Nonlinear Word Embedding Homeomorphism Approximation

William Bernardoni, Joshan Bajaj, and Riley Scott

December 7, 2018

Contents

1 The Problem: Semantic Topology and Embeddings

Word embedding at its core is the idea that instead of representing a word as a one-hot vector in an extremely highly dimensional one hot space (i.e. $\mathbf{Z}/2\mathbf{Z}^n$ space, where n is the number of unique words, and $\mathbf{Z}$ are the integers) – a totally separable space – we could represent it in a lower dimensional continuous vector space.

The goal of word embeddings is to encode each word into a semantic space, one that preferably would be independent of language. This lead to the idea that the embeddings of parallel English and French corpora would be identical up to homeomorphism.

Most research into analyzing the similarity of word embeddings assumes that the homeomorphism between the spaces of parallel corpora would be in the form of a linear transformation - specifically an orthogonal one. We do not believe that this is necessarily a safe or correct assumption.

Different languages roll different semantic components into each word. For instance, in the ancient semetic language of Akkadian, each word's core semantic meaning is determined by three consonant roots found within the sentence, and each noun is heavily inflected to reflect meaning. Whereas the verb is rather sparse – as a result of this heavy inflection each sentence contains relatively few words each with much semantic context. In English we tend not to encode specific semantic information into specific words and tend to have much more verbose sentences with semantic details spread over the sentence. So we could imagine that the *inverse embedding* from the semantic space to Akkadian, and from the semantic space to English would look very different. We would expect English to be much more *stretched out* with each vector in the English space corresponding to a semantic vector that would be much closer in angle to an axis, whereas in a heavily inflected language such as Akkadian we would expect each Akkadian one-hot vector to correspond to semantic vectors with greater angles from the *semantic basis* vectors. Each word would have more information from different axes of the semantic space.

In this highly abstracted view of word embeddings, intuition would lead us to believe that the images of the English embeddings and the French embeddings of a parallel corpora as subspaces of a grand semantic space would be at best homeomorphic, and would likely differ by a nonlinear transformation.

2 Our Solutions

The goal of this project was to develop a more accurate method of finding a continuous transformation between two vector spaces of parallel corpora word embeddings. In particular we developed a nonlinear method, derived from a nonlinear dimensionality reduction algorithm called an Elastic Map[1].

Our Elastic Map algorithm finds a smooth manifold representation of a given space. In its pure form it does not preserve the orientation of the space or relations between specific domain and codomain points. As we want a specific mapping wherein certain given points (e.g. the English word *the*) get mapped to specific points (the corresponding French word *le*), we created a modified version that converges to a specific map, rather than just a manifold over the data.

The Elastic Map views the generated manifold as a spring which wants to return to a minimal identity function. Then applies the training data as physical pressures on this elastic spring to deform it into the correct manifold. This creates a smooth manifold over the training data.

The precise algorithm we use is described in a later section.

In addition to running our Elastic Map algorithm, we also ran the data through a neural network implemented by PyTorch as a nonlinear baseline, as well as a linear orthogonal algorithm as a linear baseline.

3 Experimental Design

We used Google's Word2Vec to generate word embeddings on the Europarl parallel corpora[3]. From these corpora we then use Giza++[4] to generate the pairings between vectors in each language. We then saved the vectors Giza++ was over 90% confident were direct translations of one another.

We implemented three functions, a orthogonal linear transformation approximation between the pairs, and two nonlinear transformations, trained and tested on a 50-50 split the word embeddings in each language we generated a dataset on.

Using the withheld word embeddings, we evaluated the error of each method primarily by mean squared error. Where error is defined via the Euclidean metric between the predicted word embedding and the actual word embedding.

The results, which we will cover in more detail in a later section, overwhelmingly supported our hypothesis, that a nonlinear map would achieve far better accuracy on this problem than a linear map.

4 Linear Orthogonal Alignment

Word Embeddings of different languages are currently mapped to each other linearly by finding a "best rotation" on a dataset, i.e. a linear orthogonal matrix such that the error between the rotated source vectors and the target vectors is minimized. To implement our linear orthogonal map, we followed Kabschs algorithm[2], but generalized it to the Nx100 dimension case to account the dimensionality of the vectors created by Google Word2Vec.

The steps are broken down as follows.

Step 1) For both the training set, create a Nx100 matrix for both the source and the target word embeddings. N represents the number of words in the training set while 100 represents the number of dimensions. We will denote the source matrix S, and the target matrix T.

Step 2) Once the two Nx100 matrices, S and T, are created, we translate them such that the mean word embedding calculated from each dataset is at the zero vector.

Step 3) With the two centered Nx100 matrices, S and T, we then calculate the covariance matrix H.

$$H = S^T * T$$

Step 4) With H calculated, the next step is to find the rotation matrix R.

To do that, we compute the Single Vector Decomposition of H to find the unitary matrix containing left-singular vectors U, the rectangular diagonal matrix D, and the transpose of the unitary matrix containing right-singular vectors V^T.
i.e.

$$H = U * D * V^T$$

We then create a diagonal $100x100$ matrix of 1s in each point except for the last point, which instead contains the determinant of $V * U^T$. We will denote this matrix Q.

Finally, we may calculate the rotation matrix R.

$$R = V * Q * U^T$$

Step 5) With R calculated, we now create a Nx100 matrix of the source language for the test data, denoted S_1, similar to the ones created in Step 1. We then multiply this Nx100 matrix by the rotation matrix R, returning our linearly mapped rotated matrix F.

$$F = S_1 * R$$

5 Neural Model

Due to the prevalence of neural networks, we elected to use one to compare the results gathered from our linear and elastic map models.

Similar to previous homeworks, the model was written in PyTorch for ease of use. The model that gathered the greatest results was simply a single linear layer with an added L2 regularization term.

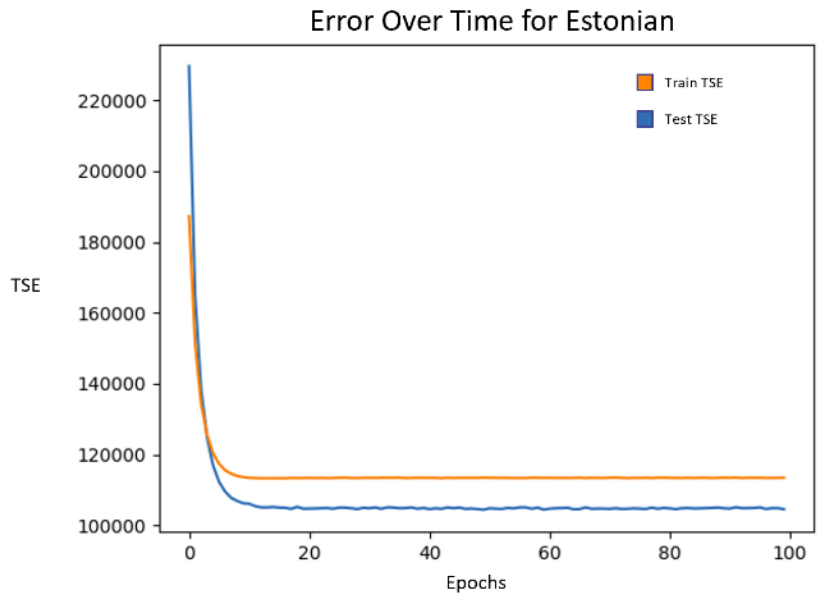
Multiple other models were written and tested which included more layers, but none performed better than the single layer network. All models ran for 100 epochs, and used the PyTorch implemented ADAM optimizer.

Mean squared loss was used to be consistent with the other methods tested, which is also provided by PyTorch.

The error reported is the average of the losses of the last 20 epochs ran, which is to ensure there aren't blatant outliers from poor performance in the beginning.

The L2 regularization was only applied to the loss used to train the model, not the mean squared error outputted.

Graphs such as the following were generated each run to illustrate the mean squared error over time.



6 Elastic Map Algorithm[1]

In this section we will use $d(a, b)$ to denote the Euclidean metric between two points in our vector space, and $||.||$ to denote the Euclidean, or l2 norm of a vector.

$\forall n \in N$ we initialize n with $n.o = n.i$.

Each data point $s \in S$ is then attached to a node $n \in N$ such that the distance between $n.i$ and $s.i$ is minimized. We denote this node as $N(s)$.

Each node $n \in N$ is then connected to each adjacent (in terms of the input lattice) node, this defines a set of elastic edges $E = \{a, b \in N : d(a.i, b.i) = 1\}$.

We then choose K random triplets of nodes in the lattice. We denote B as the set of these triplets. B will be the *spines* of the lattice.

From there we can calculate the current *tension* of our map via the following three energies:

$$P_S = \frac{1}{2} \sum_{s \in S} ||s.o - N(s).o||^2$$

P_S is the *approximation energy* of our map – the pull from our dataset onto our lattice.

$$P_E = \frac{1}{2} \lambda \sum_{(a,b) \in E} ||a.o - b.o||^2$$

P_E is the *elastic energy* of our map. The elastic energy is the tendency for nearby nodes to stay nearby. We call λ our elastic coefficient, which governs the degree to which the map resists stretching.

$$P_B = \frac{1}{2} \mu \sum_{(a,b,c) \in K} ||a.o - 2(b.o) + c.o||^2$$

P_B is the *bending energy* of our map. The bending energy is the tendency for our map to resist being bent and deformed. We call μ our bending coefficient.

The total tension of our map is:

$$P = P_S + P_E + P_B$$

We choose the positions of our nodes to minimize this total tension via a slightly modified gradient descent. We train in multiple steps, starting with a high λ and μ and then decreasing those pressures over time.

At each step of our gradient descent, for all nodes in S we only incorporate the gradient from P_S, as these nodes are our "fixed points", and we know exactly what output we want them to obtain. For all other nodes, we calculate the gradients from P_E and P_B, depending on which sets they belong to.

We then use this lattice of nodes as a nonlinear map between the spaces by interpolating between these nodes using an inverse distance weighing scheme[5]:

$$f(x) = \begin{cases} n.o & d(x, n.i) = 0 \text{ for some n} \\ \frac{\sum_{n \in N} w(n,x) * n.o}{\sum_{n \in N} w(n,x)} & \text{otherwise} \end{cases}$$

where:

$$w(n, x) = \frac{1}{d(x, n.i)^p}$$

and p being a hyperparameter, a higher p weighing further away nodes less. The p used in our implementation was $p = 5$.

Our algorithm differs from the standard implementation of an Elastic Map, as each node in our map has an input and an output vector associated with it. This allows our map to be used in either direction, although as it is nonlinear the performance in the reverse direction would not be expected to be as high as a map trained in the reverse direction. In addition, the lattice used to determine the energies of the map is fixed, and not optimized for, as in the standard elastic map algorithm. Instead our lattice is derived from the fixed input positions of each node.

As a result the training of our map is quite different (e.g. the difference in calculating gradients for members of S versus the rest of the nodes), and our map is not just a generic manifold onto the points, but instead is approximating a specific map with specific defined points.

7 Results

We ran our models on five different languages – Hungarian, Estonian, Bulgarian, Latvian, and Polish.

In terms of total squared error and mean squared error, our Elastic Map implementation consistently had a third of the error of the linear orthogonal baseline that we implemented. Even at the upper bound of one standard deviation from the mean, our Elastic Map performs better than the mean performance of the Linear Orthogonal method. Across all languages the increase in accuracy

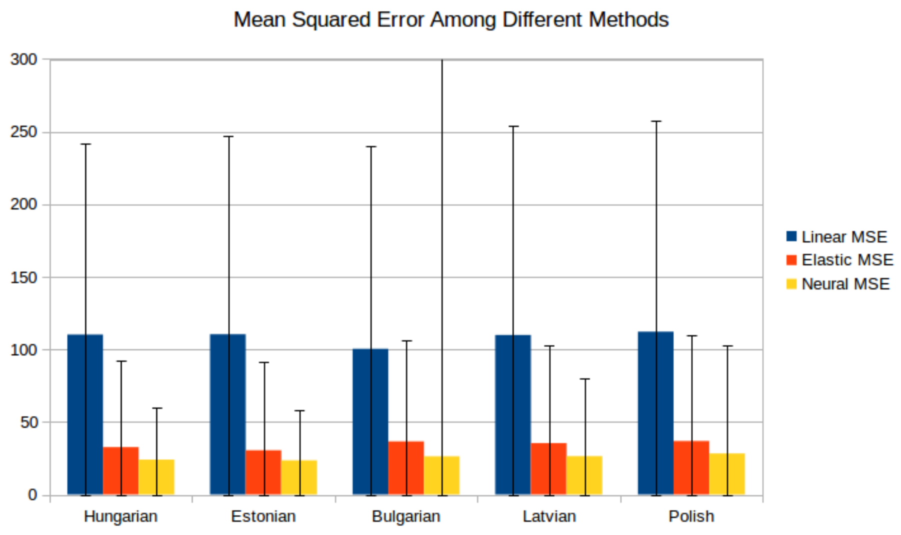


Figure 1: Chart of Mean Squared Error on each language. The error bars representing one standard deviation from the mean. Note the variance on Bulgarian in the Neural Model

is around the same proportional increase.

The only metric we measured, of Mean Squared Error, Cosine Similarity, and Pearson's R Coefficient, in which the Elastic Map had worse performance than the Linear Othogonal method was the Pearson's R Coefficient. This is to be expected however, as since the Elastic Map method is nonlinear, we would expect the error to similarly be nonlinear.

Compared to our highest performing neural model, we find the neural model consistently has a mean squared error around 20% lower than our Elastic Map's. However there is an interesting quirk of the Neural Model, in that the majority of the time it's standard deviation is proportionally better than the Elastic Map's, except for around a quarter of the models. Around a quarter of the models have standard deviations of over 100, some even higher than 400. The model we chose for Figure 1 outlines this quirk, in that for Bulgarian the upper bound for one standard deviation was over 300.

We believe this quirk to come from a potentially non-connected component of the data. Neural models produce a smooth and differentiable manifold, and if there is a ”shear” in the training space then the neural model would be forced to create an asymptotic section on that ”shear”, whereas our Elastic Map has no such requirements. The Elastic Map converges on a smooth map, but if there is a non-smooth region of the space the Elastic Map has far fewer restrictions on

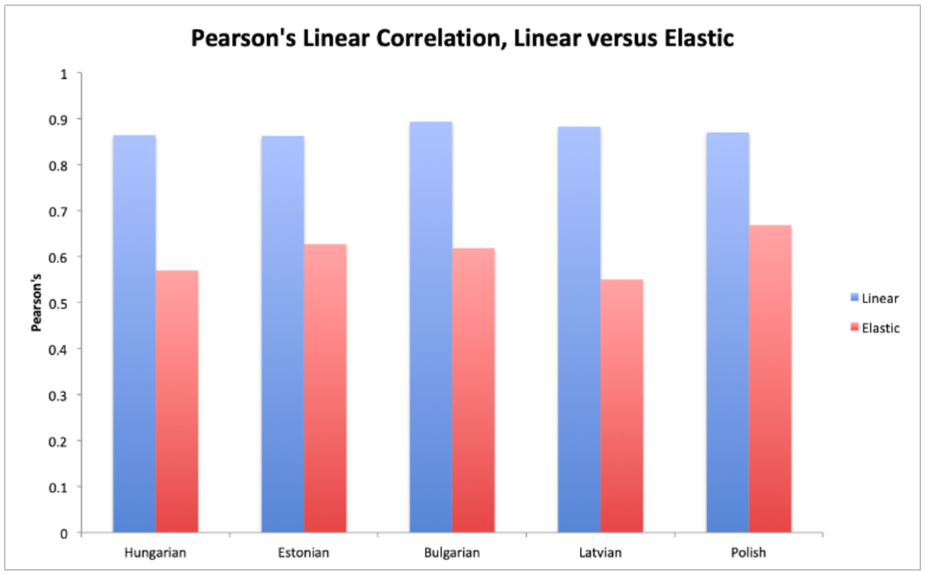


Figure 2: Chart of the Pearson's R Coefficient on the Linear and Elastic Maps.

the manifold it creates, so it could approximate that region in a less problematic way.

So while on average the neural model outperforms our current iteration of our Elastic Map, our Elastic Map performs far more regularly, and consistently. The mean error and the deviation are within a relatively tight bound across the languages we tested over. We believe that with more tuning and a more elegant interpolation model the Elastic Map could consistently outperform the neural models.

You'll note from across all methods, the data is heavily skewed, with many outliers of high error and many vectors of low error balancing them out. This tail of oddly behaving words is common in many NLP tasks, and we believe to be indicative of a poor embedding into the euclidean vector space.

Our code and data can be found at: https://github.com/wrbernardoni/mt_final

The exact mean squared errors and standard deviations are noted below.

Language	LO MSE	Std. Dev	EM MSE	Std. Dev.	Neural MSE	Std. Dev.
Hungarian	110.209	131.553	32.650	59.782	24.014	36.223
Estonian	110.393	137.015	30.427	61.188	23.526	34.589
Bulgarian	100.324	138.849	36.577	69.555	26.280	470.018
Latvian	109.879	143.777	35.394	66.971	26.387	53.276
Polish	112.161	145.219	36.846	72.523	28.280	74.816

We only examined the cosine similarity between the Linear Orthogonal and the Elastic Map, as those were the primary focii of this paper. They are as below:

Language	LO Cos. Sim.	Std. Dev.	EM Cos. Sim.	EM Std. Dev.
Hungarian	0.035	0.096	0.405	0.234
Estonian	-0.0013	0.099	0.408	0.218
Bulgarian	0.0017	0.104	0.373	0.225
Latvian	0.0036	0.105	0.380	0.236
Polish	0.0046	0.0959	0.365	0.208

The Pearson's R coefficients generated on the correllation between the output of the map and the target vector are below:

Language	LO Pearson's	EM Pearson's
Hungarian	0.864	0.570
Estonian	0.862	0.627
Bulgarian	0.893	0.618
Latvian	0.883	0.551
Polish	0.870	0.668

References

[1] Gorban, A. N., and Zinovyev, A. Y. Elastic maps and nets for approximating principal manifolds and their application to microarray data visualization. In *Principal manifolds for data visualization and dimension reduction*. Springer, 2008, pp. 96–130.

[2] Kabsch, W. A solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A 32*, 5, 922–923.

[3] Koehn, P. Europarl: A parallel corpus for statistical machine translation. In *MT Summit 2005*.

[4] Och, F. J., and Ney, H. A systematic comparison of various statistical alignment models. *Computational Linguistics 29*, 1 (2003), 19–51.

[5] Shepard, D. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM National Conference* (New York, NY, USA, 1968), ACM '68, ACM, pp. 517–524.