Representing Words in a Geometric Algebra

Arjun Mani
Princeton University
arjuns@princeton.edu

Abstract

Word embeddings, or the mapping of words to vectors, demonstrate interesting properties such as the ability to solve analogies. However, these models fail to capture many key properties of words (e.g. hierarchy), and one might ask whether there are objects that can better capture the complexity associated with words. Geometric algebra is a branch of mathematics that is characterized by algebraic operations on objects having geometric meaning. In this system objects are linear combinations of *subspaces* rather than vectors, and certain operations have the significance of intersecting or unioning these subspaces. In this paper we introduce and motivate geometric algebra as a better representation for word embeddings. Next we describe how to implement the geometric product and interestingly show that neural networks can learn this product. We then introduce a model that represents words as objects in this algebra and benchmark it on large corpuses; our results show some promise on traditional word embedding tasks. Thus, we lay the groundwork for further investigation of geometric algebra in word embeddings.

1 Motivation

Words are complex creatures; a single word has one (or multiple) semantic meanings, plays a particular syntactic role in a sentence, and has complex relationships with different words (simple examples of such relationships are synonyms and antonyms). The representations of words as vectors, and the ability of these models to capture some of this complexity, has been a seminal advance in the field of natural language processing. Such word embedding models (first introduced by Mikolov et al. in 2013 [1, 2]) learn a mapping from words to vectors by following the basic principle that "a word is characterized by the company it keeps". Using this idea the model is trained by sliding a window along a large text corpus and using the center word in each window to predict the words in the surrounding context [1]. Should two words appear in similar contexts, they are encouraged to have similar vector representations.

The vector space that results from training these models turns out to be surprisingly rich; in particular training such a model buys us two useful properties we should want from objects representing words. The first is a notion of *similarity*: computing the cosine similarity between two words gives us neighbors of a word that are semantically or syntactically similar. For example, neighbors of the word 'president' might be 'leader', 'government', 'presidency', and 'cabinet'. The second and perhaps more surprising property is that relationships between words are captured as vector directions in this space. This allows the the model to solve analogies; the canonical example is $v(\text{king}) - v(\text{man}) + v(\text{woman}) = v(\text{queen})$; because the model learns some the vector direction represented by a change in gender, the closest vector to the vector sum on the left is indeed the word *queen*. The ability to express word similarity and solve analogies already make vectors seem like a compelling way in which to represent words.

[1] This describes the "skip-gram" model, one of the two variants of word2vec. Please see 2.1 for details.

With this said, word2vec represents a (limited) attempt at capturing the complexity associated with words. If we were to step back and ask what might we want from objects representing words, we see that such a model does not capture several attractive properties. Here are some of these properties:

- **Hierarchy**: Hierarchial structure is an important feature of words (indicated by the organization of WordNet [3] into hierarchies). A space of objects representing words should be able to indicate that labrador is a type of dog, which is a type of animal. The analogy solving capabilities of word2vec might give us a weak *is-type-of* relationship, but ideally there should be an intrinsic way to represent hierarchy.
- **Part of Speech**: The role a word plays in a sentence (noun, verb, etc.) should be captured by the object representing that word. This is not captured by word2vec.
- **Closedness**: The principal operation in word2vec is the dot product, which reduces two vectors to a scalar. However, we might want some notion of closedness, where in addition to a dot product we have some operation $F : O \times O \to O$ that maps two words to another word. In this sense a word can act both as an object and as an *operator*. For example, the word mother could operate on father such that mother $\times$ father $=$ grandmother.

It is worth noting that these are aspirational goals for a set of objects representing words. However, they form the motivation for this paper, where we introduce objects in a *geometric algebra* as a better way to represent words. The rest of the paper is organized as follows. In Sec. 2 we provide formal background on word embedding models and then introduce geometric algebra and motivate it as having good theoretical properties as word representations. Subsequently we describe how the geometric product is implemented and integrated into word embedding models. We then present experiments showing promising results on traditional word embedding tasks, motivating further exploration of this space.

2 Background

2.1 Word Embeddings

Word embedding models come in two flavors, (1) “skip-gram", in which the center word is used to predict each word in the context, and (2) “CBOW", where the context words are typically mean-pooled and then used to predict the center word. We focus on the former which we use throughout the paper.

Formally, for the skip-gram model a training example consists of a center-context word pair (c, o). The model consists of two weight matrices $W, W' \in \mathbb{R}^{V \times d}$, where V is the size of the vocabulary and d is the dimension of the word embeddings. The dot product is taken between the vector representing the center word in W and the vector representing the context word in W'; this dot product $w_c^T w'_o$ represents the unnormalized probability (logit) of these words appearing in the same context. When normalizing it is intractable to take a softmax over the entire (often very large) vocabulary. Thus, instead Mikolov et al. introduce a technique called *negative sampling*. A logistic loss is applied on top of the logit $w_c^T w'_o$, such that the model is trained to maximize the probability of o appearing in the context of c. Then N negative samples $\{c, n_i\}_{i=1}^N$ are drawn from a distribution over the vocabulary, and the model should minimize their dot-product similarity. The final maximum likelihood objective looks like the following:

$$\log \sigma(w_c^T w'_o) + \sum_{n_i \sim P(w)} \log \sigma(-w_c^T w'_{n_i}) \tag{1}$$

Mikolov et al. introduce several tricks that significantly improve the performance of their model (all of which we implement in our word2vec benchmark) [1]. In particular, they (1) discard words that appear too rarely in the dataset using some empirically tuned threshold (typically 5-15), and (2) *subsample* frequent words, in other words removing words from the dataset with probability proportional to their frequency. The motivation behind the latter is to remove filler words from the dataset which co-occur with other words indiscriminately. Details of these operations (e.g. which distributions are used for negative sampling and subsampling) can be found in the original paper and are not discussed here.

The entire word2vec model is differentiable and is typically trained using some gradient-based optimizer such as Adam or SGD. The corpuses used to train these models tend to be large (a dataset of 50-100 MB of text is usually seen as a minimum requirement to learn good-quality embeddings).

2.2 Geometric Algebra

Having briefly introduced word2vec, we now introduce the mathematics of geometric algebra and motivate it as proving a useful alternative to vectors for word embeddings. At a high level, geometric algebra consists of a set of objects (*multivectors*) endowed with a multiplication (the *geometric product*). A useful starting point to understand multivectors is the basis elements $\mathbf{e}_1, ..., \mathbf{e}_n$ that span the vector space $\mathbb{R}^n$. In the geometric algebra $\mathbb{G}^n$, the basis elements are not just vectors but *spans* of vectors; for example $\mathbb{G}^2$ contains the basis element $\mathbf{e}_1\mathbf{e}_2$ which can be envisioned as the plane spanned by the two unit vectors (see Fig. 1 for a visualization). In other words, the basis elements of a geometric algebra represent *subspaces*; this is a key point, as it is the first of a recurring theme that algebraic properties/operations in this space have geometric meaning (hence 'geometric algebra'). Since the basis elements of $\mathbb{G}^n$ represent spans of vectors, there are 2^n basis elements (the powerset of the basis vectors). A general multivector is then a linear combination of these basis elements or subspaces (also called basis *blades*). For example, in G^2 an example of a multivector is:

$$A = 5 + 3\mathbf{e}_1 + 4\mathbf{e}_2 + 6\mathbf{e}_1\mathbf{e}_2. \tag{2}$$

The other important terminology is that of a *grade*. In $\mathbb{G}^n$ there are $n + 1$ grades, where the grade of a particular basis element is the dimension of the subspace. For example, the multivector above has three grades (0 for scalar, 1 for $\mathbf{e}_1$ and $\mathbf{e}_2$, and 2 for $\mathbf{e}_1\mathbf{e}_2$). For a multivector A, the grade projection $\langle A \rangle_r$ maps A to only its grade-r components; for example in the multivector above $\langle A \rangle_1 = 3\mathbf{e}_1 + 4\mathbf{e}_2$.

Note that it is important to disentangle the definitions of a multivector that are basis-independent and basis-dependent. The basis-independent definition of a multivector is a sum of *blades*, where a blade is the outer product of r linearly-independent vectors. The outer product represents the subspace spanned by these vectors, so a multivector is a *sum of subspaces*. A grade projection maps to only those subspaces of a certain grade or dimension. The definition of multivectors above is basis-dependent (I chose the basis of standard unit vectors and their spans because it makes it clearer in the next section how to implement this product). However, the discussion below of the geometric product is necessarily basis-independent.

Geometric algebra comes with a multiplication termed the *geometric product*. Although difficult to define concisely in full generality, a convenient starting point is the geometric product of two vectors, which is defined as:

$$ab = \frac{1}{2}(ab + ba) + \frac{1}{2}(ab - ba) := a \cdot b + a \wedge b. \tag{3}$$

The first term on the right is the familiar inner product, which outputs a scalar and is a commutative operation. The second term is called the 'outer product' and is anti-commutative; it is zero when a and b are parallel, and we can identify it with the signed area of the parallelogram spanned by a and b (analogous to the cross product in three dimensions). In fact it is indeed this parallelogram which is termed a *bivector*. More generally, the outer product of r linearly independent vectors (which is equivalent to the geometric product if they are orthogonal) represents the subspace spanned by these vectors. Note by the way that from the above equation it follows that $\mathbf{e}_1^2 = 1$ and $\mathbf{e}_1\mathbf{e}_2 = -\mathbf{e}_2\mathbf{e}_1$.

In general, the geometric product of two multivectors is $AB = \sum_{r,s} A_r B_s$, where A_r and B_s are the grade-r and grade-s components of A and B respectively. An important theorem of geometric algebra states that the geometric product $A_r B_s$ consists of terms of grades $r - s, r - s + 2, ...r + s - 2, r + s$. The inner product is then roughly defined as the grade-$(r - s)$ projection of this product (one might observe from this two types of inner products, the other being the grade-$(s - r)$ projection) and the outer product as the grade-$(r + s)$ projection of this product. (These statements are not vital to understand.)

A natural question at this stage is why this product is useful. It happens that many of the algebraic operations in this space have geometric interpretations. For example, take two blades A_r and B_s (where again an r-blade is the outer product, or in other words the span of r linearly-independent

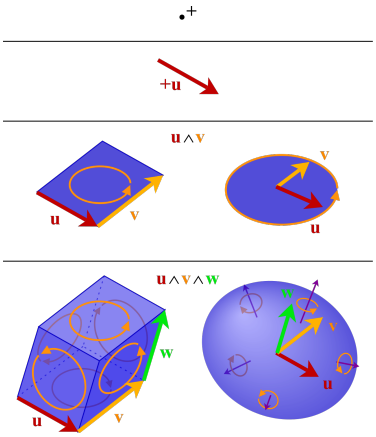


Figure 1: Pictorial representation of the elements of a geometric algebra. The top row shows the scalar element, followed by a vector, bivector (span of two vectors), and trivector (span of three vectors). A general multivector is a linear combination of such objects.

Figure credit: commons.wikimedia.org/wiki/File:N_vector_positive.svg

vectors). It can be shown that (1) the outer product between A_r and B_s represents the sum of their subspaces (or 0 if they share a common vector), and (2) the left inner product represents the orthogonal complement of A_r in B_s (and vice versa for the right inner product). These and other properties contained within the geometric product create richer *geometric* interactions between the multivectors representing words than the vector dot product. In particular, one could imagine the following benefits of representing words as multivectors:

- **More geometric objects = better at standard word embedding tasks**: Because multivectors are more complex objects and the geometric product encodes rich geometric interactions, it is possible that a multivector space of words could perform better on the standard tasks assigned to vector word embeddings, such as word similarity and analogy solving.
- **Ability to represent hierarchy / syntax**. The representations of words as subspaces and the graded nature of the algebra offers opportunities to capture more complex aspects of language. For hierarchy one could imagine assigning smaller grades to more specific concepts and larger grades to more general concepts. The graded nature of the algebra offers a natural possibility for creating extremely high-level groupings of words, and these grades could also capture elements of syntax (an important example is part-of-speech). In general it seems more natural to represent words and concepts as subspaces that *span* different semantic and syntactic aspects.
- **Words as Operators**. Having a meaningful product that maps two word objects to another object could give us the ability to use words as operators acting on other words. This could be a very useful notion as described in the Introduction.

3 Implementing the Geometric Product

With geometric algebra introduced and motivated in the previous section, we now describe how we implement the geometric product. In general, we can represent some multivector in $\mathbb{G}^n$ as a linear combination of products of the basis vectors $\mathbf{e}_1, ...\mathbf{e}_n$ (example in Eq. (2)). These have the property (generalizing Eq. 3) that $\mathbf{e}_i^2 = 1$ and $\mathbf{e}_{\sigma(1)}...\mathbf{e}_{\sigma(n)} = (\text{sgn}(\sigma))\mathbf{e}_1...\mathbf{e}_n$, since the product of two orthogonal vectors is anti-commutative. Thus for the product of two basis elements (i.e. two basis blades) we know **(1)** the basis element in the product it maps to is the exclusive-or of the basis vectors in each blade (if say $\mathbf{e_1}$ appears in both basis blades, we can permute the product such that it contains $\mathbf{e_1 e_1}$, which disappears into a scalar). We also know **(2)** the sign of the product is the number of swaps to rearrange the product into a basis element (which happens to be the number of inversions). This gives us a natural way to compute the product by representing basis blades as binary vectors, where element i is '1' if the blade contains e_i. Concretely, the geometric product is implemented by

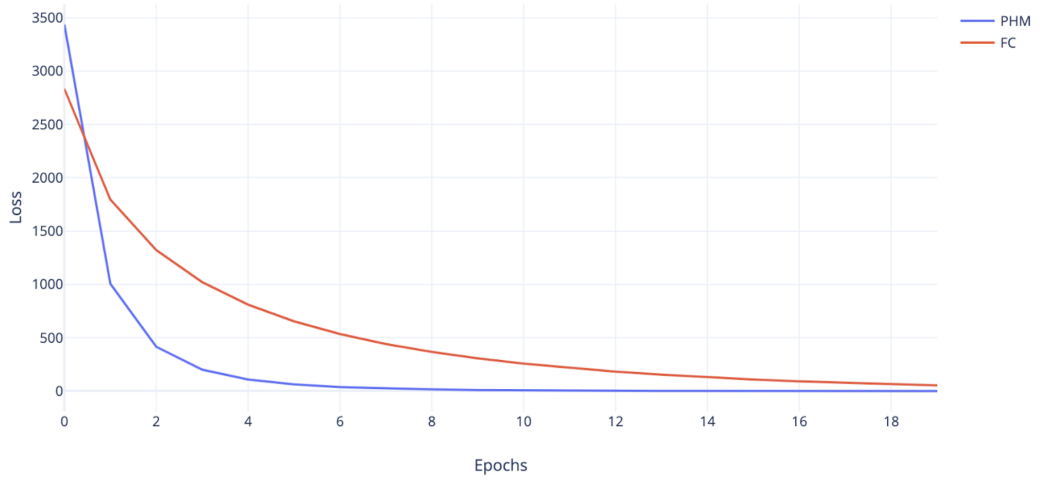


Figure 2: The figure above shows the loss curves of the PHM and FC layers on our synthetic dataset of multivector products. Note that both models successfully learn the geometric product; however, the PHM parametrization is able to learn the product quicker and the loss converges more rapidly.

computing a 3D sparse tensor with each dimension 2^n; the element (i, j, k) represents the coefficient of basis blade k in the product from multiplying basis blades i and j. The product is then the tensor product of the first multivector A with this tensor, followed by a dot product of this matrix with B.

To make this more concrete, we can take an example in $\mathbb{G}^2$. Let's say we are multiplying two multivectors:

$$(a_1 + a_2\mathbf{e}_1 + a_3\mathbf{e}_2 + a_4\mathbf{e}_1\mathbf{e}_2)(b_1 + b_2\mathbf{e}_1 + b_3\mathbf{e}_2 + b_4\mathbf{e}_1\mathbf{e}_2).$$

Due to the properties above, we would find that the coefficient of $\mathbf{e}_1$ in the product is $a_1b_2 + a_2b_1 - a_3b_4 + a_4b_3$ (we expand the product, and the reason for the negative sign is that $\mathbf{e}_2\mathbf{e}_1\mathbf{e}_2 = -\mathbf{e}_1$ by anti-commutativity. The full product can be written as $M = Ab$, where b is the vector of coefficients for the second multivector and A is the matrix given by:

$$a_1 \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} + a_2 \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} + a_3 \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} + a_4 \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad (4)$$

.

The tensor is the stack of these signed binary matrices, and this matrix is above is exactly the tensor product of A with the tensor. This provides a concrete example of how we implement the geometric product.

3.1 Neural Networks Can Learn the Geometric Product

Since the geometric product of two multivectors can be written as a matrix product, it is interesting to explore whether a neural network can learn the geometric product. We train an simple one-layer linear network ($y = Wx + b$) on a synthetic dataset we construct of 100 training examples in $\mathbb{G}^2$; concretely 100 random multivectors are generated and multiplied by a random 'hidden' multivector to produce an output multivector. Given the input and output multivectors, the linear network is able to learn the matrix product corresponding to the hidden multivector (i.e. the matrix described in the previous section) (see Fig. 2)

We also evaluate a neural network parametrization introduced in [4]. This parametrization replaces fully-connected layers with "parametrized hypercomplex multiplication" (PHM) layers $y = Hx + b$.

These layers are termed as such because they generalize several products such as the quaternion product and our own geometric product. Here H is derived as a sum of n matrix Kronecker products:

$$H = \sum_{i=1}^{n} A_i \otimes S_i,$$

where A_i and S_i are learned parameters and the Kronecker product is a block matrix where each block of $A \otimes S$ is $a_{ij}S$. The authors show that replacing FC layers with this parametrization (where the dimensions of A and S are only constrained such that the shape of H is the original shape of W) leads to state-of-the-art performance across several NLP tasks such as machine translation. They motivate this parametrization by observing that it could subsume several types of more complex products such as the quaternion product and its generalizations; hence the success of this approach provides some empirical motivation for our work. This parametrization indeed subsumes the geometric product; we can observe from Eq. 4 that if we constrain A_i to be 1×1 then H could learn the geometric product. Interestingly, this parametrization leads to faster convergence of the loss compared to the fully-connected layer (Fig. 2). The reason that we constrain the model to carrying out the geometric product rather than allowing it to learn the product it wants is the following: geometric algebra gives us a space of objects that have some notion of geometric interpretability, and this is a key aspect of word embeddings.

4 Word2MVec: Multivector Word Embedding Model

In our approach we aim to represent words as multivectors that interact with the geometric product. We note two difficulties that arise in developing such a model:

1. For the CBOW model, we would replace the mean operation on the context vectors with the geometric product of all the context multivectors. However, with a typical window size of 10 we found empirically that this led to coefficient explosion (which we were unable to easily fix with normalization or better initialization). Thus, we focus on the skip-gram model (which achieves similar word2vec performance) where the training examples are (context, center) word pairs.
2. More fundamentally, the geometric product between two multivectors is itself a multivector. However, the logits of the model are scalars; geometric algebra does not easily admit an operation that reduces multivectors to scalars. To solve this we add an $N \times 1$ fully-connected layer on top of the model that maps the multivector product to a scalar. This effectively allows the model to learn which information to extract from the geometric product.

Solving these issues allows us to arrive at our word embedding model. For a given (center, context) training example, we take the geometric product between the multivectors $m_o m_c$, where m_o is the context and m_c the center multivector. Similar to word2vec, the center and context word multivectors are found by looking up the parameter matrices $W \in \mathbb{R}^{V \times d}$ and $W' \in \mathbb{R}^{V \times d}$ respectively. This is followed by a fully-connected layer that maps the resulting product to a logit, followed by the familiar logistic loss. Exactly as in word2vec we draw negative samples from a distribution over the vocabulary and compute their logits through the geometric product + FC layer. This model, which we call **Word2Mvec**, is shown in Fig. 1.

This is the model that we use in all experiments and compare to word2vec. However, we note that due to the fully-connected layer Word2Mvec has more parameters than Word2vec (assuming same dimension of vectors/multivectors). To disentangle this component we introduce a second baseline which we term **Word2Vec+FC**; the dot product is replaced with an element-wise product followed by a fully-connected layer. This model has exactly the same number of parameters as Word2Mvec.

One important thing to note: although we are replacing the dot product with the geometric product in Word2Mvec, we still *retain* the notion of dot product similarity. This is because words that appear in similar contexts will tend to have similar representations and thus similar multivector coefficients; this similarity can be still be quantified through the dot product. This is significant, because it means we don't lose anything by shifting to multivectors but can potentially gain more from their more complex algebraic and geometric structure.

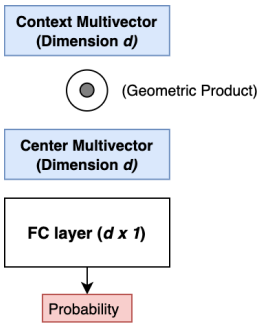
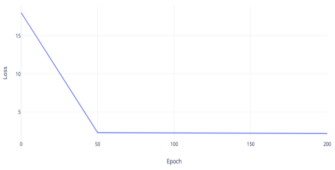


Figure 3: **Figure of the Word2Mvec model**. The geometric product is taken between the center and context multivectors, followed by a fully-connected layer that outputs an unnormalized probability. Similar to word2vec, the center and context multivectors are found by looking up the parameter matrices $W \in \mathbb{R}^{V \times d}$ and $W' \in \mathbb{R}^{V \times d}$ respectively.



Nuclear	Struggle, Capability, Ideological
Ronald	Reagan, Papers, Ford, Address
Mankind	Acting, Common, Problem, Nations

Figure 4: Left: loss converges rapidly for World-Order Dataset. Right: Word2Mvec model learns embeddings with good similarity properties on this small dataset.

5 Experiments

5.1 "World Order"

As a sanity check, we begin with a smaller corpus, the text "World Order" by Henry Kissinger. The text is only 1.2MB, and so is not appropriate for any large-scale benchmarking of word embedding models; however, we can use it to verify that our Word2Mvec model is successfully training. Indeed, the loss does converge rapidly (within 50 epochs of 200 total epochs of training) and the model is able to learn some notion of similarity. For example, the most similar words to 'ronald' are reagan, papers, ford, and address; similarly the most similar words to 'nuclear' are struggle, capability, and ideological (Fig. 4). Thus we can be satisfied that our geometric algebra based model is capable of learning satisfactory word embeddings and move to a larger-scale corpus which is the focus of our experiments.

5.2 Text8 Corpus

The *text8* corpus consists of the first 100 million characters of Wikipedia (approximately 100 MB of text). This is a sizable enough dataset to train word embedding models and analyze their performance. We train the Word2Mvec model as well as the baseline Word2Vec and Word2Vec+FC models described in Sec. 4. For all models we train for 10 epochs and use the Adam optimizer; we experiment with different learning rate / batch size configurations, which are mentioned in the tables which show our results. Throughout our experiments we use an embedding dimension of 128, which represents multivectors in $\mathbb{G}^7$.

As discussed above, Word2Mvec conveniently retains the notion of dot product similarity. Thus, a good starting point is to check whether Word2Mvec is learning embeddings that have reasonable notions of similarity. Table 1 shows qualitatively this is indeed the case. Both the Word2vec and Word2Mvec models demonstrate the ability to relate a given word to semantically and syntactically similar words.

President	Vice, Minister, Presidency, Presidential, Cabinet
Banana	Pear, Potatoes, Beets, Plantations, Juice
Tennis	Connors, Hockey, Sportsman, Borg, Wimbledon

Table 1: Example words and similarities learned by the Word2Mvec model. Similar words on the right column are taken from the top ten most similar words to the words in the left column. Note that the model learns an effective notion of similarity, both semantically (e.g. banana->pear, tennis->hockey) and syntactically (e.g. president->presidential). This justifies what we have argued above that we still retain the ability to compute cosine similarity for multivector word embeddings trained using Word2Mvec.

Method	Accuracy (%)
Word2Vec	10.49
Word2Vec+FC	15.72
Word2MVec	**16.27**

Table 2: Accuracies on the analogy-solving task with fixed hyperparameters lr = 0.002, batch size 1024. The Word2Mvec model outperforms both baselines.

5.3 Analogy Solving

Word vectors can successfully solve many analogies in the following way. Given for example the analogy man:king:woman:___(man is to king as woman is to ___), we can compute the vector $v(\text{king}) - v(\text{man}) + v(\text{woman})$. The vector closest to this (by cosine similarity) is $v(\text{queen})$ (although an important trick to get this to work is to disallow the model from predicting any of the words in the analogy itself). The text8 corpus is large enough that word vectors can demonstrate this ability. We evaluate our models on a standard dataset of 19,943 analogies that range from semantic (e.g. Son : Daughter : Husband : Wife) to syntactic (Young : Younger : Large: Larger). The accuracies are shown in Table 2. For a fixed configuration of hyperparameters (lr = 0.002, batch size = 1024) the Word2Mvec model slightly outperforms the two baselines; since the Word2Vec+FC model has the exact same number of parameters, this indicates that representing words as multivectors over vectors and taking the geometric product might have some benefit. It should be noted that this table comes with caveats; the models seem unusually sensitive to hyperparameter tuning, and increasing the batch size from 1024 to 5000 improves the accuracy of Word2Vec and Word2Vec+FC to 20.31% and 20.17% respectively. However, at the very least this provides evidence that Word2Mvec is able to learn analogies with comparable accuracy to current vector word embedding models.

5.4 Exploring Sparsity

An interesting question to ask is what happens to word vectors/multivectors when enforcing a sparsity constraint. For example, we might enforce an ℓ_1 penalty on the coefficients $\lambda||w||_1$, which encourages most coefficients to go to zero. Interestingly, enforcing such a sparsity penalty when training the model leads to highly interpretable dimensions. See Table 3 for examples of the words associated with the highest magnitude coefficient for a particular dimension of the vector or multivector. For both Word2Vec and Word2Mvec we find that introducing an ℓ_1 penalty creates a high level of interpretability. A particular detail is of interest; we enforce the ℓ_1 penalty $\max_{w_i} ||w_i||_1$, i.e. the maximum ℓ_1 norm from the word vectors/multivectors rather than the sum across all of them as conventionally. We found that the traditional ℓ_1 penalty did not lead to as interpretable word embeddings, possibly because our approach provides a more balanced tradeoff between accuracy and sparsity (since the gradient would only act on one word for a given step). This is a novel finding, as previous efforts to generate interpretable word embeddings have dismissed the ℓ_1 penalty as not having high coherence and favoring more complex methods.

Multivectors offer an additional, natural notion of sparsity, that is sparsity by *grade* rather than by coefficient. The idea is that each word should be restricted to a particular grade, or in other words a linear combination of basis subspaces of fixed dimension. A possible motivation for such a sparsity

Word2Vec	Isomorphic, Automorphism, Idempotent, Bijective, Injective Industrialized, Colonized, Subtropical, Westerly, Exporter Quicker, Slower. Faster, Taller, Cheaper
Word2Mvec	Voltage, Latency, Throughput, Amplitude, Capacitance Screenwriters, Improv, Cartoonists, Playwrights, Screenplay Airplanes, Avionics, Helicopters, Boeing, Airlines

Table 3: Words with the highest magnitude for selected coefficients in the vector or multivector (e.g. highest values of coefficient 44). For both the Word2Vec and Word2Mvec models, applying an ℓ_1 sparsity penalty results in a surprising amount of coherence for a given coefficient.

is clustering of words into extremely high-level semantic or syntactic examples; a natural example of this is part-of-speech (e.g. nouns, verbs, adverbs, etc.), something which is not captured by word2vec. Grade sparsity can be implemented using the technique of *group sparsity*, which is designed to send entire coefficient groups to zero for predefined groups. In particular for a vector w, group sparsity is computed as $\lambda \sum_{g=1}^{G} ||w_g||_g$, where G is the number of groups and w_g is the slice of coefficients in group g. The term $||w_g||_g$ is simply the ℓ_2 norm of the coefficients in group g, i.e. $||w_g||_g = \sqrt{\sum_{j=1}^{g} (w_g^{(j)})^2}$. For the Word2Mvec model, we experiment with applying group sparsity to the model based on divisions of the basis elements by grade. Thus far limited experiments have not shown any definitive kind of clustering created by enforcing group sparsity; however, the term has been plagued by gradient instability and we intend to solve this and investigate this further.

5.5 Recognizing Sentence Boundaries

The text8 corpus is typically treated as one long stream of words with no sentence delimiters, and training examples are collected across sentence boundaries. However, it is conceivable that respecting sentence boundaries matters for capturing better syntax in the word embeddings. Thus, we experiment with retraining the models in the previous section while only collecting (context, center) training examples that appear in the same sentence. Interestingly, the word2vec and word2vec+FC models improve by 1.23% and 0.75% respectively compared to Table 2 when trained in this way; by contrast the wordmvec model's performance decreases by 2.81%. This may be because respecting sentence boundaries reduces the total number of training examples for the word2mvec model where more training data is needed to capture information from the more complex geometric product.

5.6 Text9 Corpus

We have begun running experiments on the text9 corpus, which is approximately 10 times larger than text8 and provides an even better test of the potential of geometric algebra embeddings. These results are not yet conclusive; however it appears that representing words as multivectors may offer some benefit for traditional word embedding tasks. The result to this effect is that when training word2vec and Word2Mvec (lr = 0.002, batch size = 4096), the Word2Mvec model achieves **27.81%** accuracy on the analogy task, significantly higher than the word2vec model at **21.69%**. This sizable gap in accuracy is emphasized by the fact that hyperparameters were chosen based on what maximized accuracy for *word2vec* on the text8 corpus. Since the multivector-based model is more complex and likely requires more data for training, this indicates that providing a large-scale training set seems to realize the benefits of such a representation most strongly. Further investigation is of course needed into what kind of multivector space is learned by this model, but this indicates some level of promise.

6 Conclusion and Future Work

This report lays the groundwork for an investigation of geometric algebra as a representation of word embeddings. We introduce and motivate multivectors and the geometric product as having useful properties for word embeddings. We then implement this product and explore its performance compared to standard vector word embedding models on first a smaller corpus and then larger

corpuses. We find some promising evidence that representing words as multivectors leads to improved performance on standard word embedding tasks, and investigate the properties of sparse models for both Word2vec and Word2Mvec. In the future we aim to investigate this claim further, as well as examine some of the other properties we motivated in the introduction (hierarchy, closedness, etc.) which may require larger corpuses like the text9 corpus we have begun exploring.

References

[1] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[2] Tomas Mikolov, Kai Chen, Greg S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[3] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[4] Aston Zhang, Yi Tay, Shuai Zhang, Alvin Chan, Anh Tuan Luu, Siu Cheung Hui, and Jie Fu. Beyond fully-connected layers with quaternions: Parameterization of hypercomplex multiplications with $1/n$ parameters, 2021.