Hyper-Dimensional Computing and its Applications in tinyML

Ellis A. Weglewski
ydnawizard@protonmail.com
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA

Abstract

As computing systems enter the realm of nano form levels, new fields of computational development have spawned, each posing their own set of challenges. Amongst these fields is Tiny Machine Learning (tinyML), which aims to install machine learning on tiny embedded systems. The restrictions imposed upon algorithms by the limited hardware of nano-scale tiny systems make contemporary approaches to machine learning non-contenders. Hyperdimensional computing is an approach to representing data as high-dimensional vectors which allows for one-pass encoding and quick all-encompassing comparison operations via an associative memory. This approach is power-efficient, robust, and can be done in-memory, all of which make it a viable candidate for tinyML.

Keywords: HDC, hyperdimensional computing, tinyml, tiny machine learning

1 Introduction

1.1 tinyML

tinyML is an area of computational development focused on deploying machine learning models on tiny embedded systems. Embedded systems are small devices optimized for the handling of a singular task, which can then be implemented as modules in a larger system. Neural networks can be implemented on many such devices to achieve satisfactory results. However, as tiny systems scale downwards into nanometer dimensions, such networks will find themselves starved of resources. Additionally, nano-scale tiny systems are sensitive to highly energetic particles present in the natural space environment (e.g. alpha particles from radiation or protons and neutrons from cosmic rays)[2]. In the event that such a particle strikes a nano-scale tiny system, distruption of the computation pipeline, change of logic state, or even permanent damage to the system may occur [2]. These errors are known as *single event upsets*[2] and occur at a rate of about 1-3 upsets (depending on the altitude of the system's location) per million integrated logic gates a day[13]. Given these challenges, a more lightweight and robust approach to tinyML is required for deployment at the nanometer form level.

1.2 Hyper-Dimensional Computing

Humans excel at quickly determining how similar or dissimilar things are to each other. For example, we instantly know that a fisherman and a lake are two separate things, but we also instantly know that they share a likeness more so than the fisherman and a volcano. This is because the fisherman has more things in common with the lake than the volcano, mainly fish and water.

Hyperdimensional Computing (HDC) aims to mimic this ability by encoding data as spatial positions or directions, which allows for very quick and all-encompassing comparison operations. If we construct our encoding process correctly, we can quickly project data onto a vector, which can then be compared to other encoded vectors for similarity. This is the fundamental concept behind HDC and makes it lightweight, tolerant of failures/noise, and low-power.

Rahimi et al.[6] and Kanerva[4] explain the foundational concepts of Hyperdimensional Computing in respective comprehensive papers. We will discuss the foundational concepts laid out in these papers in Sections 2 and 3. After this, we will discuss how these concepts are applied to form a functional machine learning model for natural language processing by Kanerva et al.[3] in Section 4. In Section 5, applications at the nano-scale form level will be explored with a low-power prosthetic hand gesture classification system by Rahimi et al[1, 11], and a proposed in-memory approach to HDC by Rahimi et al.[5].

2 Hypervectors

A vector is a set of coordinates where each coordinate corresponds to a single position on a single axis. When all these coordinates are plotted, they form a position or a direction. A hypervector is merely a vector with a very high dimensionality (generally in the thousands). In the context of this paper, dimensionality refers to how many coordinates a vector has.

2.1 Representation

In HDC, we project data onto hypervectors using different approaches to representation. Given the scope of this paper, it is necessary to understand two of them.

2.1.1 Symbolic Representations.

We as humans typically gain an intuitive understanding of symbols early on in our lives. Therefore, they are natural to us and are widely used in computers [6]. In symbolic representations, objects or items are represented by a symbol. In the context of this paper, an object is an item of varying nature and complexity, like physical features, letters, relations, and so on. Symbols are naturally combinatorial meaning that an indefinite amount of more complex symbols can be composed from simpler ones [6]. In HDC, a good example of a symbolic representation is natural language applications, where millions of word hypervectors can be composed from a small alphabet of letter hypervectors. Symbols have an all-or-nothing similarity, meaning that the same symbols have maximal similarity and different symbols have no similarity[6].

2.1.2 Distributed-Holographic Representations.

Electrical recording from neurons shows even mundane and simple mental events activate a widespread set of neurons [4]. While trying to mimic this is very difficult, we can take pointers from it. Through distributing low-dimensional information across a high-dimensional medium, we develop a Distributed-Holographic representation that is extremely robust to noise. Distributed-Holographic hypervectors are defined as a class of vector representations, where each object or characteristic is represented by a subset of vector components[6]. This kind of representation maximizes redundancy[4] which means that if components are perturbed, the information degrades only in respect to the number of perturbed components irrespective of their position[4].

2.2 Composition

2.2.1 Density.

HDC deals with either densely populated or sparsely populated hypervectors, this paper deals specifically with dense hypervectors. In this context, dense means that the hypervector is approximately half composed of one value from a set of size two, and the other half is composed of the remaining value from the same set. In the case of a hypervector with bipolar components from the set {-1,1}, a dense hypervector would be composed of about half -1's and half 1's. In the case of hypervectors with binary components from the set {0,1}, a dense binary hypervector would be composed of about half 1's and half 0's.

2.2.2 HDC Models.

When dealing with hypervectors, it is important to consider the makeup of their components. HDC models use different component values which give them unique properties. This makes certain models better tailored for different problem types. There are a handful of these models. For the sake of efficiency, this paper will cover two of them. The *Binary Spatter Code* (BSC) model was developed by Pentti Kanerva. In BSC, hypervectors are dense and binary with components from the set {0,1} [6]. The *Multiply Add Permute* (MAP) model was proposed by Gayler in 1998[6] and has several variants, we will be covering MAP-b which is bipolar in nature and isomorphic to the BSC model [6]. In the MAP model, hypervectors are dense, bipolar, and comprised of components from the set {-1,1}.

2.3 Comparison

Hyperdimensional Computing revolves around hypervectors and how similar they are, therefore, a method is needed to gauge similarity. To accomplish this, we can compare how close they are or how similar their directions are. Three such comparison operations are needed to understand this paper, they are as follows (D is dimensions while a and b are two separate hypervectors):

Hamming Distance:

$$dist_{ham}(a,b) = \frac{1}{D}|a \oplus b|;$$

Dot Product:

$$sim_{dot}(a,b) = \sum_i a_i b_i = a^{\perp} b;$$

and *Cosine Similarity*:

$$sim_{cos}(a,b) = \frac{a^{\perp} b}{\|a\|_2 \|b\|_2}$$

Hamming Distance calculates how many components differ between two binary hypervectors in relation to their dimensions and is how we ascertain similarity amongst BSC hypervectors. This is done by running a and b through a component-wise XOR (denoted by $\oplus$), counting the amount of non-zero components in the resulting vector, and dividing that number by D-dimensions. The *Dot Product* returns a scalar (one-dimensional vector) product of two vectors. *Cosine Similarity* measures the angle between two vectors and is useful for gauging similarity amongst MAP hypervectors. It works by taking the dot product of a and b, and dividing it by the product of their Euclidean norms. A vector's Euclidean norm is: $\sqrt{a_1^2 + a_2^2 + ...a_n^2}$.

2.4 Orthogonality and Similarity

Two vectors are orthogonal if they are perpendicular and quasi-orthogonal if they are almost perpendicular. In terms of HDC models, two MAP hypervectors are orthogonal if their cosine similarity is 0 and quasi-orthogonal if their cosine similarity is within a small threshold around 0. In the case of two BSC hypervectors, they would be orthogonal if their Hamming distance was .5, and quasi-orthogonal if their Hamming distance was within a small threshold of .5. A cosine similarity of 0 in regards to MAP hypervectors and a hamming distance of .5 in regards to BSC hypervectors can only be obtained when half the components of the two hypervectors being compared are the same and half the components are different. Therefore, orthogonal hypervectors are half correlated (similar) and half anti-correlated (dissimilar), which cancel each other out. This gives them the unique property of no correlation (neither similar nor dissimilar). Quasi-orthogonal vectors operate similarly in that they are

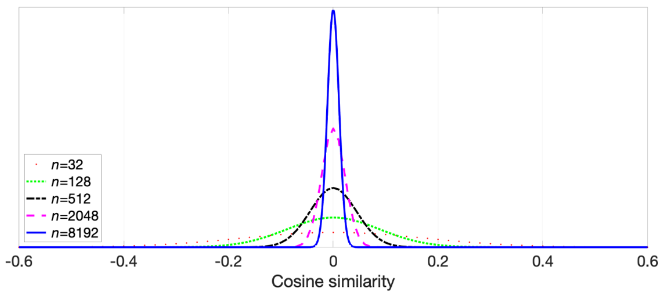

Figure 1. Cosine similarity distribution amongst 2,000 random MAP hypervectors in D-Dimensional spaces.

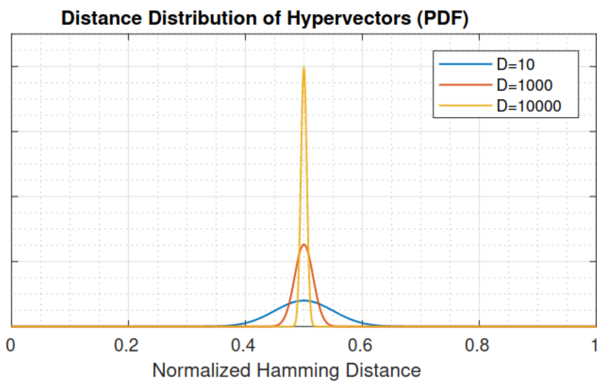

Figure 2. Hamming distance distribution of random BSC hypervectors in D-dimensional spaces.

approximately half correlated (similar) and approximately half anti-correlated (dissimilar) which gives them roughly no correlation or similarity/dissimilarity.

2.4.1 Concentration of Measure and The Blessing of Dimensionality.

If we use the properties of a normal (Gaussian) distribution to approximate the discrete binomial distribution of random BSC hypervectors, the standard deviation is $\sqrt{D}/2$ (where D is dimensions)[11]. In a normal distribution, approximately 68.2% of the space lies within one standard deviation from the mean or within $\sqrt{D} \pm 1$ standard deviations from a point in hyperspace[11]. Further, within 6 standard deviations from the mean, approximately 99.99% of the space is found[11]. From this, in respect to the BSC model, we can establish an orthogonality threshold of $\frac{\sqrt{D}\cdot(\sqrt{D}\pm 6)}{2\cdot D}$ [11]. This means that there is an approximately 99.99% chance that two random 10,0000 dimensional BSC hypervectors will have a Hamming distance in the range {0.47, 0.53}[11]. The same logic can be applied to the MAP model due to it being isomorphic to BSC. Therefore, as D (where D is dimensions) increases, the number of orthogonal and quasi-orthogonal hypervectors that can be generated in regards to either the BSC or MAP models becomes incredibly large. This is known as the concentration of measure phenomenon [6] and can be observed in Figure 1 for the MAP model and Figure 2 for the BSC model. Additionally, as dimensions increase, the mathematical properties of quasi-orthogonal hypervectors get closer to the properties of exactly orthogonal hypervectors[6]. This is known as the "Blessing of Dimensionality". These properties are very useful and make the encoding process in HDC simple.

3 Encoding

3.1 Atomic Hypervectors

It is common in HDC to first encode the base components of a problem to be solved into hypervectors, which we refer to as atomic hypervectors [6]. This process is often referred to as projection or mapping. If we were solving a problem dealing with natural language, we would project the letters of the alphabet onto random hypervectors using a symbolic representation. Letters are symbols so they should have an all-or-none similarity; They should be orthogonal or quasi-orthogonal to each other. Therefore, we can generate symbolic random hypervectors for each letter of the alphabet because of the concentration of measure phenomenon. There may also be problems where the atomic hypervectors need to have a degree of similarity amongst them [6]. For instance, if atomic hypervectors were to represent signals, it might make sense for signals with similar strengths to have similar atomic hypervectors by distributing the signal strength in a manner that preserves the signal threshold. In any case, we need to choose atomic hypervectors so that that the similarity amongst them corresponds to the characteristics which we care about [6]. Once atomic hypervectors have been assigned, they can be manipulated via operations to represent more complex objects and information.

3.2 Bundling

By nature, higher concepts are compositional. Likewise in HDC, higher concepts are hypervectors composed of atomic hypervectors. Further, these compositional hypervectors can also be combined to encode even higher abstractions to the nth degree. Thus, there needs to be a way to represent multiple hypervectors as one hypervector, like a superposition in a sense [6]. We accomplish this through *bundling*, which is a component-wise addition of two hypervectors. Through adding the component of one hypervector to its cor- responding component of the next hypervector, we end up with a hypervector that represents a more complex symbol or concept, which is composed out of lower concepts, and maintains its similarity to these components [6]

3.2.1 Bundling via Thresholded Addition.

Depending on a hypervector's composition, a straight-ahead component-wise addition can throw its components out of bounds. For instance, if two MAP hypervectors are bundled via basic component-wise addition, the resulting hypervector would have components from the set {-2,0,2}. Likewise with BSC, bundling two hypervectors via basic component-wise addition would result in a hypervector with components from the set {0,1,2}. When bundling, we need the resulting hypervector to only have components from the set that was used to generate it. This is accomplished by establishing thresholding parameters that restrict the resulting hypervector's component values to the respective model's component set.

In BSC, thresholded addition works by outputting 0 if both components are 0, outputting 1 if both components are 1, and in the case that there is a 0 component and a 1 component, the corresponding component a from a third random dense binary hypervector with components from the set {0,1} that is included in this special addition operation is taken as output instead. A simple example looks like this (where V_1 and V_2 are portions of the input hypervectors being bundled, V_3 is a portion of the tie-breaker hypervector, and V_r is a portion of the resulting hypervector):

$$V_1 \begin{bmatrix}1\\0\\1\\0\end{bmatrix} + V_2 \begin{bmatrix}0\\0\\1\\1\end{bmatrix} \begin{matrix}\nRightarrow\\ \Rightarrow\\ \Rightarrow\\ \nRightarrow\end{matrix} V_r \begin{bmatrix}1\\0\\1\\0\end{bmatrix} \begin{matrix}\Leftarrow\\ \nLeftarrow\\ \nLeftarrow\\ \Leftarrow\end{matrix} V_3 \begin{bmatrix}1\\1\\0\\0\end{bmatrix} \tag{1}$$

This generates a new hypervector V_r that retains approximately 3/4 similarity to V_1 and V_2.

In MAP, thresholded addition is similar to BSC. It is a component-wise addition using the sign function with 0 ties broken by a third random dense bipolar hypervector with components from the set {-1,1} that is included in the operation [6]. The sign function preserves the sign throughout the thresholding operation. A simple example looks like this (where V_1 and V_2 are portions of the input hypervectors being bundled, V_3 is a portion of the tie-breaker hypervector, and V_r is a portion of the resulting hypervector):

$$V_1 \begin{bmatrix}1\\-1\\1\\-1\end{bmatrix} + V_2 \begin{bmatrix}-1\\-1\\1\\1\end{bmatrix} \begin{matrix}\nRightarrow\\ \Rightarrow\\ \Rightarrow\\ \nRightarrow\end{matrix} V_r \begin{bmatrix}1\\-1\\1\\-1\end{bmatrix} \begin{matrix}\Leftarrow\\ \nLeftarrow\\ \nLeftarrow\\ \Leftarrow\end{matrix} V_3 \begin{bmatrix}1\\1\\-1\\-1\end{bmatrix} \tag{2}$$

This generates a new hypervector V_r that retains approximately 3/4 similarity to V_1 and V_2.

3.3 Binding

We cannot rely on bundling alone, however, because during recursive application, the information about combinations of the initial objects (e.g., grouping) is lost since, e.g., $(a + b) + (c + d) = (a + c) + (b + d) = \cdots = a + b + c + d$ [6]. There needs to be a way to establish that a was with b, that c was with d, and the sequence in which they appear in the first expression. We can accomplish this through what is called *binding*.

3.3.1 Binding via Multiplication/XOR. In a situation where the order of operand hypervectors is irrelevant, if we want to establish that a and b were grouped together, for MAP hypervectors, we bind a and b via component-wise multiplication [6]. In the case of BSC hypervectors, we bind a and b via component-wise XOR [6]. This effectively creates a new hypervector that is dissimilar to its input hypervectors and represents their grouping because its only replicable when the same input hypervectors go through the same operation [6].

3.3.2 Binding via Permutation. If we want to encode the sequence in which hypervectors appear, we can use a fixed permutation operation ρ on the hypervector coordinates. In respect to a and b, the sequential context of $(a + b)$ can be encoded by permuting a once, leaving b as it is, and then binding them via multiplication/XOR. This distinguishes the sequence (a+b) from (b+a) [3].

3.4 Item Memory and Associative Memory

In HDC, encoded hypervectors are stored in a matrix which we refer to as the item memory but a system that is built on HDC might employ an auto-associative memory [6]. In an auto-associative memory, addresses and its contents are one and the same, therefore it is commonly also referred to as a content-addressable memory.

4 Natural Language processing

4.1 Language Identification

Halseth and Kanerva present a method of language identification that is known as random indexing using the MAP model in their paper [3]. Random indexing represents information by projecting data onto random hypervectors [3]. The goal of this implementation is identifying languages given the frequency of three character sequences under the assumption that given enough text, the character distribution will approach the distribution of the language. This approach is fast, scalable, and space efficient [3].

4.1.1 Mapping. In order to encode these three letter sequences, we need to first assign atomic hypervectors (10,000-dimensional MAP hypervectors) to each character that is to be encountered in the text. We will assume for the sake of example that we are analyzing text written in the Latin alphabet, so the atomic hypervectors would be a symbolic representation of the alphabet, plus one more that represents the space character.

4.2 Encoding

To begin encoding three letter sequences, atomic hypervectors are bound via permutation in accordance to their position in the three letter sequence, and then bound via multiplication to form a *trigram hypervector*. For instance, if we were encoding the three letter sequence "the", we would permute the atomic hypervector for "t" by rotating it two shifts left, permute the atomic hypervector for "h" by rotating it one shift left, take the atomic hypervector for "e" as it is, and then bind all of them via multiplication. This process can be

```
                       Rotation of coordinates
                 .->------------->------------->.
                /                                \
               /                                  \
T =    (+1 -1 -1 +1 -1 -1 . . . +1 +1 -1 -1) .    .
   H =    (+1 -1 +1 +1 +1 +1 . . . +1 -1 +1 -1) .
      E =    (+1 +1 +1 -1 -1 +1 . . . +1 -1 +1 +1)
-------------------------------------------------------
THE    =     (+1 +1 -1 +1 . . . . . . +1 +1 -1 -1)
```

Figure 3. Visualization of the permutative and multiplicative binding operations for the trigram encoding process.

observed in Figure 3. We then iterate over the text character by character, creating a trigram for every sequence of three letters. If the first five letters of a text are "three", a trigram would be created for "thr", then "hre", and then "ree" etc... The resulting trigrams are then bundled together to create a profile hypervector that has the frequency of the three letter sequences encoded into it. The encoded profile hypervectors are then stored in either an item memory or an associative memory. This algorithm was used to create profile hypervectors for 21 languages with 10,000 sentences of news material for each language[3]. In Figure 4, the profile hypervectors can be observed clustering with other languages that share similar three letter frequencies. The entire encoding process took just over 7 minutes when programmed in Python on a 64-bit, 2.70GHz (100MHz clock) Intel processor, 4 cores, and 32GB of 1600 MHz memory computer [3]. Letters outside of the 26 in the Latin alphabet were replaced by their Latin equivalents by hand-coding and using the Unidecode package [3].

4.2.1 Identification. Input data can then be encoded in the same manner (but is not stored) and compared to the profile hypervectors in the item memory/associative memory via cosine similarity. The most similar profile hypervector is taken as a positive identification. The detection success rate of this method is 97.3 percent [3].

5 tinyML

5.1 Biosignal Classification

HDC is extremely robust in the presence of failures[11]. On top of this, its viability at low form levels make it ideal for things such as biosignal processing. Rahimi et al[1, 11] lay out an HDC encoding module utilizing a distributed BSC approach for electromyographic (EMG) signal-based prosthetic hand gesture classification at a nanometer form level with extremely low power consumption. .

5.1.1 Layout. The signal encoding layout can be observed in Figure 5. Two channels are shown, however, an indefinite amount of channels can be implemented.

5.1.2 Mapping. To begin, atomic hypervectors must be assigned. Random BSC hypervectors are generated for each

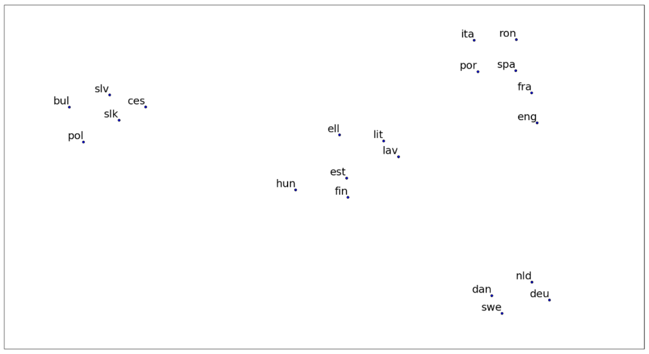

Figure 4. 10,000-dimensional language vectors for 21 languages roughly cluster based on the known relations between the languages. The Language Vectors were based on letter trigrams and were projected onto a plane using t-stochastic neighbor embedding which is a compression algorithm for visualizing high-dimensional data in low-dimensional spaces [3]. Observe the cluster of Portuguese (POR), Spanish (SPA), Italian (ITA), and English (ENG), as well as the cluster of Danish (DAN), Swedish (SWE), Dutch (NLD), and German (DEU).

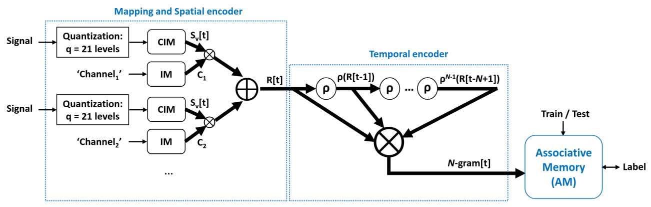

Figure 5. Example of an HD architecture for hand gestures learning and classification from EMG biosignals [11]

channel and stored in the item memory (IM). An additional atomic hypervector that is designated as the *seed* is also generated and stored in the continuous item memory (CIM). The seed is used to generate hypervectors for each possible incoming quantized signal strengths (between 1-21) in a manner such that the stronger the signal, the closer to being orthogonal with the seed its hypervector is. This is accomplished by randomly choosing $D/2$ (where D is Dimensions) components of the seed and splitting them into $q-1$ (where q is the quantized signal strength) groups which equally contain $(D/2)/(q-1)$ components[11].

5.1.3 Encoding. The encoding process produces what are called N-gram hypervectors, which are composed of bundled hypervectors. The concept of an N-gram hypervector is similar to the trigram hypervectors from the natural language processing model in that they are records of hypervector combinations. When a signal comes in, it transforms the seed by feeding it through $(D/2)/(q-1)$ (where D is dimensions and q is the quantized signal strength). The resultant hypervector ($S_v[t]$) is then bound via XOR (denoted in the diagram as ⊗) to its respective channel hypervector (C_n), and then bundled (denoted in the diagram as ⊕) with the hypervectors produced by the other channels to create a record hypervector ($R[t]$). The record hypervector represents that specific combination of those signal strengths on their respective channels. This whole process can be observed in Figure 5 in the section labeled "Mapping and Spatial encoder".

Record hypervectors are generated with respect to the sampling rate of the sensors and the length of the training period, meaning that a stream of record hypervectors is constantly being generated as time goes on. The record hypervectors, however, dont have a sequential context, only a combinatorial one. There needs to be a way to encode what order the record hypervectors were generated in. In order to make a record of signal/channel combination sequences, the record hypervectors are bound via permutation such that the first record hyper vector is rotated N-1 times, the second N-2 times, so on and so forth. The permuted hypervectors are then bundled with each other to create an N-gram hypervector that is only replicable when the same signal strengths hit the same channels in the same sequence. The N-grams are defined as N-gram$[t] = \prod_{i=0}^{N-1} \rho^i(R[t-i])$ The permutation sequence can be observed in Figure 5 in the section labeled "Temporal encoder". The resulting N-gram hypervectors are then stored in the associative memory for easy comparison operations. The whole module is referred to as the "Temporal-Spatial encoder" [11].

5.1.4 Classification.

After encoding, the associative memory has an array of N-gram hypervectors stored, each corresponding to a different gesture. When a patient attempts a gesture that was encoded, the EMG signals are fed into the Temporal-Spatial encoder which produces an N-gram hypervector. The N-gram hypevector is then compared via Hamming distance to the N-grams in the associative memory. The closest one is taken as a positive classification and its respective gesture is executed.

5.1.5 Power Consumption.

Rahimi et al[1] physically implemented an identical EMG gesture classification model on a Parallel Ultra-Low-Power (PULP) system on a chip (SoC) called Mr.Wolf[10]. Mr.Wolf is a microcontroller class RISC-V core augmented with an autonomous IO subsystem for efficient data transfer from a wide set of peripherals and is implemented in a 40-nm LP CMOS technology[10]. RISC-V is an instruction set that is mainly aimed at implementation on tiny devices[8]. CMOS stands for complementary metal oxide semi-conductor and is a techonology that is utilized today's integrated circuits due to its low power consumption[9]. The implementation was capable of remembering 11 gestures, with a power consumption of .0832mJ per classification, and a classification accuracy of 85 percent[1].

5.2 In-Memory HDC

5.2.1 Resistive Memory Devices.

In the current computing paradigm, a major topic of interest is subverting the energy and time consumption of shuttling data back and forth between the memory and the CPU. This is commonly referred to as the Von-Neumann bottleneck. In-Memory computing aims to combat this by utilizing resistive memory devices. A resistive memory device is a nano-scale resistor-type circuit component that can store data by defining bits with corresponding bipolar resistance states[7].

5.2.2 Crossbar Array.

We can use resistive memory devices to create a circuit that stores information and does computations with it in the same location, hence the name "In-Memory". These circuits are known as crossbar arrays. The structure of a crossbar array consists of nodes where a resistive-memory device is sandwiched between perpendicular running nano-wires with one direction running below the device as an input drive and the other direction running on top as an output drive. Data can be written and read via application of electrical fields of varying natures depending on the class of resistive memory device being used [12]

5.2.3 Hyperdimensional Crossbar.

Rahimi[5] proposes a way to leverage in-memory computing in tandem with HDC for very low-latency, low-power, and low-form factor system using PCM resistive memory devices. PCM techonology operates by switching a material between an amorphous state (high resistance) and a crystalline state (low resistance)[5]. The system works by storing the atomic hypervectors in the nodes of a crossbar array that acts as an item memory. The item memory can be pulsed with low voltage down certain input drives to send the atomic hypervectors down the output drives into CMOS gates which perform the bundling and binding. The encoded hypervectors are then stored in the nodes of an associative memory crossbar array for easy comparison operations.

6 Conclusion

In this paper, we have explained hyperdimensional computing as it was laid out by Rahimi et al. [6] and Kanerva[4]. We explored how Kanerva [3] proposed how it might be employed for natural language processing and how Rahimi et al. [1, 11] deploy a prosthetic hand gesture classification on a nano-scale ultra low power system. Finally, we covered in-memory computing and how Rahimi et al.[5] propose HDC might be done in-memory.

Acknowledgments

I would like to thank Elena Machkasova for guiding me through the writing process and pushing me to understand this topic. I would also like to thank Peter Dolan for taking the time to help me understand this complex topic, and

Wenkai Guan for his leading of the senior seminar course. Finally, I would like to extend thanks to alumn Joseph Walbran for his valuable feedback and interest in my topic.

References

[1] Benatti, S., Montagna, F., Kartsch, V., Rahimi, A., Rossi, D., and Benini, L. Online learning and classification of emg-based gestures on a parallel ultra-low power platform using hyperdimensional computing. *IEEE Transactions on Biomedical Circuits and Systems 13*, 3 (2019), 516–528.

[2] Dodd, P., and Massengill, L. Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Transactions on Nuclear Science 50*, 3 (2003), 583–602.

[3] Joshi, A., Halseth, J., and Kanerva, P. Language recognition using random indexing, 2015.

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

[5] Karunaratne, G., Gallo, M. L., Cherubini, G., Benini, L., Rahimi, A., and Sebastian, A. In-memory hyperdimensional computing, 2020.

[6] Kleyko, D., Rachkovskij, D. A., Osipov, E., and Rahimi, A. A survey on hyperdimensional computing aka vector symbolic architectures, part i: Models and data transformations. *ACM Comput. Surv. 55*, 6 (dec 2022).

[7] Lin, W.-P., Liu, S.-J., Gong, T., Zhao, Q., and Huang, W. Polymer-based resistive memory materials and devices. *Advanced Materials 26*, 4 (2014), 570–606.

[8] Lu, T. A survey on risc-v security: Hardware and architecture, 2021.

[9] Martins, R., Nathan, A., Barros, R., Pereira, L., Barquinha, P., Correia, N., Costa, R., Ahnood, A., Ferreira, I., and Fortunato, E. Complementary metal oxide semiconductor technology with and on paper. *Advanced materials (Deerfield Beach, Fla.) 23* (10 2011), 4491–6.

[10] Pullini, A., Rossi, D., Loi, I., Tagliavini, G., and Benini, L. Mr.wolf: An energy-precision scalable parallel ultra low power soc for iot edge processing. *IEEE Journal of Solid-State Circuits 54*, 7 (2019), 1970–1981.

[11] Schmuck, M., Benini, L., and Rahimi, A. Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory. *J. Emerg. Technol. Comput. Syst. 15*, 4 (oct 2019).

[12] Slesazeck, S., and Mikolajick, T. Nanoscale resistive switching memory devices: a review. *Nanotechnology 30*, 35 (jun 2019), 352003.

[13] Wang, F., and Agrawal, V. D. Single event upset: An embedded tutorial. In *21st International Conference on VLSI Design (VLSID 2008)* (2008), pp. 429–434.