A Robust and Energy-Efficient Classifier Using Brain-Inspired Hyperdimensional Computing

Abbas Rahimi
EECS Department
University of California
Berkeley
abbas@eecs.berkeley.edu

Pentti Kanerva
Redwood Center for
Theoretical Neuroscience
University of California
Berkeley
pkanerva@berkeley.edu

Jan M. Rabaey
EECS Department
University of California
Berkeley
jan@eecs.berkeley.edu

ABSTRACT

The mathematical properties of high-dimensional (HD) spaces show remarkable agreement with behaviors controlled by the brain. Computing with HD vectors, referred to as "hypervectors," is a brain-inspired alternative to computing with numbers. Hypervectors are high-dimensional, holographic, and (pseudo)random with independent and identically distributed (i.i.d.) components. They provide for energy-efficient computing while tolerating hardware variation typical of nanoscale fabrics. We describe a hardware architecture for a hypervector-based classifier and demonstrate it with language identification from letter trigrams. The HD classifier is 96.7% accurate, 1.2% lower than a conventional machine learning method, operating with half the energy. Moreover, the HD classifier is able to tolerate 8.8-fold probability of failure of memory cells while maintaining 94% accuracy. This robust behavior with erroneous memory cells can significantly improve energy efficiency.

1. INTRODUCTION

Reducing the size of CMOS transistors no longer guarantees the customary gains in performance and energy efficiency of integrated computing platforms. The manufacture of devices near atomic feature dimensions is particularly challenging. Any variation in dimensions, doping, etc. has a large effect on the resulting device and circuit behavior [1]. Solutions that improve energy efficiency – performance per Watt – in the presence of such variations, are highly desirable.

Bio- and brain-inspired information processing architectures are a promising new avenue to energy efficiency, asymptotically approaching the efficiency of brain computation, while tolerating variations in nanoscale fabrics [2, 3, 4]. Among brain-inspired computing paradigms, hyperdimensional computing is founded on the mathematical properties of high-dimensional spaces which show remarkable agreement with behaviors controlled by the brain [5, 6, 7, 8, 9]. Instead of computing with numbers, we compute with hypervectors that are high-dimensional (HD; e.g., 10,000-*D*) and holographic, i.e., every piece of information contained in the vector is distributed equally over all the components of the vector. These features allow hyperdimensional computing to achieve robustness in the presence of hardware-induced errors, greatly improving energy efficiency at the expense of slight, or no functional performance degradation.

The algorithm that forms the basis of this paper is available on [9]. It identifies the language of text samples based on letter *N*-grams. Here, we introduce an HD classifier for that task as the first energy-efficient and robust hardware design of hyperdimensional computing. We propose a modular, scalable, and memory-centric architecture where a hypervector for a text sample is produced and compared concurrently with a set of trained hypervectors. Our design optimizations include substituting a high-precision hypervector of integers with a low-precision binary hypervector, reducing switching activities, and lowering the complexity of search operations. We compare classification accuracy, energy consumption and robustness of the HD classifier with a conventional machine learning method of the same hardware complexity. Compared to this conventional method: 1) the HD classifier enables 53% energy saving at the expense of 1.2% lower accuracy in language recognition. 2) the HD classifier exhibits extremely robust behavior with low-precision components and tolerates hardware-induced errors in them; it tolerates 8.8-fold probability of failure per individual memory cells, while maintaining a recognition accuracy of above 94%. In addition, the same architecture can be retrained to perform other tasks such as text classification by topic with similar success rates [10].

In Section 2, we present the concepts of hyperdimensional computing. Our proposed memory-centric architecture with energy optimizations for HD classifier are described in Section 3. In Section 4, we present experimental results followed by discussion. Section 5 concludes this paper.

2. HYPERDIMENSIONAL COMPUTING

The brain's circuits are massive in terms of numbers of neurons and synapses, suggesting that large circuits are fundamental to the brain's computing. Hyperdimensional computing [7] is based on the understanding that brains compute with *patterns of neural activity* that are not readily associated with numbers. In fact, the brain's ability to calculate with numbers is feeble. However, due to the very size of the brain's circuits, we can model neural activity patterns with points of a high-dimensional space, that is, with hypervectors. When the dimensionality is in the thousands (e.g., D =10,000), it is called hyperdimensional.

Hypervectors are holographic, and (pseudo)random with i.i.d. components. A hypervector contains all the information combined and spread across all its bits in a full holistic representation so that no bit is more responsible to store any piece of information than an-

ISLPED '16, August 08-10, 2016, San Francisco Airport, CA, USA

DOI: http://dx.doi.org/10.1145/2934583.2934624

other. Hypervectors are combined with operations akin to addition, multiplication, and permutation that form an algebra over the vector space (e.g., a field). Hypervectors can be compared for similarity using a distance metric over the vector space. These operations on hypervectors can be combined into interesting computational behavior with unique features that make them robust and efficient. In this paper, we target an application of hyperdimensional computing for identifying the language of text samples, based on encoding consecutive letters into hypervectors.

Recognizing the language of a given text is the first step in all sorts of language processing, such as text analysis, categorization, translation, etc. High-dimensional vector models are popular in natural-language processing and are used to capture word meaning from word-use statistics. The vectors are often called semantic vectors. Ideally, words with a similar meaning are represented by semantic vectors that are close to each other in the vector space, while dissimilar meanings are represented by semantic vectors far from each other [11]. Latent semantic analysis [11] is a standard way of making semantic vectors. It relies on singular value decomposition of a large matrix of word frequencies. It is computationally heavy and scales poorly.

Random indexing [5, 6] is an algorithm based on high dimensionality and randomness and it provides a simple and scalable alternative to methods based on principal components, including latent semantic analysis. It is incremental and computes semantic vectors in a single pass over the text data. With the dimensionality in the thousands it is possible to calculate useful representations with fast, and highly scalable algorithms. We use random indexing for identifying the source language of text samples by compiling their N-grams – N consecutive letters – into hypervectors, and by comparing the vectors to each other.

2.1 Random Indexing

Random indexing represents information by projecting data onto vectors in a hyperdimensional space. There exist a huge number of different, nearly orthogonal hypervectors in such a space [12]. This lets us combine two such hypervectors into a new hypervector using well-defined vector-space operations, while keeping the information of the original two with high probability. We consider a variant of the multiplication, addition, and permutation (MAP) coding described in [13] to define the hyperdimensional vector space. The hypervectors are initially taken from a 10,000-dimensional space and have an equal number of randomly placed 1s and -1s. Such hypervectors are used to represent the basic elements, i.e., the 26 letters of the Latin alphabet and the (ASCII) space.

The MAP operations on the hypervectors are defined as follows. Componentwise addition of two hypervectors A and B, is denoted by $A+B$. Information from a pair of hypervectors A and B is stored and utilized in a single hypervector by exploiting the addition operation. That is, the sum of two separate hypervectors naturally preserves unique information from each hypervector because of the mathematical properties of vector addition. The vector addition is well suited for representing sets. Componentwise multiplication is denoted by $A*B$. Multiplication of two hypervectors produces a vector that is dissimilar to its constituent vectors; hence it is well suited for binding hypervectors. The third operation is a permutation, ρ, that rotates the hypervector coordinates. The permutation operation generates a dissimilar vector by scrambling that is good for storing a sequence of hypervectors. For example, the sequence trigram of A-B-C, is stored as the following hypervector, $\rho(\rho A*B)*C = \rho\rho A*\rho B*C$. This efficiently distinguishes the sequence A-B-C from A-C-B, since a rotated hypervector is uncorrelated with all the other hypervectors.

Cosine similarity is used to measure similarity between two hypervectors by measuring the cosine of the angle between them using a dot product. It is defined as $\cos(A,B) = |A'*B'|$, where A' and B' are the normalized vectors of A and B, respectively, and $|C|$ denotes the sum of the elements in C.

3. MEMORY-CENTRIC ARCHITECTURE FOR HD CLASSIFIER

In this section, we first describe our proposed architecture for the HD classifier and in Sections 3.1 and 3.2 describe its main modules. In Section 3.3, we show how the hyperdimensional computing for language recognition is very robust to low-precision binary components, followed by our optimizations for energy efficiency in Section 3.4.

The proposed design uses the same strategy as presented in [9] for recognizing a text's language by generating and comparing text hypervectors: the text hypervector of an unknown text sample is compared for similarity to precomputed text hypervectors of known language samples – the former is referred to as a query hypervector, while the latter are referred to as language hypervectors. As shown in Figure 1, the design is based on a memory-centric architecture where logic is tightly integrated with the memory and all computation is fully distributed. The architecture has two main modules: encoding and similarity search. The encoding module projects an input text, composed of a stream of letters, to a hypervector in high-dimensional space. Then this hypervector is broadcast to the similarity search module for comparing with a set of precomputed language hypervectors. The search module returns the language that has the closet match. In the following two sections, we describe the architectural details of these two modules.

3.1 Encoding Module

The encoding module accepts a stream of letters from a text and computes a hypervector that represents the text. First, an item memory assigns a unique but random hypervector, that is called letter hypervector, to an input letter. The item memory is a catalog of meaningful patterns, and it is implemented as a lookup table. In the binary implementation of our encoding module, a hypervector has an equal number of randomly placed 1s and 0s. This assignment is fixed throughout the computation, and formed 27 approximately orthogonal hypervectors as the basic elements of our alphabet here with 27 symbols.

Second, we need to compute a hypervector for a block of N consecutive letters, for example, a window of three letters or a trigram. Hence, we consider three stages of memory, in the FIFO style, each of which stores a letter hypervector. A trigram hypervector is created by permuting the letter hypervectors and multiplying them as described earlier. The random permutation operation ρ is fixed, and implemented as a cyclic rotation to right by 1 position as shown in Figure 1. In geometry sense, this permutation rotates the vector in the space. For instance, considering the trigram of A-B-C, A hypervector is rotated twice ($\rho\rho A$), B hypervector is rotated once (ρB), and there is no rotation for C hypervector. Once the letter C is reached, its corresponding C hypervector is fetched from the item memory and is directly written to the first stage of encoder (i.e., letter_3 hypervector in Figure 1). To apply $\rho(\rho A, B)$ on the two previous letters, they are rotated as they pass through the encoder stages. The pointwise multiplications are then applied between these new hypervectors to compute the trigram hypervector, i.e., $\rho\rho A*\rho B*C$. Since the trigram hypervector is binary, the multiplication between two hypervectors is implemented with D XOR gates.

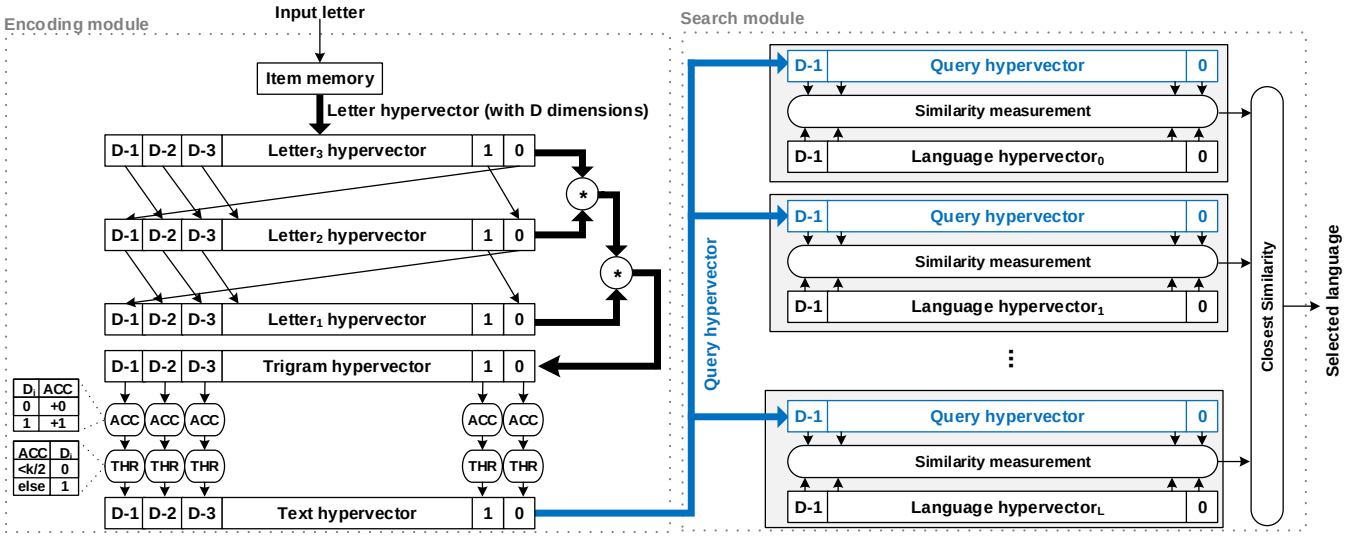

Figure 1: Memory-centric architecture for hyperdimensional computing: encoding module and search module.

Third, a text hypervector for an input text is computed by adding all the trigram hypervectors using a sliding window of three letters across the text. This pointwise addition, or summation, produces another D-dimensional hypervector where each component is an integer value. Section 3.4.2 shows how we can substitute such a high-precision text hypervector with a binary hypervector. The output of the encoding module is the text hypervector.

The encoding module is used for both training and testing. During training when the language of the input text in known, we refer to the text hypervector as a language hypervector. Such language hypervectors are stored in the search module. When the language of a text is unknown, as it is during testing, we call the text hypervector as a query hypervector. The query hypervector is sent to the similarity search module to identify its source language.

3.2 Similarity Search Module

The search module stores a set of language hypervectors that are precomputed by the encoding module. These language hypervectors are made in exactly the same way described above, by making the text hypervectors from samples of a known language. Therefore, during the training phase, we feed texts of a known language to the encoding module and save the resulting text hypervector as a language hypervector in the search module. We consider 21 European languages, consequently at the end of the training phase, we will have 21 language hypervectors, each of which is stored separately in a row of the search module.

Determining the language of an unknown text is done by comparing its query hypervector to all the language hypervectors. This comparison is effectively performed in a distributed fashion using an associative memory. The cosine similarity is used as the similarity metric [8, 9]. It measures $\text{distance}_{\cos}$ between a language hypervector (LV_i) and an unknown query hypervector (QV) as follows:

$$\text{distance}_{\cos} = \frac{\text{LV}_i \cdot \text{QV}}{|\,\text{LV}_i\,||\,\text{QV}\,|} \qquad (1)$$

where $\text{LV}_i \cdot \text{QV}$ is the dot product between the two hypervectors, $|\text{LV}_i\,|$ and $|\,\text{QV}\,|$ are the magnitudes of LV_i and QV, respectively. If $\text{distance}_{\cos}$ is close to 1, it means that the trigram frequencies of the unknown text presented in QV are similar to the trigram frequencies of the language vector_i, and therefore the text is likely to be written in the same language. We design a modular similarity-measurement block that calculates such $\text{distance}_{\cos}$ between a precomputed LV_i and QV. This block is replicated L times within the search module; L is the number of languages in our application. The QV is broadcast across the search module, hence all the similarity-measurement blocks compute their cosines concurrently. Finally, a combinational comparison block selects the highest cosine and returns its associated language as the language that the unknown text has been written in.

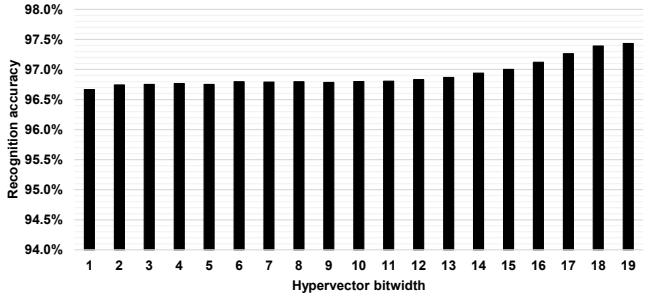

Figure 2: Recognition accuracy while varying the component bitwidth of 10,000-dimensional hypervectors.

3.3 Robustness in the Presence of Low-Precision Components

Here we assess the robustness of hyperdimensional computing for the language recognition by replacing high-precision components with low-precision components. As described in Section 3.1, a text hypervector contains integer components due to the addition operation that accumulates the trigram hypervectors over the text. Hence, each component in the hypervector requires a multibit cell memory. For learning a megabyte of text, each hypervector component will need 19 bits precision for summing a million of randomly placed 0s and 1s. Figure 2 shows the accuracy of language recognition as a function of bitwidth of the hypervector components. As shown, the recognition accuracy is slightly decreased by reducing the bitwidth (i.e., the precision of each component). Such a robust behavior enables us to turn the high-precision hypervec-

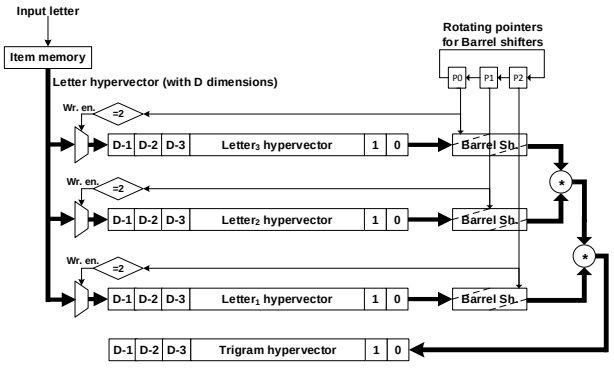

Figure 3: Barrel shifters for trigram hypervector generation.

tors to binary hypervectors, with the same dimensionality, while slightly degrading the recognition accuracy from 97.4% to 96.7%. This bitwidth reduction saves the required memory for the search module by a factor of 19×. Moreover, computing with such binary hypervectors requires fewer hardware resources in both encoding and search modules, motivating us to look for further optimizations presented in the following section.

3.4 Optimizations for Energy Efficiency

In this section, we describe our design optimization techniques for energy-efficient hyperdimensional computing with binary components. Our energy analysis in Section 4.2.2 shows that more than 55% of the total power consumption goes to the encoding module. This is because the encoding module, shown in Figure 1, involves power-hungry operations: trigram hypervector generation, and text hypervector generation. The former requires high amount of switching activity, and the latter requires a large number of resources for accumulation and thresholding. In the following two sections, we provide effective solutions to address each of these concerns. In Section 3.4.1, we describe a technique for reducing memory switching activity during trigram hypervector generation. In Section 3.4.2, we demonstrate savings in resources by using binary hypervectors rather than high-precision hypervectors. In Section 3.4.3, we reduce the complexity of hardware implementation of the similarity search module.

3.4.1 Encoding Trigrams with Minimal Switching

Generating a trigram hypervector involves permutation and multiplication operations. As we describe in Section 3.1, the permutation is implemented as a cyclic 1-bit rotation to right. This rotation operation imposes high amount of switching activity in the memory stages where the hypervectors for the letters are stored. Because the hypervectors have equal number of randomly placed 0s and 1s, rotating them in the memory consumes a lot of energy. To address this issue, we propose a new design for trigram encoding that avoids such high switching activity in the memory.

Figure 3 illustrates the proposed encoding of trigrams. The letter hypervectors, retrieved from the item memory, are stored in a separate letter memory in their arrival order. The design uses three Barrel shifters to rotate the letter hypervectors as desired, before sending them into the multipliers as opposed to rotating them in each cycle and storing the rotated hypervectors in the memory stages (see Figure 1). The design includes a set of rotating pointers (P0, P1, and P2) that rotate values of 0, 1, and 2 among themselves. These pointer values for the Barrel shifters implement the no rotate, 1-bit rotate (ρ), and 2-bit rotate ($\rho\rho$) operations. Every Barrel shifter rotates the letter hypervector based on the assigned pointer value. The rotated letter hypervectors are multiplied (i.e., XORed) and the resulting hypervector is written to the trigram memory. This design inhibits the undesirable switching activities due to the rotate operations in the memory while generating the trigrams as accurately as the naive encoder, therefore it does not degrade the recognition accuracy.

3.4.2 Binary Hypervector Generation

Here, we focus on resource optimizations for the second part of the encoder that uses the trigrams. As described in Section 2.1, a text hypervector is computed by adding all the trigrams over the input text. To produce a binary hypervector, we implement such pointwise addition through a set of D accumulators (ACC) and threshold units (THR) as shown in Figure 1. Every accumulator is assigned to a dimension of the hypervector, and counts the number of 1s in that component location. Once a new trigram hypervector is generated, i accumulators will be accordingly incremented where i is the number of 1s in the generated trigram hypervector. An input text with k letters generates $k-2$ trigram vectors. Finally, to compute the corresponding binary text hypervector, the encoding module applies a majority function of (k, $k/2$) to every accumulator value. The accumulator values are compared to a threshold of $k/2$ by the encoding module and passed on to the text hypervector as either 0 or 1. Left side of Figure 1 shows such a dedicated accumulation and thresholding for every hypervector component.

3.4.3 Low-Cost Modular Similarity Search

The last optimization focuses on the similarity search module that is composed of a set of similarity-measurement blocks. In a similarity-measurement block, the broadcast query hypervector (QV) is compared to a precomputed language hypervector, LV_i. Our optimization reduces the complexity of hardware resources that are required to compare these two hypervectors while providing a reliable distance measurement. In Section 3.2, the cosine is suggested as a measure of similarity between hypervectors [7, 9]. The cosine is a non-Euclidean distance that is based on angles between vectors and not their "locations" in space. We find that Hamming distance measures the similarity of hypervectors as reliably as the cosine without any degradation in the recognition accuracy.

Hamming distance counts the number of components at which two binary hypervectors disagree. Hamming distance reduces the energy consumption of similarity-measurement block since it does not require any normalization calculation as opposed to the cosine. We use a set of D XOR gates to identify mismatches between QV and LV_i. To ensure the scalability, the module compares only one component each clock cycle. Hence, the similarity-measurement block takes O(D) cycles to compute the Hamming distance between the two hypervectors. Thanks to its modularity, this block is replicated L times in the search module as shown in Figure 1. The search module selects a language that has the minimum Hamming distance with QV.

4. EXPERIMENTAL RESULTS

In this section, we first present our application of language recognition and its dataset. Next, we describe a conventional machine learning method as a baseline to compare our HD classifier to. We provide[1] both Matlab and RTL implementations for these two classifiers. We then compare their classification accuracy, memory footprints, energy consumption and robustness. Finally, we discuss our observations.

[1]Available for download at https://github.com/abbas-rahimi/HDC-Language-Recognition

4.1 Language Recognition Dataset

We consider an application for recognition of 21 European languages. The sample texts are taken from the Wortschatz Corpora [14] where large numbers of sentences in these languages are available. We train each language hypervector based on about a million bytes of text. To test the ability of identifying the language of unseen text samples, we select test sentences from Europarl Parallel Corpus [15] as an independent text source. This corpus provides 1,000 samples of each language, and each sample is a single sentence. The accuracy recognition metric used throughout this paper is the percentage of these 21,000 test samples that are identified correctly. This accuracy is measured as the microaveraging that gives equal weight to each per-sentence classification decision, rather than per-class.

4.2 Baseline Machine Learning Method

As the baseline technique, we choose a nearest neighbor classifier that uses histograms of N-grams. To compute distance between histograms, the dot product is used. A histogram is generated for each language to capture the frequency of N-grams that are observed during the training text. Hence, the outcome of the training phase is a set of 21 histograms that represent the language profiles. In the same vein, a histogram is generated from a test sentence. To find out the language of the test sentence, we compute the dot product of its histogram with the 21 precomputed histograms. The highest dot product score identifies the language that the test sentence is written in. Considering N-grams as the input features, a histogram requires $\#\text{elements}^N$ integer components where #elements is 27 in our application. To reduce this memory footprint, we convert the integer components of histograms to binary using their mean value as the threshold.

We choose such a nearest-neighbor classifier among histograms as the baseline for two reasons. First, the histogram has full information about the N-gram statistics, so it sets the highest standard of comparison. Second, from a hardware point of view, this baseline shares similarity with the HD classifier. Both use N-grams as input for the encoding, and involve operations with the same complexity. For instance, computing the frequency of an N-gram in the baseline is a lookup action followed by addition. During search operations, both use dot product to measure the distances; it is then simplified to Hamming distance for the binary components. Essentially, this baseline uses the same hardware components as the HD classifier does, but excludes the item memory. In the following sections we compare them in detail.

4.2.1 Classification Accuracy and Memory Usage

Table 1 compares the two classifiers when using binary components with different N-grams. The first two columns summarize the classification accuracy; the last two columns list the memory footprint with the binary components. Using bigrams of letters, the baseline has 2.3% lower recognition accuracy compared to the HD classifier. However, using N-grams with $N \geq 3$, the baseline displays slightly higher accuracies. For example, the baseline shows 97.9% recognition while the HD classifier shows 96.7% using trigrams. In this case, the HD classifier requires 1.2× as many memory cells as the baseline. On the upside, the HD classifier is able to represent many more N-grams within the same hardware structure. It scales very well: for instance, by moving from trigrams ($N = 3$) to pentagrams ($N = 5$), the HD classifier must add memory cells only for 2 extra hypervectors whereas the memory required by the baseline grows exponentially with N. Using pentagrams of letters, the baseline shows an accuracy of 99.8% (4.8% higher than the HD classifier), at the expense of 500× larger memory size. Of course the exact counts of all pentagrams that appear in a million bytes of text can be captured in much less memory than that but the algorithm is no longer as simple.

Table 1: Classification accuracy and memory footprint of HD and baseline classifiers.

	Accuracy		Memory (Kb)	
	HD	Baseline	HD	Baseline
Bigrams (N=2)	93.2%	90.9%	670	39
Trigrams (N=3)	96.7%	97.9%	680	532
Tetragrams (N=4)	97.1%	99.2%	690	13837
Pentagrams (N=5)	95.0%	99.8%	700	373092

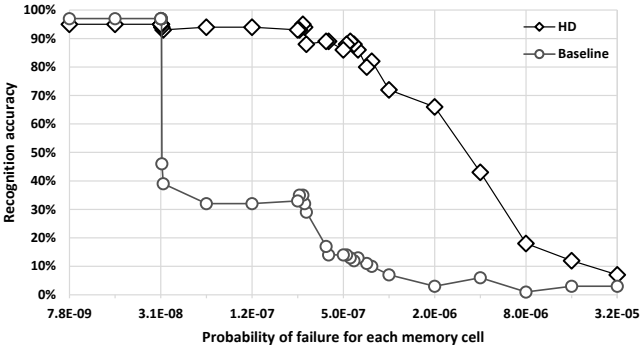

Figure 4: Accuracy of classifiers with faulty memory cells.

4.2.2 Energy Efficiency

We use a standard ASIC flow to design dedicated hardware for these two classifiers. We describe the classifiers, in a fully parameterized manner, using RTL SystemVerilog. We apply the identical constraints and flow to both designs. For the synthesis, we use *Synopsys Design Compiler* with the TSMC 65-nm technology library, the low-power process with high V_{TH} cells. The designs are optimized for a cycle time of 1 ns. We extract the switching activity of these classifiers during postsynthesis simulations in *ModelSim* by applying the test sentences. Finally, we measure their power consumptions using *Synopsys PrimeTime* at (1.2V, 25 °C, TT) corner.

We compare the power consumption for trigrams only, since with N-grams of $N \geq 4$, the baseline classifier becomes increasingly less efficient compared to the HD classifier due to exponential growth in the amount of memory required. With 47% of the energy required by the baseline classifier, the HD classifier is only 1.2% less accurate: 96.7% versus 97.9% for the baseline. Although both designs use binary components and low-cost Hamming distance for similarity measurements, the HD classifier achieves higher energy efficiency thanks to its one-shot computation with highly scalable and local operations, in addition to the optimizations presented in Section 3.4.

4.2.3 Robustness Against Memory Errors

Here we assess the classifiers' tolerance for memory errors. We target RTL fault simulations where we inject memory bit flips during every clock cycle of execution. We consider a wider range of probability of failures for each memory cell; the fault simulations cover all the memory elements in both designs.

Fig. 4 shows the recognition accuracy with the erroneous memory cells; the X-axis displays the probability of failure for each memory cell in every clock cycle. The baseline is able to maintain

its high accuracy of 97% using faulty memory cells with the probability of failure at 3.16E-08, and lower values. At 3.17E-08 the accuracy falls sharply to below 46%. However, the HD classifier exhibits a very robust behavior: it maintains the recognition accuracy of 94% and higher for the probability of failure up to 2.78E-07. At or near peek performance (94% for the HD classifier and 97% for the baseline), the HD classifier tolerates 8.8-fold probability of failure compared to the baseline. By further increasing the probability of failure by 2.8× to 7.69E-07, the HD classifier is still 80% accurate or better. Finally, the accuracy of the HD classifier drops to 43% when using memory cells with probability of failure at 4.00E-06, i.e., ≈120× higher that the failure rate that the baseline could tolerate for the same accuracy.

4.3 Discussion

Here, we further discuss energy efficiency and robustness benefits of hyperdimensional computing. At its very core, hyperdimensional computing is about manipulating and comparing large patterns, stored in memory. The operations are either local or can be performed in a distributed fashion leading to a substantial energy reduction. These properties of hyperdimensional computing make it an excellent match to emerging 3D nanoscale device platforms. This presents a fundamental departure from traditional computational architectures, where data has to be transported to the processing unit and back, creating the infamous memory wall.

Hyperdimensional computing also exhibits a robust behavior enabling further energy saving by operating in low signal-to-noise ratio conditions, or by utilizing emerging imprecise nanoscale devices. Such robustness to the low-precision and faulty components is achieved thanks to the special brain-inspired properties of hyperdimensional computing: (pseudo)randomness with i.i.d. components, high-dimensionality, and holographic representations. In the following, we briefly discuss their contributions to such robustness.

The algorithm starts with seed letter vectors with i.i.d. components and combines them with the MAP operations. Componentwise multiplication and addition are i.i.d.-preserving. When the permutation is combined with the multiplications to encode *N*-grams, we end up with vectors whose components are identically distributed and nearly independent. This means that the components of the language vectors are identically distributed and nearly independent; hence, a failure in a component is not contagious. At the same time, failures in a subset of components are compensated by the holographic representation, i.e., the error-free components can still provide a useful representation that is "good enough" for distinction. This property inherently eliminates any needs for the asymmetric error protection in the memory units.

Further, the algorithm for hyperdimensional computing is one-shot and incremental. It involves componentwise and local operations to compute and compare the hypervectors without any control flow conditions, which brings another degree of robustness for the algorithm.

5. CONCLUSION

We propose a robust and energy-efficient hardware design for hyperdimensional computing. The proposed HD classifier forms a memory-centric architecture with modular and scalable components. We compare it with the conventional nearest neighbor classifier that uses histograms of trigrams: the HD classifier uses half the energy and tolerates 8.8-fold probability of failure for individual memory cells, while displaying a recognition accuracy of 94% (at maximum, 3% lower than the conventional method). This excellent performance with low-precision and faulty components is accomplished by appeal to the mathematical properties of high-dimensional spaces, including the high-dimensional, holographic, and (pseudo)random representation with i.i.d. components, in addition to the absence of control flow during execution. Our ongoing work is focused on efficient encoding with hierarchical searches, as well as their realization on 3D nanofabrics.

6. ACKNOWLEDGMENT

This work was supported by Systems on Nanoscale Information fabriCs (SONIC), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

7. REFERENCES

[1] S. Borkar, et. al. Parameter variations and impact on circuits and microarchitecture. In *Proc. of the Design Automation Conference*, pages 338–342, June 2003.

[2] T.-T. Liu and J.M. Rabaey. A 0.25 V 460 nW asynchronous neural signal processor with inherent leakage suppression. *Solid-State Circuits, IEEE Journal of*, 48(4):897–906, 2013.

[3] D. Kuzum, et. al. Low-energy robust neuromorphic computation using synaptic devices. *Electron Devices, IEEE Transactions on*, 59(12):3489–3494, Dec 2012.

[4] Beinuo Zhang, Zhewei Jiang, Qi Wang, Jae sun Seo, and Mingoo Seok. A neuromorphic neural spike clustering processor for deep-brain sensing and stimulation systems. In *Proc. of the International Symposium on Low Power Electronics and Design*, 2015.

[5] Pentti Kanerva, Jan Kristoferson, and Anders Holst. Random indexing of text samples for latent semantic analysis. In *Proc. of the Conference of the Cognitive Science Society*, 2000.

[6] Magnus Sahlgren. An introduction to random indexing. In *Methods and Applications of Semantic Indexing Workshop at the 7th International Conference on Terminology and Knowledge Engineering, TKE 2005*, 2005.

[7] Pentti Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation*, 1(2):139–159, 2009.

[8] Pentti Kanerva. Computing with 10,000-bitwords. In *Proc. 52nd Annual Allerton Conference on Communication, Control, and Computing*, 2014.

[9] Aditya Joshi, Johan Halseth, and Pentti Kanerva. Language geometry using random indexing. In *Quantum Interaction 2016 Conference Proceedings*, in press.

[10] Fateme Rasti Najafabadi, Abbas Rahimi, Pentti Kanerva, and Jan M. Rabaey. Hyperdimensional computing for text classification. *Design, Automation Test in Europe Conference Exhibition (DATE), University Booth*, March 2016.

[11] T.K. Landauer and S.T. Dumais. A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review*, 104(2):211–240, 1997.

[12] Pentti Kanerva. *Sparse Distributed Memory*. MIT Press, Cambridge, MA, USA, 1988.

[13] Ross W. Gayler. Multiplicative binding, representation operators & analogy. *Advances in analogy research*, 1998.

[14] Uwe Quasthoff, Matthias Richter, and Christian Biemann. Corpus portal for search in monolingual corpora. In *Proc. of the International Conference on Language Resources and Evaluation*, 2006.

[15] Philipp Koehn. Europarl: A parallel corpus for statistical machine translation. http://www.statmt.org/europarl/, 2005.