Algorithm-Hardware Co-Design for Efficient Brain-Inspired Hyperdimensional Learning on Edge

Yang Ni[1], Yeseong Kim[3⋆], Tajana Rosing[2] and Mohsen Imani[1⋆]

[1]University of California Irvine, [2]University of California San Diego, [3]Daegu Gyeongbuk Institute of Science and Technology

⋆Corresponding authors: yeseongkim@dgist.ac.kr; m.imani@uci.edu

***Abstract*—Machine learning methods have been widely utilized to provide high quality for many cognitive tasks. Running sophisticated learning tasks requires high computational costs to process a large amount of learning data. Brain-inspired Hyperdimensional Computing (HDC) is introduced as an alternative solution for lightweight learning on edge devices. However, HDC models still rely on accelerators to ensure real-time and efficient learning. These hardware designs are not commercially available and need a relatively long period to synthesize and fabricate after deriving the new applications. In this paper, we propose an efficient framework for accelerating the HDC at the edge by fully utilizing the available computing power. We optimize the HDC through algorithm-hardware co-design of the host CPU and existing low-power machine learning accelerators, such as Edge TPU. We interpret the lightweight HDC learning model as a hyper-wide neural network to take advantage of the accelerator and machine learning platform. We further improve the runtime cost of training by employing a bootstrap aggregating algorithm called bagging while maintaining the learning quality. We evaluate the performance of the proposed framework with several applications. Joint experiments on mobile CPU and the Edge TPU show that our framework achieves 4.5× faster training and 4.2× faster inference compared to the baseline platform. In addition, our framework achieves 19.4× faster training and 8.9× faster inference as compared to embedded ARM CPU, Raspberry Pi, that consumes similar power consumption.**

I. Introduction

With the emergence of the Internet of Things (IoT), many applications run machine learning algorithms to perform learning and cognitive tasks [1]. Examples are smart homes, smart cities, smart manufacturing, and smart transportation. Today's systems typically rely on sending all the data to the cloud to complete learning and training, which leads to a significant communication cost. This communication cost can be eliminated by scaling the learning tasks in a distributed fashion where different devices collect data. Edge computing tries to realize such a distributed computing paradigm by bringing the computation closer to the location where the data are generated [2], [3]. A mainstream of the research is *federated learning* [4] that trains a central model over multiple devices. However, these techniques use complex algorithms, e.g., Deep Neural Networks (DNNs), which require billions of parameters and many hours to train in a powerful and reliable computing environment [5]–[7]. Considering memory and resource limitations of embedded devices on edge, which also have potential issues of network noises and hardware failure due to the unstable nature of IoT systems, current computing environments are still far from real-time learning [8].

To better deploy deep learning applications to the edge environment, recent efforts have focused on the energy-efficient accelerator designs, e.g., Eyeriss [9] and UNPU [10]. Google also releases its standardized DNN accelerator at the edge, i.e., the Edge TPU. However, they mainly target DNN inference and thereby cannot handle the dynamics of many IoT practices, which require model updates frequently to follow the rapidly changing inputs. DNN training, however, heavily relies on more powerful platforms like cloud servers. Thus, a lightweight model design is needed to enable training at the edge.

In contrast to existing machine learning algorithms, the human brain can train effortlessly and efficiently without much concern of noisy and broken neuron cells [11]. To more closely model the human brain, earlier researchers proposed HyperDimensional Computing (HDC) as an alternative computing method, which mimics important brain functionalities towards high-efficiency and noise-tolerant computation [12], [13]. HDC is motivated by the observation that the human brain operates on high-dimensional representations of data. In HDC, objects and data are thereby encoded with high-dimensional vectors, called *hypervectors*, which have 10,000 or more elements. HDC can then perform various learning tasks with computation in the high dimensional space. HDC is well suited to address learning on edge systems as HDC models are computationally efficient to train [14]–[17], offer intuitive and human-interpretablity [18], provide strong robustness to noise [19], [20], and support lightweight privacy [21], [22]. These features make HDC a promising solution for today's embedded devices with limited storage, battery, and resources, as well as embedded devices that often depend on the unreliable source of battery.

To exploit the sparsity and orthogonality of high-dimensional space, HDC operations are defined over long vectors with thousands of elements, called hypervectors. Therefore, HDC requires many number of multiplication and addition operations, which are significantly costly for today's low-power platforms. To ensure fast HDC operation, most prior works rely on ASIC or emerging hardware acceleration. However, these hardware designs are not commercially available and need a relatively long period to synthesize and fabricate after deriving the new applications. As such, to ease the deployment of HDC in the real world, we need a framework solution to run HDC on a highly parallel but general-purpose platform.

In this paper, we deploy HDC by exploiting and reusing readily available and standardized DNN accelerators, which are already in mass production. Particularly, we choose Google Edge TPU since it works closely with one of the most widely-used machine learning frameworks, i.e., TensorFlow (TF) [23]. We propose a framework for efficient acceleration of the HDC in the edge environment by optimizing its algorithm to fully utilize the low-power Edge TPU as well as the host CPU. The main contributions of the paper are listed as follows:

- We propose a framework to efficiently perform HDC by optimizing its algorithm via co-design of CPU and TPU procedures at the edge. We interpret the lightweight HDC learning model as a hyper-wide neural network to take advantage of the accelerator and machine learning platform. Our framework maps the HDC model to TensorFlow that works seamlessly with the Edge TPU. The bulky matrix operations in HDC efficiently utilize the hardware parallelism. We also explore how dataset parameters, e.g., the number of input features, influence the performance of our framework on the Edge TPU.
- We further accelerate the training runtime on the host CPU by employing the ensemble methods called bagging. We optimize the HDC training algorithm towards the host CPU since training class hypervectors, i.e., weight updates, is not supported by edge accelerators. The optimized algorithm accelerates the HDC training runtime on the CPU by reducing the overall computation costs. We present a training method employing the bootstrap aggregating algorithm, i.e., bagging. It utilizes multiple learners for a consensus-based prediction, which relaxes requirements for the quality of each learner. Using the ensemble method, we can reduce the training iterations and the hypervector width for each model, which leads to faster HDC learning. More importantly, our

978-3-9819263-6-1/DATE22/©2022 EDAA

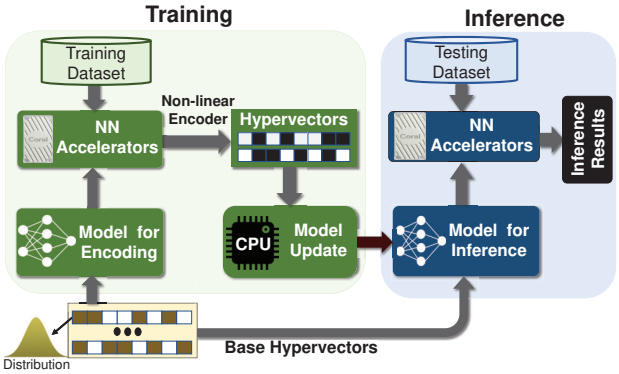

Fig. 1: HDC training and inference procedures with acceleration provided by our framework

method still achieves high-quality results with similar inference accuracy.

We evaluate the performance of our proposed framework using five large-scale datasets. Our method significantly improves the training/inference runtime of HDC on low-power edge platforms. We further improve the runtime cost of training by employing a bootstrap aggregating algorithm called bagging while maintaining the learning quality. We evaluate the performance of the proposed framework with several applications. Joint experiments on mobile CPU and the Edge TPU show that our framework achieves 4.5× faster training and 4.2× faster inference compared to the baseline platform. In addition, our framework achieves 19.4× faster training and 8.9× faster inference as compared to embedded ARM CPU, Raspberry Pi, that consumes similar power consumption. We will provide a fully open source library of our framework based on TensorFlow.

II. Related Work

HDC, inspired by the large neural circuits inside the human brain, is both efficient in the calculation and robust against noise. Recent researchers have shown those advantages in multiple HDC applications, e.g., gesture/object detection [24], [25], DNA pattern matching [26], [27], regression [28], and manufacturing [29], and clustering [30]. Recently, an increasing number of researchers focused on the acceleration of HDC utilizing the parallelism of its hardware-friendly operations. Multiple hardware platforms have been used to accelerate the training and inference process. However, this simulated design is not available as a physical chip due to the high manufacturing cost. In this paper, we present an HDC acceleration framework based on a readily available low-power platform, i.e., Google Edge TPU accelerator.

Edge TPU is one of many accelerators that focus on low-power, highly efficient computation to satisfy the increasing need for computation in the edge environment. The computation power of the Edge TPU lies in its relatively large matrix multiplication unit (MXU), which adopts a spacial architecture called the systolic array. This architecture efficiently reuses all the inputs by pumping them through each processing element [31]. For standardized accelerators like the Edge TPU, Intel also released their own Movidius VPU [32]. Another example is Eyeriss [9], an ASIC-based experimental accelerator that utilizes a spatial architecture for minimal data movements with an energy-efficient dataflow. All of them target low-power scenarios and support several wide-used machine learning frameworks. Most of the accelerators at the edge target traditional NN tasks. For example, the Edge TPU only officially supports NN inference; however, in our work, we accelerate the HDC using a co-design approach over both Edge TPU and host CPU.

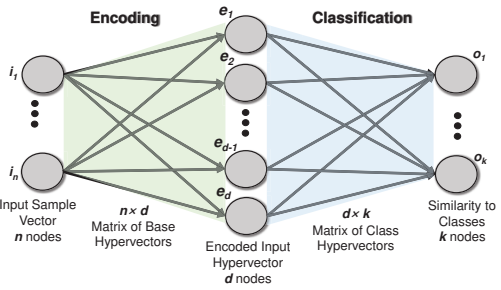

Fig. 2: Interpretation of HDC as a wide NN: base hypervectors and class hypervectors map to hidden layer weights

III. Hyperdimensional Computing at the Edge

Figure 1 shows an overview of our proposed framework for accelerating HDC training and inference on Edge TPU. During both training and inference tasks, our key aim is to map the HDC operations to valid DNN-like models such that they can be accelerated by low-power Edge TPU. For training, base hypervectors are generated randomly using the normal distribution. Then, our framework takes samples from the training dataset and encodes them on Edge TPU. These encoded hypervectors are sent to the host CPU for class hypervectors update. For inference, our framework is based on an inference neural network model with its parameters determined by trained class hypervectors and base hypervectors. The model is loaded to the accelerator and takes samples from the testing dataset. Our framework completely maps the inference process to Edge TPU, enabling real-time and efficient prediction.

A. Mapping the hyperdimensional computing to Edge TPU

Figure 2 shows how three major operations in HDC, i.e., input vectors encoding, class hypervectors update, and classification, are mapped to a three-layer wide neural network. First, the three-layer network is sliced in half. The first part of the network includes the input layer with n nodes and the wide hidden layer with d nodes, and it maps the inputs to higher dimensions. The second part of the network takes the hidden layer as inputs and generates the classification results at the output layer with k nodes.

Encoding: As the basis of HDC, encoding maps the input space to higher dimensions. Suppose an n-feature input sample vector has the form $\vec{F} = \{f_1, f_2, \ldots, f_n\}$, and each component stands for a single feature value. At present, this vector is in relatively lower dimensions, and the information stores largely in those values rather than the patterns, which is not ideal for HDC operations later. To achieve better results, the inputs are mapped to hypervector with the width $d = 10,000$. Most prior works have tried to encode the input using linear mapping [21]. However, in this work, we adopt a non-linear mapping which achieves higher learning accuracy. This encoding maps linearly inseparable data to a higher dimension for possible linear separation.

The mapping relies on randomly generated $1 \times d$ base hypervectors $\{\vec{\mathbf{B}}_1, \vec{\mathbf{B}}_2, \ldots, \vec{\mathbf{B}}_n\}$ for each input feature, and the randomness is realized through the normal distribution for components in these hypervectors, i.e., $b \sim \mathcal{N}(\mu, \sigma^2)$ with $\mu = 0$ and $\sigma = 1$. The components of these random hypervectors follow a symmetric distribution around zero so that the dot product between any two base hypervectors is very close to zero. Thus, we also regard them as near orthogonal. Then these hypervectors multiply with corresponding feature values, and the outputs are aggregated as $1 \times d$ encoded hypervectors:

$$\vec{\mathbf{E}} = \tanh(f_1 \times \vec{\mathbf{B}}_1 + f_2 \times \vec{\mathbf{B}}_2 + \ldots + f_n \times \vec{\mathbf{B}}_n)$$

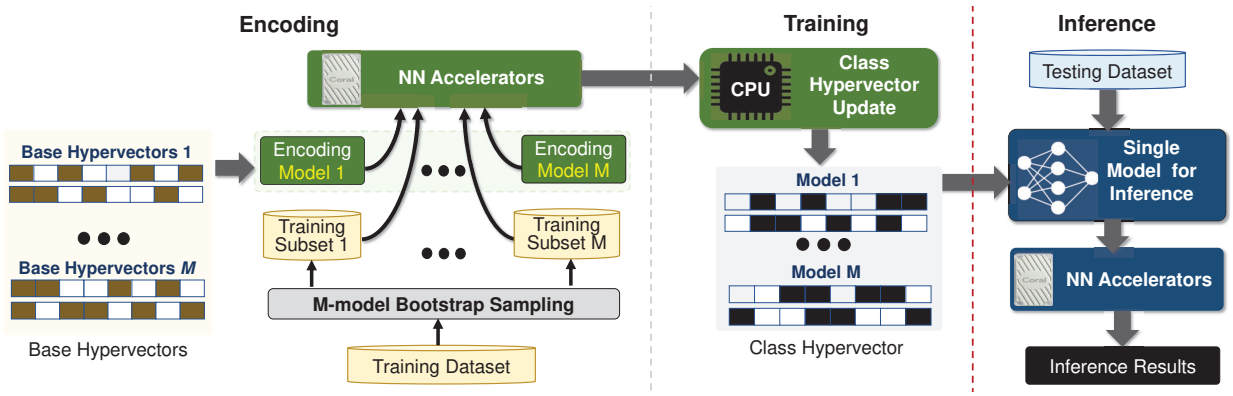

Fig. 3: HDC acceleration framework with multi-model bootstrap sampling (Bagging) for runtime reduction

We refer to this hypervector addition as the bundling operation, which preserves the information of each hypervector. Especially for non-linear encoding, we take the hyperbolic tangent value of each component in the encoded hypervector.

To map the encoding process to Edge TPU, we notice that bundling in the encoding process is essentially a large number of MAC operations. In other words, the encoding is indeed a vector-matrix multiplication that is ready to accelerate on most hardware accelerators. We map these MAC operations to the first half of that three-layer wide NN. Within that half, the input is the $1 \times n$ sample vector, and all the base hypervectors form the $n \times d$ weight matrix for edges connecting the input layer and the hidden layer. The non-linear term can be integrated into the NN as an activation function of nodes in the wide hidden layer. Particularly for Edge TPU, these two fully connected layers will be used to generate a TPU TensorFlow Lite (TFlite) model for encoding acceleration. The output from this model is the encoded hypervector samples.

Class hypervectors update: The process for generating and updating k class hypervectors uses lightweight operations, i.e., bundling and its reverse called detaching. Starting with all zeros in the $1 \times d$ class hypervectors $\{\vec{\mathbf{C}}_1, \vec{\mathbf{C}}_2, \cdots, \vec{\mathbf{C}}_k\}$, these hypervectors are updated according to the classification correctness of each input. For instance, if one encoded hypervector $\vec{\mathbf{E}}_m$ that belongs to class a, is instead classified to b incorrectly. Then the HDC training algorithm will update the class hypervectors for both class a and b:

$$Bundling: \vec{\mathbf{C}}_a = \vec{\mathbf{C}}_a + \lambda \vec{\mathbf{E}}_m$$
$$Detaching: \vec{\mathbf{C}}_b = \vec{\mathbf{C}}_b - \lambda \vec{\mathbf{E}}_m$$

λ is the learning rate.

Class hypervector training is interpreted as weights update[1] for the classification part of the neural network, which we will mention in the next paragraph. Most edge accelerators are not designed for training or weights update. For example, Edge TPU lacks the support for element-wise operations, so the acceleration for class hypervectors update is not available. In our framework, we deploy this part on the host CPU due to this limitation. However, the ensemble method introduced in Section B will help reduce the training computation cost on the CPU and thus achieve a faster training speed.

Classification: In the final classification step of the HDC, the associative search checks the similarity between encoded query hypervectors and class hypervectors. A common classification method is to calculate the cosine similarity between two hypervectors:

$$\delta(\vec{\mathbf{E}}_m, \vec{\mathbf{C}}_k) = \frac{\vec{\mathbf{E}}_m \cdot \vec{\mathbf{C}}_k}{\|\vec{\mathbf{E}}_m\| \|\vec{\mathbf{C}}_k\|}$$

[1]In this paper, we interchangeably refer to class hypervector training as the weights update process in Edge TPU.

We approximate the similarity using the dot product: $\delta(\vec{\mathbf{E}}_m, \vec{\mathbf{C}}_k) = \vec{\mathbf{E}}_m \cdot \vec{\mathbf{C}}_k$ to accelerate it on Edge TPU. After repeating this for each class hypervector, the final result is the class with the highest similarity.

Similarity check maps to the second half of that three-layer fully connected network. The input to this half is the encoded hypervectors coming out of the wide hidden layer, and the network parameters, i.e., the $d \times k$ weight matrix, are determined by the trained class hypervectors. Through the network, the output is the sum of product between hidden node outputs and the weight on edge, which is the same calculation of the similarity check.

B. Boost the Efficiency with Bagging

As mentioned in Section III-A, a large portion of the training process is not accelerated due to the limitation of Edge TPU, which has no support for on-device weights update. Low-power host machines at the edge also lack the computing power for fast training with wide hypervectors. This leads to our idea behind the use of the bagging method, which is to achieve more efficient training without compromising the HDC learning quality. We take advantage of this meta-learning method to reduce the overall computation cost of weights update. Figure 3 is an overview of our framework with the bagging method, which includes the training and inference process. Comparing to the training without bagging, we need multiple groups of base hypervectors to generate the sub-models, which take inputs from different training subsets. These subsets are sampled from the original training dataset through bootstrap sampling. Each subset and sub-model go through the encoding process before the class hypervector training in the host CPU. For the inference, a single inference model is constructed using all the base hypervectors and trained class hypervectors. In the following, we will discuss the details of our proposed method.

Bagging for faster HDC training: As one of the ensemble methods, bagging [33] aggregates multiple weak learners or models and takes an average of them to produce a more stable model for unstable procedures, e.g., NN and classification. Usually, the training of a prediction model is based on a single learning dataset. However, the bagging method aims at using weak models based on multiple sub-datasets to provide better predictions. To generate multiple datasets for training different models, bagging relies on bootstrap feature sampling and dataset sampling. Feature sampling randomly chooses features of each training sample, and dataset sampling randomly chooses samples from the original dataset to form new training subsets.

Bagging and other ensemble methods provide higher accuracy with the cost of longer training runtime because of the need for training multiple sub-models. Thus, an unmodified implementation of the bagging method is not ideal for efficient training in HDC. However, we interpret its accuracy advantage from another angle. Because of

the aggregation and the consensus-based prediction process, bagging is able to improve the accuracy of each sub-model. In other words, multiple smaller-sized models with fewer training iterations can still provide similar accuracy comparing with fully-trained models. This outcome aligns with our targets, i.e., lower runtime cost of weights update. Our experimental results in Section IV show that the bagging method indeed achieves our aims.

The cost of weights update is directly related to the class hypervector width. However, training multiple sub-models significantly increases the computation cost if the hypervector width stays unchanged. Thus, we decrease the hypervector width to $d' = d/M$, where d is the original hypervector width and M is the number of sub-models generated. We choose this relationship between sub-model and full model dimensions mainly for a fairer comparison because it allows us to generate an inference model of similar size after the training. Next, we further reduce the runtime through fewer training iterations. We utilize the property of bagging that it does not require fully-trained sub-models. The bagging method achieves similar accuracy with fewer than half iterations, which dramatically lowers the runtime. Then through the bootstrap sampling in bagging, we also speed up the training process. Both dataset and feature sampling generate a training subset, which means less computation cost.

For weights update, we achieve a smaller computation cost C' comparing to the original computation cost C. We estimate the new computation cost by:

$$C' = C \times M \times \frac{d'}{d} \times \frac{I'}{I} \times \alpha \times \beta$$

I and I' are the original and reduced training iterations, α and β are the dataset and feature sampling ratios.

Inference model generation: Even though we utilize the bagging method mainly for optimizing the training runtime on the host CPU, the inference process on Edge TPU also needs to adapt accordingly. As the bagging method trained multiple sub-models, which also take inputs from different sub-datasets, accelerating these models on Edge TPU in series is not efficient. Most Edge TPU only take one model at a time, and the weights have to be loaded to the on-chip buffer every time. This brings the overhead for preparing the accelerators for multiple models. Also, these sub-models trained through bagging are smaller, so running them in series may not fully utilize accelerator hardware parallelism. Thus, we design a technique to combine multiple trained sub-models as a single full-sized inference model.

For an n-feature input $\vec{F}$, it first passes through the encoding process, where its features are sampled and then different groups of base hypervectors encode it. Suppose the bagging process uses M different sub-models, and each of them is based on an $n \times d'$ matrix of base hypervectors $\boldsymbol{\mathcal{B}}^m = \{\vec{\mathbf{B}}_1^m, \vec{\mathbf{B}}_2^m, \cdots, \vec{\mathbf{B}}_n^m\}$. For this matrix of each sub-model, some of the columns are set to zero, because they correspond to features that are not sampled. In this way, the feature sampling process is automatically finished. Then we stack these matrices horizontally to form a full $n \times d$ weight matrix $\mathbb{B}$ for the encoding part of the inference model. The output $1 \times d$ encoded hypervector $\vec{\mathbf{E}}$ is calculated as below:

$$\vec{\mathbf{E}} = \vec{F} \times \mathbb{B} = \vec{F} \times [\boldsymbol{\mathcal{B}}^1 \ \boldsymbol{\mathcal{B}}^2 \ \ldots \ \boldsymbol{\mathcal{B}}^M]$$

Then these encoded hypervectors enter the second half of the inference model for classification of k different classes. For the different groups of class hypervectors trained through the bagging, we also stack them vertically to form a bigger $d \times k$ matrix $\mathbb{C}$ with the same dimension as before:

$$\vec{\mathbf{O}} = \vec{\mathbf{E}} \times \mathbb{C} = \vec{\mathbf{E}} \times [\boldsymbol{\mathcal{C}}^1 \ \boldsymbol{\mathcal{C}}^2 \ \cdots \ \boldsymbol{\mathcal{C}}^M]^T$$

where the $d' \times k$ matrix $\boldsymbol{\mathcal{C}}^m = \{\vec{\mathbf{C}}_1^m, \vec{\mathbf{C}}_2^m, \cdots, \vec{\mathbf{C}}_k^m\}$. Instead of aggregating the results of similarity checks for each sub-model, i.e., multiple vector-matrix multiplications followed by an element-wise addition, we could perform a single time vector-matrix multiplication and the classification result is available directly from the output:

$$Class\ Prediction = \underset{1 \le i \le k}{\arg\max}(\vec{\mathbf{O}}_i)$$

Because the full inference NN model generated here has the same dimensions as the one generated without using the bagging method, we make the inference process free of extra overhead.

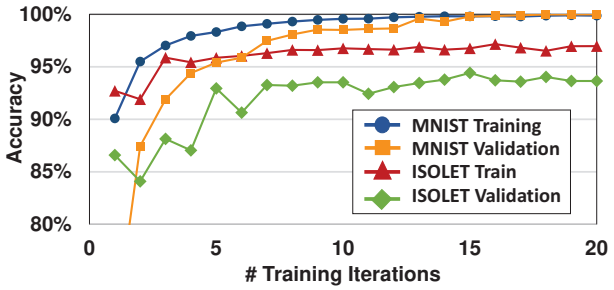

Fig. 4: Training and validation accuracy for CPU experiments

TABLE I: Details of the datasets used for experiments

Datasets	# Samples	# Features	# Classes	Descriptions
FACE [34]	80854	608	2	Facial images
ISOLET [35]	7797	617	26	Speech Data
UCIHAR [36]	7667	561	12	Human Activity Logs
MNIST [37]	60000	784	10	Handwritten Digits
PAMAP2 [38]	32768	27	5	Human Activity Logs

IV. Experimental Results

A. Experimental setup

We implement our framework on the Google Edge TPU accelerator, which is connected through USB 3.0 to a lower-end laptop. The laptop has a mobile Intel CPU i5-5250U. We choose the Edge TPU for its better compatibility with TensorFlow. All the experiments are based on TensorFlow tflite_runtime version 2.1.0. We first run different learning tasks using HDC only on the host machine CPU as the baseline for our framework. Then we perform the same tasks using our proposed framework, which includes the acceleration of the Edge TPU. In the following sections, we present three groups of experimental results for different settings, i.e., the CPU baseline, our proposed framework without bagging (the TPU baseline) and with bagging. We compare and analyze the results from two aspects: accuracy and runtime costs. We also present the parameter search process for our choices of bagging parameters. At last, we discuss the influence of input feature sizes on the performance of our framework.

For the CPU baseline experiments and those on TPU without bagging, we train the model for 20 iterations to achieve fully trained models as in Figure 4. For experiments on TPU with bagging, we trained 4 sub-models with hypervector width $d = 2500$ for 6 iterations. We choose the dataset sampling ratio as 0.6, i.e., using 60% of the training dataset for each sub-model, and the feature sampling ratio is disabled. Our choice is based on a simple parameter search to balance accuracy and runtime, and it is applied to all datasets that may not be optimal for every dataset we test. For the learning tasks of HDC, in Table I, we choose five different datasets of practical applications.

B. Training Efficiency

In Figure 5, we include the runtime measurement for each component of the training process in HDC. Our framework without bagging, i.e., the TPU baseline, maps the training set encoding from the host CPU to Edge TPU. Comparing with the CPU baseline, we observe that the encoding time, with the acceleration of Edge TPU,

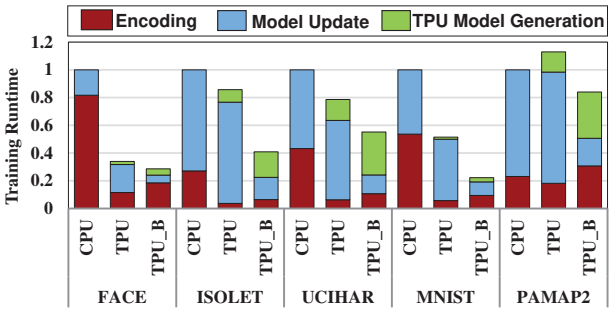

Fig. 5: Training runtime cost comparison of three different framework settings: CPU baseline, TPU (without bagging method) and TPU_B (with bagging method). The runtime costs are normalized to the measurement on CPU within each dataset.

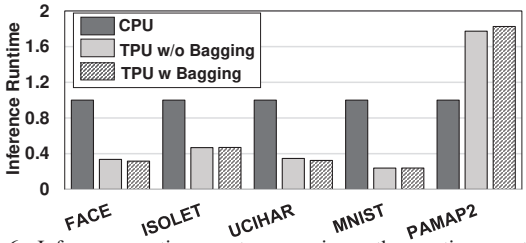

Fig. 6: Inference runtime cost comparison: the runtime costs are normalized to the measurement on CPU within each dataset.

significantly decreases for 4 out of 5 datasets except for PAMAP2. The maximum encoding runtime speedup is $9.37\times$ for the MNIST dataset. For datasets such as FACE, the encoding runtime takes up a large portion of the total training time, and our framework with the Edge TPU significantly reduces the overall runtime with $2.95\times$ speedup. Our framework with bagging, i.e., TPU_B in the figure, achieves dramatically lower runtime for class hypervector update in host CPU with the optimized HDC training algorithm. For example, compared to the CPU baseline, the bagging method brings a maximum $4.74\times$ speedup for the hypervector update process in the host CPU. Comparing with the overall training runtime in CPU baseline, our efficient framework brings a maximum $4.49\times$ speedup on the MNIST dataset by optimizing operations on both the Edge TPU and host CPU. It also achieves significantly faster training on FACE, ISOLET, and UCIHAR datasets with $3.49\times$, $2.45\times$ and $1.81\times$ speedup, respectively. In the figure, we include the runtime cost of generating Edge TPU models on the host CPU. It includes the runtime for generating TFlite model files and compiling those files for Edge TPU. The cost of transferring models onto the accelerator is negligible. Thus, these extra runtime for preparing TPU models are one-time costs, and we do not include them for the inference runtime later.

C. Inference Efficiency and Accuracy

Figure 6 shows the runtime measurements for the inference process of HDC on the Edge TPU. Besides our counterexample PAMAP2, our framework significantly accelerates the inference process. For example, the maximum inference speedup achieved on MNIST with the bagging method is $4.19\times$. The speedups for other 3 datasets are: $3.16\times$(FACE), $2.13\times$(ISOLET), $3.08\times$(UCIHAR). The bagging method, with a unified inference model, can achieve the inference runtime with no extra overhead compared with the TPU baseline.

Figure 7 summarizes the accuracy results of our experiments on five different datasets. The comparison illustrates that our proposed method is able to achieve similar inference accuracy on the Edge TPU. Because the ensemble method compensates for the possible

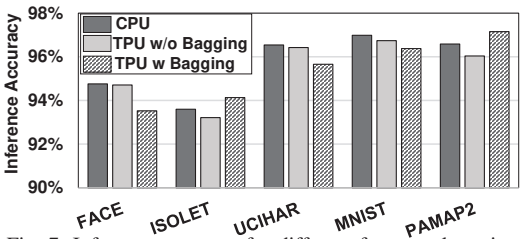

Fig. 7: Inference accuracy for different framework settings

TABLE II: Our Edge TPU-based efficiency vs. Raspberry Pi 3.

	FACE	ISOLET	UCIHAR	MNIST	PAMAP2
Training	21.5 ×	15.6×	17.9×	23.6×	18.6×
Inference	11.4×	7.2×	7.9×	11.1×	6.8×

incorrect classification of each sub-model, the final inference model even achieves higher accuracy than the full-sized, fully-trained model for some datasets, e.g., ISOLET and PAMAP2.

We also compare our Edge TPU-based platform with an embedded CPU which consumes similar average power consumption. Table II shows the performance improvement of our framework as compared to Raspberry Pi 3 using ARM Cortex A53 processor. Our results indicate that our framework can provide $19.4\times$ and $8.9\times$ faster training and inference compared to Raspberry Pi 3.

D. Parameter Search for Bagging

Figure 8 explains how we choose the sampling ratios for the bagging process with a parameter search on ISOLET dataset. We recall the accuracy changes over epochs in Figure 4, and temporally select 6 training iterations for searching the ratio parameters so that the learners in bagging are relatively weak and do not consume a long time for training. Focusing on the training runtime, we observe that the dataset sampling ratio has a significant influence on runtime for both the encoding and the class hypervector update. Thus, we choose $\alpha = 0.6$ because it only needs about 70% of the training time comparing to using the full dataset, and the accuracy does not decrease. We also find that the feature sampling ratio does not provide the ideal runtime reduction, rather compromises the accuracy when the feature sampling ratio reaches $\beta = 0.6$. Thus, the feature sampling is disabled in our following experiments. In Figure 9, we present the change of accuracy and runtime for ISOLET with 3 to 8 training iterations, and we fix the sampling ratios to our choices before. This parameter only affects the runtime for class hypervector update in the host CPU. In the figure, 4 to 6 training iterations can save around 20% of runtime comparing to that of 8 iterations with similar accuracy. Based on Figure 4 and 9, we continue using 6 iterations in our experiments on all datasets.

E. Encoding Scalability with Number of Input Feature

In Figure 5 and Figure 6, it is obvious that the PAMAP2 dataset does not perform well on our framework. From a simple analysis of these figures with information from Table I, we notice that the PAMAP2 dataset has a significantly smaller number of features in its samples. The ratio of feature numbers between the PAMAP2 and MNIST dataset is only 3.4%. This shows that the number of input features of a certain dataset will influence the performance.

We further investigate this by calculating the encoding runtime speedup over the CPU baseline after mapping the encoding process to Edge TPU. We construct a few synthetic datasets with a different number of input features changing from 20 to 700. In Figure 10, we observe that the speedup increases when the feature number is larger. For example, our framework achieves an $8.25\times$ speedup for the encoding runtime when the input sample has 700 features. However, the speedup decreases to $1.06\times$ when the feature number is 20. This observation shows that our framework is able to provide high

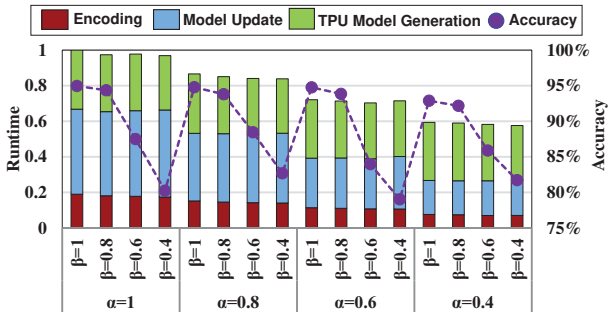

Fig. 8: Inference accuracy and training runtime of different ratios: the runtime costs are normalized to the value with $\alpha = 1$, $\beta = 1$ (α: Dataset Sampling Ratio, β: Feature Sampling Ratio)

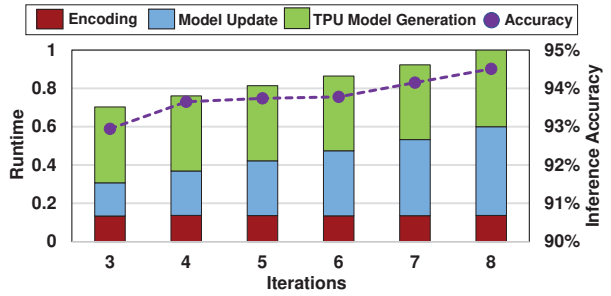

Fig. 9: Inference accuracy and training runtime of different iterations: the runtime costs are normalized to the value for 8 iterations

encoding runtime speedup for datasets with large inputs. Therefore, the PAMAP2 dataset with fewer features in inputs is not suitable for acceleration on the Edge TPU. In addition, sampling a subset of features for bagging settings is not an ideal design choice for our proposed framework since we hardly obtain the appropriate runtime and accuracy trade-off.

V. Conclusion

In this paper, we propose a framework for efficient acceleration of HDC in the edge environment by optimizing its algorithm to fully utilize the low-power Edge TPU as well as the host CPU. We interpret the lightweight HDC learning model as a hyper-wide NN to take advantage of the accelerator and machine learning platform. We further accelerate the training runtime on the host CPU by employing the ensemble methods called bagging. We test our framework on the Google Edge TPU in the low-power scenario. Joint experiments on mobile CPU and the Edge TPU show that our framework achieves faster training and inference compared to the baseline platform as well as embedded CPU platforms.

Acknowledgments

This work was supported in part by National Science Foundation (NSF) #2127780, #2112167, #2052809, #2003279 and Semiconductor Research Corporation (SRC) Task #2988.001, SRC CRISP, and Department of the Navy, Office of Naval Research, grants #N00014-21-1-2225 and #N00014-22-1-2067, and a generous gift from Cisco. Yeseong Kim was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (NRF-2018R1A5A1060031).

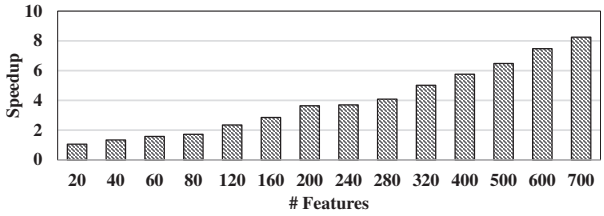

Fig. 10: Encoding runtime speedup of our framework on the TPU compared to the CPU baseline for different number of features

References

[1] N. Dall'Ora, S. Centomo, and F. Fummi, "Industrial-iot data analysis exploiting electronic design automation techniques," in *IWASI*. IEEE, 2019, pp. 103–109.
[2] A. Marchisio *et al.*, "Deep learning for edge computing: Current trends, cross-layer optimizations, and open research challenges," in *ISVLSI*. IEEE, 2019, pp. 553–559.
[3] S. Wang *et al.*, "When edge meets learning: Adaptive control for resource-constrained distributed machine learning," in *INFOCOM*. IEEE, 2018, pp. 63–71.
[4] L. K. Kalyanam *et al.*, "A distributed framework for real time object detection at low frame rates with iot edge nodes," in *ISES*. IEEE, 2020, pp. 285–290.
[5] J. Pan and J. McElhannon, "Future edge cloud and edge computing for internet of things applications," *IEEE IOT*, vol. 5, no. 1, pp. 439–449, 2017.
[6] H. Li, K. Ota, and M. Dong, "Learning iot in edge: Deep learning for the internet of things with edge computing," *IEEE network*, vol. 32, no. 1, pp. 96–101, 2018.
[7] M. Shafique *et al.*, "Robust machine learning systems: Challenges, current trends, perspectives, and the road ahead," *IEEE Design & Test*, 2020.
[8] S. K. Ram *et al.*, "Sehs: Solar energy harvesting system for iot edge node devices," in *Progress in Advanced Computing and Intelligent Engineering*. Springer, 2021, pp. 432–443.
[9] Y. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *JSSC*, vol. 52, no. 1, pp. 127–138, 2017.
[10] J. Lee *et al.*, "UNPU: A 50.6tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *ISSCC*. IEEE, 2018.
[11] 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.
[12] M. Imani *et al.*, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *ICRC*. IEEE, 2017, pp. 1–8.
[13] A. Hernández-Cano *et al.*, "Real-time and robust hyperdimensional classification," in *GLSVLSI*, 2021, pp. 397–402.
[14] M. Imani *et al.*, "Exploring hyperdimensional associative memory," in *HPCA*. IEEE, 2017, pp. 445–456.
[15] P. R. Genssler *et al.*, "Brain-inspired computing for wafer map defect pattern classification," in *ITC*. IEEE, 2021, pp. 123–132.
[16] Y. Halawani *et al.*, "Rram-based cam combined with time-domain circuits for hyperdimensional computing," 2021.
[17] A. Hernandez-Cane *et al.*, "Onlinehd: Robust, efficient, and single-pass online learning using hyperdimensional system," in *DATE*. IEEE, 2021, pp. 56–61.
[18] Z. Zou *et al.*, "Scalable edge-based hyperdimensional learning system with brain-like neural adaptation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.
[19] P. Poduval *et al.*, "Stochd: Stochastic hyperdimensional system for efficient and robust learning from raw data," in *DAC*. IEEE, 2021, pp. 1195–1200.
[20] M. Imani *et al.*, "Revisiting hyperdimensional learning for fpga and low-power architectures," in *HPCA*. IEEE, 2021, pp. 221–234.
[21] ——, "A framework for collaborative learning in secure high-dimensional space," in *CLOUD*. IEEE, 2019, pp. 435–446.
[22] A. Hérnandez-Cano *et al.*, "Prid: Model inversion privacy attacks in hyperdimensional learning systems," in *DAC*. IEEE, 2021, pp. 553–558.
[23] K. Dinghofer and F. Hartung, "Analysis of criteria for the selection of machine learning frameworks," in *ICNC*. IEEE, 2020, pp. 373–377.
[24] A. Moin *et al.*, "A wearable biosensing system with in-sensor adaptive machine learning for hand gesture recognition," *Nature Electronics*, vol. 4, no. 1, 2021.
[25] Z. Zou *et al.*, "Spiking hyperdimensional network: Neuromorphic models integrated with memory-inspired framework," *arXiv preprint arXiv:2110.00214*, 2021.
[26] Y. K. others, "Geniehd: Efficient DNA pattern matching accelerator using hyperdimensional computing," in *DATE*. IEEE, 2020, pp. 115–120.
[27] P. Poduval *et al.*, "Cognitive correlative encoding for genome sequence matching in hyperdimensional system," in *DAC*. IEEE, 2021, pp. 781–786.
[28] A. Hernández-Cano *et al.*, "Reghd: Robust and efficient regression in hyperdimensional learning system," in *DAC*. IEEE, 2021, pp. 7–12.
[29] R. Chen *et al.*, "Joint active search and neuromorphic computing for efficient data exploitation and monitoring in additive manufacturing," *Journal of Manufacturing Processes*, vol. 71, pp. 743–752, 2021.
[30] M. Imani *et al.*, "Dual: Acceleration of clustering algorithms using digital-based processing in-memory," in *MICRO*. IEEE, 2020, pp. 356–371.
[31] H. T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, 1982.
[32] B. Barry *et al.*, "Always-on vision processing unit for mobile applications," *IEEE Micro*, vol. 35, no. 2, pp. 56–66, 2015.
[33] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, 1996.
[34] Y. Kim *et al.*, "ORCHARD: visual object recognition accelerator based on approximate in-memory processing," in *ICCAD*. IEEE, 2017, pp. 25–32.
[35] D. Dua and C. Graff, "Isolet dataset, UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/isolet
[36] D. Anguita *et al.*, "A public domain dataset for human activity recognition using smartphones," in *ESANN*, 2013.
[37] Y. LeCun *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
[38] A. Reiss *et al.*, "Introducing a new benchmarked dataset for activity monitoring," in *ISWC*. IEEE, 2012, pp. 108–109.