and converts it into a D-dimensional hypervector ($D >> n$). The encoding is performed in three steps, which we describe below. The encoding scheme assigns a unique channel ID to each feature position. These IDs are hypervectors which are randomly generated such that all features will have orthogonal channel IDs, i.e., $\delta(ID_i,\ ID_j) < 5,000$) for $D = 10,000$ and $i \neq j$; where the δ measures the element-wise similarity between the vectors. The HD computing encoder also generates a set of *level* hypervectors to consider the impact of each feature value [7]. To create these level hypervectors, we compute the minimum and maximum feature values among all data points, $\mathbf{v}_{min}$ and $\mathbf{v}_{max}$, then quantize the range of $[\mathbf{v}_{min}, \mathbf{v}_{max}]$ into Q levels, $\mathbb{L} = \{L_1, \cdots, L_Q\}$. Each of these quantized scalars corresponds to a D-dimensional hypervector [7]. To encode a feature vector, the encoder looks at each position of the feature vector and element-wise multiplies the channel ID (ID_i) with the corresponding level hypervector (L_i). The following equation shows how an n length feature vector is mapped into HD space with this encoding scheme:

$$\mathbf{H} = [hv_1 * ID_1 + hv_2 * ID_2 + \ldots + hv_n * ID_n]$$
$$hv_j \in \{L_1,\ L_2, \ldots,\ L_m\},\quad 1 \leqslant j \leqslant n$$

It is clear that this encoding scheme is inefficient due to consistent random memory accesses to find the corresponding level hypervector for each feature value. In addition, the amount of computations needed is large and does not take advantage of hardware optimizations like data sparsity.

B. Random Projection

We desire a fast and hardware-friendly algorithm that can take a vector of real-valued data and generate a binary code such that the encoding preserves the cosine similarity. Let us assume $\mathbf{A}, \mathbf{B} \in \mathbb{R}^n$ are two feature vectors in the original domain with real values. We wish to define an encoding operation $\lambda(*)$ such that:

$$\{\mathbf{X} = \lambda(\mathbf{A}), \mathbf{Y} = \lambda(\mathbf{B})\ ,\ \mathbf{X}, \mathbf{Y} \in \{1, -1\}^D\}$$
$$\delta(\mathbf{A}, \mathbf{B}) = \delta(\mathbf{X}, \mathbf{Y})$$

where $\delta(*)$ is the cosine similarity. Since the cosine angle of binary vectors is determined by how many bits match, the cosine angle and Hamming distance are proportional. This type of encoding can be performed using Locality Sensitive Hash algorithms, such as Random Projection [41]. Let us assume a feature vector $\mathbf{F} = \{f_1,\ f_2, \ldots,\ f_n\}$, with n features ($f_i \in \mathbb{N}$) in original domain. The goal of random projection is to map this feature vector to a D dimensional space vector: $\mathbf{H} = \{h_1,\ h_2, \ldots,\ h_D\}$. As Figure 3a shows, random projection generates D dense bipolar vectors with the same dimensionality as original domain, $\{\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_D\}$, where $\mathbf{P}_i \in \{-1, 1\}^n$. The inner product of a feature vector with each randomly generated vector gives us a single dimension of a hypervector in high-dimensional space. For example, we can compute the $i-th$ dimension of the encoded data as:

$$h_i = sign(\mathbf{P}_i \cdot \mathbf{F})$$

where $sign$ is a sign function which maps the result of the dot product to +1 or -1 values. This type of hashing involves a large amount of multiplications/additions which is inefficient in hardware. For example, to map a feature vector from n to D dimensions, this encoding involves $n \times D$ multiplication and addition operations.

C. Sparse Random Encoding

The efficiency of random projection can be improved by sparsifying each projection vector. Instead of generating dense projection vectors, we can generate sparse projection vectors(Figure 3b). Consider s as a sparsity of each projection vector. Then, for each sparse projection vector, only $s\%$ of the vector's elements are randomly generated and the rest are set to zero. For example, if $s = 5\%$, each projection vector only has $0.05 \times n$ non-zero elements. Therefore, each dimension of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3069139, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

encoded hypervector can be computed with only $0.05 \times n$ multiplication/addition operations. Therefore, encoding a single hypervector takes $s \times n \times D$ multiplication/addition operations, compared to $n \times D$ multiplication/addition operations with dense projection vectors. Although the sparsity significantly reduces the number of arithmetic operations, it introduces random accesses to the algorithm, which is hard on the cache and slows down the computation.

D. Locality-based Sparse Random Projection

Here we propose a novel approach that keeps the advantages of a sparse projection matrix, i.e., fewer operations while removing random accesses to make the algorithm more hardware friendly. We propose a locality-based random projection encoding that uses a predictable access pattern. Instead of selecting s% random indices of the projection matrix to be non-zero, we approximate sparse random projection by selecting pre-defined indices to be non-zero. Figure 3c shows the structure of the locality-based matrix. Our approach selects the first $s \times n$ of the $\mathbf{P}_1$ vector to be non-zero (indices $[1...s \times n]$). Similarly, P_2 projection vector only has $s \times n$ non-zero elements on indices $[2...s \times n - 1]$. Finally, $\mathbf{P}_D$ contains non-zero elements on the last $s \times n$ dimensions. This creates a clear spacial locality pattern that hardware accelerators can take advantage of.

Figure 4 shows the overview of ReHD encoding mapping each n dimensional feature vector to a D dimensional binary hypervector. ReHD simplifies the projection matrix to a single dense random projection vector with D bipolar values. Our approach first replicates the feature vector, $\mathbf{F}$, such that it extends to D dimensions, the same as our desired high-dimensional vector. For example, to encode a feature vector with $n = 500$ features to $D = 4,000$ dimensions, we need to concatenate 8 copies of a feature vector together. Then, it generates a random D dimensional projection vector, P, next to the extended feature vector (as shown in Figure 4). To compute the dimensions of the high-dimensional vector, ReHD takes the dot product of the extended feature vector with each projection vector in an N-gram window. The first N-gram calculates the dot product of the first N features and N projection vector elements:

$$h_1 = sign(f_1 * p_1 + f_2 * p_2 + ... + f_N * p_N)$$

Similarly, the N-gram window shifts by a single position to generate the next feature values. So, we can compute the i^{th} dimension of an encoded hypervector using:

$$h_i = sign(f_i * p_i + f_{i+1} * p_{i+1} + ... + f_{i+N} * p_{i+N})$$

Each step of the N-gram window corresponds to a multiplication with a sparse projection vector in the projection matrix. Although this encoding has the same number of computations as sparse random projection, it provides the following advantages: (i) it removes random accesses from the feature selection by introducing spacial locality, which significantly reduces the cost of hardware implementation. (ii) There is an opportunity for computation reuse, as every neighboring dimension shares $N - 1$ terms.

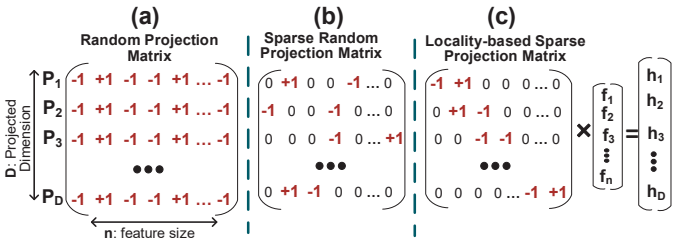

Fig. 3. Random projection encoding using dense, sparse, and locality-based projection matrix.

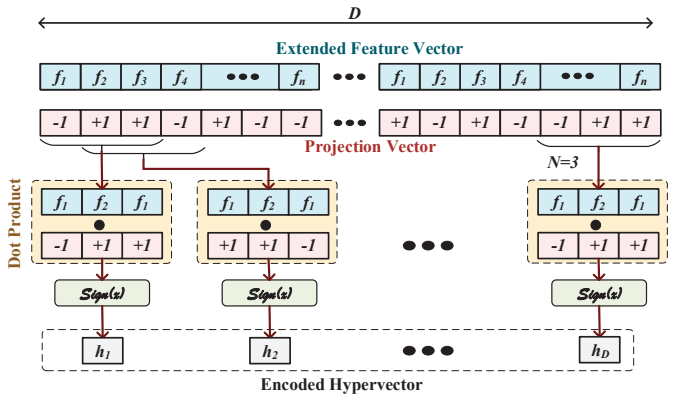

Fig. 4. Locality-based random projection encoding.

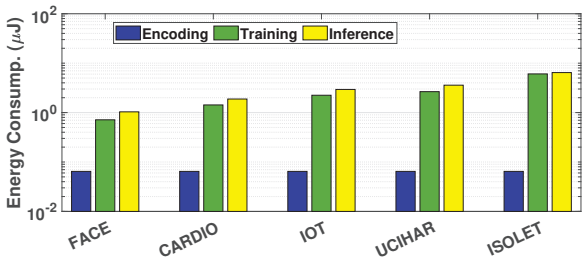

Fig. 5. Energy consumption of HD encoding, training, and inference after utilizing the proposed encoding module.

IV. Training in ReHD

After utilizing our new hardware-friendly encoding, we observe that training and inference are now the energy-intensive parts of the HD algorithm. Figure 5 shows the updated energy consumption of encoding, training, and inference (associative search) when running a single data point on five practical applications when utilizing the proposed encoding. Our evaluation shows that training and inference on average take 43% and 55% of the total energy when using the new proposed encoding. This is mainly due to the usage of full precision 32-bit HD models. In this work, we propose a novel approach which (i) allows the HD computing model hypervectors to be represented with n-bit integers, where n ranges from 1 to 32, and (ii) allows for fine-grained control between accuracy and energy efficiency compared to the previous approach of utilizing full 32-bit precision or 1-bit binary models.

A. Baseline HD Computing

Initial Training: Figure 2Ⓑ shows the functionality of Baseline HD Computing during training. In baseline HD computing, the model used in training is initialized through by element-wise addition of all encoded hypervectors in each existing class. The result of training is k hypervectors each with D dimensions, where k is the number of classes. For example, the i^{th} class hypervector can be computed as: $\mathbf{C_i} = \sum_{\forall j \in class_i} \mathbf{H_j}$. This training operation involves a large number of integer additions, which makes the HD computation costly [10].

Retraining: HD computing performs model adjustment by iterating through the training dataset. Figure 2Ⓑ shows the functionality of baseline HD computing during retraining. In a single iteration of model adjustment, HD computing checks the cosine similarity of all training data points, say $\mathbf{H}$, with the class hypervectors in the trained binary model. If a data point is incorrectly classified by the model, HD updates the model by (i) adding the incorrectly classified hypervector to the class the input data point belongs to ($\widetilde{\mathbf{C}}^{correct} = \mathbf{C}^{correct} + \mathbf{H}$), and (ii) subtracting it from the class to which it is wrongly matched ($\widetilde{\mathbf{C}}^{wrong} = \mathbf{C}^{wrong} - \mathbf{H}$). After each retraining iteration, we check the classification accuracies from the last three iterations and stop the retraining if the change in error is less than 0.1% (Figure 2Ⓒ). The retraining stops after 20 iterations if the convergence condition is still not satisfied. We are able to stop after just 20 retraining iterations as prior work has shown that HD is a fast learning algorithm and often only needs 10 iterations of retraining compared to 100s for DNNS [42].

B. Binary Model Quantization

Previous work proposed quantization to a binary model for improved speed and efficiency [43].

Initial Training: An integer model is first initialized through element-wise addition of all encoded hypervectors in each existing class. Like in Baseline HD Computing, the result is k hypervectors, each with D dimensions, where k is the number of classes. For example, i^{th} class hypervector can be computed as $\mathbf{C_i} = \sum_{\forall j \in class_i} \mathbf{H_j}$. We then binarize each class hypervector from the integer model to create the binary model. We perform this binarization operation by taking the sign bit of each dimension from the accumulated class HVs.

Retraining: We train the binarized model by iterating through the training set. Throughout training, we maintain both a binary model and an integer model of the class hypervectors. In a single iteration of model adjustment, HD computing checks the similarity of all training data points, say $\mathbf{H}$, with the class hypervectors in the trained binary model. The data point is assigned to the class with which it has the closest Hamming distance. If a data point is incorrectly classified by the model, HD updates the model by (i) adding the incorrectly classified hypervector to the class the input data point belongs to ($\widetilde{\mathbf{C}}^{correct} = \mathbf{C}^{correct} + \mathbf{H}$), and (ii) subtracting it from the class to which it is wrongly matched ($\widetilde{\mathbf{C}}^{wrong} = \mathbf{C}^{wrong} - \mathbf{H}$). These changes are made to the integer model saved from training because adding to and subtracting from the binary model would drastically change the model. To update the binary model, the updated class hypervectors from the integer model are binarized via the same process described in training.

C. N-Bit Model Quantization

The Binary Model results in faster and more efficient training because the model is represented with integers smaller than 32 bits, but a sharp decline in accuracy often accompanies the increase in speed and efficiency. The binary model quantization, where we represent the dimensions of model hypervectors with 1 bit, maximizes efficiency but also yields the lowest classification accuracy. This forces us to choose between two extremes: low accuracy but high efficiency (binary), and high efficiency but low accuracy (32-bit). To solve the problem of having to choose between two extremes, we can achieve more granular control over this trade-off by representing dimensions with n bits, where n ranges from 1 to 32. Hence, we no longer have to choose between 1-bit and 32-bits. As we represent dimensions with more bits, we increase the accuracy but make classification less efficient.

Initial Training: The initial training for model quantization is very similar to the initial training for the binary model, as the integer model is created through the same process. The training process for model quantization diverges from that of past work after the initial addition, as rather than an adjacent binary model, we create an adjacent n-bit model. To represent the dimensions with n-bits, we utilize the integer model and clip all dimensions that fall outside of the range of integer values we can represent with n bits. For an n-bit model quantization, we can represent the range $[-2^n, 2^n - 1]$. Therefore, for all elements of class hypervectors, we discard any overflow beyond this range.

Retraining: The retraining process for model quantization is also similar to the retraining process for the binary model. Throughout training, we store both an integer model and an n-bit representation model of the class hypervectors. Model quantization performs model adjustment by iterating through the training dataset, making changes to the integer model, and reflecting those changes to the n-bit representation model similar to the initial training process.

D. Model Quantization Inference

After training and retraining, the HD model can now be used for inference (Figure 2Ⓓ). The input data is encoded as a binary *query* hypervector. Model quantization then computes the similarity between the binary *query* hypervector and each n-bit class hypervector. 1-bit model quantization computes similarity using Hamming distance and n-bit model quantization using cosine similarity over n bits. The input data is classified into the class whose hypervector it is most similar to. As the number of bits used to represent dimensions increases, so does the inference accuracy, but the training, retraining, and inference processes become more complex.

V. Online Dimension Reduction

The gradient descent during retraining gives equal weight to all features when the data is binarized. This includes

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3069139, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

noisy, low strength features as well as features with high intra-class differences. In fact, gradient descent moves the hyperplane in the direction of these features with equal strength as the important features, which results in possible overfitting. The challenge is to amplify the learning rate of "significant" dimensions, while not amplifying the learning rate of "insignificant" or "noisy" features. Online dimension reduction attempts to remove insignificant "noisy" dimensions from the model to improve energy efficiency. We can define insignificant dimensions using either high absolute values or low variance as a metric. We define s as the sparsity level denoting what percentage of the dimensions will be removed, regardless of which metric is used, dropping the $s\%$ most insignificant dimensions from the model, results in an efficiency improvement of approximately $s\%$.

We drop the $s\%$ most insignificant dimensions from the model rather than using a thresholding technique because the range of values varies between datasets, as it depends on how many samples there are. Datasets with larger amounts of samples result in significantly larger accumulated dimensions compared to those with fewer samples. This is because of how the initial model is created by accumulating all the encoded samples. Therefore, with more samples, the dimensions that agree across all samples will accumulate much higher values. However, we can account for this difference in datasets by removing a proportion rather than an absolute threshold.

To use high absolute values as a metric of insignificance, we first compute the element-wise addition of all binarized sample hypervectors and examine the sum of each dimension. Because all training hypervectors are initially binary with +1 or -1, dimensions with a very high sum indicate that most training instances have a +1 for that dimension, and dimensions with a very low sum indicate that most training instances have a -1 for that dimension. Such dimensions have low differentiation between training instance data points and low differentiation between classes, so we declare dimensions with high absolute value sums to be "insignificant", as Figure 6 shows. This is because to distinguish the classes from each other, we want to emphasize their differences and not their similarities.

We can choose insignificant dimensions more intelligently by using low variance as a metric of noise and low-strength. Before the encoded hypervectors are binarized by taking the sign bit, we calculate the variance of each dimension. The dimensions with low variances indicate that those dimensions contain mutual information among all the samples, and thus do not help the model differentiate between classes. Dimensions with high variance are declared "significant', while dimensions with low variances are "insignificant". As stated above, we must emphasize inter-class differences rather than similarities. This method drops the dimensions with the lowest variances from the model as shown in Figure 7. Comparing the distributions of the variances shown in Figure 7 and the distributions of absolute values in Figure 6, we can see that the variance metric can cluster and identify more insignificant dimensions compared to the absolute value metric. Thus, using variance as the metric to determine insignificant dimensions is able to reduce dimensionality further with less accuracy loss than using high absolute values.

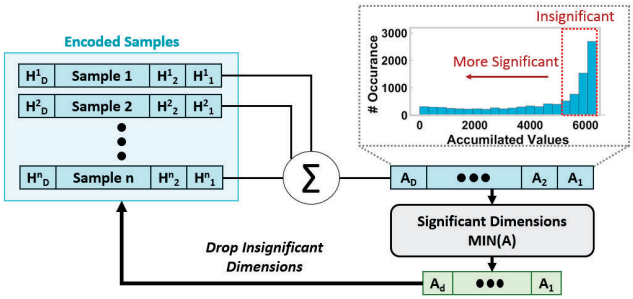

Fig. 6. Online dimension reduction with absolute value.

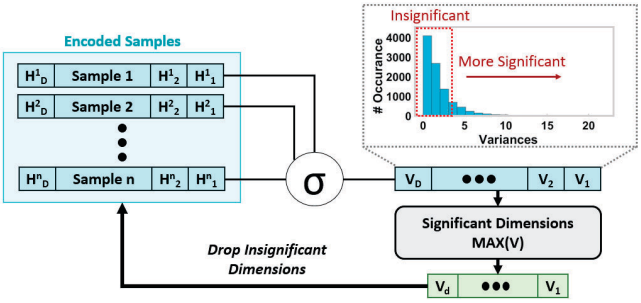

Fig. 7. Online dimension reduction with variance.

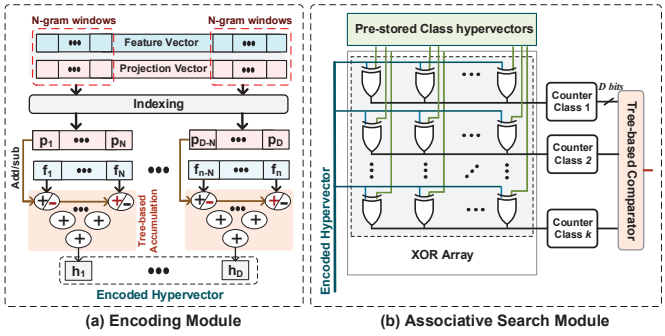

Fig. 8. FPGA implementation of the encoding and associative search block.

VI. FPGA Acceleration

ReHD can be accelerated on different platforms such as CPU, GPU, FPGA, or ASIC. FPGA is one of the best options as ReHD computation involves bitwise operations among long vector sizes. General strategies of optimizing the performance of ReHD are (i) using a pipeline and partial unrolling on low levels (dimension levels) to speed up each task and (ii) using dataflow design on a high level (task level) to build a stream processing architecture that lets different tasks run concurrently. In the following, we explain the functionality of ReHD in encoding, training, retraining, and inference phases.

A. Encoding Implementation

As we explained in Section III-C, we used the locality-based random projection encoding to implement the encoding module. Due to the sequential and predictable memory access patterns as well as the abundance of binary operations, this encoding approach can be implemented efficiently on an FPGA.

In the hardware implementation, we represent all $\{-1, +1\}$ values with $\{0, 1\}$ respectively. This enables us to represent each element of the projection vector using a single bit. Figure 8a shows the hardware implementation of the ReHD encoding module. The encoding process includes reading a feature vector from off-chip DDR memory and generating a binary hypervector from them.

Calculating the inner product of a feature vector and a projection vector, $P \in \{1, -1\}^D$, can be implemented with no multiplications. Each element of the projection vector decides the sign of each dimension of the feature vector in the accumulation of the dot product. Thus, the dot product can be simplified to the addition/subtraction of the feature vector elements. Right after the encoding, the hypervectors are used for initial model training. We also need to store the encoded hypervectors for retraining. However, the FPGA does not have enough BRAM blocks to store all encoded hypervectors, so, our design stores them into DDR memory.

B. Training Implementation

Initial Training: Like previously, initial training for ReHD with model quantization is a single-pass process. The training module accesses the encoded hypervectors and accumulates them in order to create a hypervector representing each class. When the training module accumulates the encoded hypervector to one of the class hypervectors, the encoding module maps the next training data into high-dimensional space, improving data throughput by increasing resource utilization. After going through all of the training data, our implementation creates an n-bit quantized representation of the model. We iterate through all hypervectors in the training and test datasets, and clip values greater than 2^n-1 to 2^n-1 and values less than -2^n to -2^n. Finally, the quantized n-bit model is stored in the BRAM blocks to be used for inference or retraining.

Retraining: The retraining phase first sequentially reads already encoded training hypervectors from the off-chip memory in batches to help hide the latency of reading from the off-chip memory. This is necessary as each read has a latency of about $15ns$, which would slow down the retraining process. Next, we check the similarity of each data point with all trained class hypervectors. Each data point gets a tag of a class in which it has the highest Hamming distance (1-bit quantized model) or cosine similarity (n-bit quantized models with $n \neq 1$). In the case of misclassification, ReHD needs to update the model by adding and subtracting a data hypervector with two class hypervectors as defined before.

C. Inference Implementation

After the retraining, the quantized ReHD model has a stable model that can be used in the inference phase. The encoding module is integrated with the similarity check module as the entire inference part. Each test data point is first encoded to high-dimensional space using the same encoding block explained in Section VI-A. Next, the quantized ReHD model checks the cosine similarity of the data point with all pre-stored class hypervectors, in order to find a class with the highest similarity. One unique advantage of our approach is its capability to enable online training during the inference phase. Our implementation stores two HD models: one with integer values used for retraining and an n-bit quantized model which is used to perform the classification task. ReHD quantizes the integer model to an n-bit model periodically to update the inference model. While the previous model computes similarity with Hamming distance, the updated quantized ReHD model computes cosine similarity. cosine similarity with n-bit quantized models may seem much more energy intensive than utilizing Hamming distance for binary models because cosine similarity involves multiplications. However, we can use the same optimization in encoding that removed the multiplications between the feature vector and a projection vector to remove the multiplications between the encoded query hypervector and n-bit quantized class hypervector. This is because each element of the encoded query hypervector is binary. Each element of the query hypervector decides the sign of each dimension of the feature vector in the accumulation of the dot product step of cosine similarity. Although Hamming distance is still faster and more computationally efficient, cosine similarity results in higher accuracy when we represent the dimensions of class and instance with hypervectors with n bits rather than 1-bit.

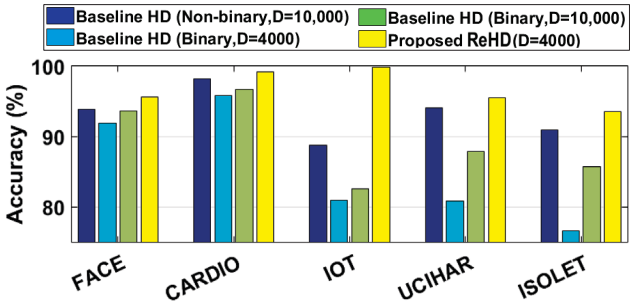

Fig. 9. Classification accuracy of ReHD and the baseline HD using binary and integer models.

VII. Evaluation

A. Experimental Setup

We implemented ReHD training, retraining, and inference in both software and hardware. In software, we implemented ReHD with Python. In hardware, we fully implemented ReHD using Verilog. We verify the timing and the functionality of the models by synthesizing them using Xilinx Vivado Design Suite [44]. The synthesis code has been implemented on the Kintex-7 FPGA KC705 Evaluation Kit. We evaluated the efficiency of the proposed ReHD on four practical classification problems listed below: Speech Recognition (ISOLET) [45], Activity Recognition (UCIHAR) [46], Face Detection (FACE) [47], Cardiotocography (CARDIO) [48], and Attack Detection in IoT systems (IoT) [49]. We compare ReHD with, baseline HD, an FPGA implementation of [7].

B. Comparison With Other State-of-the-Art Light-Weight Classifiers

Table I compares HD computing with other light-weight classifiers including support vector machines (SVM), gradient

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3069139, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

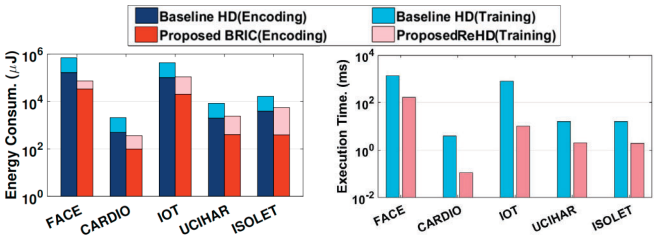

Fig. 10. Energy consumption and execution time of ReHD and the baseline HD during training.

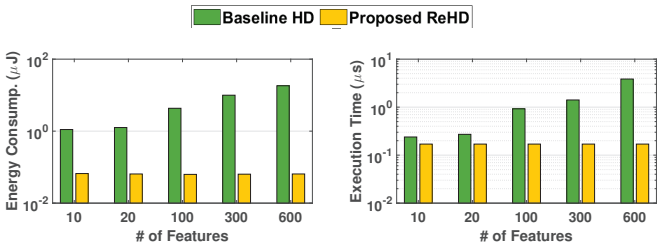

Fig. 11. Scalability of the encoding module in ReHD and the baseline HD with the feature size.

TABLE I
CPU-BASED COMPARISON OF HD AND OTHER CLASSIFIERS.

	SVM	Perceptron	MLP	HD
Training Exe.(ms)	480.3	320.2	1,229.2	168.3
Testing Exe.(μs)	813.7	102.4	286.2	59.4

boosting classifiers (Boosting), perceptrons, and multi-layer perceptrons (MLP) in terms of accuracy and training/inference efficiency. All results are reported when applications are running on an embedded device (Raspberry Pi 3) using an ARM Cortex A53 CPU. Our evaluation shows that HD computing can provide comparable accuracy to algorithms such as SVM and MLP. In terms of efficiency, HD computing can provide much faster computation in both training and testing. For example, in a CPU implementation, HD computing is 7.3× and 4.8× (2.9× and 13.6×) faster than MLP (SVM) during training and testing respectively. These results demonstrate that HD computing is the clear choice among light-weight classifiers for low-powered energy efficient machine learning.

C. ReHD Accuracy and Memory Requirement

Figure 9 compares the impact of hypervector dimensions on the classification accuracy of ReHD and the baseline HD computing encoding [10]. As we explained, ReHD always encodes data points into D binary dimensions. However, for the baseline HD computing encoding, we consider two cases when HD encodes data points to binary and integer domains. Our results in Figure 9 indicate that ReHD requires significantly fewer dimensions to provide the same accuracy as the baseline. For example, ReHD using $D = 4,000$ binary dimensions provides the same accuracy as the baseline with $D = 10,000$ integer dimensions. In addition, the baseline with a binarized model provides significantly lower accuracy than ReHD and the baseline with an integer model. ReHD is on average 11.5% more accurate than the baseline using a binary encoding and binary model. However, as we explore in Section VII-E, ReHD is able to achieve even higher accuracies when utilizing n-bit quantization compared to binary quantization.

Here we compare ReHD and the baseline in terms of the training memory requirement. As we explained in Section IV-A, the baseline/ReHD store all encoded training data in memory. Going into high dimensional space intuitively means increasing the data size, since we map each feature vector from n into D dimensional space, where $D >> n$. Let us assume a feature vector with $n = 500$ integer features. For the baseline with integer values, the data size increases by approximately 20×. Even the baseline with a binary encoding ($D = 10,000$) increases the data size by 2.5×, while it provides much lower accuracy. In contrast, the proposed ReHD encodes data points to a much lower dimensionality, e.g., $D = 4000$, in order to provide the same accuracy as the baseline. Our evaluation shows that ReHD can ensure 1:1 ratio of high-dimensional data to original data, while providing the same accuracy as baseline HD [10], proving that ReHD is more capable to run on embedded devices with limited memory.

D. Hardware Efficiency

We compare the efficiency of ReHD with the state-of-the-art HD computation algorithms on a Kintex-7 FPGA. To have a fair comparison, we consider an optimized implementation of the baseline [10], running on the same architecture as ReHD (explained in Section VI).

Encoding & Training: Due to the predictable memory access pattern and lower ReHD dimensionality, ReHD encoding can process with higher efficiency as compared to the baseline. For instance, to get maximum accuracy, the baseline needs to work with $D = 10,000$ dimensionality while ReHD can provide the same accuracy with $D = 4,000$. Figure 11 shows the scalability of ReHD and the baseline efficiency in terms of the feature size. Our evaluation shows that the execution time of the baseline increases with the number of features, while it takes the same time for ReHD to encode any size feature vector. For applications with 600 features, ReHD provides 282× more energy efficiency and a 22.7× speed up as compared to the baseline.

In training, to create class hypervectors, the baseline accumulates integer hypervectors, while ReHD training accumulates binary hypervectors. Figure 10 compares the energy consumption and execution time of ReHD and the baseline during initial training. The results are reported when both designs encode and train the model in a pipeline structure. For the baseline, encoding dominates the execution time, thus the training execution hides under the encoding module. However, in ReHD, the encoding can process faster than the training, thus the training is the bottleneck of the execution time (as it is shown in Figure 10). Our evaluation shows that ReHD can provide 64.1× more energy efficiency and a 9.8× speed up as compared to the baseline during training.

Retraining/Inference Efficiency: ReHD stores all encoded hypervectors in order to perform iterative retraining. The existing HD computing algorithms map data points to integer values, where each encoded data is around 20 times larger than

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3069139, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

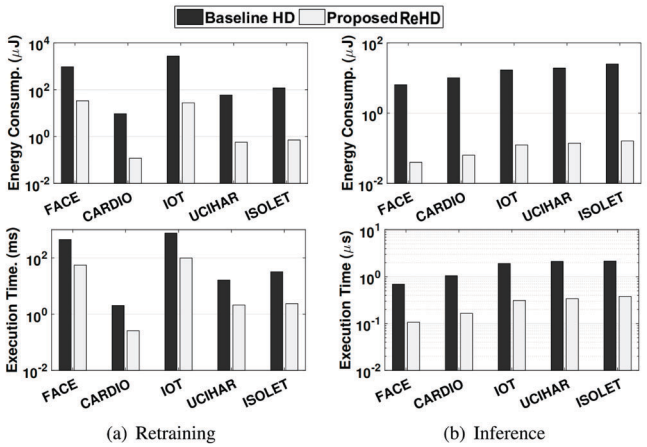

Fig. 12. Energy consumption and execution time of ReHD and the baseline HD running (a) a single retraining iteration, and (b) a single query at inference.

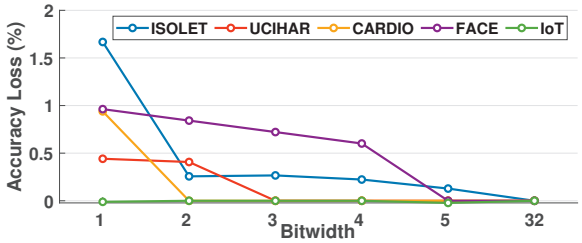

Fig. 13. Accuracy loss of ReHD utilizing n-bit model quantization.

the data in the original domain. During retraining, the FPGA needs to sequentially access the encoded values which are pre-stored on off-chip memory. The limited memory bandwidth between the off-chip memory and the FPGA BRAM blocks significantly slows down the baseline computation during retraining. In contrast, ReHD maps the training data to lower dimensions, where each dimension can be represented using a binary value. This enables ReHD to speed up the retraining by loading hypervectors faster than the baseline.

During inference and retraining, HD checks the similarity of each encoded hypervector with all existing class hypervectors. To achieve a high classification accuracy, the existing HD computing algorithms generate an integer model. Therefore, they require the use of an expensive similarity metric such as cosine to find the most similar class. In contrast, ReHD performs the similarity check with Hamming distance. Figure 12 shows the energy consumption and execution time of the FPGA accelerating a single retraining iteration and a single query during inference. The results show that ReHD can achieve on average a 61.6× energy efficiency improvement and a 7.9× speed up as compared to the existing HD computation algorithms. Similarly, in inference, the FPGA implementation of ReHD can achieve on average a 43.8× energy efficiency improvement and a 6.1× speed up running a single query (Figure 12b).

E. Model Quantization Trade-off

In Figure 13, we explore the impact of representing the HD Computing model with bit lengths ranging from 1 to 32 on quality loss. Due to significant information loss when converting to a binary hypervector, 1-bit model quantization, which computes with binary hypervectors, yielded the lowest inference accuracy. 1-bit quantization leads to an accuracy loss of up to 1.7%. However, because the 1-bit model quantization enables using Hamming distance as the similarity function, it is the most efficient quantized model. Figure 14, shows the energy improvement of n-bit model quantization over a 32-bit model. The 32-bit model uses the same encoding method as the n-bit models proposed in ReHD. The only difference is that there is no quantization during training and retraining. 1-bit model quantization results in 150× less energy consumption as compared to a 32-bit model. Therefore, 1-bit model quantization is most useful in scenarios when we primarily prioritize computational efficiency, such as on very low-resource devices. We also primarily prioritize computational efficiency when the classification task is trivial, as is the case with the IoT and UCIHAR datasets.

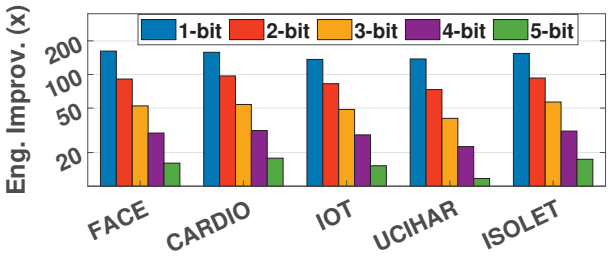

Fig. 14. Energy improvement of ReHD utilizing n-bit model quantization normalized to a 32-bit integer model.

In scenarios where resources are constrained, but high accuracy is still required, larger bitwidth model quantization is required. By allowing for less efficiency in training and inference, higher bit models allot higher inference accuracy. Using larger bit widths, hypervector dimensions take on an exponentially larger range of values, allowing for more information to be preserved. Larger bit widths yield better inference accuracy, but at the cost of less efficiently than 1-bit model quantization. This is because we have to use cosine similarity as our similarity metric, which is much more expensive than Hamming distance. Larger bitwidth model quantization is useful for datasets that are sufficiently complex that a certain number of information needs be preserved, such as for ISOLET and FACE. On ISOLET, 1-bit model quantization achieves 1.8% lower accuracy than the full 32-bit model. However, but just increasing to a 2-bit model, we are able to reduce the quality loss to 0.25% and use 93× less energy. In our experiment, models which represented hypervectors with 5 or more bits performed with comparable accuracy to models which represented hypervectors with 32 bits. Representing hypervectors with more than 5 bits is more computationally expensive, but yields no accuracy increase. Therefore, by utilizing 5-bit model quantization, we can achieve on average, 15× less energy consumption at no accuracy loss.

F. Online Dimension Reduction

Table II compares the online dimension reduction techniques of (i) computing the element-wise sum of training hy-

TABLE II
AVERAGE CHANGE IN CLASSIFICATION ACCURACY DUE TO ONLINE DIMENSION REDUCTION.

Dimension Reduction	20%	40%	60%	70%	80%	90%	95%
ABS	+0.38%	0%	-0.54%	-4.1%	-5.64%	-9.6%	-14.2%
VARIANCE	+0.4%	0%	0%	0%	-0.3%	-0.8%	-4.4%

pervectors and removing dimensions with high absolute value sums (ABS) and (ii) computing the variance across all dimensions and removing dimensions with low variance (VARIANCE). The values compute the average quality loss(accuracy drop) over the five datasets described in Section VII-A. More directly, Table II shows the impact of each dimension reduction technique on classification accuracy. When using ABS as a metric of insignificance, our results indicate that dropping 20% of "insignificant" dimensions slightly improves accuracy because we remove noisy features. As listed in Table II, dropping up to 60% of "insignificant" dimensions almost no impact on accuracy, but dropping further dimensions will lead to a decline in accuracy because we begin to drop significant dimensions.

Dropping dimensions with low intra-class differences allow for a more intelligent selection of "insignificant" dimensions than summing all training hypervectors and dropping dimensions with high absolute values. With ABS, we were able to drop 70% dimensions before we started losing significant dimensions. But since VARIANCE selects dimensions to drop more intelligently, we can drop the 90% most insignificant dimensions with only a 0.3% average loss in accuracy as a result, meaning we improve training efficiency by 90% with only a negligible decline in accuracy. The energy efficiency improves proportionally with the dropped dimensions because operations in HD are done with hypervectors. Therefore, by reducing the dimensionality of all hypervectors, all operations reduce in complexity. When we drop more than 90% dimensions, we begin to drop too many significant dimensions and lose a significant amount of accuracy.

Figure 15b shows the classification probability over an image, where yellow and blue colors indicate low and high face probability respectively. The results show that ReHD working with $D = 4,000$ dimensions can perfectly detect the faces in the image. ReHD in lower dimensionality after online dimension reduction has lower quality and detects "non-face" regions. Online dimension reduction improves ReHD efficiency linearly during both retraining and inference. For example, an 80% dimension reduction results in approximately 80% energy efficiency improvement and a $5\times$ speed up while providing less than 0.3% quality loss as compared to ReHD with full dimensionality.

VIII. CONCLUSION

In this paper, we propose ReHD, a novel HD computing framework that significantly improves the computation efficiency of HD computing. ReHD exploits the predictable memory access of our proposed encoding to design an efficient encoding approach that maps data into binary hypervectors. ReHD enables quantized training and retraining on the encoded hypervectors and simplifies the inference similarity metric. N-bit model quantization, allows us to represent our model hypervectors with n-bits where n ranges from 1 to 32, whereas previously designs chose between 1-bit or 32-bit representations. This enables more granular control over the trade-off between model classification accuracy and efficiency. We additionally implemented a dimension reduction technique that removes unnecessary dimensions to further improve the efficiency of ReHD. We also designed a fully pipelined FPGA implementation to accelerate ReHD. Our evaluations show that ReHD can achieve 64.1× and 9.8× (43.8× and 6.1×) energy efficiency and speed up as compared to the baseline during training (inference) while providing the same classification accuracy.

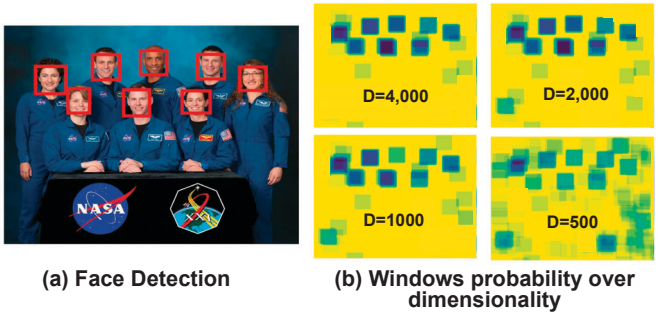

Fig. 15. Visualization of ReHD face detection accuracy over different dimensionality.

ACKNOWLEDGEMENTS

This work was supported in part by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, in part by SRC-Global Research Collaboration grant Task No. 2988.001, and also NSF grants 1527034, 1730158, 1826967, 1830331, 1911095, 2003277, and 2003279.

REFERENCES

[1] J. Gubbi *et al.*, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.
[2] M. Hassanalieragh *et al.*, "Health monitoring and management using internet-of-things (iot) sensing with cloud-based processing: Opportunities and challenges," in *IEEE SCC*, pp. 285–292, IEEE, 2015.
[3] Y. Sun *et al.*, "Internet of things and big data analytics for smart and connected communities," *IEEE Access*, vol. 4, pp. 766–773, 2016.
[4] S. Sicari *et al.*, "Security, privacy and trust in internet of things: The road ahead," *Computer networks*, vol. 76, pp. 146–164, 2015.
[5] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, 2009.
[6] O. Rasanen and J. Saarinen, "Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, no. 99, pp. 1–12, 2015.
[7] M. Imani *et al.*, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *ICRC*, pp. 1–6, IEEE, 2017.
[8] A. Rahimi *et al.*, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *ISLPED*, pp. 64–69, ACM, 2016.
[9] B. Khaleghi, M. Imani, and T. Rosing, "Prive-hd: Privacy-preserved hyperdimensional computing," *arXiv preprint arXiv:2005.06716*, 2020.
[10] M. Imani *et al.*, "Hierarchical hyperdimensional computing for energy efficient classification," in *DAC*, p. 108, ACM, 2018.
[11] Y. Kim *et al.*, "Efficient human activity recognition using hyperdimensional computing," in *IoT*, p. 38, ACM, 2018.

[12] P. Kanerva *et al.*, "Random indexing of text samples for latent semantic analysis," in *CogSci*, vol. 1036, Citeseer, 2000.

[13] M. Imani, X. Yin, J. Messerly, S. Gupta, M. Niemier, X. S. Hu, and T. Rosing, "Searchd: A memory-centric hyperdimensional computing with stochastic training," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[14] M. Imani *et al.*, "Hdcluster: An accurate clustering using brain-inspired high-dimensional computing," in *DATE*, IEEE/ACM, 2019.

[15] M. Imani *et al.*, "Hdna: Energy-efficient dna sequencing using hyperdimensional computing," in *IEEE BHI*, pp. 271–274, IEEE, 2018.

[16] M. Imani *et al.*, "A framework for collaborative learning in secure high-dimensional space," in *IEEE CLOUD*, pp. 1–6, IEEE, 2019.

[17] A. Rahimi *et al.*, "Hyperdimensional biosignal processing: A case study for emg-based hand gesture recognition," in *ICRC*, pp. 1–8, IEEE, 2016.

[18] M. Imani *et al.*, "A binary learning framework for hyperdimensional computing," in *DATE*, IEEE/ACM, 2019.

[19] M. Imani *et al.*, "Sparsehd: Algorithm-hardware co-optimization for efficient high-dimensional computing," in *IEEE FCCM*, pp. 1–6, IEEE, 2019.

[20] J. Morris, M. Imani, S. Bosch, A. Thomas, H. Shu, and T. Rosing, "Comphd: Efficient hyperdimensional computing using model compression," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2019.

[21] M. Imani, S. Bosch, S. Datta, S. Ramakrishna, S. Salamat, J. M. Rabaey, and T. Rosing, "Quanthd: A quantization framework for hyperdimensional computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[22] Y.-C. Chuang, C.-Y. Chang, and A.-Y. A. Wu, "Dynamic hyperdimensional computing for improving accuracy-energy efficiency trade-offs," in *2020 IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 1–5, IEEE, 2020.

[23] T. Wu *et al.*, "Brain-inspired computing exploiting carbon nanotube fets and resistive ram: Hyperdimensional computing case study," in *IEEE ISSCC*, IEEE, 2018.

[24] H. Li *et al.*, "Hyperdimensional computing with 3d vrram in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition," in *IEDM*, pp. 16–1, IEEE, 2016.

[25] S. Gupta *et al.*, "Felix: Fast and energy-efficient logic in memory," in *IEEE/ACM ICCAD*, pp. 1–7, IEEE, 2018.

[26] M. Imani *et al.*, "Fach: Fpga-based acceleration of hyperdimensional computing by reducing computational complexity," in *ASPDAC*, pp. 493–498, ACM, 2019.

[27] S. Salamat *et al.*, "F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing," in *FPGA*, pp. 53–62, ACM, 2019.

[28] M. Imani *et al.*, "Exploring hyperdimensional associative memory," in *HPCA*, pp. 445–456, IEEE, 2017.

[29] G. Karunaratne, M. Le Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian, "In-memory hyperdimensional computing," *Nature Electronics*, pp. 1–11, 2020.

[30] S. Salamat, M. Imani, and T. Rosing, "Accelerating hyperdimensional computing on fpgas by exploiting computational reuse," *IEEE Transactions on Computers*, 2020.

[31] M. Imani, S. Salamat, B. Khaleghi, M. Samragh, F. Koushanfar, and T. Rosing, "Sparsehd: Algorithm-hardware co-optimization for efficient high-dimensional computing," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 190–198, IEEE, 2019.

[32] M. Schmuck *et al.*, "Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory," *arXiv preprint arXiv:1807.08583*, 2018.

[33] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[34] B. Lesser, M. Mücke, and W. N. Gansterer, "Effects of reduced precision on floating-point svm classification accuracy," *Procedia Computer Science*, vol. 4, pp. 508–517, 2011.

[35] M. Wess, S. M. P. Dinakarrao, and A. Jantsch, "Weighted quantization-regularization in dnns for weight memory minimization toward hw implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2929–2939, 2018.

[36] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 267–278, IEEE Press, 2016.

[37] Y. Zhou, S.-M. Moosavi-Dezfooli, N.-M. Cheung, and P. Frossard, "Adaptive quantization for deep neural network," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[38] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8612–8620, 2019.

[39] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, P. Chuang, and L. Chang, "Compensated-dnn: Energy efficient low-precision deep neural networks by compensating quantization errors," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[40] S. Ye, T. Zhang, K. Zhang, J. Li, J. Xie, Y. Liang, S. Liu, X. Lin, and Y. Wang, "A unified framework of dnn weight pruning and weight clustering/quantization using admm," *arXiv preprint arXiv:1811.01907*, 2018.

[41] T. I. Cannings *et al.*, "Random-projection ensemble classification," *Journal of the Royal Statistical Society*, vol. 79, no. 4, pp. 959–1035, 2017.

[42] M. Imani, J. Morris, S. Bosch, H. Shu, G. De Micheli, and T. Rosing, "Adapthd: Adaptive efficient training for brain-inspired hyperdimensional computing," in *2019 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pp. 1–4, IEEE, 2019.

[43] M. Imani, J. Morris, J. Messerly, H. Shu, Y. Deng, and T. Rosing, "Bric: Locality-based encoding for energy-efficient brain-inspired hyperdimensional computing," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[44] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.

[45] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/ISOLET.

[46] "Uci learning repository." https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Acti

[47] G. Griffin, A. Holub, and P. Perona, "Caltech-256 object category dataset," 2007.

[48] "Uci machine learning repository." https://archive.ics.uci.edu/ml/datasets/cardiotocography.

[49] "Uci machine learning repository." https://archive.ics.uci.edu/ml/datasets/detection_of_IoT_botnet_attacks_N_BaIoT.