Programming word embeddings in Snap!

Ken Kahn[1], Yu Lu[2], Jingjing Zhang[3], Niall Winters[1], Ming Gao[3]

1. Department of Education, University of Oxford
2. Advanced Innovation Center for Future Education, Beijing Normal University
3. Big Data Centre for Technology-mediated Education, Beijing Normal University

toontalk@gmail.com

Abstract

Word embeddings is a technique in natural language processing whereby words are *embedded* in a high-dimensional space. They are used in sentiment analysis, entity detection, recommender systems, paraphrasing, text summarisation, question answering, translation, and historical and geographic linguistics. We describe a Snap! library that contains 20,000 word embeddings in 15 languages. Using a block that reports a list of 300 numbers for any of the known words, one can create programs that search for similar words, find words that are the average of other words, explore cultural biases, and solve word analogy problems. These programs can work in a single language or rely upon the alignment of the word embedding spaces of different languages to perform rough translations.

To compute with word embeddings one needs to perform vector arithmetic. This can be accomplished by providing vector arithmetic blocks. More advanced users can instead take advantage of Snap!'s support of higher-order functions to use list mapping blocks to perform the vector operations.

Word embeddings place within Natural Language Processing

Natural language processing is a broad field which includes translation, question answering, parsing, sentiment analysis, summarization, chatbots, speech recognition, and speech synthesis. Here we focus on one piece that is an underlying foundation of many natural language processing systems constructed in the last decade. The ability to place words in a high-dimensional space is also itself a way to computationally explore the semantics and morphology of words. Unlike other natural language processing technologies, the mathematics behind word embeddings is at a high school level or lower.

Introduction to word embeddings

Word embeddings are very useful in machine learning since nearly all machine learning systems work only with numbers (typically multi-dimensional arrays of numbers called tensors).

Textual input to a machine learning system is enabled by mapping each word in the text to a vector of numbers.

Word embeddings themselves can be used directly for various tasks including solving word analogy problems. They can solve problems such as

1. Man is to woman as king is to X
2. Cat is to kitten as dog is to X
3. Slow is to slower as fast is to X
4. Paris is to France as Berlin is to X

The first example can be recast as finding the word closest to the result of adding to the vector associated with ‘king’ the difference between the vectors for ‘woman’ and ‘man’. In ratio notation we can say:

woman : man :: queen : king

Similar to how one says:

40 : 60 :: 2 : 3

Or it can be expressed as this approximate equality between vectors:

woman - man ≈ queen - king

Given the ability to map words to a high-dimensional space students can create programs to explore the space. They can attempt to answer questions like:

1. Do some words have many close neighbours and others few?
2. Are opposites near each other?
3. Do colours form their own neighbourhood? Emotions? Animals?
4. What words does this technique get wrong?
5. What problems do multiple senses of the same word create?
6. Does the membership of clusters of “nice” words (e.g. flowers) and “not nice” words (e.g. insects) reveal biases?

Where do word embeddings come from?

There are three ways researchers have created word embeddings:

1. By “hand”
2. By frequencies

3. By machine learning

Creating word embeddings “by hand”

Creating word embeddings “by hand” involves creating a list of features or attributes for each word. For example, one might describe a puppy as

<0.3,	*size measure*
0.9,	*cuteness measure*
0.1,	*age measure*
1,	*animals are the first of 6 phyla*
1,	*nouns are the first of 9 parts of speech*
…	*many more are possible*
>	

Metric dimensions such as size, cuteness, and age work well but category ids such as phyla and parts of speech are problematic. Is part of speech id 1 closer to 2 than to 3 or 9? These are arbitrary encodings without any geometric meaning. There is a technical solution called “one hot” which turns any category into a long list of binary (0 or 1) properties. A puppy would become

<...	
0.1,	*age measure*
1,	*animal*
0,	*plant*
0,	*fungus*
0,	*protist*
0,	bacteria
0,	*archaea*
1,	*noun*
0,	*verb*
0,	*participle*
…>	

Note that to make distance calculations between vectors the values need to be standardized, for example, constrained to values between 0 and 1. This can be a problem for dimensions such as size that would need to span the range from quarks to galactic superclusters.

Creating word embeddings by frequencies of co-occurrence

Another way of creating word embeddings is to use the frequency with which a word occurs near other words in a large corpus of text (e.g. all of Wikipedia in a particular language). This builds upon the concept of distributional semantics [Harris 1954]. The underlying idea that “a word is characterized by the company it keeps" was popularized by Firth [Firth 1957].

Puppy might be something like

<.0023,   *‘kitten’ occurs nearby sometimes*
  .027,   ‘*and’ occurs nearby frequently*
  .000007,   ‘*algebra’ occurs nearby but rarely*
…>

The major shortcoming to this approach is that each vector needs to be as long as the size of the entire vocabulary. Vectors consisting of one million numbers are expensive to store and compute with. Limiting the vocabulary to the *n* most common words helps but is still expensive and limits the utility of the embeddings.

Extracting word embeddings for machine learning models

There are now several deep machine learning models that estimate the probabilities of what the missing words are from sentences with one or two removed [Mikolov et al 2013a]. They have been trained on text containing billions of words. The hidden layer in the neural network that feeds into the final output layer that predicts the probabilities can be used to generate word embeddings. The weights from each node in that penultimate layer can be collected as they connect to the output nodes for each word. Figure 1 illustrates this.

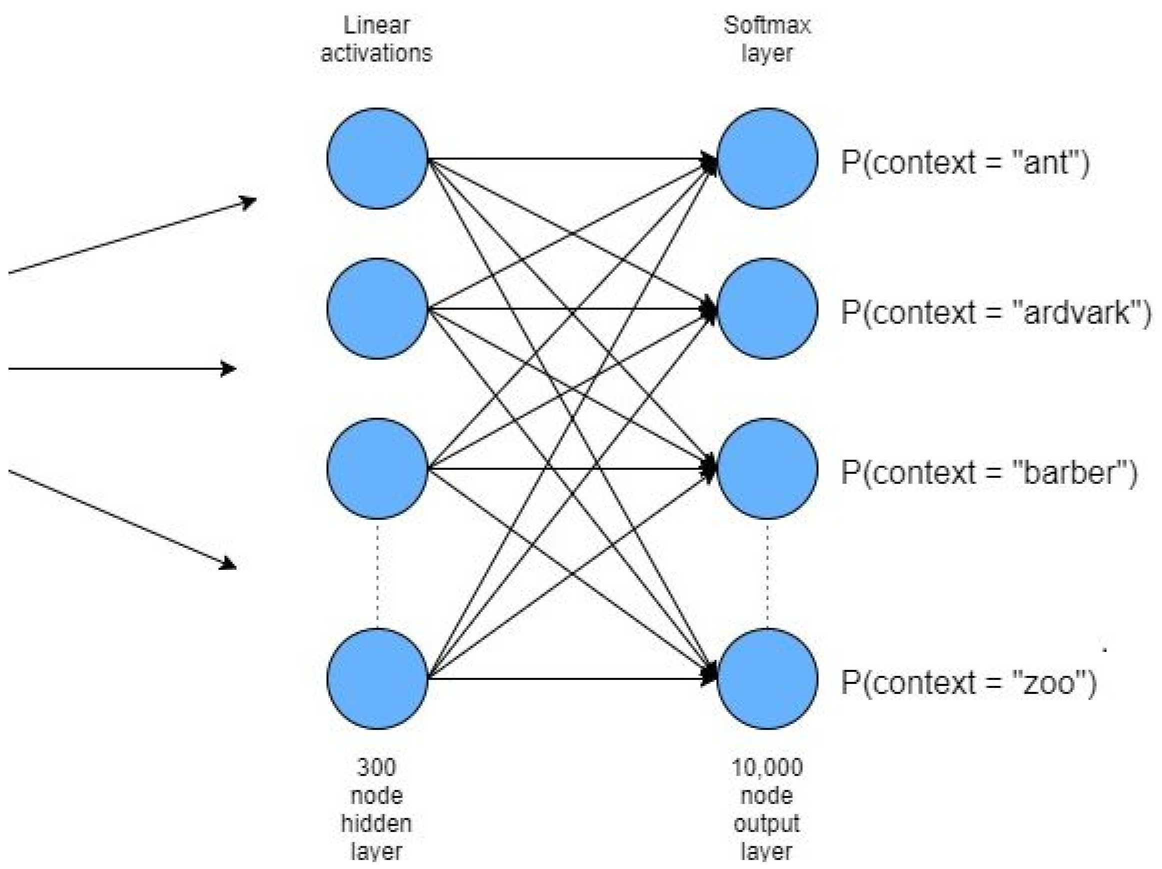


Figure 1 - a sketch of a neural net that can be used to generate a vector of 300 numbers for each possible output word

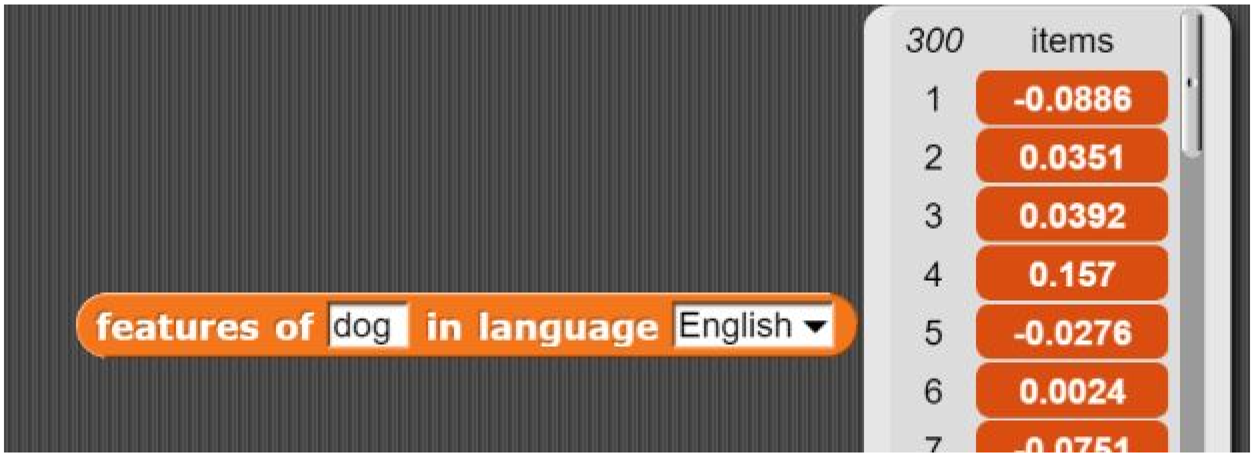
Unlike the frequency approach, each word embedding is only a few hundred numbers. The word embeddings are the result of very computationally intensive training with huge amounts of data but once extracted they can simply be implemented as a lookup table. Word analogy problems, for example, can be solved without using machine learning models at run-time.

Each node in the penultimate layer receives inputs from a deep network that has extracted a very large number of features. Typically these networks have millions of parameters. This makes it nearly impossible to say what any of the dimensions of the word embeddings mean. What is clear about these embeddings is that closely related words tend to be close together in this high-dimensional space and unrelated words are far apart.

Word embedding Snap! Blocks

We developed the word embedding Snap! blocks as part of the eCraft2Learn AI programming library [Kahn and Winters 2018]. It relies upon Snap!'s ability to create blocks using JavaScript. The library is free and open-source and available as both exported Snap! Blocks as well as a project that includes sample block usage and comments. Furthermore, we have provided a programming guide for each AI programming library including the one for word embeddings. Links to all of these resources and more is at https://ecraft2learn.github.io/ai/.

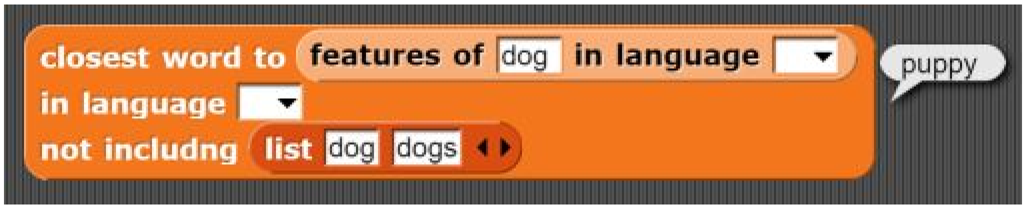
The basic word embedding block is one that reports a list of 300 numbers for any of 20,000 words in any of 15 languages (see Figure 2). We selected the most common lower case words lacking punctuation or digits from databases with one million words [Grave et al 2018]. To include more words and proper nouns would not be difficult but would result in longer load times, more memory consumption, and slower searches for nearby words. We chose the 15 languages based upon the needs of the eCraft2Learn project [Kahn and Winters 2018] and the languages' popularity. We have documented how one can add word embeddings for additional languages to Snap! [https://ecraft2learn.github.io/ai/word-embeddings/adding-more-languages.html].


features of dog in language English

300	items
1	-0.0886
2	0.0351
3	0.0392
4	0.157
5	-0.0276
6	0.0024

Figure 2 - Obtaining the word embedding for 'dog'

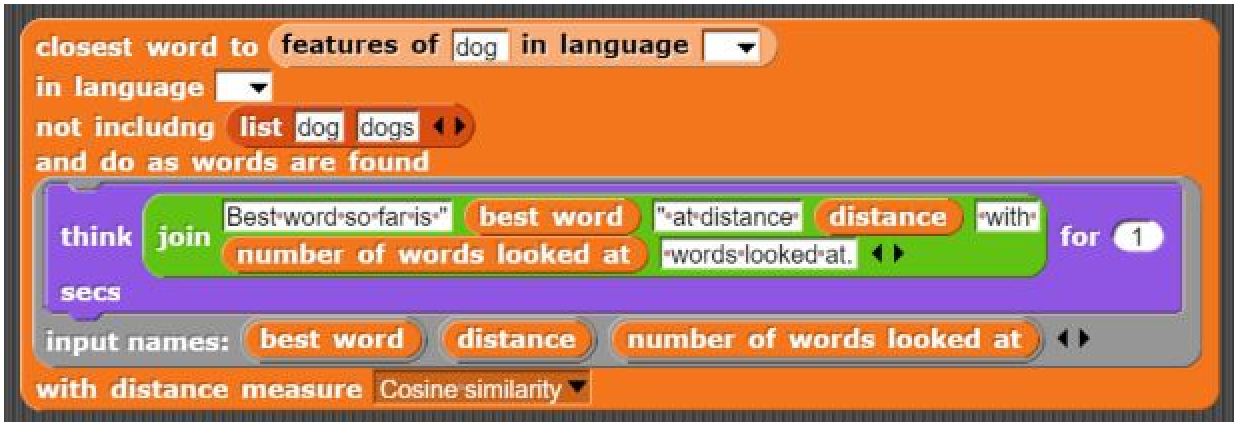
To find the closest word to a word embedding we provide the block illustrated in Figure 3.


closest word to features of dog in language
in language
not includng list dog dogs

puppy

Figure 3 - Searching for the word closest to 'dog' (not including 'dog' or 'dogs')

We provide a full-featured version of closest word for advanced users. It enables one to provide blocks that run as better and better matches are found. Also it supports a choice between two different distance measures: Euclidean distance and cosine similarity. An example of its use is illustrated in Figure 4.

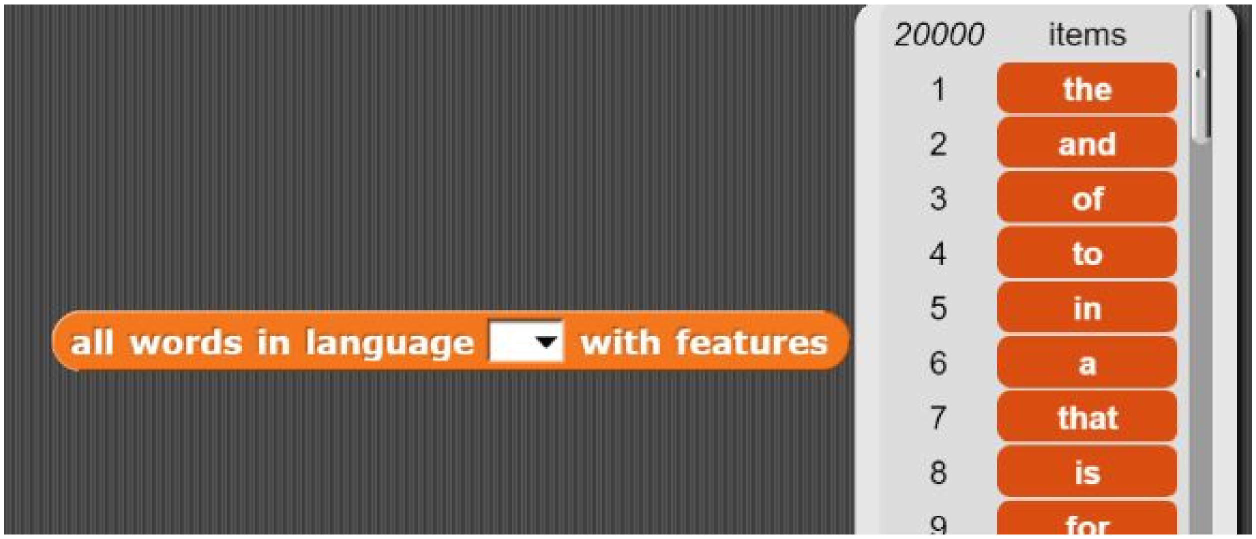


```
closest word to features of dog in language
in language
not includng list dog dogs
and do as words are found
  think join Best word so far is" best word "at distance distance with number of words looked at words looked at. for 1 secs
  input names: best word distance number of words looked at
with distance measure Cosine similarity
```

Figure 4 - Observing progress as the search for the closest word proceeds

We also implemented a block that sorts all the available words by how close they are to a vector input. It uses cosine similarity to measure closeness. Since each comparison entails many hundred multiply and add operations and we need to do 20,000 comparisons we chose to implement this using TensorFlow.js [https://www.tensorflow.org/js]. If the device has a GPU it can typically perform this in less than one second.

A block that reports all words with embeddings in a given language is illustrated in Figure 5.

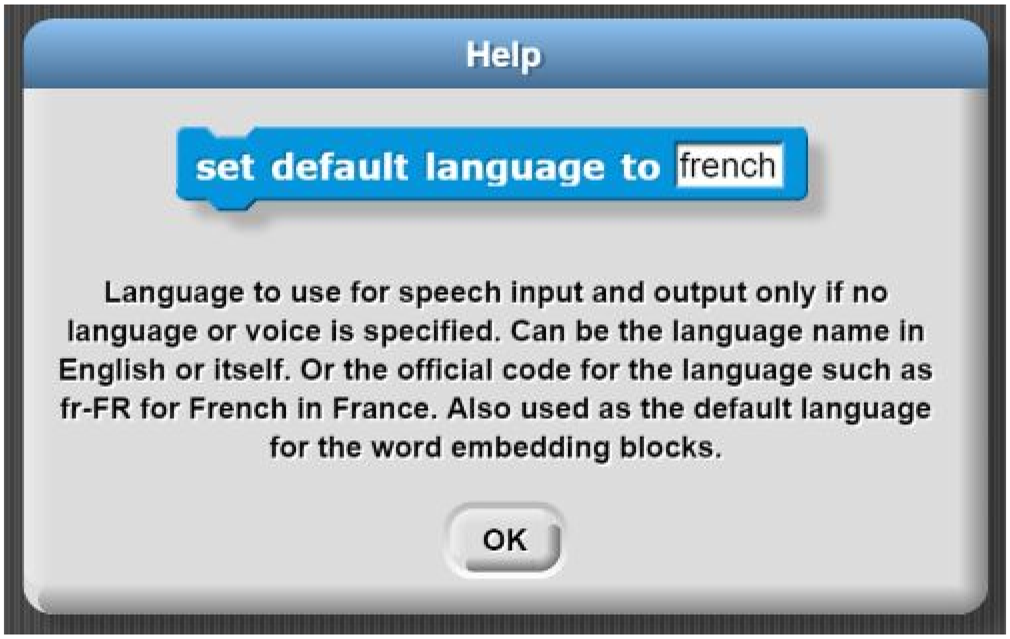


```
all words in language with features
```

20000	items
1	the
2	and
3	of
4	to
5	in
6	a
7	that
8	is
9	for

Figure 5 - Obtaining all the words with embeddings in the specified language

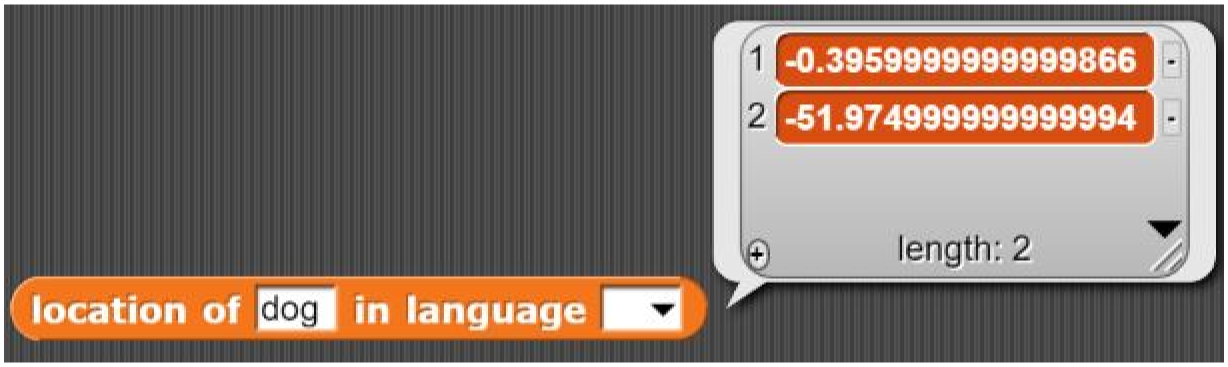
The default language can be set by the block illustrated in Figure 6. Note that the default language is also used by Snap! blocks for speech synthesis and recognition.



Language to use for speech input and output only if no language or voice is specified. Can be the language name in English or itself. Or the official code for the language such as fr-FR for French in France. Also used as the default language for the word embedding blocks.

Figure 6 - The block for setting the default language

While a great deal of information is lost when mapping a point in 300 dimensions to two it can be useful. Figure 7 illustrates a block that relies upon t-SNE [Maaten and Hinton 2008] so that words nearby in 300 dimensions are near in two dimensions and those that are far apart in 300 dimensions are far apart in two dimensions.



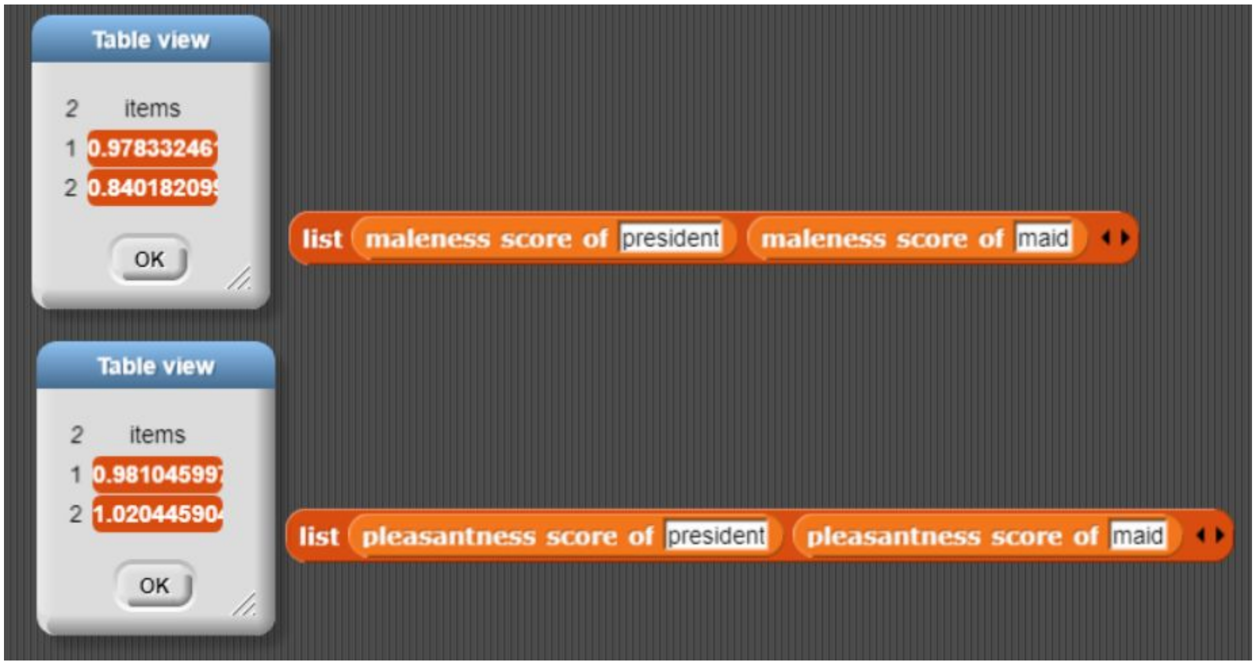
```
1 -0.3959999999999866
2 -51.974999999999994
```

Figure 7 - a block for obtaining a two-dimensional approximation of a word embedding

Finding words close to (and far from) other words

One can explore programmatically how a word can be closer to one set of words and further from another set of words while another word may be the opposite. This might reveal that flowers are close to pleasant words and far from unpleasant words while the opposite is true of insects. One can discover biases in this manner. These biases are implicit in the training text (Wikipedia and a web crawl).

A paper published in *Science* “Semantics derived automatically from language corpora contain human-like biases” [Caliskan et al 2017] proposed a way to measure word biases. The idea is to use the average distance two words have to two sets of "attribute" words. To explore gender bias, for example, the attribute word lists can be "male, man, boy" and "female, woman, girl". The difference of the average distances provides a score that can be used to compare words.



```
Table view
2   items
1   0.97833246
2   0.84018209
OK

list (maleness score of president) (maleness score of maid)

Table view
2   items
1   0.98104599
2   1.02044590
OK

list (pleasantness score of president) (pleasantness score of maid)
```

Figure 8 - Relative maleness and pleasantness scores of “president” and “maid”

Figure 8 illustrates a bias that “president” has a high maleness score while “maid” has a lower score. (Also “maid” is associated more closely with pleasant terms than “president”.) Some caution is required when exploring biases. A well-known bias is that “doctor” is more male than “nurse”. This may be a bias or due to the fact that only women *nurse* babies.

Solving word analogy problems

An amazing aspect of word embeddings is how they can be used to solve word analogy problems. For example, the problem “Man is to woman as father is to X” can be recast in terms of vector arithmetic. Starting with the expectation illustrated in Figure 9 that

$$\text{woman} - \text{man} \approx \text{mother} - \text{father}$$

we can “solve” for X as

$$X \approx \text{father} + (\text{woman} - \text{man})$$

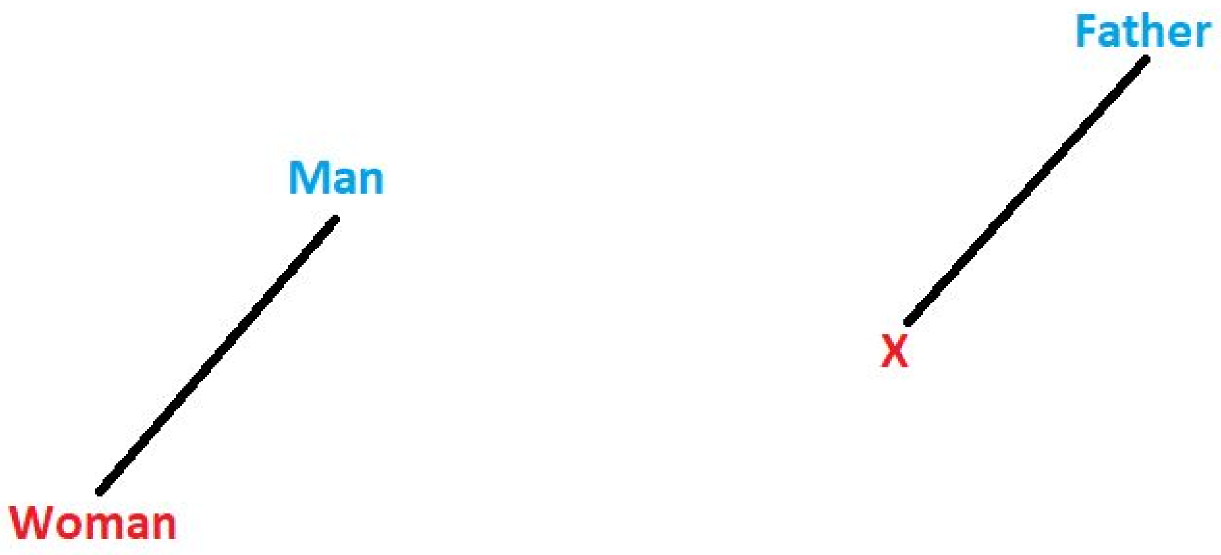
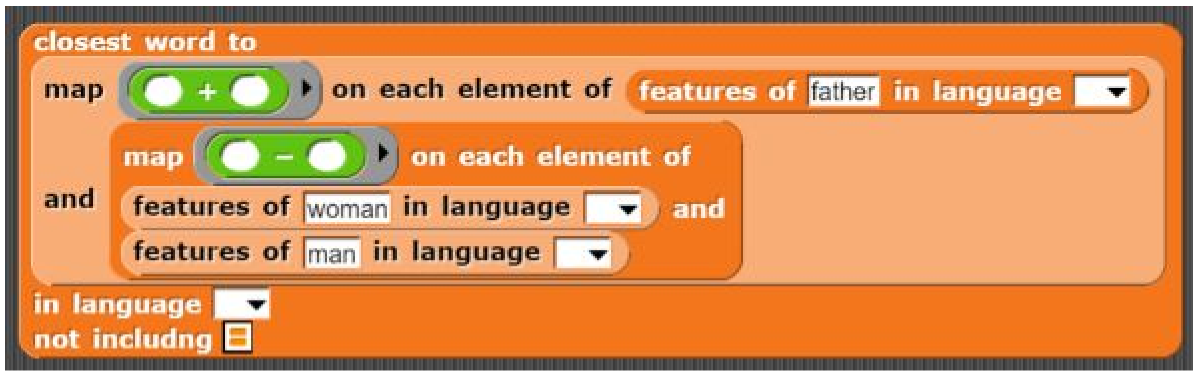


Figure 9 - The geometric relationship between man and woman is similar to that of father and mother (except the relationship is not two-dimensional but is in a very high-dimensional space)



```
closest word to
  map (( ) + ( )) on each element of (features of [father] in language [ ])
  and
    map (( ) - ( )) on each element of
      (features of [woman] in language [ ]) and
      (features of [man] in language [ ])
in language [ ]
not includng [ ]
```

Figure 10 - Solving a word analogy problem in Snap!

One way to interpret the equation is that if we compute “femaleness” by (woman - man) and add that to “father” we get “a female father” or a “mother”. Note that an equivalent equation

$$X \approx \text{woman} + (\text{father} - \text{man})$$

Can be understood as computing the essence of “parent” by (father - man) and then adding that to “woman” to obtain “mother”.

Word analogy problem solving using word embeddings is not limited to the meaning of words. Word embeddings also encode word morphologies so, for example, they can solve “fast is to X as slow is to slower”

Visualising word embeddings

While it is very useful to place words into a 300-dimensional space it makes it hard to visualize. Two popular means of visualizing high dimensional space is to map them to two or three dimensions. This can rely upon Principal Component Analysis (PCA) or T-distributed Stochastic Neighbor Embedding (t-SNE). A free tool provided by Google provides a way to interactively explore our word embeddings [https://projector.tensorflow.org/?config=https://ecraft2learn.github.io/ai/word-embeddings/en/projector.json].

A programmatic interface in Snap! Is also provided as illustrated in Figure 7. The “location of” block uses t-SNE to report two-dimensional coordinates for any word. Words close to each other in 300-dimensional space tend to be close in this two-dimensional mapping and those that are far away tend to be mapped to distant locations. Using this block one can define the “display word” block illustrated in Figure 11.

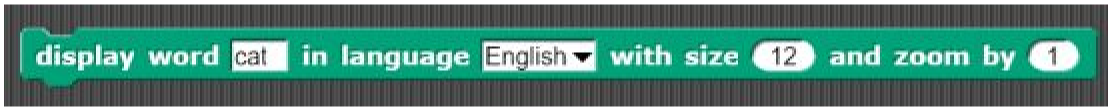

display word cat in language English with size 12 and zoom by 1

Figure 11 - a block for displaying words in Snap!’s Stage

Using Snap!’s pen and movement commands one can visualise a word embedding as vertical lines whose length corresponds to the value of each element in the word embedding vector. Figure 12 illustrates this by depicting ‘dog’ and ‘cat’. One can see how close these words are but also see several differences.

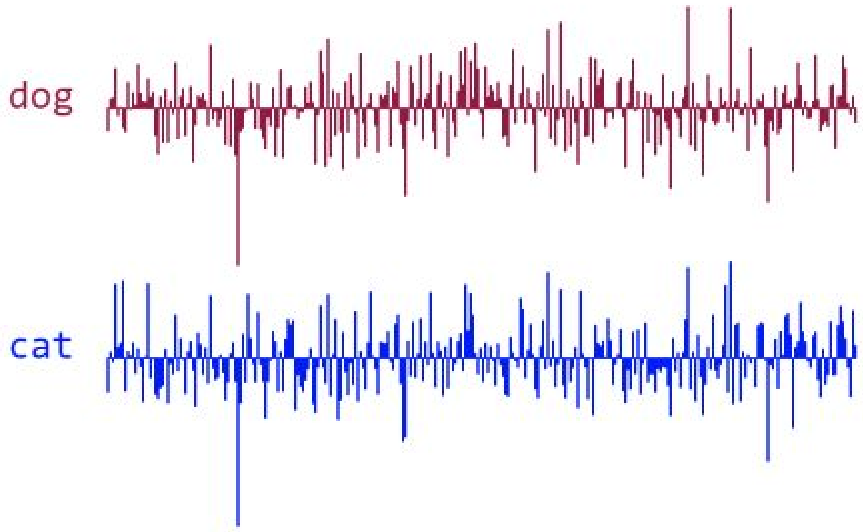


Figure 12 - Visualizations of the ‘cat’ and ‘dog ’embeddings

Implementation issues

Unlike the rest of the Snap! AI library the entire word embedding subset could have been implemented purely in Snap!. For reasons of performance and memory usage we relied instead upon Snap!'s ability to define blocks in JavaScript. Regarding memory usage we currently support 15 languages where each language needs to associate 300 numbers with each of 20,000 words (90,000,000 numbers in total and more as more languages are supported). Even if the memory requirements were satisfiable, we suspect that the time to load a Snap! project with so much data would be very slow. We addressed these problems by implementing each language's embedding as a JavaScript file that is loaded on demand.

A very heavily used block finds words close to a given vector. The distance between two points in the 300-dimensional space can be measured as the Euclidean distance or using cosine similarity. They behave roughly similarly but researchers prefer cosine similarity. Cosine similarity is defined as the dot product divided by the product of the magnitudes. A straightforward implementation in Snap! is over 100 times slower than the equivalent JavaScript program. Dot product was particularly slow when implemented using "map" so we used the uglier "for" command instead. On a modern laptop Snap! took 5110 milliseconds without "warp" and 71 milliseconds with it. The JavaScript equivalent took 0.18 milliseconds. Recall that a frequent operation is to search through all 20,000 words for the closest word. As described earlier when one wants the ranking of all 20,000 words an implementation that can take advantage of an available GPU is faster still.

The word embeddings used in this chapter were created by Facebook [https://fasttext.cc/docs/en/support.html] . They trained their machine learning models on 157 different languages on all Wikipedia articles in each language [Grave et al 2018]. Even though that was about a billion words each that wasn't enough, so they also trained their models on tens of billions more words found by crawling the web. They created tables for each language of at least a million different words. The blocks described here provide the 20,000 most common words for 15 languages.

Each language's embedding is independent of each other. Techniques have been developed, however, to rotate one language's embedding to roughly match another's. We relied upon a technique to align two languages given a small word list [Mikolov et al 2013b]. We provided a word list of 500 words (2.5%) to obtain a good alignment for the other 97.5%. The alignment was done in Python relying upon a student project [Xiaohong 2017]. As illustrated in Figure 13, due to this alignment one can fetch features in one language and search a different one.

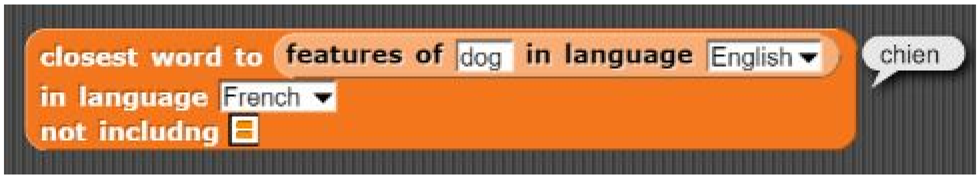


Figure 13 - Using 'closest word' between languages

Learning outcomes

At the time of writing we know of only two high school students who created projects using the word embedding blocks. One girl, for example, created a spelling quiz that relied upon finding the next closest word. More trials are being planned. We are investigating if any of the following learning outcomes can be observed:

1. Linear algebra (appreciation, creative uses, and practice)
2. Higher-order programming (understanding and appreciation)
3. Semantics of words (synonyms, antonyms, word senses, …)
4. Geometry (high-dimensional spaces, correspondences with vector arithmetic)
5. AI - Natural language processing and machine learning
6. Computational thinking (in a context rarely considered)

An introduction of word embedding programming to teachers in training

We conducted a study of N students at the Beijing Normal University…

Study methodology

Research questions

Findings

Possible projects

A few suggestions for projects using these word embedding blocks are:

1. Use word embeddings to explore the similarity of sentences. One idea is to average all the words in the sentence. This is called the bag of words technique since it ignores the order of the words just as if they were put in a bag.
2. Find a chain of similar words by finding the nearest word to the starting word. Then repeatedly find the nearest word to that while making sure to never repeat the same word. Use this to repeatedly change random words one at a time in a famous poem or text (e.g. "roses are red and violets are blue").
3. Make word games using word embeddings. For example, something like Semantris [https://experiments.withgoogle.com/semantris], a semantic version of Tetris. A bilingual version Semantris might be a good idea.
4. Create a program that searches for new word analogies. Hint: If A is to B as C is to D then A-B "is close to" C-D.
5. Search for biases that arise from the way people write about things. See if any biases found also apply in other languages.
6. Use word embeddings as a way to input textual data into the deep neural nets. The deep learning models can be created with other Snap! AI blocks [Kahn et al 2019]. This opens up a wide range of natural language processing possibilities.

Future developments

We chose to limit the number of words supported to 20,000 words and ignored all proper nouns. If proper nouns were added one would be able to solve analogies such as “Paris is to France as Berlin is to X”. Perhaps more interestingly one can explore biases associated with given names that are associated with particular cultures, genders, or religions. Or biases against baby names that were popular during particular decades. Relationships between occurrences of the names of historical figures can be explored. Word embeddings generated from texts from different periods in history can be explored to see how the meanings of some words have changed (e.g. “awful”).

A weakness with word embeddings is that different senses of a word are combined. “Duck” is close to “chicken” and close to “jump”. “Rat” ends up near “screen” because “rat” is near “mouse” and (computer) “mouse” is near (computer) “screen”. Researchers are exploring word sense embeddings that address this problem.

Another problem is that we consider only single words. “Sherbet” and “sorbet” are close to each other but “ice cream” has no entry. Similarly proper nouns such as “New York City” are currently not handled well.

References

[Caliskan et al 2017] Caliskan, Aylin, Joanna J. Bryson, and Arvind Narayanan. "Semantics derived automatically from language corpora contain human-like biases." *Science* 356, no. 6334 (2017): 183-186.

[Firth 1957] Firth, J.R. (1957). "A synopsis of linguistic theory *1930-1955". Studies in Linguistic Analysis: 1–32*. Reprinted in F.R. Palmer, ed. (1968). Selected Papers of J.R. Firth 1952-1959. London: Longman.

[Grave et al 2018] Grave, Edouard, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. "Learning word vectors for 157 languages." arXiv preprint arXiv:1802.06893 (2018).

[Harris 1954] Zellig S. Harris (1954) "Distributional Structure", *WORD*, 10:2-3, 146-162, DOI: 10.1080/00437956.1954.11659520

[Kahn and Winters 2018] Ken Kahn and Niall Winters, "AI Programming by Children" *Proceedings of Constructionism Conference*, Vilnius, Lithuania, August 2018

[Kahn et al 2019] "Deep Learning Programming by All", submitted for publication.

[Maaten and Hinton 2008]Maaten, Laurens van der, and Geoffrey Hinton. "Visualizing data using t-SNE." *Journal of machine learning research 9*, no. Nov (2008): 2579-2605.

[Mikolov et al 2013a] Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. "Distributed representations of words and phrases and their compositionality." In *Advances in neural information processing systems*, pp. 3111-3119. 2013.

[Mikolov et al 2013b] Mikolov, Tomas, Quoc V. Le, and Ilya Sutskever. "Exploiting similarities among languages for machine translation." arXiv preprint arXiv:1309.4168 (2013).

[Xiaohong 2017] Ji Xiaohong, Translation Matrix: how to connect "embeddings" in different languages?, https://rare-technologies.com/translation-matrix-in-gensim-python/