**Finding the optimal number of dimensions for word embeddings**

[[Matthew Kwan](https://medium.com/@matti.kwan?source=post_page-----f19f71666723--------------------------------)](https://medium.com/@matti.kwan?source=post_page-----f19f71666723--------------------------------)

[Matthew Kwan](https://medium.com/@matti.kwan?source=post_page-----f19f71666723--------------------------------)

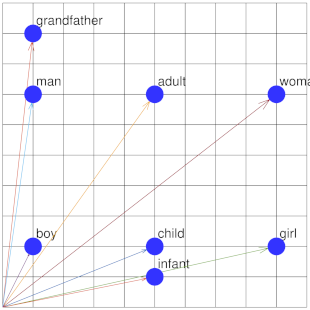
·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F4f67732626e9&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40matti.kwan%2Ffinding-the-optimal-number-of-dimensions-for-word-embeddings-f19f71666723&user=Matthew+Kwan&userId=4f67732626e9&source=post_page-4f67732626e9----f19f71666723---------------------post_header-----------)

5 min read

·

Sep 6



*Embeddings*are vectors of numbers that represent a word. They have some useful properties for machine learning, such as

* Words with a similar meaning have similar embeddings.
* Embeddings are made up of numbers which, unlike text, can be used to train neural networks.
* Embeddings can be derived from any corpus of text, unsupervised.

And you can perform linguistic arithmetic with them. For example, the embedding for **king**, minus **man**, plus **woman**, is very close to the embedding for **queen**.

It’s as though an embedding encodes the abstract representation of a concept. A language of the gods, in a way.

But how many numbers does it take to represent the attributes of a word? In other words, what’s the optimal size (a.k.a. *dimension*) of an embedding?

The authors of [**word2vec**](https://en.wikipedia.org/wiki/Word2vec#Dimensionality) are somewhat vague, recommending somewhere between 100 and 1000 dimensions. Other authors pick numbers like 100 or 300. **OpenAI** uses [1536 dimensions](https://openai.com/blog/new-and-improved-embedding-model) — although their model handles lots of human languages, not just English.

But given the amount of computing power needed to process embeddings in large language models, it would be useful to find the optimal embedding size, rather than just guessing.

**Skip grams**

The original embedding model, **word2vec**, uses the skip-gram approach, where each word is paired with some words in its nearby context (e.g. within 5 words distance in the same sentence), and those pairs are used to train the model. In addition, *negative samples* — pairs that *don’t* occur nearby — are added to the training mix.

The drawback to this approach is that you need to re-ingest the entire text corpus every time you want to change a parameter in your model. And when your corpus contains billions of words, that can take a while.

Another drawback is that word2vec doesn’t produce an objective accuracy metric, so evaluations of embeddings tend to be subjective.

**Matrix factorization**

Another way to calculate embeddings is though *matrix factorization*. The classic example of this technique involves a sparse matrix of movie ratings, with a row for each user, and a column for each movie.

Each of the *u* users and *m* movies are represented by a *d* dimensional embedding, so you have *d*x *u* and *m*x *d* matrices. You then use backpropagation to populate these two matrices, so that when multiplied together they reproduce the sparse ratings as closely as possible.

Clearly, the higher the value of *d*, the more accurately the matrix can be factorized, but at some point you’re going to over-fit the data. So how do you detect that (and thus find the optimal value of *d*)?

The solution is to hide some fraction of the ratings matrix as a validation set. You then calculate the user and movie embeddings from the visible ratings, and evaluate the result using the hidden values. Then you find the value of *d* (through trial and error) that best predicts the hidden validation ratings.

**Embeddings from a matrix**

It turns out that word embeddings can also be calculated through matrix factorization, although the matrix isn’t sparse. Kian Kenyon-Dean has written an [excellent series of blog posts](https://medium.com/radix-ai-blog/unifying-word-embeddings-and-matrix-factorization-part-1-cb3984e95141) explaining the mathematics of word2vec matrices, but in short you factorize a matrix containing the PMI (point-wise mutual information) of each word pair.

The Kenyon-Dean solution factorizes the matrix using a custom Keras loss function, which is a clever approach, but wasn’t suited to my needs. Because the full matrix has to be resident in memory, I couldn’t hide the validation values. And besides, my GPU didn’t have enough RAM.

Instead, I needed to specify explicit training values rather than loss values for the cells in the matrix. But many PMI values are negative infinity, so the matrix was populated instead with the [*sigmoid*](https://en.wikipedia.org/wiki/Sigmoid_function) of the PMI values, and the model optimized the sigmoid of the dot product of the embedding matrices.

As a result, each cell of the matrix contained N*ij* / (N*ij* + N*i*.N*j* / N), where N*ij* is the number of times word *i* occurs near word *j*, N*i* is the sum of N*ix*, N*j* is the sum of N*xj*, and N is the sum all N*ij*. The skip-words window is typically symmetric, so N*i* should always be the same as N*j*.

The Keras model for calculating the embeddings was as follows:

row\_input = layers.Input(shape=(1,))  
 row\_embedding = layers.Embedding(  
 token\_count,  
 dimensions,  
 name='Embedding'  
 )(row\_input)  
 row\_vec = layers.Flatten()(row\_embedding)  
  
 column\_input = layers.Input(shape=(1,))  
 column\_embedding = layers.Embedding(token\_count, dimensions)(column\_input)  
 column\_vec = layers.Flatten()(column\_embedding)  
  
 dot\_product = layers.Dot(axes=1)([column\_vec, row\_vec])  
 dot\_product = layers.Activation('sigmoid')(dot\_product)  
  
 model = models.Model(inputs=[row\_input, column\_input], outputs=dot\_product)  
 optimizer = optimizers.Adam(lr=0.02)  
 model.compile(loss='mean\_squared\_error', optimizer=optimizer)

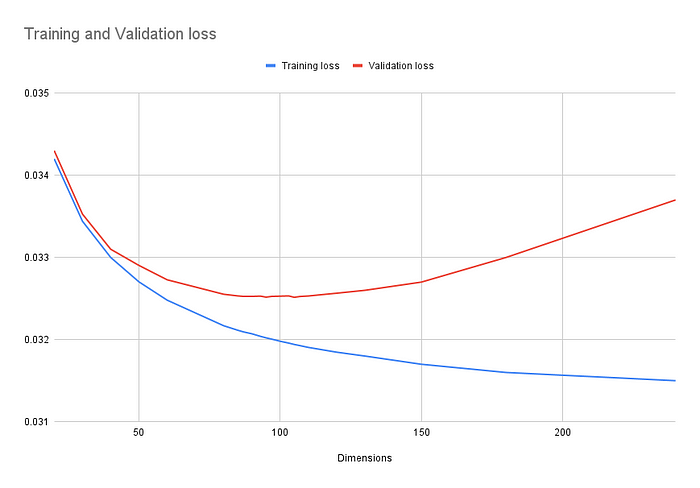
And here’s sample code for training with a 1% validation set, returning the resulting word embeddings. Although in practice the token and value lists often won’t fit into memory, so a generator may be needed.

model.fit(  
 (token1\_list, token2\_list),  
 values\_list,  
 epochs=epochs,  
 batch\_size=1048576,  
 validation\_split=0.01)  
  
 embeddings = model.get\_layer('Embedding').get\_weights()[0]

**Experimental results**

The model was trained on the English Gutenberg corpus, roughly 4 billion words in total. 24,830 tokens were used, with a skip window of 5 words, resulting in a matrix with around 600 million cells, of which 33% were populated.

The following graph shows the training and validation loss after 10 epochs, and how it varied with the number of embedding dimensions.



Training and validation loss vs dimensions

As expected, the training loss fell steadily, while the validation loss leveled out and started increasing. It turns out the validation loss was fairly constant between 85 and 110 dimensions, varying by less than 0.05% across the range.

So what’s the best number of dimensions? At the low end of that range, where over-fitting and computational load are both minimized? Or at the high end, where the training loss is 0.65% lower?

To find out I ran some subjective (and unscientific) tests, finding the closest matches to the word “anger” and the equation “king”-“man”+“woman”.

With 85 dimensions I got:

anger 1.0000  
wrath 0.8815  
rage 0.8788  
indignation 0.8678  
resentment 0.8222  
  
king - man + woman  
king 0.9075  
queen 0.9039  
prince 0.8950  
wife 0.8709  
brother 0.8583

Fairly good. And with 110 dimensions:

anger 1.0000  
wrath 0.8732  
rage 0.8671  
indignation 0.8542  
resentment 0.8090  
  
king - man + woman  
king 0.9021  
prince 0.8847  
queen 0.8710  
wife 0.8441  
brother 0.8416

A similar result for the word “anger”, but the king/queen result was less accurate.

But unexpectedly, the result for 30 dimensions was really good (subjectively):

anger 1.0000  
rage 0.9320  
indignation 0.9181  
wrath 0.9118  
reproach 0.8785  
  
king - man + woman  
queen 0.9686  
prince 0.9412  
king 0.9395  
brother 0.9086  
husband 0.9071

**Conclusion**

It’s a bit awkward that the best subjective results came from low-dimension embeddings that had poor *objective* loss values, but I’ll trust the numbers and conclude that 85 dimensions are optimal.

Results might vary with a different text corpus or tokenization algorithm, but probably not by much if it really is picking up fundamental properties of English text.

Moving forward, I’m keen to build a large language model that’s more efficient and elegant than the transformer architecture, and smaller embeddings are a good place to start.