Parallel Architectures for Hyperdimensional Computing

Ryan Moughan

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-67
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-67.html

May 13, 2021

Parallel Architectures for Hyperdimensional Computing

by Ryan Moughan

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Bruno A. Olshausen
Research Advisor

May 9, 2021

(Date)

* * * * * * *

John DeNero
Second Reader

May 13, 2021

(Date)

Parallel Architectures for Hyperdimensional Computing

Abstract

Parallel Architectures for Hyperdimensional Computing

by

Ryan Moughan

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Bruno Olshausen, Chair

Hyperdimensional computing represents a relatively different way to approach artificial intelligence than what has become mainstream. It focuses on the use of connectionist paradigms with a set of simple algebraic operations to form a powerful framework to represent objects. In this thesis, we show how these algebraic operations can be used to build parallel algorithms for hyperdimensional language models. We first ask the question of why this is useful from both an engineering and scientific point of view. Then we show how different parallel algorithms can be built to answer each of these questions. One algorithm focuses on distributing the data to different workers in order to minimize the runtime, while the other algorithm focuses on distributing the different embedding techniques in order for parallel learning to occur in a process inspired by the brain. Both algorithms are able to achieve superior efficiency, however the one that distributes the data over multiple workers is ultimately the most efficient. We further compare these methods to the popular word2vec models and show how they are able to outperform them on one of the original metrics used to test word embeddings, the TOEFL test. Finally we describe our vision for future work, in particular the use of algorithms for learning multimodal embeddings in parallel with joint hyperdimensional models of language and vision.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

There are many people I would like to thank for helping me get to where I am today. Chief among them are my parents, who have continually supported me throughout the often perilous adventure of my education. I would also like to express my great appreciation for Professor Olshausen, who helped me get into research and advised me on this thesis. I am likewise grateful to many others at the Redwood Center for Theoretical Neuroscience, especially Pentti Kanerva, for giving me guidance and many interesting conversations throughout my time here. I would further like to express my deepest gratitude towards Professor Denero and Professor Garcia for building such an incredible teaching community in the EECS department, of which I have benefited from immensely. Lastly, I would like to thank the innumerable family members, friends, and random students I have met while having the privilege of attending this great school for the support and inspiration they have provided.

Chapter 1

Introduction

Traditional computation is based on rigid architectures where each bit plays a significant role. This came about as a result of the natural development of the fields of electrical engineering and computer science. The first computers had minuscule memories compared with today's machines, and as a result computer programs had to be written to maximize the efficiency of every bit. When the field of cybernetics and artificial intelligence rose to prominence in the 1950s, many of the models inherited the limitations of traditional computation [6]. These models still used the small n-bit architectures that the traditional von-Neumann architecture was based off. One such example of this was the first work in artificial intelligence, the Logic Theorist, which ran on a 40-bit JOHNNIAC computer [11, 27]. This program attempted to write proofs by representing mathematical expressions as lists that could then be solved using tree search [11, 27]. In many ways, this type of architecture is in contrast with the entity they were trying to replicate, the brain. While the exact number will vary by individual, the brain has around 80 billion neurons with 150 trillion synapses [1]. This constitutes a very different foundation to build an architecture from than the limited hardware in the 1950s.

In contrast with the framework of traditional computation, a new paradigm called connectionism came from the field of psychology. Connectionism focused on using distributed representations as the basis for knowledge [33]. Instead of the third bit of a 16-bit unsigned integer encoding the 2^2 value, the representation may be scattered throughout an encoding of 100 bits. In this design, each individual bit no longer has an understandable significance, but collectively they form a robust and error-tolerant representation [33]. Computational models such as Hopfield Networks arose from this idea as a way to store these distributed representations, and they've been shown to be promising models for parts of the brain such as the hippocampus [14, 29]. Perhaps the most famous connectionist design is the multilayer Perceptron, commonly referred to now as the deep neural network [32]. While these models are encouraging, the capacity to learn and build robust distributed representations isn't enough by itself. This has been pointed by multiple works that challenge whether or not these architectures can solve one of the original problems posed to artificial intelligence, symbolic reasoning [9, 16].

Rising to the challenge of expanding on connectionism to incorporate symbolic logic was hyperdimensional computing, also known as Vector Symbolic Architectures (VSAs) [9]. Hyperdimensional computing represents quantities as vectors in extraordinarily high dimensions of space, usually 1,000 - 10,000 [18]. The elements of these vectors can be composed in a variety of ways, including binary, ternary, and complex values [18]. What is important is the properties that arise from this type of representation. The first property is the robustness of the representations. Lose a single bit in a 32-bit architecture and the entire system could fail, lose a single bit in hyperdimensional computing and with a high probability, the system will remain intact [19]. This is because of the redundant and distributed representation across all the bits. A system organized in such a way is known as a *holographic* or *holistic* representation, as it spreads information throughout the components of its system [19]. This type of organizations seems to mirror that of the brain, which has relatively large circuits composed of many neurons [20]. Another property is the randomness when building the representations. While every computer may have the exact same representation of the number 128 or the logo for a popular company, it is highly unlikely our brains do [20]. Therefore it is important to consider how architectures can be built that give rise to similar behavior despite potentially different initial conditions. Hyperdimensional computing builds randomness as an assumption in its framework and leverages properties of it when constructing representations [20].

Once the architecture is constructed, hyperdimensional computing defines a set of operators over its space that is both highly efficient and allows for complex data structures to be built. The three operators that define the algebra over the space are multiplication, addition, and permutation [8]. While simple in nature, when used in combination with the holographic representation of the data, they allow for a powerful and robust form of symbolic computation. For example, a seminal work in hyperdimensional computing showed how these operations can build representations that are able to understand analogies in language, such as answering the question, "What is the dollar of Mexico?" [21]. These operations are also able to be done with a high degree of efficiency, with learning not needing to calculate expensive gradients on the data. They also have the necessary properties that could allow them to be applied in parallel architectures.

An important property of the nervous system is its ability to operate in parallel [13]. After all, if neurological processes didn't happen in parallel, it would be incredibly hard for organisms to survive. The very act of reading this sentence is dependent upon an innumerable amount of computational complexity to organize and coordinate visual perception and symbolic reasoning, not to mention the many underlying actions needed to sustain a functioning body. The brain can thus be thought of as a parallel machine that is able to integrate simultaneous percepts into sequential actions [28, 35]. This ability to operate many different processing tasks at once is known as multiple-instruction parallelism in computer science. It is widely accepted that the brain operates with this types of parallelism, and we therefore think it is an important goal of any model inspired by cognition to also demonstrate such capabilities [13].

In this thesis we will cover the basics of hyperdimensional computing with a particular emphasis on their application to natural language processing. Then we will show how the learning in hyperdimensional computing lends itself to a parallel computing framework, specifically focusing on two models of language previously used in Kanerva et al. [34]. Here language represents a useful medium to model as a parallel process given the multiple independent, parallel components that some believe are needed for comprehension [15, 16]. This parallel framework we introduce will include a detailed explanation of an architecture we call high-performance hyperdimensional computing, as well as an architecture that allows for learning multiple embeddings in parallel. Finally we will discuss how our results can be applied to other subfields of hyperdimensional computing and the broader implications on the discipline.

Chapter 2

Background

We begin our background by giving a survey of the fundamentals of hyperdimensional computing. We then discuss the specific applications that have used it in various tasks for natural language processing. Lastly we discuss the basics of parallel computation as they apply to this thesis.

2.1 Overview of Hyperdimensional Computing

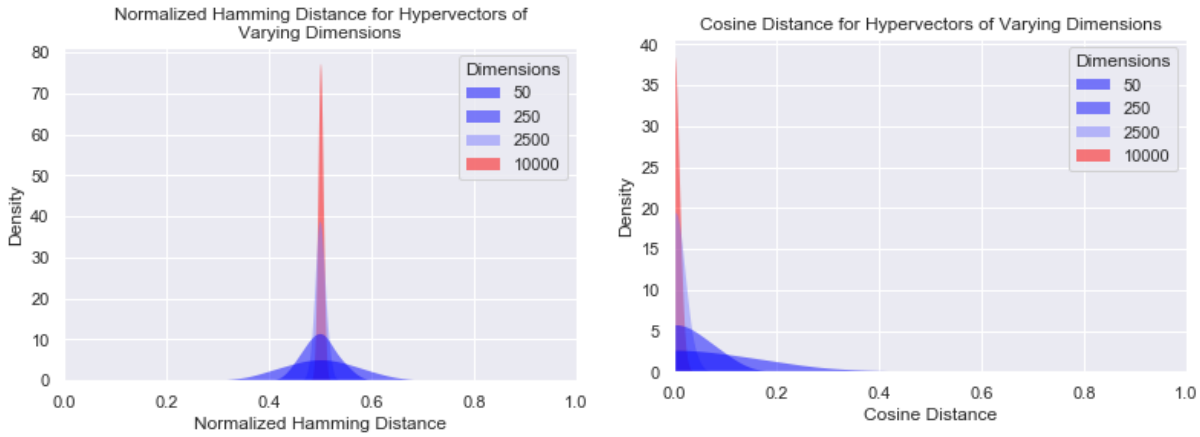
Hyperdimensional computing is a framework in which the atomic unit of computation is a vector with extremely high dimensionality, often 1,000 - 10,000 [18]. This type of vector is referred to as a *hypervector* and the overall space is referred to as a *hyperspace* [20]. Each hypervector generally represents a symbol, which can be anything from a word to a position in an image. The hypervector can be composed of a variety of different elements, including bipolar vectors that are randomly sampled from {-1, 1}, binary vectors that are randomly sampled from {0, 1}, and k-sparse ternary vectors that are mostly zeros but randomly sample k 1s and k -1s [20]. Each different way of defining the elements corresponds to a slightly different framework with small differences in how the operators we will discuss are implemented. In this work we will use k-sparse ternary hypervectors.

When defining a new way to represent quantities, it is important to also define a way to compare them. With hyperdimensional computing, this is traditionally done in one of two ways. The first is to use the Hamming distance, which measures the number of elements or bits in which two vectors differ:

$$d_{\text{Hamming}}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} \mathbb{1}\{\mathbf{x}_i \neq \mathbf{y}_i\}$$

The second way to compare vectors is using the cosine similarity, which represents the normalized distance between the vectors in the inner product space:

$$d_{\cos}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{|\mathbf{x}||\mathbf{y}|}$$



(a) Normalized Hamming Distance

(b) Cosine Distance

Figure 2.1: Distribution of Cosine and Hamming Distances of Randomly Initialized Hypervectors

In this work we will primarily use cosine distance as our metric for comparing hypervectors. An important attribute of either metric is the dissimilarity of randomly-generated hypervectors. Random initialization is usually how hypervectors are created, and as such, a desirable property is that before learning any associations between entities, all hypervectors are dissimilar. This is where the dimensionality plays a key role. As is shown in Figure 2.1, increasing the dimensionality concentrates the randomly created hypervectors to be dissimilar with an extremely high probability. Then, when learning associations between quantities, it only takes a small movement away from the random initialization to have two hypervectors be significantly correlated with each other. This allows for relatively fast and efficient learning, which will be done using the operators for hyperdimensional computing.

There are three primary operators used in hyperdimensional computing: multiplication, addition, and permutation [8]. We will briefly introduce each operator, how it is computed, and what the general use is below.

Multiplication

Multiplication is used to bind hypervectors together. It is generally used to represent (or capture) role-filler relationships, such as creating a single entity that represents the currency of the United States of America by binding one hypervector that represents the broad notion of currency with another that represents the dollar [21]. The multiplication operator takes in two hypervectors and computes the element-wise XOR between them:

$$\mathbf{m} = \mathbf{x} \oplus \mathbf{y}$$

The result of the multiplication operator is a hypervector that is dissimilar to either of its operands. Note that for different frameworks of hyperdimensional computing, multiplication can be defined in different ways. These include the circular convolution and the element-wise product between hypervectors [8, 30]. Regardless of definition, the importance of the operator is its properties: that it is both invertible and distributes over addition.

Addition

Addition is used to superimpose hypervectors together. It is generally used to learn that two hypervectors are associated with each other. The addition operator takes in two hypervectors and computes the element-wise sum between the two:

$$\mathbf{a} = \mathbf{x} + \mathbf{y}$$

The result of the addition operator is a hypervector that is similar to both of its operands. Depending on the hyperdimensional computing framework, this sum can either be thresholded or kept as is [8]. The addition operator can be thought of as a way to form a set from the hypervectors that are being added, as well as a form of learning associations between hypervectors [20].

Permutation

Permutation is used to encode order in hypervectors. It is generally used to help build structural relationships between hypervectors. The permutation operator takes in a single hypervector and shuffles the dimensions, usually by shifting each element up or down a given number of positions:

$$\mathbf{p} = \rho(\mathbf{x})$$

The result of the permutation operator is a hypervector that is dissimilar to its operand. Note that the permutation operator can also be represented as a multiplication operator where the permutation is computed by calculating the matrix product of a vector with a shifted identity matrix. Following from this, an important property of the permutation operator is that it distributes over addition, such that $\rho(\mathbf{x} + \mathbf{y}) = \rho(\mathbf{x}) + \rho(\mathbf{y})$. In general it can be thought of as a rotation of the hypervectors into another part of hyperspace that is approximately orthogonal to the original one.

While simple, these operators collectively define a powerful algebra over the hyperdimensional space. They can be combined to form complex data structures seen in traditional computing, such as trees and linked lists [6, 20]. They are also versatile in the data they can represent, as they've been used for everything from image to text embeddings. In this work, we will focus on the use of hyperdimensional computing for language, but the methods discussed should apply to many other models used in the literature.

2.2 Hyperdimensional Language Models

One of the early uses of hyperdimensional computing came in the form of natural language processing. Latent Semantic Analysis (LSA) is a form of natural language processing that constructs embeddings for words based on how frequently they occur in similar contexts, often times defined as the same paragraph [23]. Each context is given a unique identifier that can be added to the word embedding. The hope is that words that occur in similar contexts have similar meanings. One of the issues with this approach is the dimensionality. The identifier for the context is generally a one-hot encoded vector, which means that to keep track of d contexts, a d-dimensional vector is needed [22]. For a corpus with a vocabulary of 80,000 words in 50,000 paragraphs, this approach requires a matrix of dimensions 80,000 x 50,000 to keep track of all the words and the contexts they could occur in. An improvement to this can be found through an idea called random indexing. This is a dimensionality reduction technique similar to hyperdimensional computing that also focuses on using randomly initialized, nearly-orthogonal hypervectors. By implementing hypervectors as the representation for contexts instead of one-hot encoded vectors, Kanerva et al. were able to achieve comparable results to the original LSA work at a fraction of the memory cost [22].

Building off their work of using random indexing with latent semantic analysis, Kanerva et al. developed more complex models of language using the algebra of hyperdimensional computing [34]. This was done by constructing what they called context and order vectors. Whereas LSA sought to define words that occurred in similar paragraphs, this new approach aimed to define an embedding in two different ways. The first way was a more narrow definition of the context that a word occurred in. Rather than just define the context as the paragraph a word occurred in, they now defined the context as the small group of words before and after a given word. In addition to this, they defined order vectors to consider the relative position of the words around a given word.

To demonstrate how these embeddings are calculated, consider the sentence "A white dog went for a walk." First a vocabulary is constructed of each unique word in the corpus, which in this case would be the six unique words in the sentence. Then label vectors are defined as randomly initiated k-sparse ternary hypervectors, which we will notate as $\overrightarrow{\text{word}}$. We will define the *focus word* as the word we are generating an embedding for and the *focus window* as the number of words to consider before or after the focus word. Then when the focus word is "dog" and the focus window is two, the context vector would be:

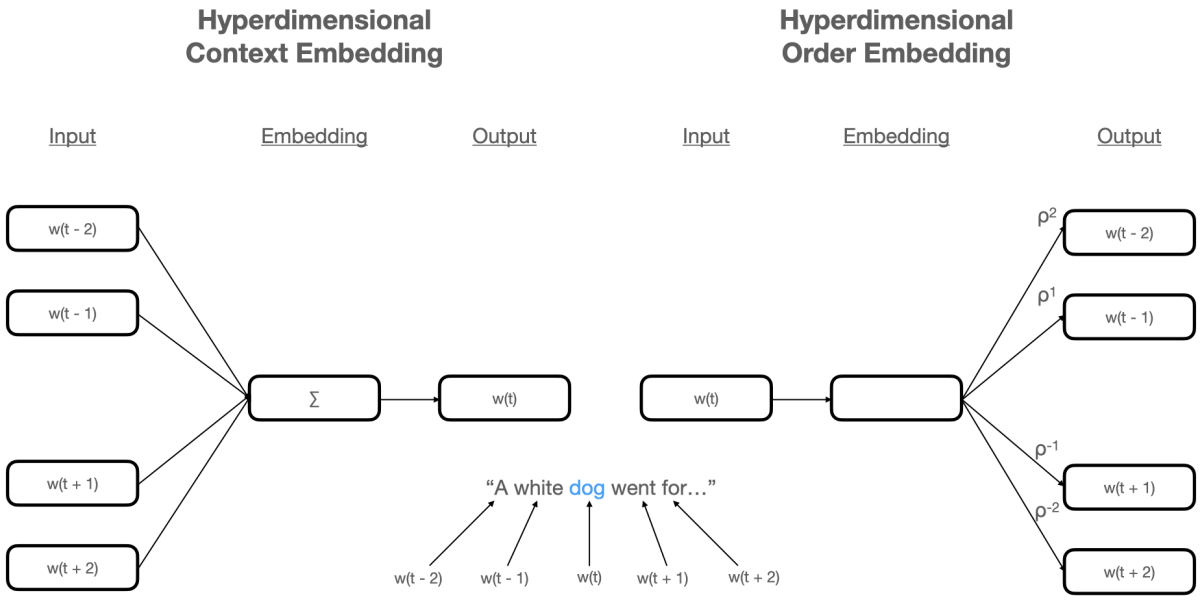


Figure 2.2: Hyperdimensional language embeddings

$\overrightarrow{\text{A}} + \overrightarrow{\text{white}} + \overrightarrow{\text{went}} + \overrightarrow{\text{for}}$. We denote this summed context vector as $[\overrightarrow{\text{dog}}]$. The order vector makes use of the permutation operator to encode order relative to the focus word. Using the same example sentence and focus word, this would be done by computing the sum: $\rho^{-2}(\overrightarrow{\text{A}}) + \rho^{-1}(\overrightarrow{\text{white}}) + \rho^{1}(\overrightarrow{\text{went}}) + \rho^{2}(\overrightarrow{\text{for}})$. This order embedding, denoted as $\langle\overrightarrow{\text{dog}}\rangle$, is then combined with the context embedding $[\overrightarrow{\text{dog}}]$ to form the overall embedding, $\overrightarrow{\text{DOG}}$.

There are many impressive aspects to this model. One in particular is its foreshadowing of better-known models in the years to come, the word2vec embeddings produced by the Skip-gram and Continuous Bag of Words (CBOW) models [25]. The similarity in ideas behind both models is outlined in Figures 2.2 and 2.3. While both groups of models use different learning methods, the goals of each type of model are similar. Both hyperdimensional context vectors and CBOW embed a given word in a high-dimensional space by considering the unordered set of words around the given word. Hyperdimensional order vectors and Skip-gram share a similar relationship. They both learn an embedding of a given word that now considers the order of words around it. Skip-gram uses softmax to determine the most likely word at a given position, whereas hyperdimensional order vectors use the permutation operator. Note that the permutation operators in Figure 2.2 appear to be the opposite of how we described calculating order embeddings in the previous paragraph. This is not a mistake. In order to recover a word at a given position from the embedding, the inverse permutation operation must be applied. Using the example from the previous paragraph, we

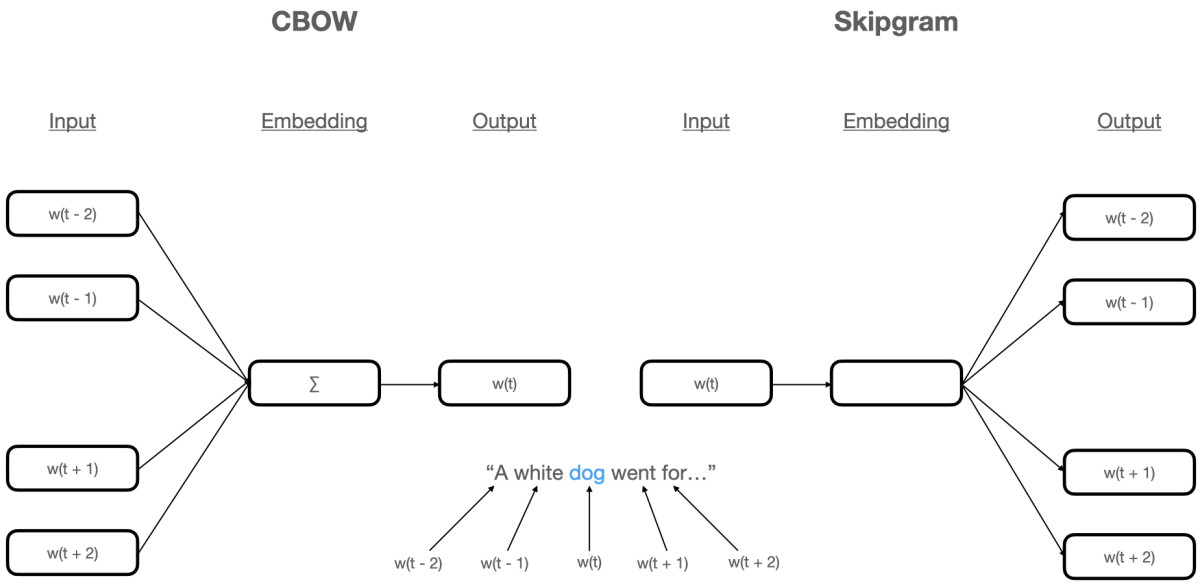


Figure 2.3: Traditional word2vec embeddings

could recover the word that occurred two positions before "dog" like: $\rho^2\langle\overrightarrow{\text{dog}}\rangle = \rho^2(\rho^{-2}(\overrightarrow{\text{A}}) + \rho^{-1}(\overrightarrow{\text{white}}) + \rho^1(\overrightarrow{\text{went}}) + \rho^2(\overrightarrow{\text{for}})) \approx \overrightarrow{\text{A}} + \text{noise}$. Then the importance of the previously established dissimilarity between randomly initialized hypervectors comes in handy, as this quantity should be able to be used to recover $\overrightarrow{\text{A}}$. This is also why the arrows point from the embedding to the output. Learning the embedding requires using the surrounding words shown as the output, however to emphasize the similarity to Skipgram we show how the most common surrounding words can be recovered from the learned embedding in a comparable way. While similar, one notable advantage that hyperdimensional models have over their word2vec counterparts is their relative simplicity, as the addition and permutation operators are the only computation required for learning the embeddings.

Many other works have used hyperdimensional computing as a model for various tasks in natural language processing. Of particular note is one that used it to understand the geometric structure of different languages [17]. In this work we will focus on the use of context and order embeddings with parallel computation, but the algebra of hyperdimensional computing should allow the same principles of parallelism to extend to most, if not all, other models.

2.3 Parallel Computation

In computer science, parallel computation refers to the ability to perform multiple calculations at the same time. It can generally exist at many different levels of abstraction in a computer program, and one of the most common is at the data level. This is appropriately named data-level parallelism [12]. A good example of of how data-level parallelism can be used is with the following example. Define $\mathbf{x}$ as a large sequence of numbers and f as some expensive function we wish to apply on $\mathbf{x}$ to find $\mathbf{y}$:

$$\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 & 4 & ... & 1000 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} f(1) & f(2) & f(3) & f(4) & ... & f(1000) \end{bmatrix}$$

Observe that there is no need to go through $\mathbf{x}$ sequentially, as there is no inherent dependence between elements in the list. So we can rewrite $\mathbf{y}$ as:

$$\mathbf{y} = \begin{bmatrix} f(1) & f(2) & f(3) & ... & f(500) \end{bmatrix} \Big\| \begin{bmatrix} f(501) & f(502) & f(503) & ... & f(1000) \end{bmatrix}$$

Note here that $\Big\|$ denotes the concatenation of two vectors. While a single processor could do all of this work, it can be done more efficiently by dividing the data among two processors. The first processor would determine $\begin{bmatrix} f(1) & f(2) & f(3) & ... & f(500) \end{bmatrix}$ and the second processor would calculate $\begin{bmatrix} f(501) & f(502) & f(503) & ... & f(1000) \end{bmatrix}$. Then the output of these two processors can be combined in order to achieve the correct result in half the time. More generally, this method allows for a roughly linear scaling of speed with the number of parallel workers as long as the data is sufficiently large.

Another form of parallelism is control or task-level parallelism [12]. This type of parallelism can be seen with the following example. Define once again $\mathbf{x}$ as a vector representing some sequence of numbers. Now say that we wish to perform two different functions, f and g, on this sequence:

$$f(\mathbf{x}) = \sum_i \mathbf{x}_i, \quad g(\mathbf{x}) = \prod_i \mathbf{x}_i$$

While a computer could do this sequentially, computing f first and g second, this is unnecessarily slow. Since the sum and product are independent, we can compute both functions in parallel on the single vector $\mathbf{x}$. Hence in this example, the code to compute the sum and product would be the two tasks operating in parallel for task-level parallelism. Note however that unlike before with data-level parallelism, task-level parallelism scales with the number of independent functions applied on the data. It is also generally slower than data-level parallelism, as the speedup is determined by the maximum runtime of all the independent functions running in parallel. If f and g take about the same amount of time, this means that there would be a 2x speedup. If f takes 20x the time of g, however, the speedup would be negligible. Because of this, task-level parallelism is often slower than data-level parallelism. However we include it in this thesis because of the brain's vast use of it [13].

Chapter 3

Methodology

We begin our methodology with an overview of the two parallel architectures developed in this thesis. We proceed to give a thorough explanation of how the data was processed for each architecture and conclude by giving a step-by-step walk-through of how the parallel algorithms were implemented.

3.1 Overview of Architectures

We experimented with two main architectures for parallel computation in hyperdimensional computing. The first architecture sought to maximize parallel performance for hyperdimensional computing, whereas the second sought to explore the use of parallelism in hyperdimensional computing from a framework more similar to the brain. All parallelism in this work was implemented in Python using a library called Ray. Ray is an open-source software project out of the RISE Lab at the University of California, Berkeley [26]. It has been noted for its ease of use and its capabilities with shared-object memory when operating with large amounts of data, which makes it an ideal library to use for natural language processing models that have large datasets such as this one [4].

High-Performance Architecture

The first architecture we developed is what we call high-performance hyperdimensional computing. The goal of this framework was to maximize the speed of the learning process in a simple and efficient way. To do so, this framework follows a similar structure to the traditional map-reduce paradigm in parallel computation [5]. A depiction of the general architecture is shown in Figure 3.1 below.

The key to this framework is that the learning process in hyperdimensional computing can be formulated as a data-level parallelism problem. For a corpus of millions, or even potentially billions, of words there are many different ways to potentially split the data up into multiple streams for parallel processing. In the case of the learning algorithms

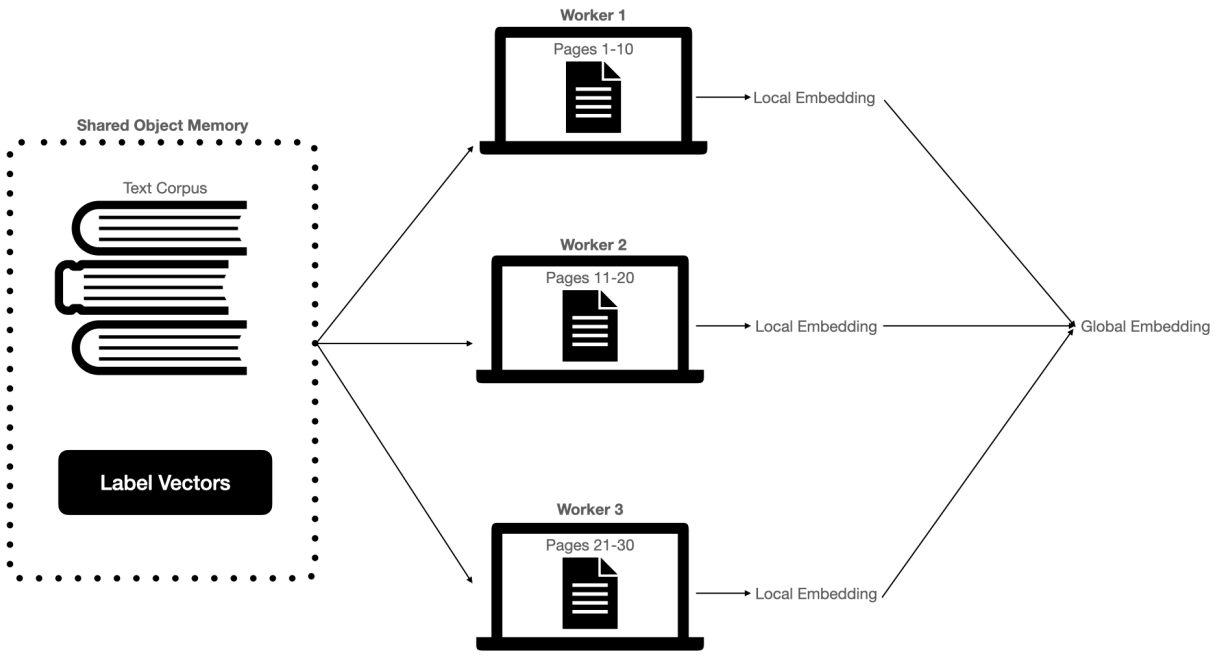


Figure 3.1: Overview of High-Performance Architecture

for hyperdimensional language modeling, each individual iteration of the algorithm is fairly inexpensive. Because of this, it isn't ideal to simply map the learning algorithm over the entire array of text. If programmed this way, the overhead of parallel computation dominates the cost of running the algorithm. Instead, it is better to divide the corpus into contiguous groups of text and run parallel processes on each of these, which is data-level parallelism.

Figure 3.1 demonstrates how a corpus is split into a series of contiguous texts that each worker can process in parallel. These workers then output what we call the local embeddings, which would be the context or order embedding for that local chunk of text. This works because there are no long range dependencies in the embedding, a given word is only dependent on the few words around it. For example, a word on page two of a book has no relationship with a word on page twenty of a book, and as such, an embedding can be computed on both at the same time. These local embeddings are then combined using the addition operator to create the global embedding over the entire corpus. Note that the use of the word "embedding" in the diagram refers to only one type of embedding. For example, it can refer to either $\langle \overrightarrow{\text{dog}} \rangle$ or $[\overrightarrow{\text{dog}}]$, but not both. So while each embedding processes the corpus in parallel, $\langle \overrightarrow{\text{dog}} \rangle$ and $[\overrightarrow{\text{dog}}]$ are learned sequentially and then combined into the overall word embedding. While highly efficient, this framework notably separates itself from one of the objectives of hyperdimensional computing given that the brain cannot temporally segment input data in this manner.

Parallel Learning Architecture

One of the original goals of hyperdimensional computing was to come up with a framework that more closely resembled processes in the brain [20]. Unfortunately, data-level parallelism breaks this relationship. Presented with the previous page of text in this thesis, it is highly unlikely that any human reader would have the innate ability to read different paragraphs in parallel. The human brain is endowed with a very different and altogether more complex form of parallelism, one that much more closely resembles task-level parallelism that can run many different processes at the same time. With this in mind, we developed a second architecture that focused on a task-level parallelism, as is shown in Figure 3.2.

This architecture takes advantage of the fact that that both types of embeddings for hyperdimensional language modeling use text as an input. With this setup, task-level parallelism is a natural framework that can be exploited. This is implemented by going through the entirety of the corpus sequentially, but for each word computing both the context and order embedding at the same time in parallel. This is notably different from the prior high-performance architecture, which exploited parallelism at the data level to break the corpus up into different segments and not process it sequentially. On the other hand, this architecture exploits parallelism at the instruction level while still processing the data sequentially. This more closely mimics the type of input the brain receives, which can generally be thought of as sequential time-series data that undergoes many parallel operations to process it. Ul-

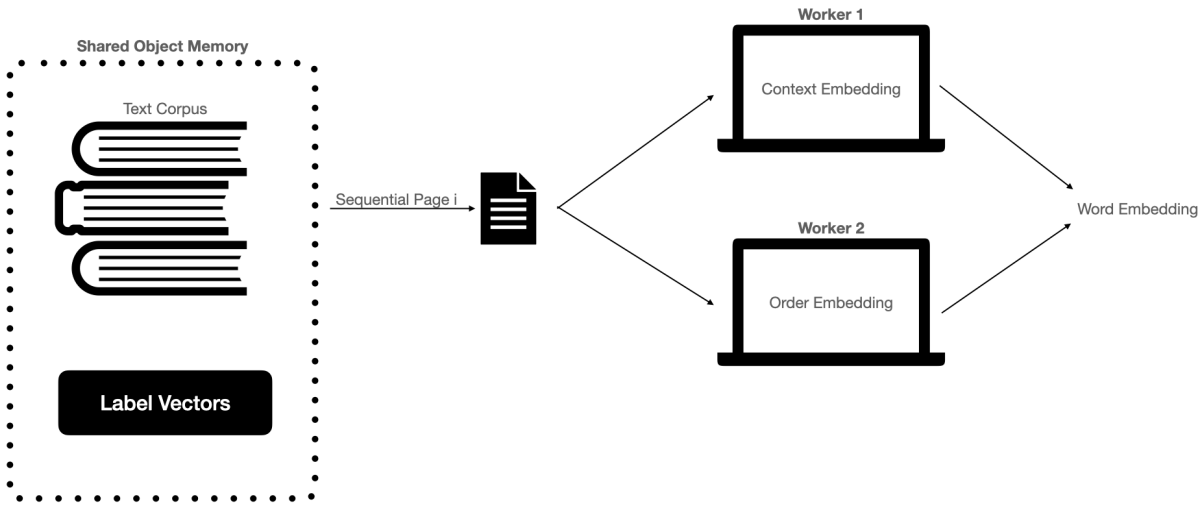


Figure 3.2: Overview of Parallel Learning Architecture

timately since there are generally less embedding techniques (instructions) than there are ways to break up the data, this architecture is generally slower than the high-performance architecture.

While the scope of this thesis is to focus on data-level and task-level parallel computation for hyperdimensional models on a single stream of input data, we hope that future works can expand on this to incorporate the type of multimodal data that the brain has to process in parallel. This is discussed further in the Conclusions.

3.2 Data Processing

Unfortunately the corpus used in the original work, the TASA corpus, is not publicly available and we could not gain access to it [34]. Therefore we chose a different corpus that is publicly available and has been used in many natural language processing studies, the British National Corpus (BNC), as our primary text for this work. The BNC is a corpus that was constructed from both written and spoken English in the late 20th century and is generally accepted to be a high quality corpus [36].

All texts that we used underwent the same cleaning process. This cleaning was done in Python with the help of the Natural Language Toolkit (NLTK) library [2]. The first step for cleaning the data was to remove any extra spaces in the corpus. Next we removed any punctuation from the corpus and kept only alphabetical words. This was done using the built-in *isalpha()* method in Python. Then we removed stop words from the corpus. Stop words are words that occur frequently in the English language but provide little semantic meaning, such as "the" or "is". While there are various ways to determine stop words, for consistency we used the built-in list of stop words in the NLTK library [2]. We also tested other techniques for generating stop words but ultimately found that they did not lead to better performance. After removing stop words from the corpus, we converted every word to lowercase using the built-in *lower()* method in Python. Following this, we converted the corpus to a vector if it wasn't already in that format. This was done using the built-in *split()* method in Python. The final step was to lemmatize each word, which is a way to convert a word to its base form. For example, the word "mice" would be changed to "mouse." This was done using the WordNet Lemmatizer in the NLTK library [2].

As an example of using this data cleaning process on some corpus, consider the sentence, "This is an example sentence for a thesis I am writing." The vector ["example", "sentence", "thesis", "writing"] would be the output after undergoing the data cleaning. After this, the vocabulary is constructed from the cleaned data and label vectors are created for each word in the vocabulary. This vector is what would then be used as an input to the learning algorithms described in the following sections.

3.3 Parallel Context Embedding

In this section we give a detailed description of how parallel context embeddings were generated for each word in the vocabulary of a corpus. The code for the embeddings can be seen in Algorithm 1 below. This code has two notable simplifications from what was tested. The first is that the focus window size is 1 to minimize the number of variables displayed, and the second is that conditional expressions checking for edge cases were omitted for visual clarity.

Before running the algorithm, the text array containing the vectorized version of the entire corpus was put in the shared object memory with the label vectors. Since both the text array and the label vectors only need to be read by each worker, they could be shared among all of them in order to save memory. For the high-performance architecture, a copy of the algorithm was then distributed to each worker with the corresponding start and end indices for the portion of the corpus they were responsible for. For the parallel learning architecture, a copy of the algorithm was distributed to a single worker to be run in parallel with the order embedding algorithm.

When the algorithm was run on each individual worker, they first defined their own local copy of the context embeddings. Since these values need to be written to, each worker needs to track its own copy so they cannot be efficiently shared in a parallel framework. Then each worker iterated through its respective range of the corpus, computing the local context embedding for each word as they go. After all the workers are complete, the global context embedding was computed by summing up the local embeddings.

Algorithm 1 Parallelized Context Embedding

```
1: Put text_array and label_vectors in Shared Object Storage
2: procedure Parallel Context Embedding(start_index, end_index)
3:     Initialize context_embeddings
4:     Define curr_word and next_word
5:     Define prev_vec, curr_vec, next_vec
6:     context_sum_extra ← prev_vec + curr_vec + next_vec
7:     for i from start_index to end_index do
8:         if curr_word in vocabulary then
9:             context_sum ← context_sum_extra - curr_vec
10:            context_embeddings[curr_word] += context_sum
11:        context_sum_extra -= prev_vec
12:        Reassign curr_word, next_word
13:        Reassign prev_vec, curr_vec, next_vec
14:        context_sum_extra += next_vec
       return context_embeddings
15: Put procedure in remote workers
```

The context embedding was computed as follows. Consider once again the sentence,"The white dog went for a walk" and a focus window size of two. Then for the word "dog", the context embedding, $[\overrightarrow{\text{dog}}]$, can be written as:

$$[\overrightarrow{\text{dog}}]_{extra} = \overrightarrow{\text{The}} + \overrightarrow{\text{white}} + \overrightarrow{\text{dog}} + \overrightarrow{\text{went}} + \overrightarrow{\text{for}}$$
$$[\overrightarrow{\text{dog}}] = [\overrightarrow{\text{dog}}]_{extra} - \overrightarrow{\text{dog}}$$

It may not be immediately apparent why it was useful to track this extra information $[\overrightarrow{\text{dog}}]_{extra}$. To see why, consider computing the context embedding for the next word, [went, which we can now write as:

$$[\overrightarrow{\text{went}}]_{extra} = \overrightarrow{\text{white}} + \overrightarrow{\text{dog}} + \overrightarrow{\text{went}} + \overrightarrow{\text{for}} + \overrightarrow{\text{a}}$$
$$[\overrightarrow{\text{went}}]_{extra} = (\overrightarrow{\text{white}} + \overrightarrow{\text{dog}} + \overrightarrow{\text{went}} + \overrightarrow{\text{for}}) + \overrightarrow{\text{a}}$$
$$[\overrightarrow{\text{went}}]_{extra} = ([\overrightarrow{\text{dog}}]_{extra} - \overrightarrow{\text{The}}) + \overrightarrow{\text{a}}$$
$$[\overrightarrow{\text{went}}] = [\overrightarrow{\text{went}}]_{extra} - \overrightarrow{\text{went}}$$

Writing the algorithm in this manner allowed for a fixed number of additions and subtractions regardless of focus window size. While this doesn't have a massive impact on the runtime for the context embedding, there is a similar trick that can be used for the order embedding that does have a significant effect. This will be discussed in the next section.

Two more important improvements were made beyond the naive way to write this algorithm. Depending on how the data was pre-processed, there are multiple ways to check if the current word is in the vocabulary. The best way we found to do so was to keep a hashmap of all words where the corresponding value represented whether or not the given word was in the vocabulary. This small change has an important effect, as the operation will be run for every word in the corpus and this improves the check from a naive $O(n)$ to $O(1)$ time, where n is size of the vocabulary. The other smaller change that was helpful was to cycle through variable assignment rather than always index into the text array. By this we mean to take advantage of the fact that for any subsequent iteration, most vectors, like prev_vec, can get their values from the corresponding next vector, like curr_vec. The only vector that needs to be found by indexing into the text array is the one that represents the farthest word in front of the focus word and in the focus window. While this may seem trivial, it adds up over millions of iterations and improved the speed by a nontrivial factor.

3.4 Parallel Order Embedding

The parallel order embedding algorithm followed many of the same design principles as the parallel context algorithm and can be seen below in Algorithm 2. The main difference is with the slightly more complex extra information it keeps track of. Since the permute function

Algorithm 2 Parallelized Order Embedding

```
1: Put text_array and label_vectors in Shared Object Storage
2: procedure PARALLEL ORDER EMBEDDING(start_index, end_index)
3:     Initialize order_embeddings
4:     Define curr_word, next_word
5:     Define prev_vec, curr_vec, next_vec
6:     order_sum_extra ← prev_vec + curr_vec + next_vec
7:     for i from start_index to end_index do
8:         if curr_word in vocabulary then
9:             order_sum ← order_sum_extra - curr_vec
10:            order_embeddings[curr_word] += order_sum
11:        order_sum_extra -= prev_vec
12:        Reassign curr_word, next_word
13:        Reassign prev_vec, curr_vec, next_vec
14:        order_sum_extra ← permute(order_sum_extra)^-1 + permute(next_vec)^1
       return order_embeddings
15: Put procedure in remote workers
```

is significantly more expensive than addition or subtraction, the algorithm is written to minimize the number of permutations necessary.

For an example of this, consider again the sentence, "The white dog went for a walk." With a focus window size of two, the first word to use all words in the window will be "dog," so we choose that as our example. Then we can compute its order embedding as:

$$\langle\overrightarrow{\text{dog}}\rangle_{\text{extra}} = \rho^{-2}(\overrightarrow{\text{The}}) + \rho^{-1}(\overrightarrow{\text{white}}) + \overrightarrow{\text{dog}} + \rho^{1}(\overrightarrow{\text{went}}) + \rho^{2}(\overrightarrow{\text{for}})$$
$$\langle\overrightarrow{\text{dog}}\rangle = \langle\overrightarrow{\text{dog}}\rangle_{\text{extra}} - \overrightarrow{\text{dog}}$$

While the use of this extra information may not be immediately apparent, consider the next word we want to compute the order embedding for, "went." We can now write it as:

$$\langle\overrightarrow{\text{went}}\rangle_{\text{extra}} = \rho^{-2}(\overrightarrow{\text{white}}) + \rho^{-1}(\overrightarrow{\text{dog}}) + \overrightarrow{\text{went}} + \rho^{1}(\overrightarrow{\text{for}}) + \rho^{2}(\overrightarrow{\text{a}})$$

Taking advantage of the distributivity of the permutation operator, this can be written as:

$$\langle\overrightarrow{\text{went}}\rangle_{\text{extra}} = \rho^{-1}(\rho^{-1}(\overrightarrow{\text{white}}) + \overrightarrow{\text{dog}} + \rho^{1}(\overrightarrow{\text{went}}) + \rho^{2}(\overrightarrow{\text{for}})) + \rho^{2}(\overrightarrow{\text{a}})$$

Substituting in the previously-calculated order vector for dog, we can arrive at:

$$\langle\overrightarrow{\text{went}}\rangle_{\text{extra}} = \rho^{-1}(\langle\overrightarrow{\text{dog}}\rangle - \rho^{-2}(\overrightarrow{\text{The}})) + \rho^{2}(\overrightarrow{\text{a}})$$
$$\langle\overrightarrow{\text{went}}\rangle = \langle\overrightarrow{\text{went}}\rangle_{\text{extra}} - \overrightarrow{\text{went}}$$

With this formulation of the algorithm, each order embedding only takes two permutation operators to calculate regardless of the focus window size. This can be seen with the $\langle \text{went} \rangle_{\text{extra}}$ variable while noting that $\rho^{-2}(\text{The})$ has already been calculated and can be reused. While this is a relatively small change, it has substantial effects on the runtime that will be discussed in the next chapter.

Chapter 4

Results

We begin our results by first proving that our parallel architecture is able to achieve similar accuracy to the models from the original work. We proceed to discuss how it is able to do so in a fraction of the time by leveraging parallel computation and finally discuss the cost of this framework in terms of memory.

4.1 Quality of Embeddings

To test the quality of the embeddings, we used the Test of English as a Foreign Language (TOEFL). This data set consists of a set of 80 multiple-choice questions, although four questions were omitted because the words were not in the corpus. The goal for each question is to pick one of four words that is most similar to the query word in the question. An example of five questions from the data set can be seen in Table 4.1, where the answer to each question is in bold. This test was kindly made available to us by Pentti Kanerva.

Question	Option A	Option B	Option C	Option D
enormously	appropriately	uniquely	**tremendously**	decidedly
slowly	rarely	**gradually**	effectively	continuously
tranquillity	**peacefulness**	harshness	weariness	happiness
feasible	permitted	**possible**	equitable	evident
bigger	steadier	closer	**larger**	better

Table 4.1: Example questions from the TOEFL test

We chose the TOEFL test as our accuracy metric in order to best compare our accuracy to multiple works. It was the metric for the original paper that described the hyperdimensional models we use, and thus we wanted to ensure the accuracy of our model by comparison [34]. However this test has also been used in many other works, including the seminal work

on Latent Semantic Analysis, among others [23]. Therefore we thought it was the most appropriate way to measure and compare the quality of our word embeddings with other works.

We present our results on the TOEFL test in Table 4.2. Since every question in the test has four multiple choice options, basic probability suggests that random embeddings should be correct 25% of the time, which we confirmed in our experiments. These random embeddings acted as our control group for the test. We then tested three different hyperdimensional language models. The first model was one of just context embeddings, the second was a model of only order embeddings, and the third was a model that consisted of both the context and the order embeddings. All three models performed substantially better than the random embeddings, indicating their ability to learn embeddings that differentiate words. These results are also similar to those achieved in the original work, although exact comparison is impossible due to the different corpora used [34].

Embedding	Random	Context	Order	Context + Order
Accuracy (%)	25.0	67.1	68.4	71.1

Table 4.2: Accuracies of Embedding Techniques

After the original work on hyperdimensional context and order embeddings was released, word2vec models were discovered and popularized. These models, as we briefly discussed, follow very similar ideas to those of the context and order embeddings. We therefore thought it would be interesting to do a comparison between the two groups of models, given that we are unaware of any other work that has done so. The results of this comparison are shown in Table 4.3.

Embedding	Best HD	Skip-gram 1	CBOW 1	Skip-gram 2	CBOW 2	Mikolov
Accuracy (%)	71.1	26.3	22.4	40.8	35.5	86.7

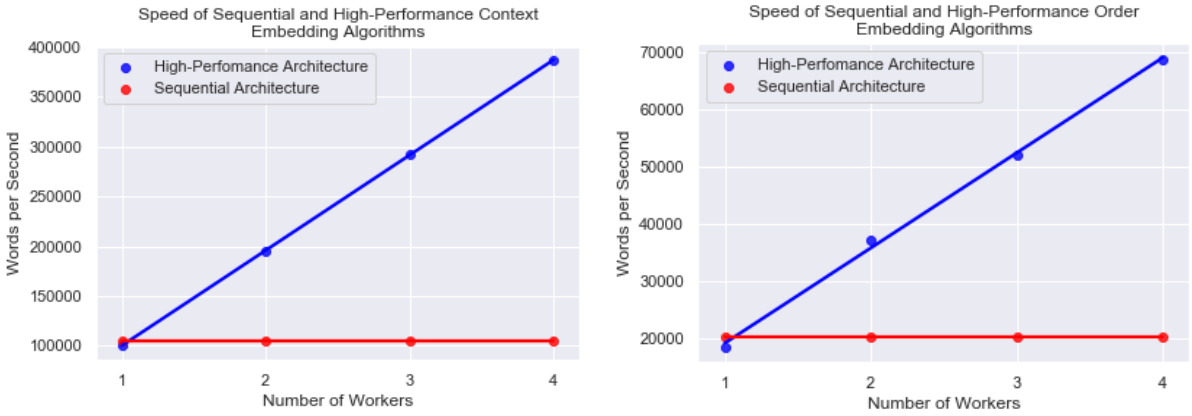
Table 4.3: Comparing Hyperdimensional Language Embeddings with Word2Vec

To compare these two forms of embeddings, we trained both Skip-gram and Continuous Bag of Words (CBOW) models using the GENSIM library on the same data that we used for our hyperdimensional models [31]. Initially, we set the hyperparameters of both word2vec models, which we call Skip-gram 1 and CBOW 1 in the table, to be as similar to the hyperdimensional models as possible. However the results were abysmal, with the word2vec models not being noticeable better than random embeddings. We therefore thought it would be best to tune the hyperparameters for each word2vec model and then test the performance. While this loses some of the ability to directly compare them with the hyperdimensional models, it gains the benefit of considering each type of model at its best. These tuned word2vec models are labeled as Skip-gram 2 and CBOW 2 in the table, and while they are

a sizable improvement over the first word2vec models, they still come nowhere close to any of the hyperdimensional models. Lastly, we tested the pre-trained word2vec model from Mikolov et al [25]. Note that this model is trained on a different data set, the Google News dataset. This corpus is approximately 100 billion words, or well over 1,000x the size of the dataset we used. Unsurprisingly, it is able to perform better than the hyperdimensional models given the amount of data it uses. However we find it promising how close the hyperdimensional models can come to its accuracy, indicating their ability to learn more efficiently than word2vec models.

4.2 Analysis of Runtime

After verifying the accuracy of our models, we measured their runtimes to determine the efficacy of the parallel architectures. In this section we primarily focus on the high-performance architecture, as it is the faster of the two parallel architectures we created. The performance of this architecture versus a traditional sequential architecture is shown in Figure 4.1. As we hoped, the parallelism allows for the algorithms to scale at a roughly linear rate with the number of workers assigned to the task. Note that it will not be perfectly linear, as there are various overheads for setting up and running the parallel computation. From these graphs, we can conclusively show how we are able to successfully implement parallelized versions of these algorithms. Impressively, they are able to compute at a rate of about 400,000 words/sec on a laptop with a four-core CPU, for a total of about 35 billion words per day. Scaled to the size and number of cores of modern supercomputers, this represents a truly tremendous amount of potential computational power, especially considering the efficiency at which the hyperdimensional algorithms seem to learn embeddings from the data relative to other embedding techniques like word2vec.



(a) Parallel Context Embedding Performance (b) Parallel Order Embedding Performance

Figure 4.1: Runtime performance for parallel architectures

For completeness, we also present the overall runtime for both architectures. Note that the high-performance architecture requires two passes over the corpus (one for each type of embedding algorithm), whereas the parallel learning architecture only requires one. Still, since the high-performance architecture is better able to divide the tasks among workers, it is the faster of the two architectures. This is to be expected, as this is generally the case with many parallel architectures in high-performance computing. Note that the reason for the relatively long runtime for the parallel learning architecture is the order embeddings. The parallel learning architecture learns both the order and context embeddings at the same time, but that means it won't finish until both are complete. Therefore when there is a significant difference in runtimes between the two, as there is with the context and order embeddings, the overall runtime is dominated by the slower embedding algorithm. Mathematically, the overall runtime for the parallel learning architecture can be thought of as maximum runtime of all the parallel algorithms it is running, and as such, the overall runtime was dragged down by the significantly slower order embedding algorithm.

Architecture	Total Runtime (min)
Sequential	54
Parallel Learning	47
High-Performance	14

Table 4.4: Overall Runtimes for Parallel Architectures

It's worth noting that the order embeddings can be almost as fast as the context embeddings, albeit at a cost to the memory used. Since the reason for the relative slowness of order embeddings is the permutation operator, all possible permutations of a given label vector can be computed and stored in a hash map. Then when a permutation is needed, instead of having to call a function, the hash map can be used to retrieve the appropriate permutation in $O(1)$ time. This effectively reduces the operations needed for the order embeddings to be the same as the context embeddings, resulting in approximately the same runtime. While this may sound expensive, we will discuss memory optimizations that can be done in the next section to make storing the label vectors negligible.

4.3 Analysis of Memory

Unfortunately the cost of a parallel architecture is generally the use of more memory. In the case of hyperdimensional computing, this is not a trivial cost. Given the natural of hyperdimensional computing, which by name is to use extremely high-dimensional vectors to represent objects, the use of memory for sequential architectures is fairly large. For parallel architectures, this large memory cost is exacerbated. Nevertheless, we did our best to minimize the amount of memory used. For the data-level and task-level parallel architectures,

both the text array containing the words in the corpus and the label vectors were in a shared memory that could be accessed by all workers. Then the only increase in memory cost for parallelism was for each worker to have their own copy of the embeddings. For the data-level architecture, this meant that each worker needed its own copy of the embeddings that corresponded to the learned embeddings on that portion of text. For the task-level architecture, this meant that each worker had its own type of embedding (context or order) that would be learned over the entire text. So while sequential architectures will pay a fixed memory cost of $O(\text{corpus} + \text{labels} + \text{embeddings})$, parallel architectures pay a memory cost of $O(\text{corpus} + \text{labels} + w \cdot \text{embeddings})$, where w is the number of workers.

We also made a small but important change to the data representation itself. Since our label vectors are static bipolar vectors, they are defined on the set $\{-1, 1\}^{\mathrm{d}}$. Given this, they can be easily represented with small signed integer data types, or even Boolean values with the proper adjustments made. This greatly reduces the memory overhead of the label vectors and makes it so that the vast majority of shared memory is consumed by the actual text array containing the corpus. We used a similar trick with reducing the amount of memory used by the semantic embeddings. Since these are dynamic, they are not theoretically bound on any set. However given the size of our corpus, we could determine a maximum possible value for them, which was less than that of a 32-bit unsigned integer. As such, we were able to use 32-bit integers to represent the embeddings, resulting in a 2x increase in memory efficiency. Collectively, these changes to the representation of the data drastically reduced the amount of memory consumed and made it so that the dominating constant term for the memory cost was the corpus size and the linear growth from the number of workers was significantly reduced.

Chapter 5

Conclusion

We begin our conclusion by summarizing our contributions in this thesis. We then conclude by discussing two ways this work can be used for future research, one for each architecture that was developed.

5.1 Summary

This work makes several key contributions. The most important contribution is to introduce the notion of parallelism in hyperdimensional computing and demonstrate how it can be used for modeling language. This was done in two different frameworks that sought to answer two different questions. The first framework sought to answer the engineering question of how to build the fastest hyperdimensional models and used data-level parallelism to do so. This architecture was able to successfully reproduce the accuracy of the original models at a fraction of runtime. The second framework used task-level parallelism to explain the more scientific question of how these models can learn different structures in parallel like the brain. While slower than data-level parallelism, this architecture demonstrated a key capability for any model inspired by the brain: how instruction-level parallelism can be successfully and efficiently implemented. For both of these tasks, we chose language as our medium to model. This was primarily because of the amount of work in hyperdimensional computing that has done so, but also because of the need for many processes to run in parallel in order to understand language [15, 16]. In doing all of this, we also established a new benchmark on a publicly available data set for future research to be based off, including an analysis that shows the promise of hyperdimensional embeddings relative to other techniques such as word2vec.

5.2 Future Work

Hyperdimensional computing is a small but growing discipline that encompasses everything from natural language processing to computer vision and robotics [7, 10]. Models in all of these areas can greatly benefit from the increased computational power demonstrated with data-level parallelism in this work. As such, the first path we envision future work exploring involves building off of our introduction of data-level parallelism to increase the computational capabilities of these hyperdimensional models. It is our hope that this will then lead to a increase in performance across models in all of these fields.

The second path we envision future work taking involves building off of the idea of learning multiple different embeddings in parallel. We primarily imagine this as expanding our task-level parallelism to include more than one stream of data. This seems natural since a crucial function of the brain is the ability to process and combine multiple streams of data in parallel. Language, after all, is not just about reading symbols on a page. Any high school student trying to learn a foreign language can attest to this. It's about being immersed in a foreign world full of percepts and having to communicate that to another individual. Models of language should consider this, and hyperdimensional computing presents a promising way to do so. In this work we considered words symbolically as a combination of their order and context, but it need not stop there. They could also include a pictorial representation of the object they represent to build a joint embedding space of both language and images. This has been done with large, complex neural networks, but has not yet be done with hyperdimensional computing [3, 24]. Since learning is primarily done through association in hyperdimensional computing, it presents a natural way to learn both words and images together. Moreover, building on what we have presented in this thesis, both embeddings could be learned at the same time, a process much more akin to how humans seemingly learn. This would create a truly powerful, yet simple and efficient multimodal model of language and vision.

Bibliography

[1] C.S. von Bartheld, J. Bahney, and S. Herculano-Houzel. "The search for true numbers of neurons and glial cells in the human brain: A review of 150 years of cell counting". In: *Journal of Comparative Neurology* 524.18 (2016), pp. 3865–3895.

[2] S. Bird and E. Loper. "NLTK: The Natural Language Toolkit". In: *Proceedings of the ACL Interactive Poster and Demonstration Sessions*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 214–217.

[3] Y. Chen et al. "UNITER: Learning UNiversal Image-TExt Representations". In: *European Conference on Computer Vision (ECCV 2020)*. Aug. 2020.

[4] D. Datta, R. Agarwal, and P.E. David. "Performance Enhancement of Customer Segmentation Using a Distributed Python Framework, Ray". In: *International Journal of Scientific & Technology Research* 9.11 (2020), pp. 130–139.

[5] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

[6] P.E. Frady et al. "Resonator networks, 1: an efficient solution for factoring high-dimensional, distributed representations of data structures". In: *Neural Computation* 32.12 (2020), pp. 2311–2331. DOI: 10.1162/neco_a_01331.

[7] S.I. Gallant and P. Culliton. "Positional binding with distributed representations". In: *2016 International Conference on Image, Vision and Computing (ICIVC)*. 2016, pp. 108–113.

[8] R. Gayler. "Multiplicative Binding, Representation Operators & Analogy". In: *Advances in analogy research: Integration of theory and data from the cognitive, computational, and neural sciences*. Jan. 1998.

[9] R. Gayler. "Vector Symbolic Architectures answer Jackendoff's challenges for cognitive neuroscience". In: *ICCS/ASCS International Conference on Cognitive Science*. Jan. 2003.

[10] R. Gayler, S. Levy, and S. Bajracharya. "Learning Behavior Hierarchies via High-Dimensional Sensor Projection". In: *Proceedings of the 12th AAAI Conference on Learning Rich Representations from Low-Level Sensors*. AAAIWS'13-12. Bellevue, Washington: AAAI Press, 2013, pp. 25–27.

[11] L Gugerty. "Newell and Simon's Logic Theorist: Historical Background and Impact on Cognitive Modeling". In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 50.9 (2006), pp. 880–884.

[12] W.D. Hillis and G.L. Steele Jr. "Data parallel algorithms". In: *Communications of the ACM* 29.12 (1986), pp. 1170–1183.

[13] G.E. Hinton and J.A. Anderson. *Parallel models of associative memory*. Cognitive science series: technical monographs and edited collections. L. Erlbaum, 1989. ISBN: 080580269X.

[14] J. Hopfield. "Neural Networks and Physical Systems with Emergent Collective Computational Abilities". In: *Proceedings of the National Academy of Sciences of the United States of America* 79 (May 1982), pp. 2554–8.

[15] R. Jackendoff. "A Parallel Architecture perspective on language processing." In: *Mysteries of meaning* 1146 (2007), pp. 2–22. ISSN: 0006-8993.

[16] R. Jackendoff. *Foundations of language: Brain, meaning, grammar, evolution*. Oxford University Press, USA, 2002.

[17] A. Joshi, J.T. Halseth, and P. Kanerva. "Language geometry using random indexing". In: *International Symposium on Quantum Interaction*. Springer. 2016, pp. 265–274.

[18] P. Kanerva. "Computing with 10,000-bit words". In: *Communication, Control, and Computing (Allerton), 2014 52nd Annual Allerton Conference on*. IEEE. 2014, pp. 304–310.

[19] P. Kanerva. "Fully Distributed Representation". In: *Proc. 1997 Real World Computing Symposium (RWC97, Tokyo)* (Nov. 1997), pp. 358–365.

[20] P. Kanerva. "Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors". In: *Cognitive Computation* 1 (2009), pp. 139–159.

[21] P. Kanerva. "What we mean when we say "What's the dollar of Mexico?": Prototypes and mapping in concept space". In: *2010 AAAI fall symposium series*. 2010.

[22] P. Kanerva, J. Kristoferson, and A. Holst. "Random Indexing of Text Samples for Latent Semantic Analysis". In: *Proceedings of the Annual Meeting of the Cognitive Science Society* 22 (2000).

[23] 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." In: *Psychological review* 104.2 (1997), p. 211.

[24] J. Lu et al. "ViLBERT: Pretraining Task-Agnostic Visiolinguistic Representations for Vision-and-Language Tasks". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019.

[25] T. Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *Proceedings of the International Conference on Learning Representations*. Jan. 2013, pp. 1–12.

[26] P. Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577. ISBN: 978-1-939133-08-3.

[27] A. Newell and H. Simon. "The logic theory machine–A complex information processing system". In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 61–79. DOI: 10.1109/TIT.1956.1056797.

[28] H. Pashler. "Processing stages in overlapping tasks: evidence for a central bottleneck." In: *Journal of Experimental Psychology: Human perception and performance* 10.3 (1984), p. 358.

[29] B. Pfeiffer and D.J. Foster. "Autoassociative dynamics in the generation of sequences of hippocampal place cells". In: *Science* 349.6244 (2015), pp. 180–183.

[30] T. Plate. "Distributed Representations and Nested Compositional Structure". PhD thesis. University of Toronto, 1994.

[31] R. Řehůřek. "Scalability of Semantic Analysis in Natural Language Processing". PhD thesis. Masaryk University, 2011.

[32] F. Rosenblatt. *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Tech. rep. Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[33] D. Rumelhart and J. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge: MIT Press, 1987, pp. 303–314.

[34] M. Sahlgren, A. Holst, and P. Kanerva. "Permutations as a Means to Encode Order in Word Space". In: (2008).

[35] M. Sigman and S. Dehaene. "Brain Mechanisms of Serial and Parallel Processing during Dual-Task Performance". In: *The Journal of neuroscience : the official journal of the Society for Neuroscience* 28 (July 2008), pp. 7585–98. DOI: 10.1523/JNEUROSCI.0948-08.2008.

[36] *The British National Corpus, version 3 (BNC XML Edition)*. Distributed by Bodleian Libraries, University of Oxford, on behalf of the BNC Consortium. 2007. URL: http://www.natcorp.ox.ac.uk/.