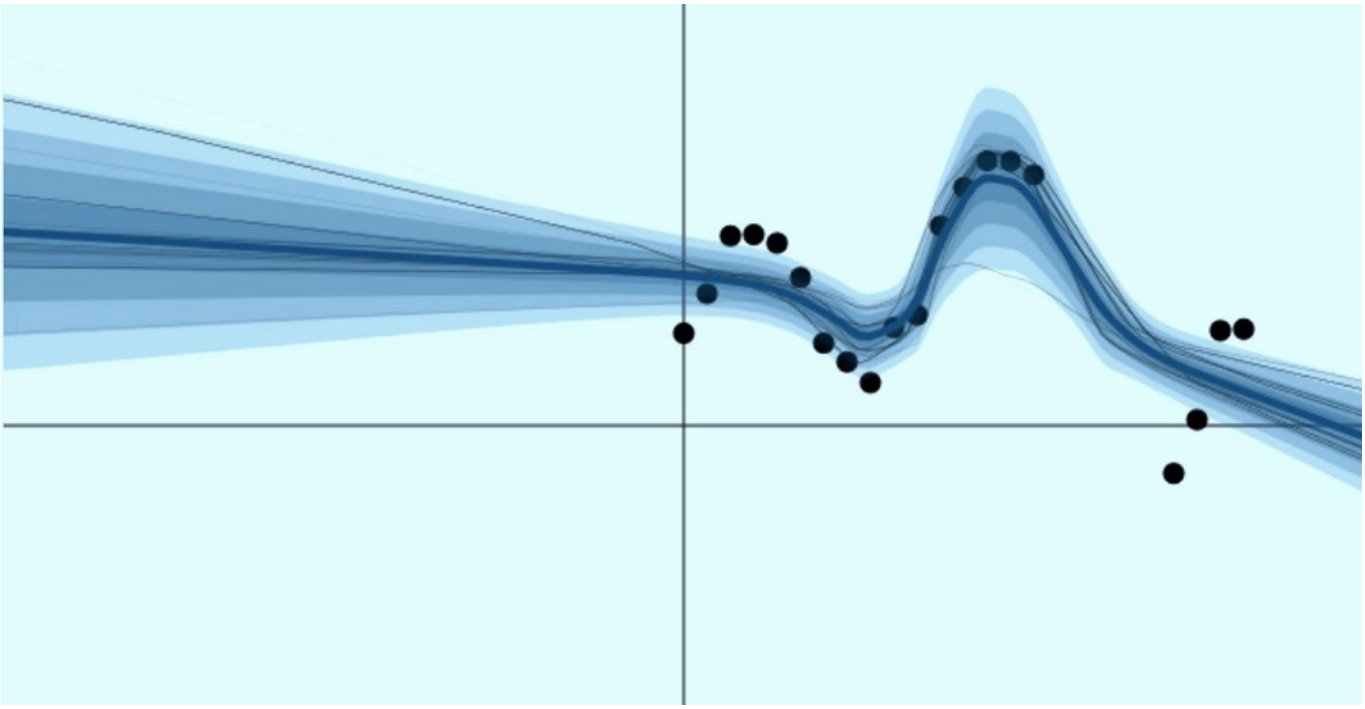
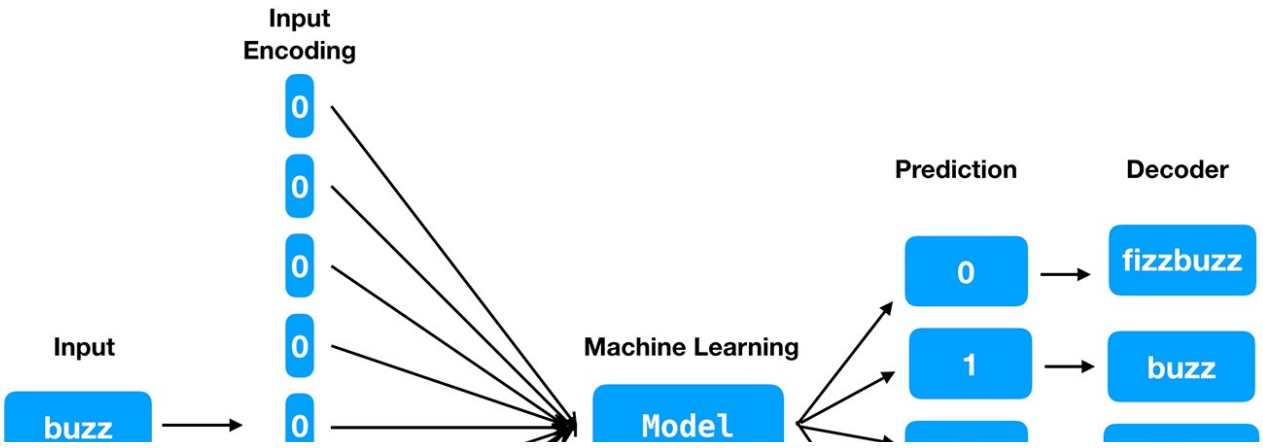
NLP: Everything about Embeddings

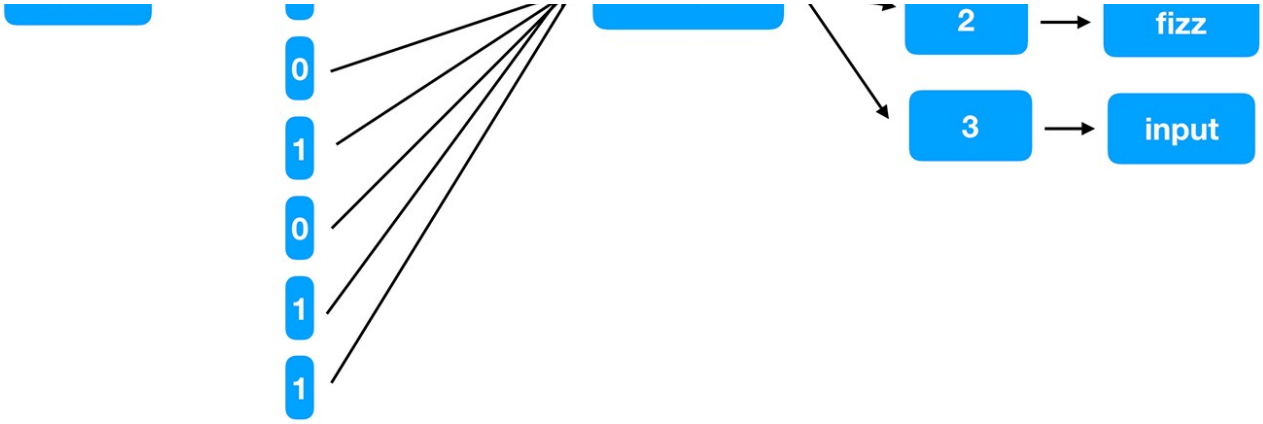
Mohammed Terry-Jack Follow
Apr 22, 2019 · 14 min read



A function (the line) approximated by a machine learning algorithm from the given training data (the dots).

Numerical representations are a prerequisite for *most* **machine learning** models (algorithms which learn to approximate **functions** that map inputs to outputs).





The machine learning model requires numerical values in and out, which are encoded from/decoded into corresponding symbolic terms

Embedding methods (alternatively referred to as “encoding”, “vectorising”, etc) convert **symbolic** representations (i.e. words, emojis, categorical items, dates, times, other features, etc) into *meaningful* numbers (i.e. real numbers that capture underlying **semantic relations** between the symbols).

one-hot encoding

If we wanted to embed a colour like “orange”, we could have a long vector wherein vector values correspond to the various colours of the rainbow.

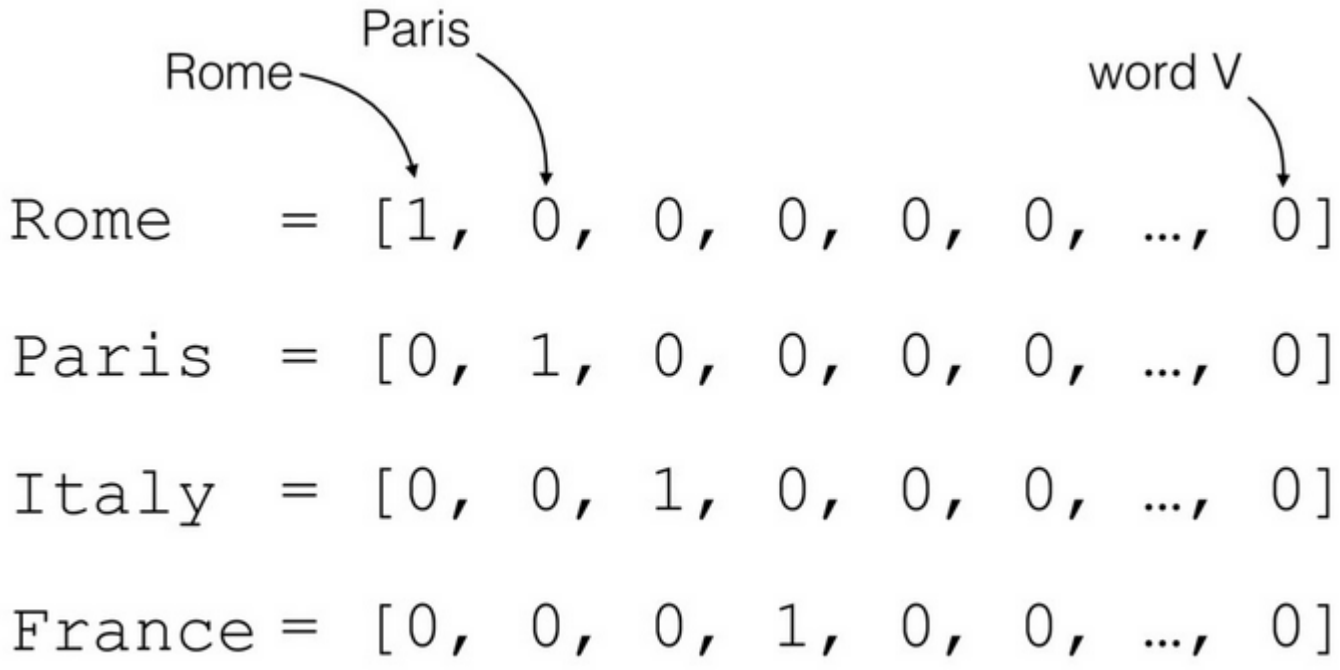
Each location in the vector represents a different colour

The location corresponding to the colour being embedded could be given the value “1” while all other values would be set to “0”. This is one of the simplest embedding methods and is known as **one-hot encoding** (because only one value in the vector has the value one).

```
orange = [1,0,0,0,0,0,0,0...]
```

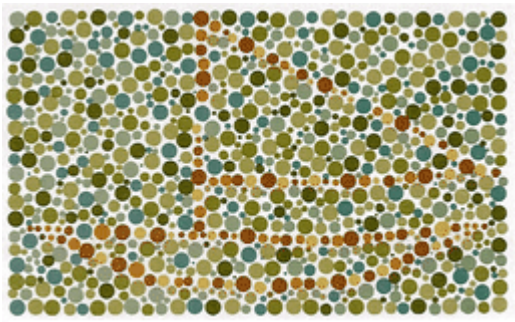
Unfortunately, this method is limited in a number of ways. For staters, one-hot vectors can be prohibitively large as we try to incorporate more and more colour variations. (If

embedding words in this manner, the size of the vector would be as large as the number of words in the entire dictionary!)



one-hot vectors are simple yet inefficient representations

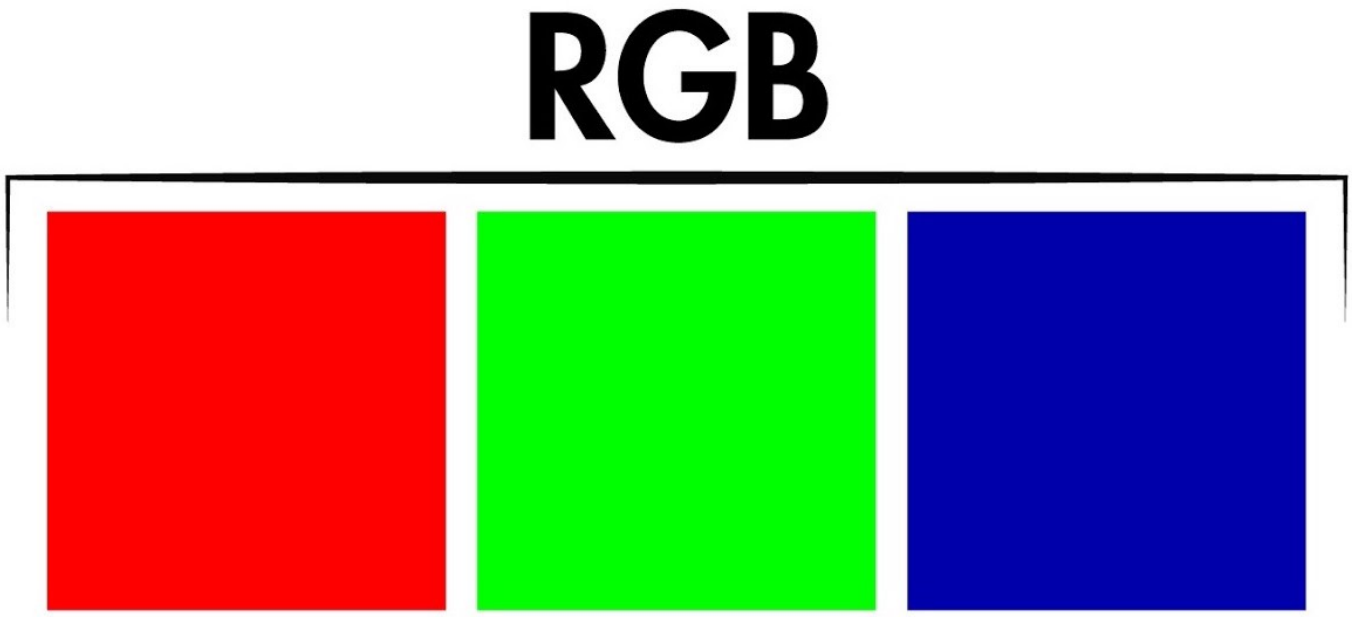
One-hot vectors also assume that there are no inherent relationship between any of the colours (or items) being embedded. For instance, if we measured the similarity between the vectors for “orange” and “red”, it would give us no difference to the similarity as measured between “orange” and “green”



Are all colours equally different from one another?

Feature vectors

Imagine another type of vector that is able to represent any colour with just three values each time (lets say the values represent red, green and blue).



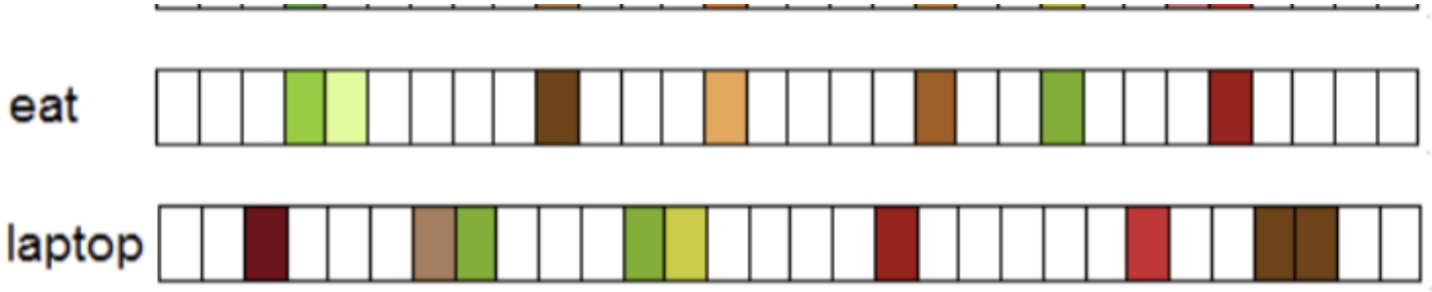
3-valued vector to represent any colour

```
red   = [1, 0, 0]
green = [0, 1, 0]
```

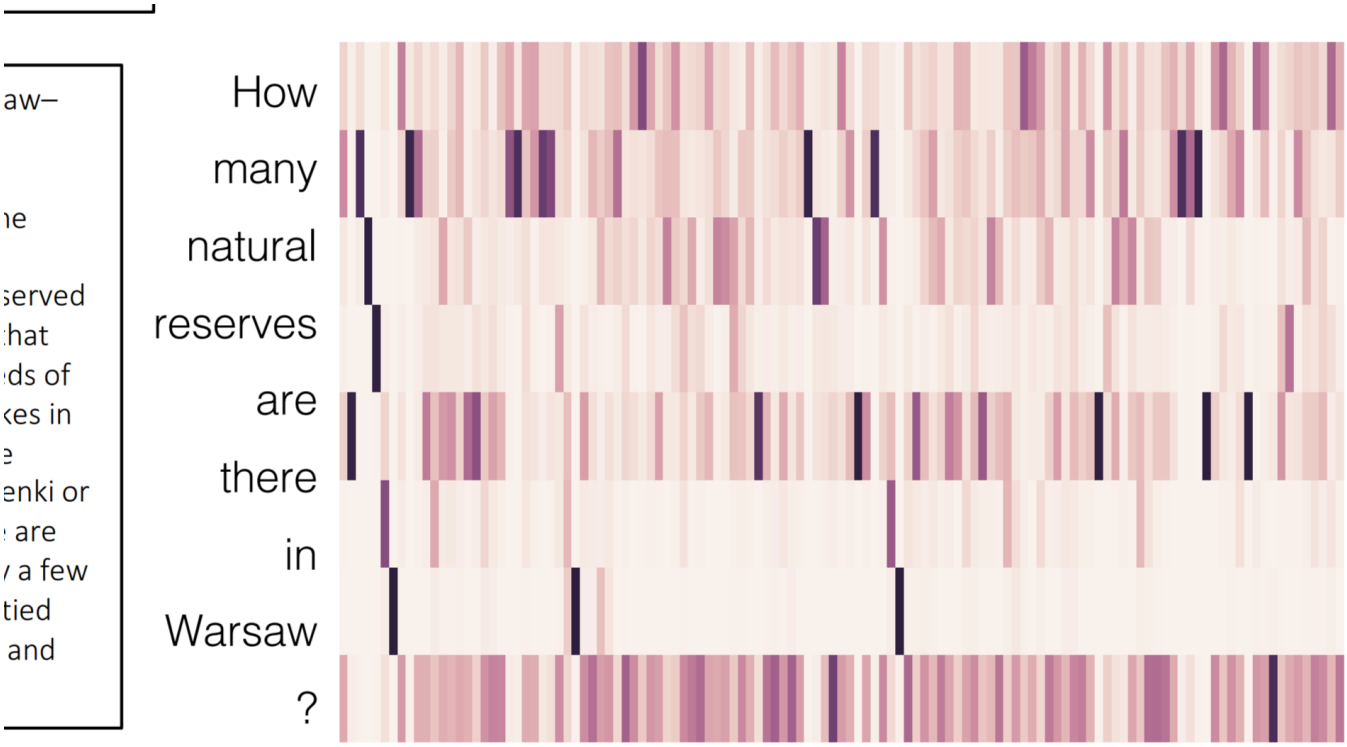
This vector could represent other colours too since all colours are actually some combination of these three primary colours (e.g. Orange would be red and a bit of green)

```
orange = [1, .5, 0]
```

When we compare the vectors for “red” and “green”, we get a very large difference, however the vectors for “red” and “orange” have a much smaller difference (correctly indicating that the two colours are more similar). Similarly, we can create fixed-length vectors that can represent other categorical items just like colours, or even words.

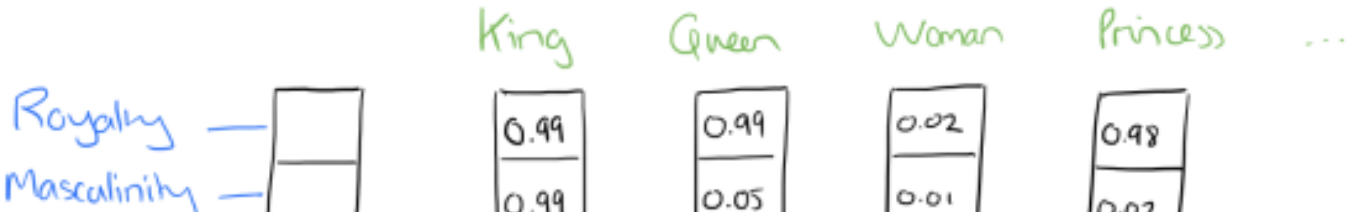


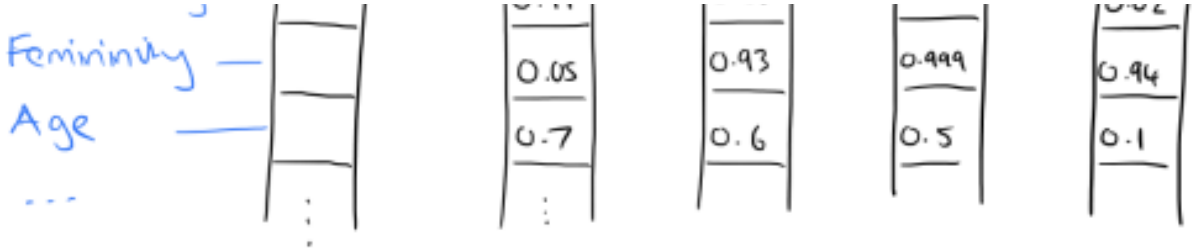
Distributed word vectors



A heat map representing the values in real word vectors

Word vectors typically range from between 50 to 300 values. A word vector of 100 values can represent 100 unique **features** (a feature can be anything that relates words to one another — e.g. a common Theme / Topic such as“Royalty”, “Masculinity”, “Femininity”, “Age”, etc.). Words are then represented as a distribution of its membership to each of the features.





E.g. the word "King" could be represented with high values for the features "Royalty" and "Masculinity" and low values for "Femininity", etc. Like the word "King", the word "Princess" would also be represented by a high value for the feature "Royalty" but, unlike "King", it would have a low value for "Masculinity". The word "Woman" would have a high value for the feature "Femininity" just like the word "Princess", but unlike "Princess" it would not have a high value for "Royalty". Etc.

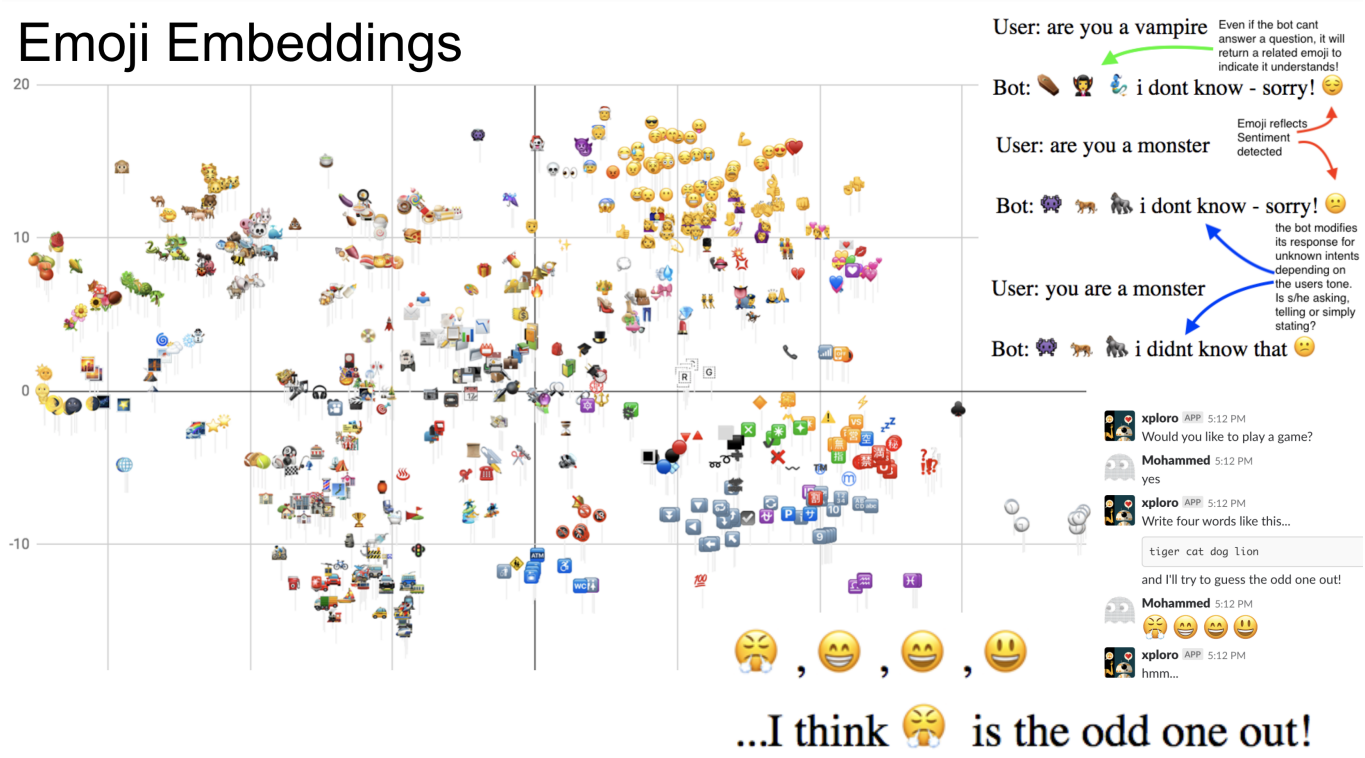
Let's take a real example using emojis. Suppose we chose the following four features ("Spring", "Summer", "Autumn" and "Winter") to represent an emoji as a vector with 4 values

Spring	🌳 🍃 🍂 🌾 🌿 🍁 🌱 🏔️ 🌄 🏕️
Summer	🌳 🌾 🌿 🍃 🏕️ 🌄 🍄 🌱 🌻 🏔️
Autumn	🎄 🎅 🎁 ⛄ 🌳 ❄️ 🍃 ❤️ 🌿 💚
Winter	🎄 ⛄ 🎅 🎁 ❄️ ❤️ 💚 🏕️ ✨ 😊

According to our matrix above, we can embed each emoji according to the features they possess

```
emoji = [spring, summer, autumn, winter]
🌳    = [1, 1, 1, 0]
🍂    = [1, 0, 0, 0]
🍁    = [1, 0, 0, 0]
🌻    = [0, 1, 0, 0]
🎁    = [0, 0, 1, 1]
🎅    = [0, 0, 1, 1]
...etc
```

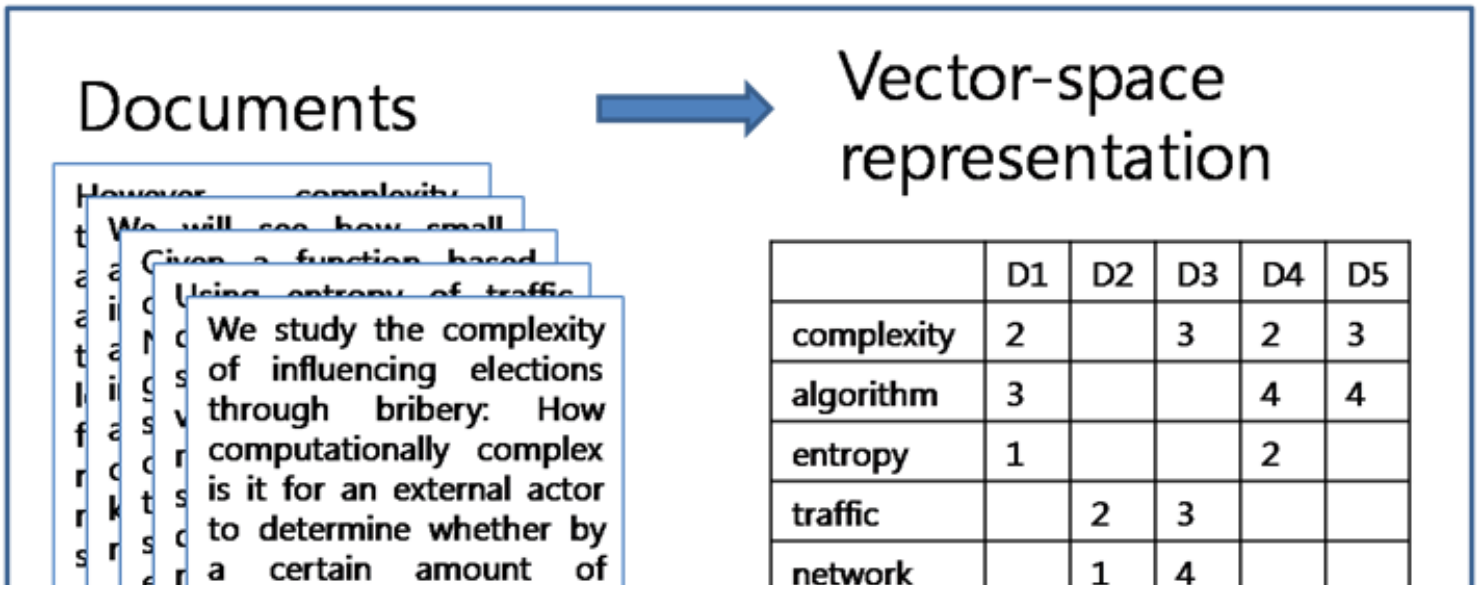
The embedded emojis can be compared with one another to identify emojis with similar meanings (e.g. 🎁 and 🎅 have identical vectors. So do 🍁 and 🍂. etc)

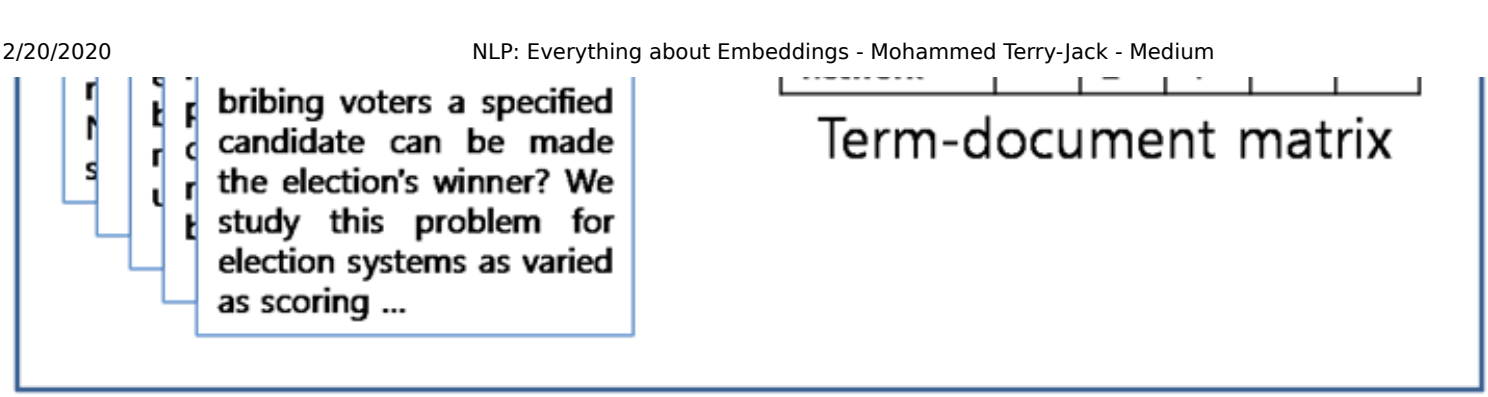


The emojis here are embedded into vectors with 50 features

Document Vectors

Features can also be more abstract relations, such as the context in which a word occurs (assuming words with similar contexts must have similar meanings). For instance, if we assume a document is generally on a single topic / theme, then we can safely take that entire document to be an unlabelled feature and generate distributed word vectors using documents as features. This is useful when the exact nature of the underlying features which unite the words are unknown.





Document-Term Matrix. In the above example, the embeddings for "traffic" [0,2,3,0,0] and "network" [0,1,4,0,0] would be almost identical.

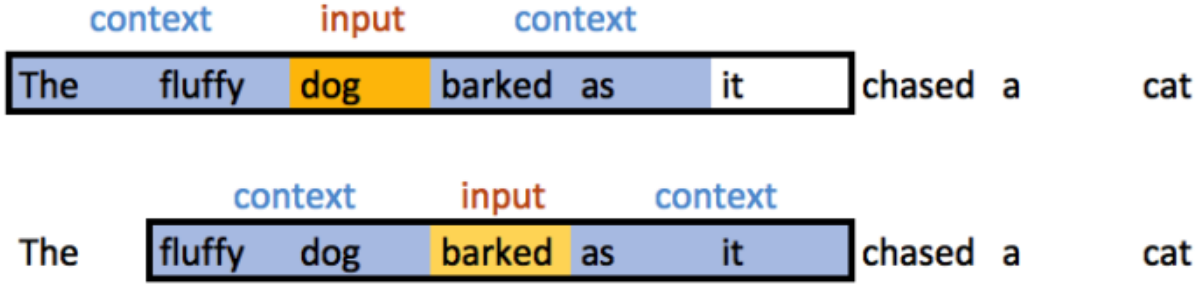
Cooccurrence Vectors

An even noisier method to embed words based on their context would be to use any **cooccurring** (surrounding) words. *(This would mean that the word vector's features are also words themselves!)*

	Terms					
Terms	data	examples	introduction	mining	network	package
data	53	5	2	34	0	7
examples	5	17	2	5	2	2
introduction	2	2	10	2	2	0
mining	34	5	2	47	1	5
network	0	2	2	1	17	1
package	7	2	0	5	1	21

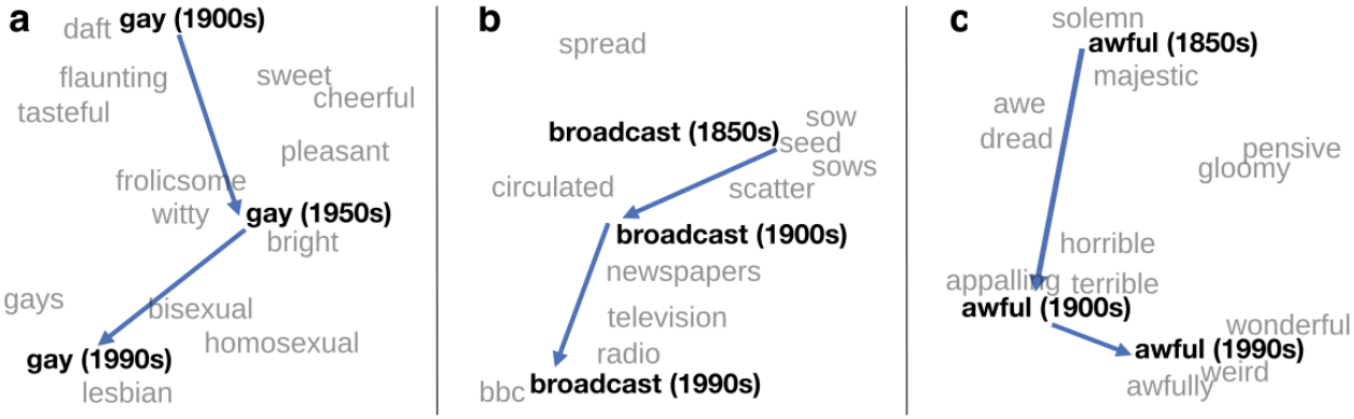
Co-occurrence Matrix. According to the example above, the embedding for "data" [53,5,2,34,0,7] and "mining" [34,5,2,47,1,5] are nearly identical (indicating the two words appear in similar contexts and thus have similar meanings)

One advantage of using this method is that it is very simple to get training examples. You simply scan a document and for each word take the surrounding words as features (using a **sliding window**)



A sliding window can be used to take the surrounding words (as vector features)

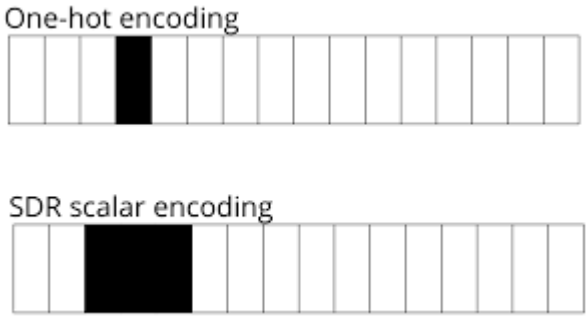
Another key advantage is that this type of embedding can accurately capture the sense in which the word is used (a word's usage can change depending on the time, community and context it is being used in)



Three examples ("gay", "broadcast", "awful") of how a word's usage has changed over time (Synonymous terms are shown in light grey beside the word)

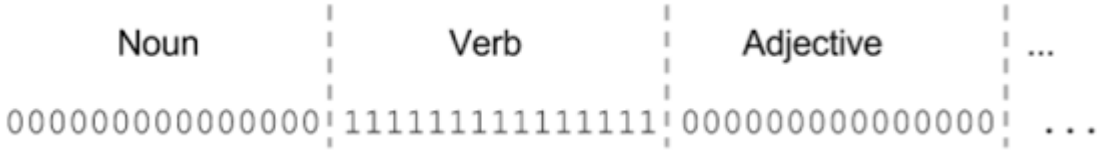
Sparse Distributed Representations

Sparse distributed representations (SDRs) are an interesting embedding variant modelled after the way the brain encodes information. Somewhat similar to one-hot encoding, the vector is only composed of only 0s and 1s. Unlike one-hot encoding, however, SDR features can be treated as "fuzzy" features because they occupy a range of values (and thus an SDR often has 1s in more than one location).

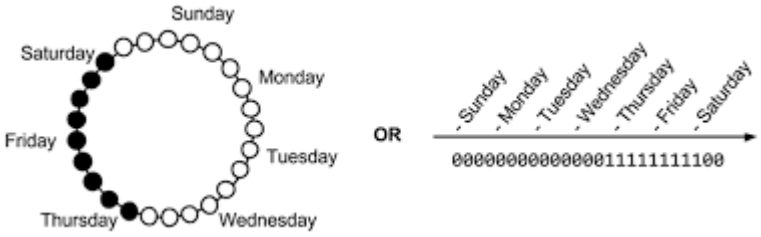


SDRs vs One-hot vectors

Increasing the range of values that a particular feature occupies is equivalent to increasing the influence of that feature over other features (in other vector representations, all features are assumed equally influential!)

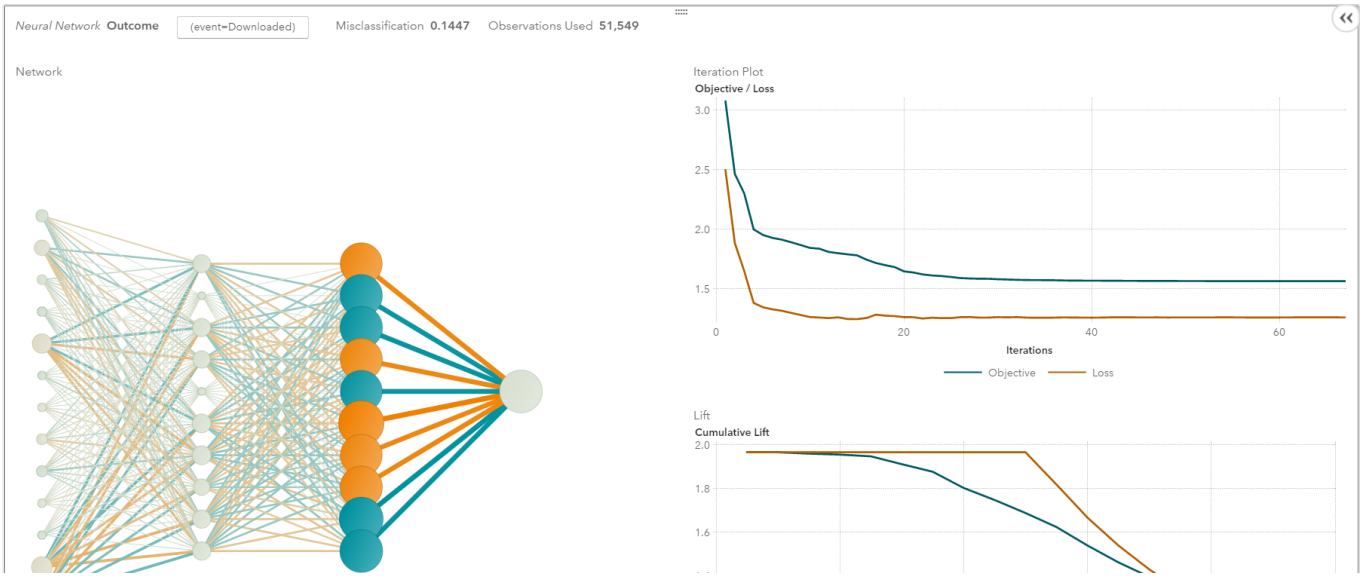


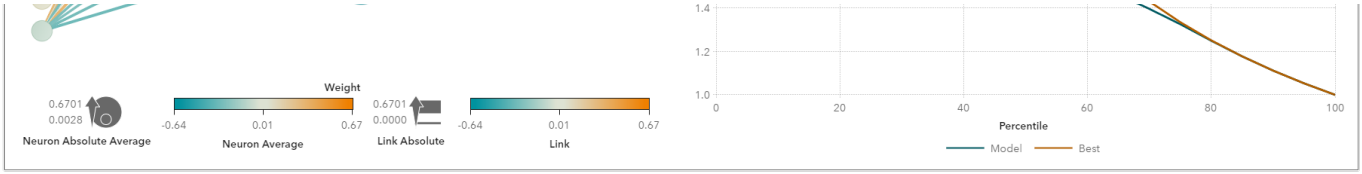
You can also encode something as having a part of a feature (E.g. the example below shows an embedding for a date which lasts from Thursday, Friday and only *part* of Saturday.)



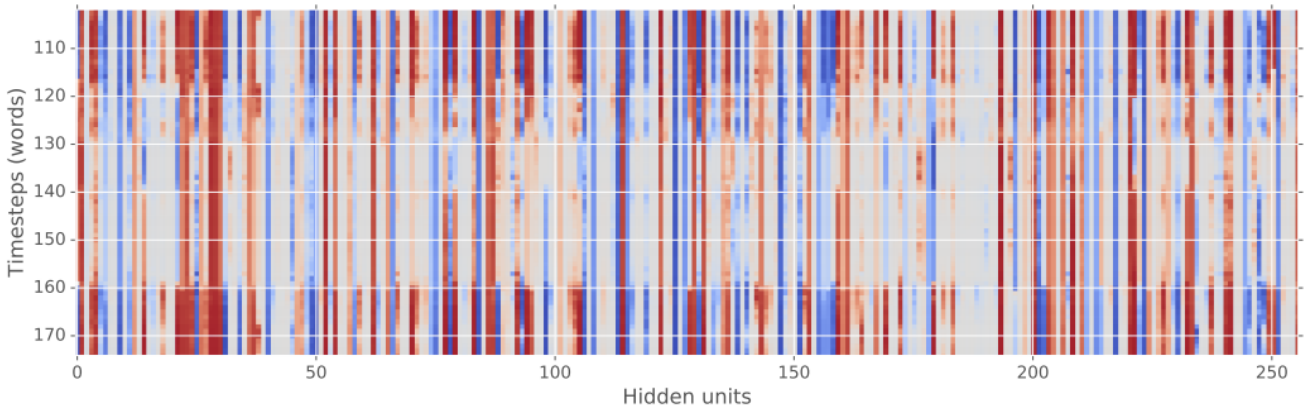
Neural Embedding Methods

When a **Neural Network** has learnt a task well (e.g. predicting the sentiment of a word), we can assume that its **hidden layers** have learnt useful features that have helped it better *understand* how the input data(e.g. words) relate to the output data (e.g. sentiments).

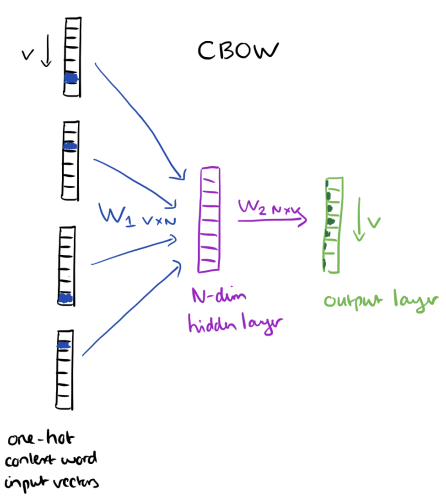


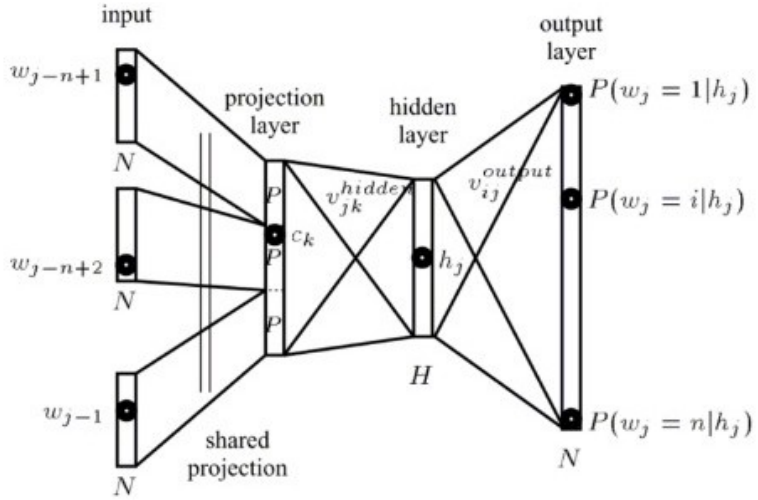


Thus we can run a word through a trained network and extract the values of the hidden layer as the word's embedding. Although the meaning of the neural network's learnt features are a mystery to us, they do not need to be known so long as all our information is consistently embedded by the same network. Nevertheless, research has shown that the features and relations learnt are similar to those we would traditionally chose (e.g. countries are clustered close together and syntactically similar words occupy similar locations in the vector space.)

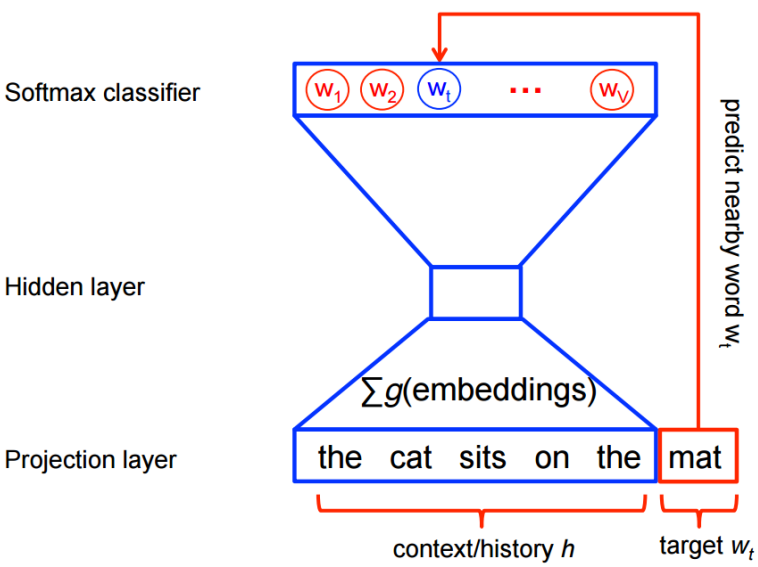


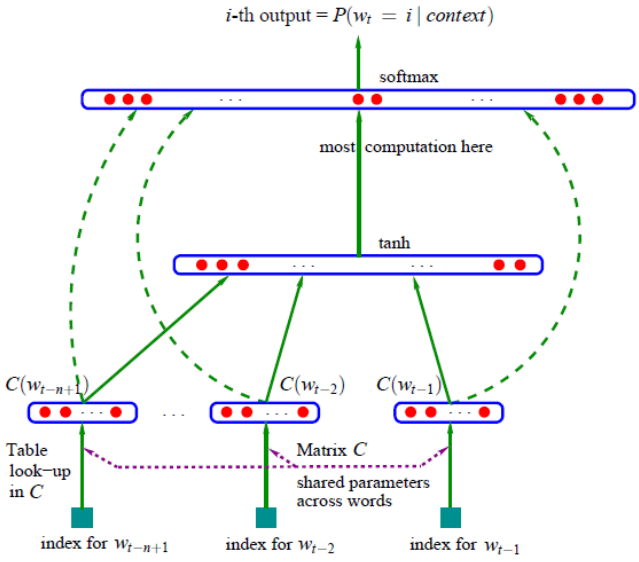
One of the earliest neural embedding methods was the **Continuous Bag of Words (CBOW)** model.



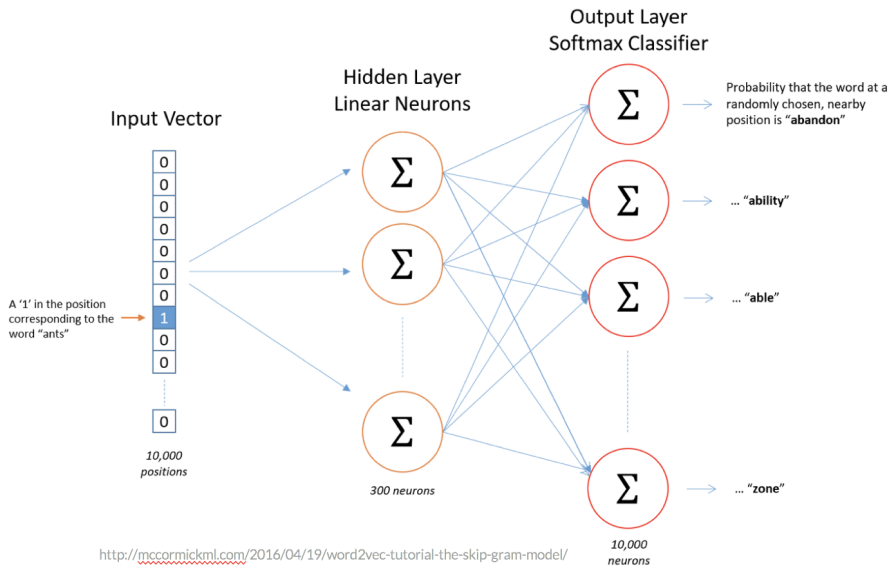


This particular method uses a neural network to predict a word (output) given a context of words (inputs). To do so successfully, the network needs to learn to model the target language





It was actually shown that training a network to predict the context of words given a particular word (the exact reverse of the task above) produces more meaningful embeddings. This is known as the **skip-gram** model.

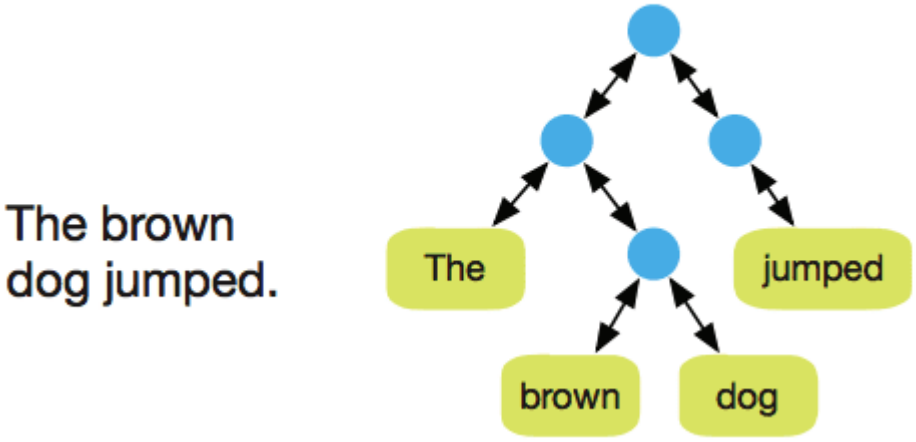


"Skip-Gram"
With one-hot encoded centre word, we can predict context words.

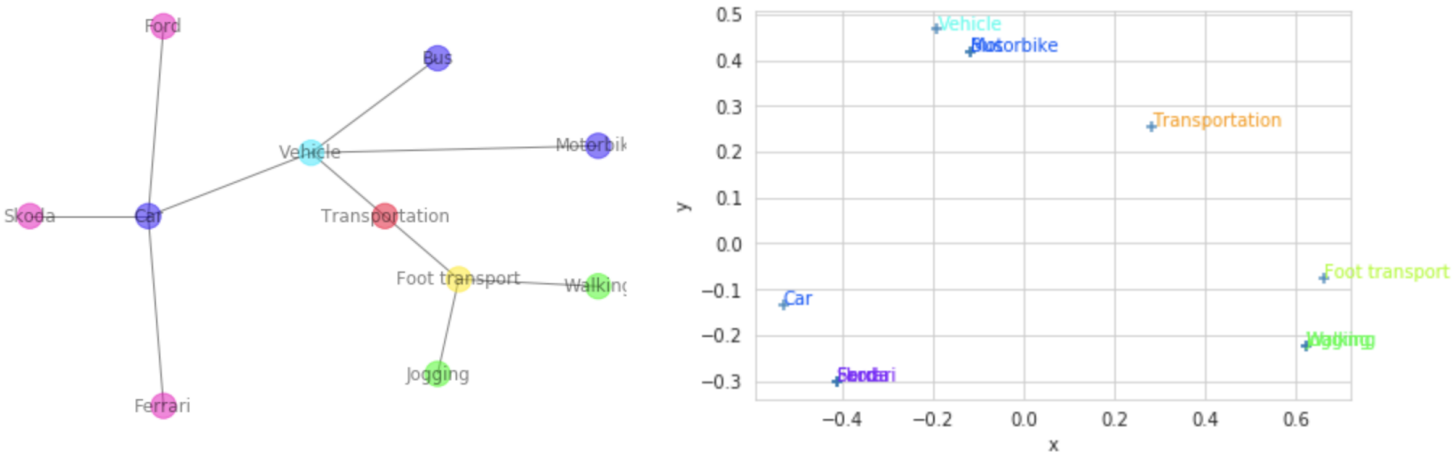
Hidden layer creates embeddings

Graph Embeddings

Sentence and Parse Tree



Graphs (like WordNet) are a powerful way to represent relationships between things like words. We can use these graphs to generate word vectors too! (*This is a very interesting field of research - I'll dedicate a separate post to explain some methods for embedding knowledge from graphs into a vector space*)

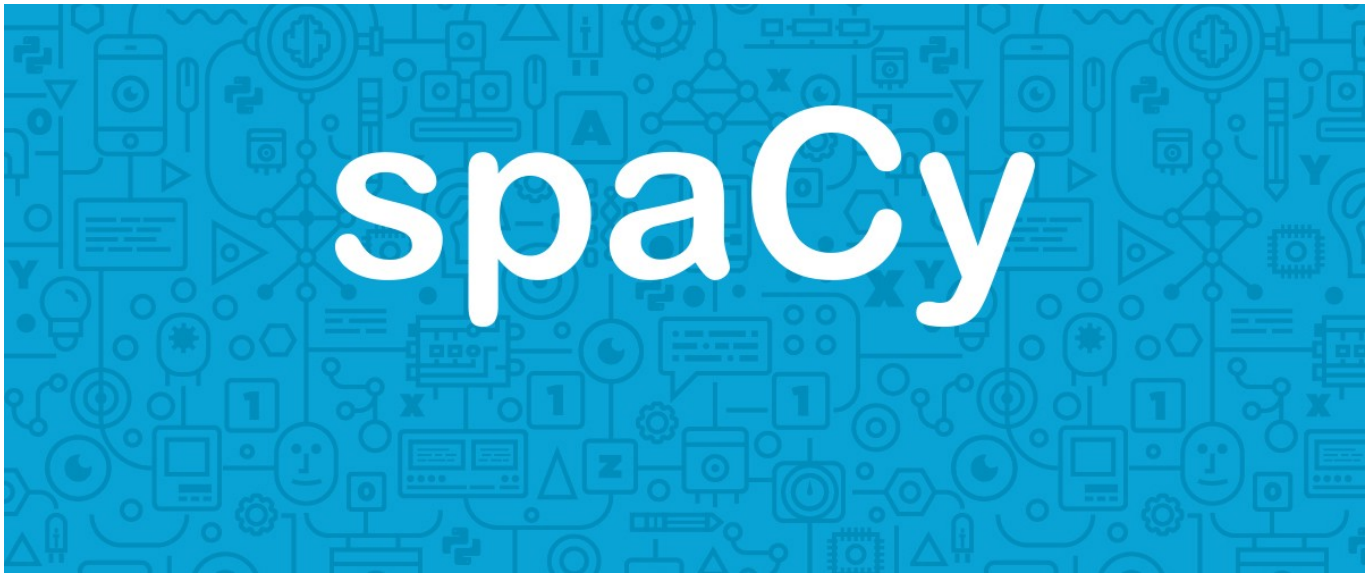


(left) graph of words and their relations to one another. (right) vector representations of those words plotted on a 2d graph

Pre-trained Word Embeddings

There are a number of pre-trained word vectors available, such as:

SpaCy (word2vec)



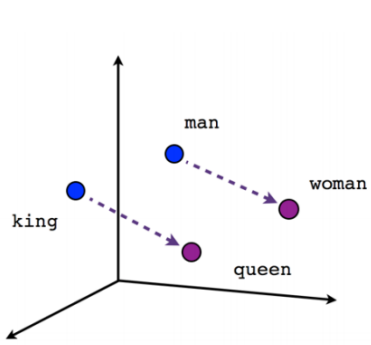

```
import spacy
!python3 -m spacy download en_core_web_lg
sp = spacy.load('en_core_web_lg')

sp(word).vector
```

FastText

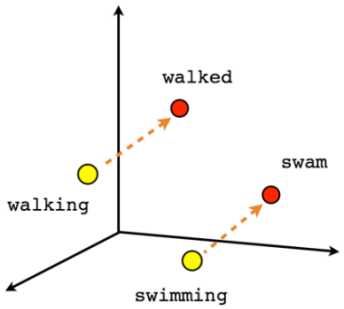
```
!pip3 install mxnet
!pip3 install gluonnlp
import gluonnlp
```


```
fasttext = gluonnlp.embedding.create('fasttext',
source='wiki.simple')
fasttext[word].asnumpy()
```

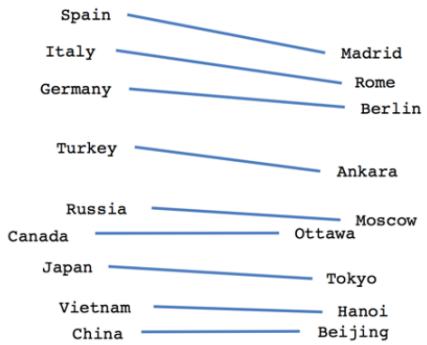
GloVe



Male-Female



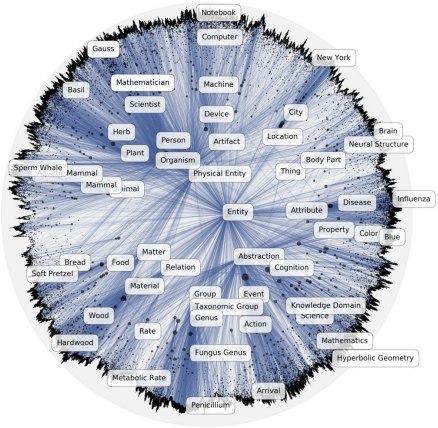
Verb tense

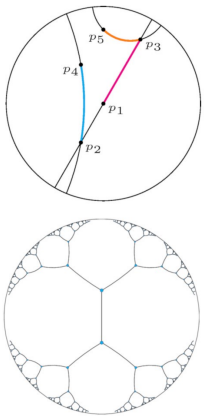


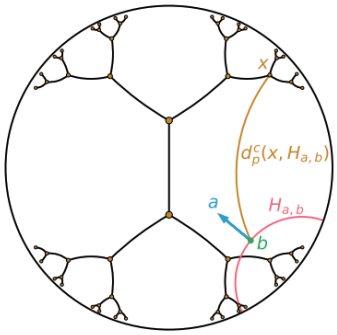
Country-Capital

```
glove = gluonnlp.embedding.create('glove', source='glove.6B.300d')
glove[word].asnumpy()
```

Poincare







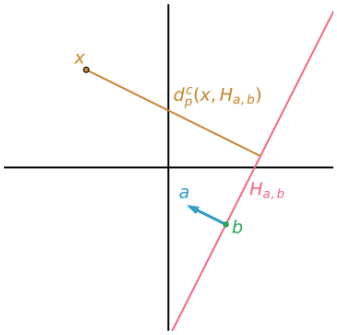


Figure 3: Illustration of an orthogonal projection on a hyperplane in a Poincaré disc $\mathbb{B}_c^2$ (Left) and an Euclidean plane (Right). Those hyperplanes are *decision boundaries*.

It was shown that hierarchical relations are better embedded using a non-euclidean space

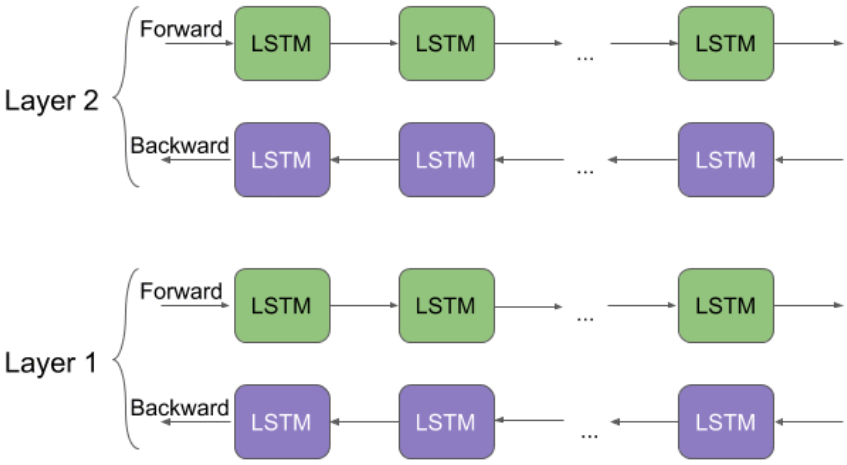
```
#https://polybox.ethz.ch/index.php/s/TzX6cXGqCX5KvAn #download
poincare vectors from here

with open('poincare.txt') as f:
  lines = f.readlines()
```

```
poincare = {line.split()[0]: [float(x) for x in line.split()[1:]] for
line in lines}

poincare[word]
```

Numberbatch


```
!wget
https://conceptnet.s3.amazonaws.com/downloads/2017/numberbatch/number
batch-17.06.txt.gz
!gunzip numberbatch-17.06.txt.gz

from gensim.models.keyedvectors import KeyedVectors

numberbatch = KeyedVectors.load_word2vec_format("numberbatch-
17.06.txt", binary=False)
numberbatch[f"/c/{en}/{word}"]
```

Elmo (Language Model)

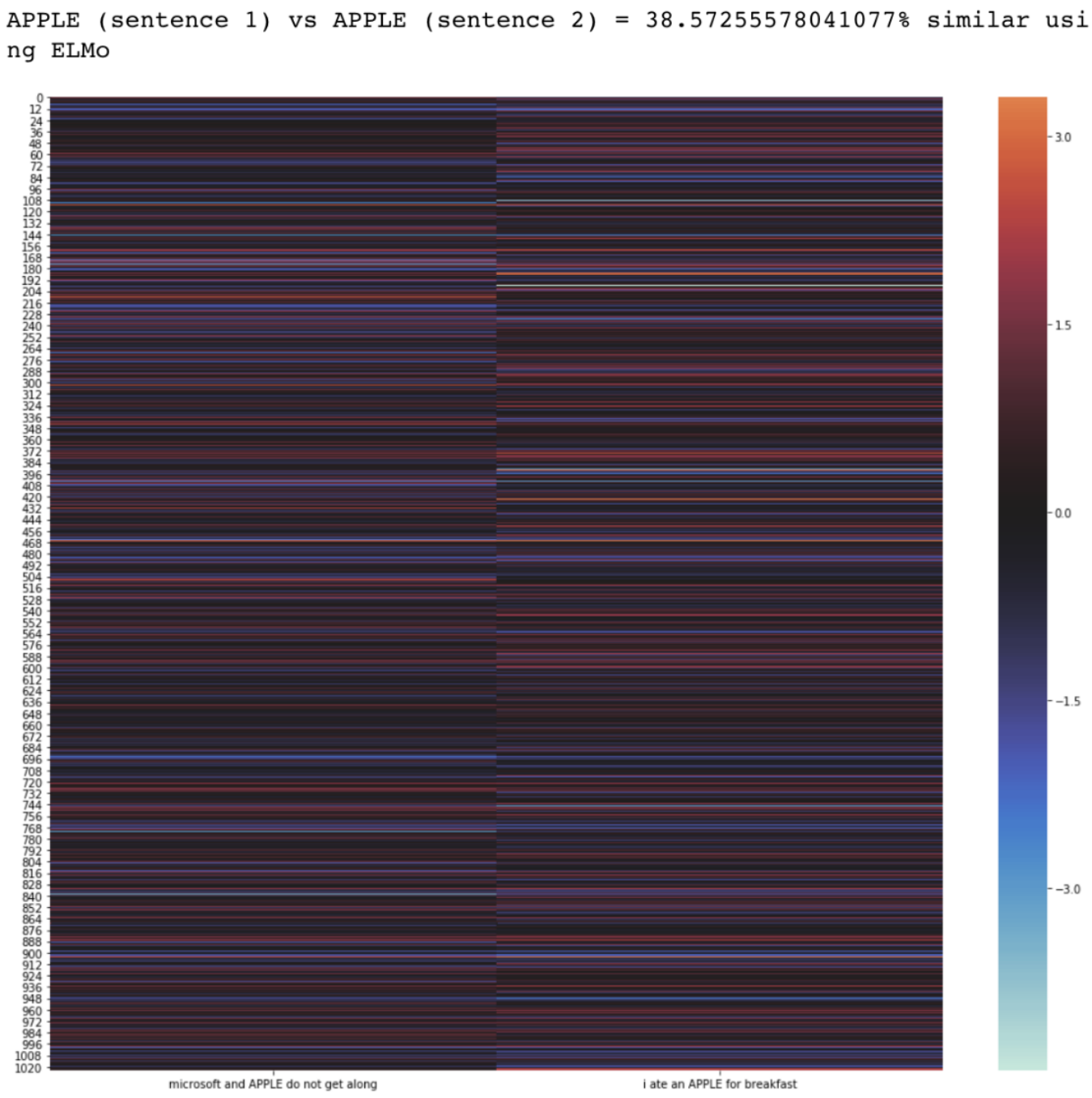


```
!pip3 install allennlp
from allennlp.commands.elmo import ElmoEmbedder
elmo = ElmoEmbedder()

elmo.embed_sentence(words.split())[-1]
```

Flair

```
!pip3 install flair
from flair.embeddings import FlairEmbeddings, StackedEmbeddings,
Sentence
stacked_embeddings = StackedEmbeddings([FlairEmbeddings('news-
forward'), FlairEmbeddings('news-backward')])

s = Sentence(words)
stacked_embeddings.embed(s)
[token.embedding.detach().numpy() for token in s]
```

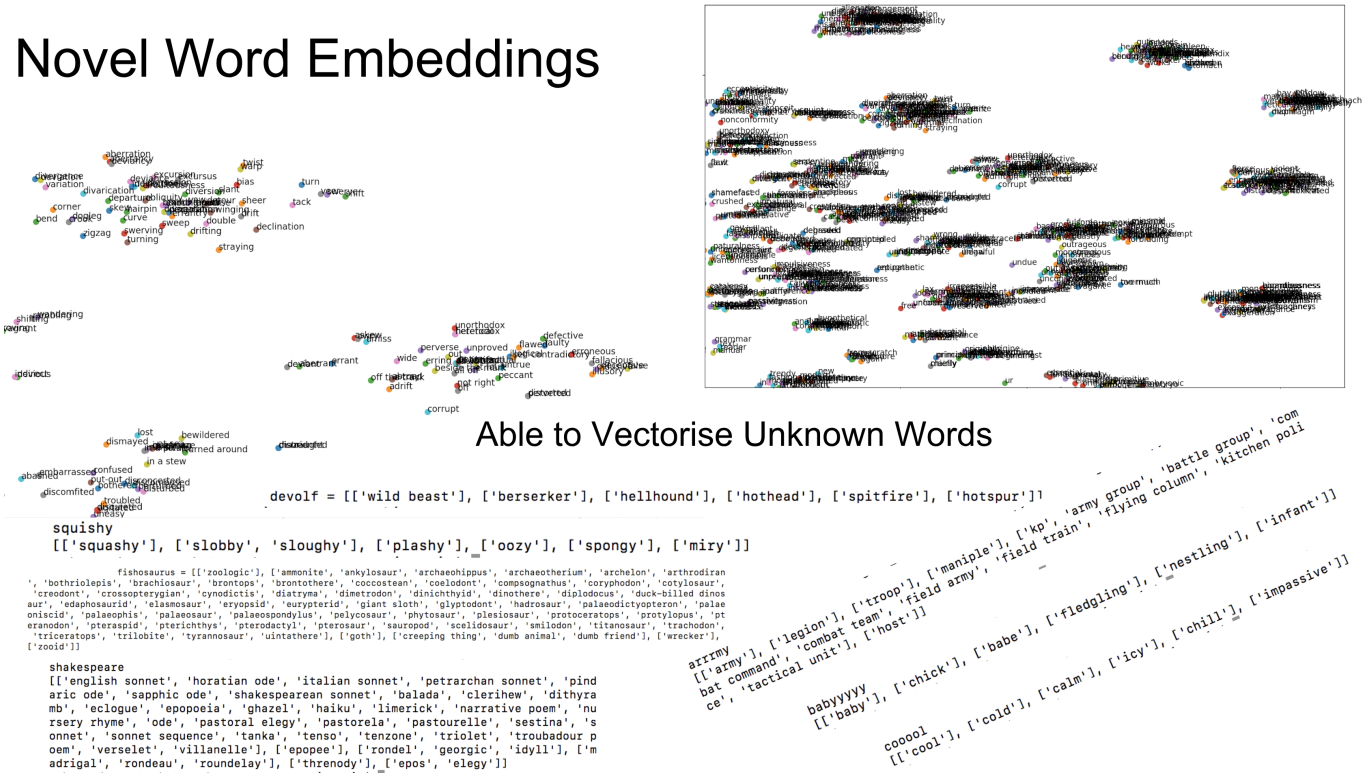
The advantage of embedding methods like flair and elmo is that they also consider a word's context when generating its vector representation. Unlike most embedding methods, a different word vector is generated for the word "apple" in the sentence "Microsoft and Apple do not get along" than for the word "apple" in the sentence "I ate an apple for breakfast".



The word vector for the word "apple" in one sentence is different from the vector for "apple" in another context

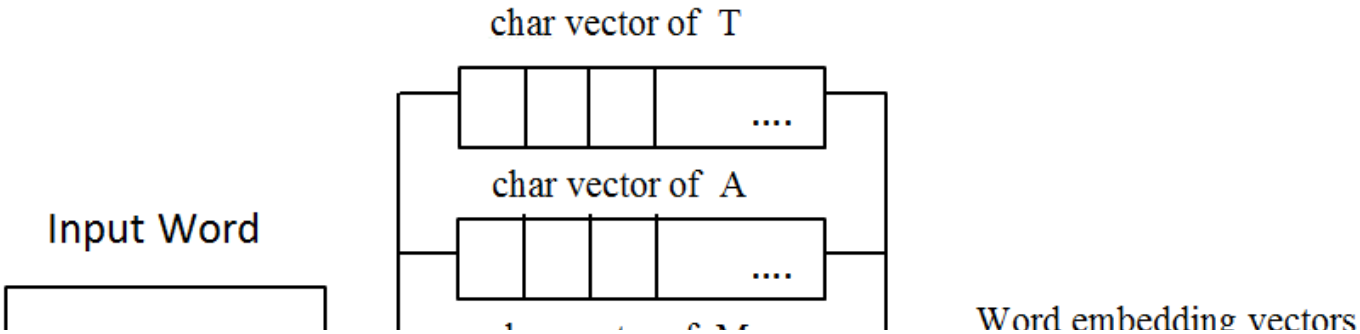
Embedding Out-of-Vocab words

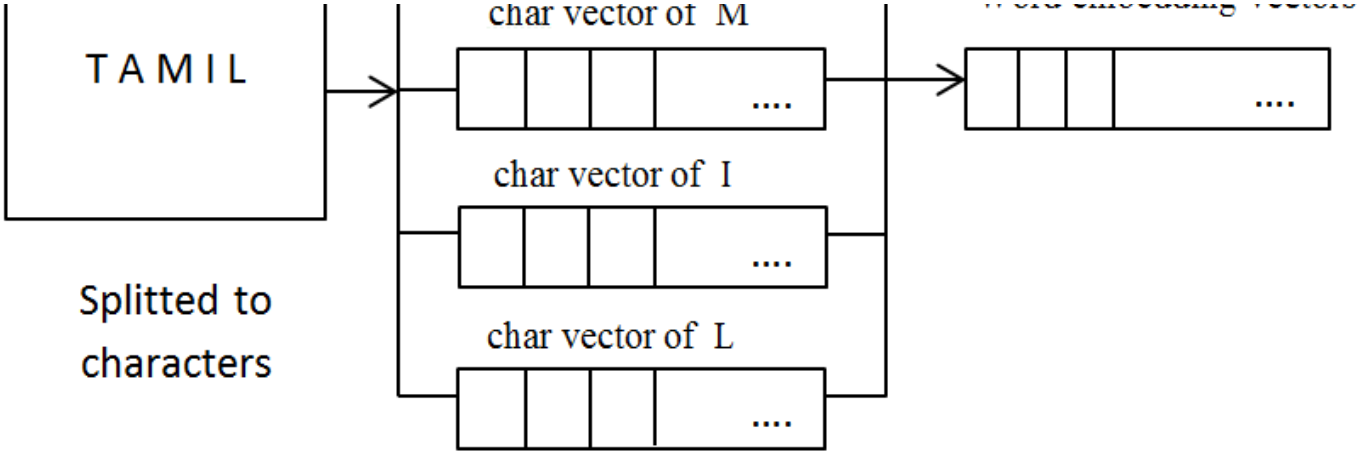
Inevitably, there will be words which are out-of-vocab (OOV) and will thus lack a vector representation. There are methods for approximating the meaning of such words so that made-up words (like “devolf”) can be understood to mean something similar to “wild beast” and “hellhound”. It can also make a system more robust to typos like “arrrrmy” and “babyyyy”, etc.



Methods for embedding OOV words can create word vectors for made-up words like ’devolf’ that are similar in meaning to the vectors for ’wild beast’ and ’hellhound’. It can also allow typos like ’arrrmy’ to be embedded nearby to words like ’army’, ’legion’ and ’troop’.

One method would be to approximate an OOV word’s embedding from pre-trained character-level embeddings (i.e. by combing pre-trained embeddings for character-level ngrams).

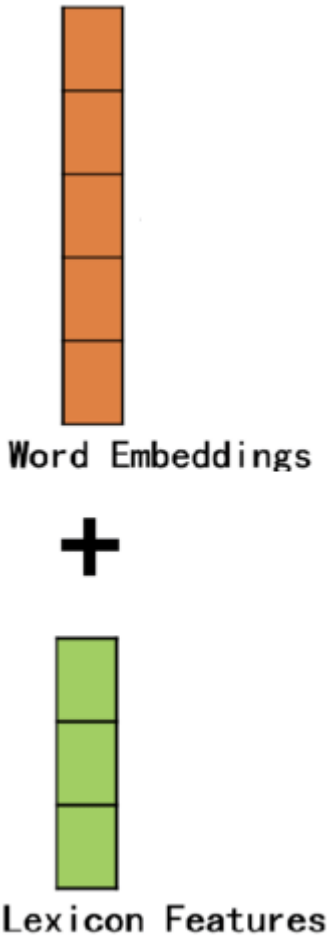




constructing a word vector for an OOV word "TAMIL" by combining character-level embeddings

Enriching Word Vectors

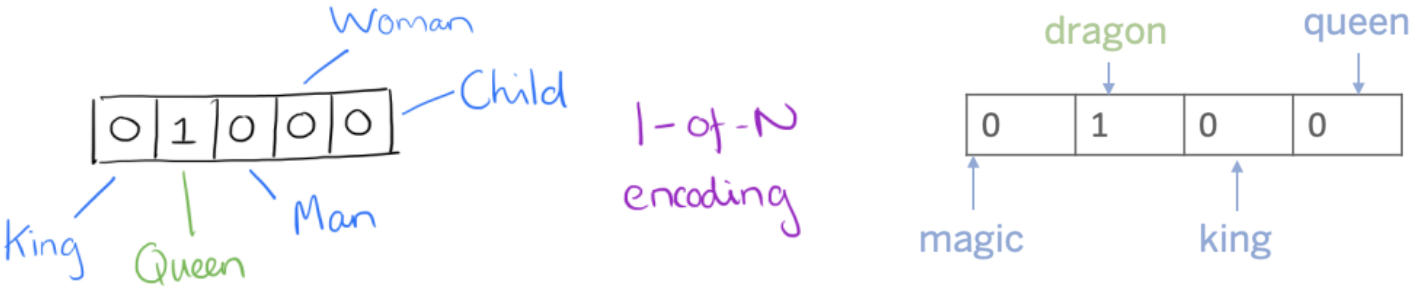
It is always a good idea to extract additional features from a word and append the values onto the end of a pre-trained word vector. Additional features may include: Title-Casing flag, upper case flag, word sentiment, word subjectivity, Named Entity Type (GPE, PERSON, ORG, etc), mood type, question flag, total number of characters, total number of characters without punctuation, total number of syllables, etc.



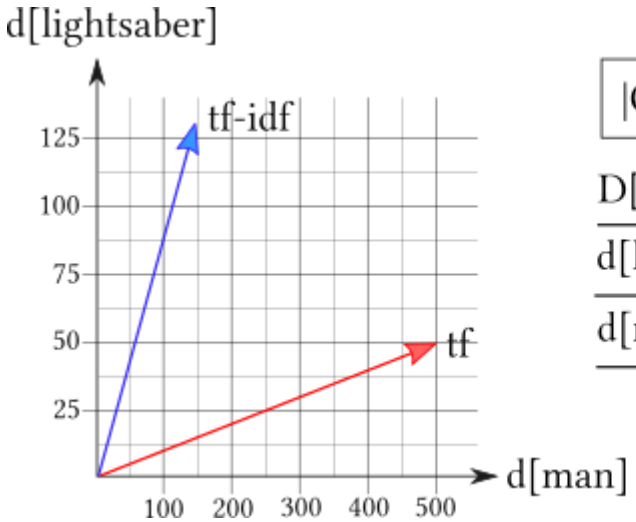
appending additional engineered features to enrich word vectors

Paragraph Embeddings

When it comes to embedding more than one word into a single, fixed-length vector, there are a few approaches to try. The most basic approach is the **bag-of-words** method which takes a vector the length of the vocab and each word occupies a position in the vector. For a given sentence, each word is counted and the frequencies are recorded in the relevant positions in the vector



A slightly better variant to this is a **Tf-Idf** vector, which uses the term-frequency inverse-document frequency (tf-idf) for each word rather than their frequencies. This essentially weights words by their rareness as well as their frequency (e.g. very common words that appear in every context like “man” will be given less importance than relatively rarer words like “lightsaber”, even if the rarer word appeared less frequently)

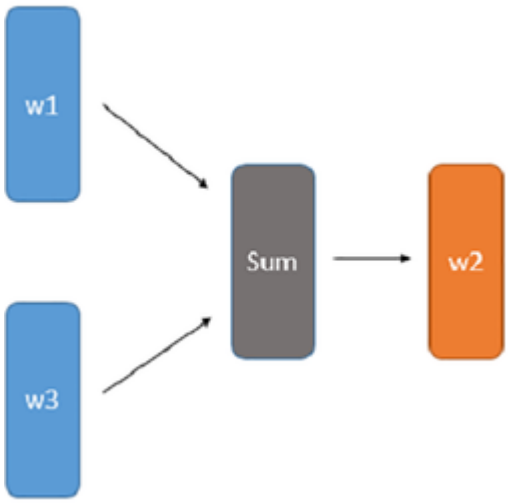


|C|=2e3

D[StarWars]	tf	df	idf	tf-idf
d[lightsaber]	50	6	2.52	126
d[man]	500	1e3	0.30	150

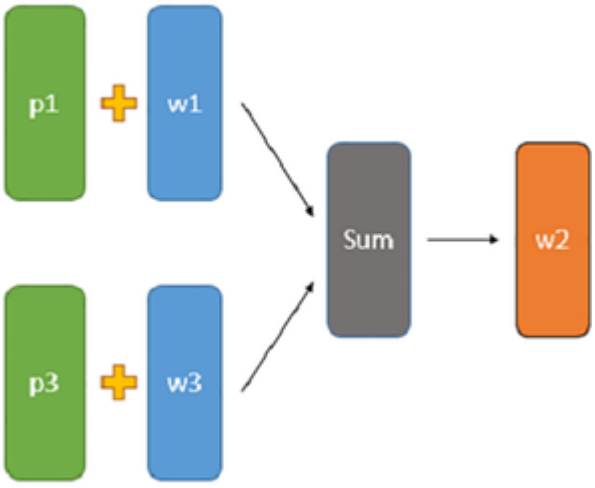
even though the word "man" appeared 500 times in a document, it receives a modest tf-idf score of 150 due to its commonness, as opposed to the rarer word "lightsaber" which received an almost equally important tf-idf score of 126 despite appearing only 50 times (ten times less than the word "man").

However, the problem with the above two methods is that each word is counted as a completely unrelated item. Other methods involve embedding each word (as discussed above) and then *merging* them together into a single, fixed-length paragraph vector.



merging multiple vectors into a single, fixed-length vector via sum pooling

When it comes to calculating the combined vectors values, it is simply the mean, max, min or sum of all the other vectors (the choice is yours). An interesting variant is to also add weights (e.g. tf-idfs) to each word vector so that the paragraph vector is less influenced by some words than others.



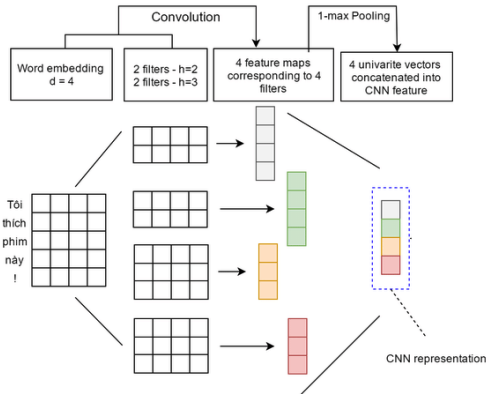
weighting each vector before merging them

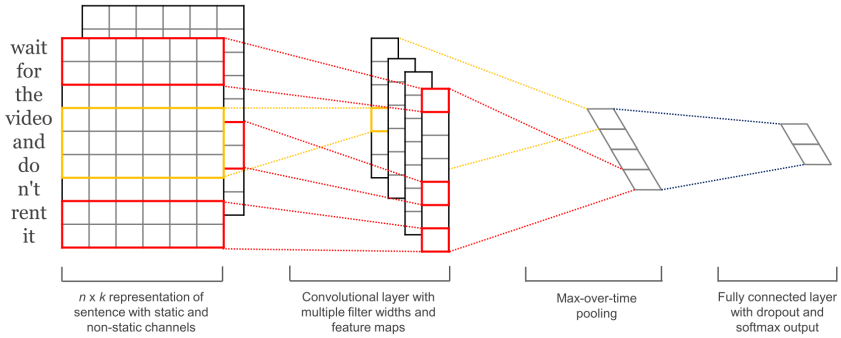
Some prefer to combine the vectors in various ways (mean pooling, max pooling, min pooling) and then taking them all via concatenation.

$$\mathbf{h}_c = [\mathbf{h}_T, \texttt{maxpool}(\mathbf{H}), \texttt{meanpool}(\mathbf{H})]$$

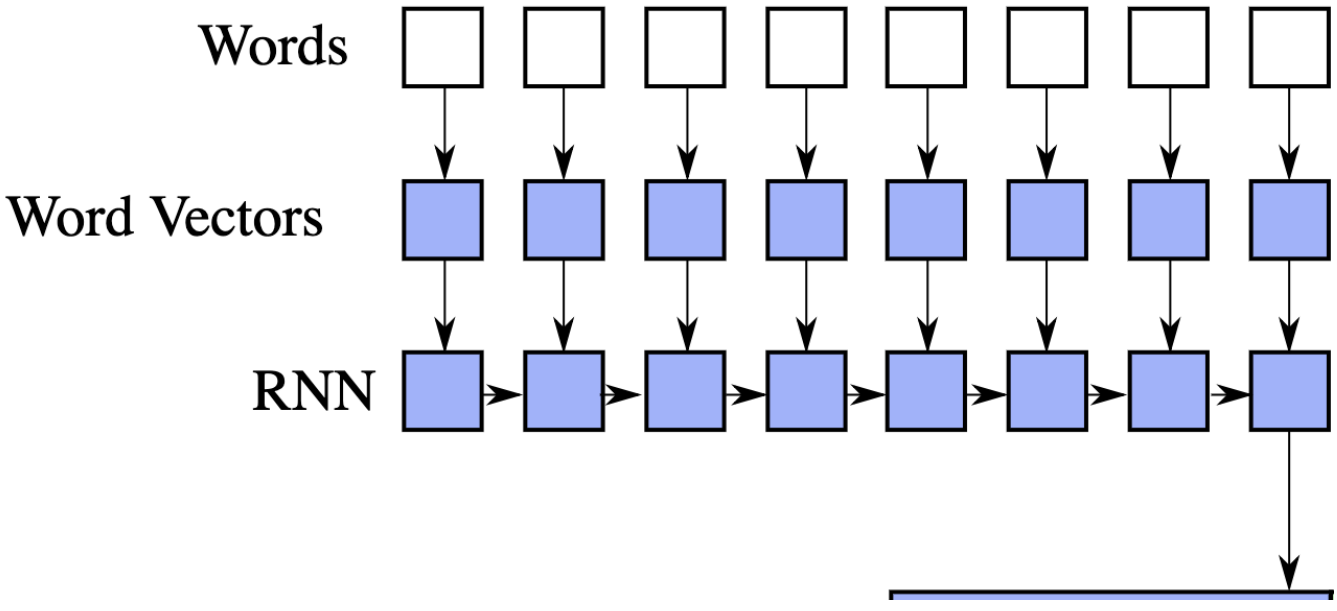
Concatenate layers. hT is the hidden state at the last time step. (Howard and Ruder, 2018)

Alternatively, there are more sophisticated methods to merge word vectors, involving a neural network (or a combination of them)





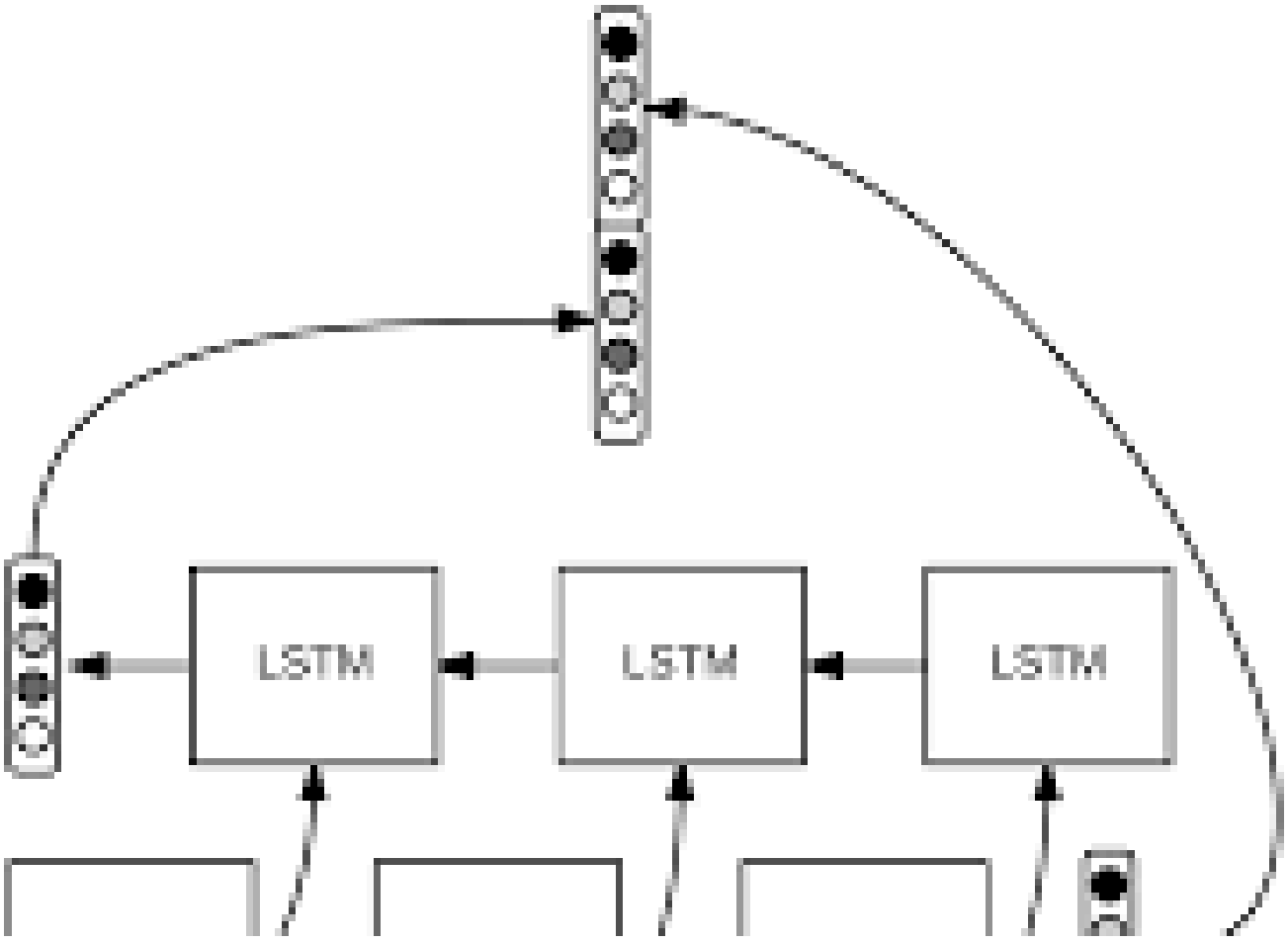
One such method is to use a **Convolutional Neural Network** (CNN) — you can find a simple code implementation of this method here: https://github.com/keras-team/keras/blob/master/examples/imdb_cnn.py

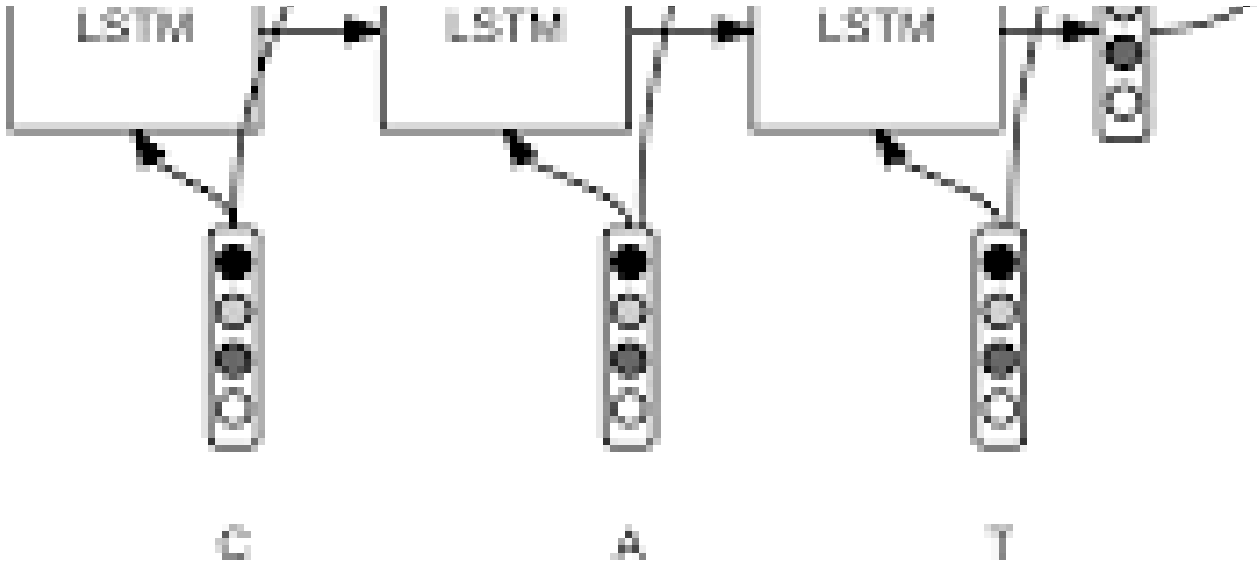


Merged output

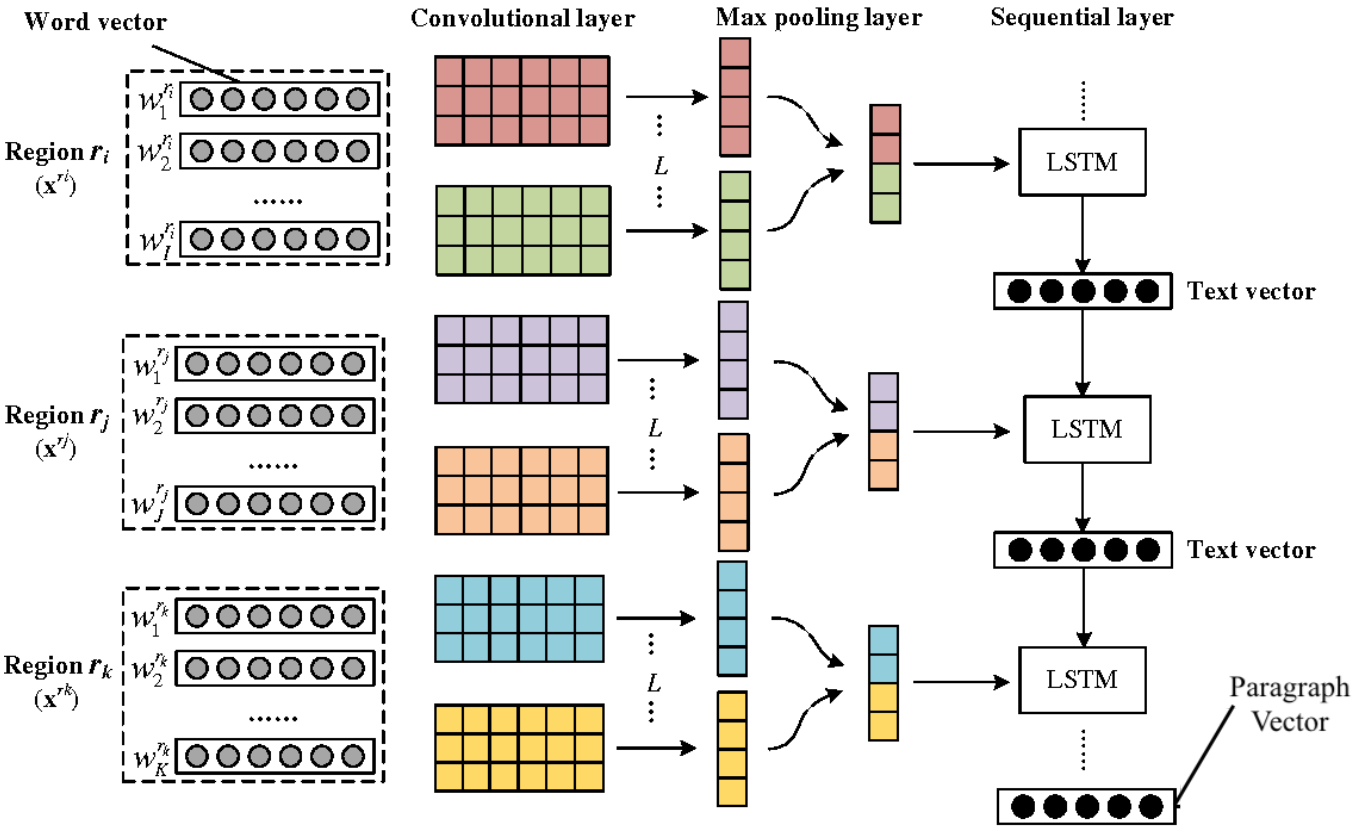
Alternatively, you can use a **Recurrent Neural Network** (RNN) and take the final RNN cell's value as your paragraph embedding (due to the nature of RNNs and sequence models, the final cell will contain a combination of all the values before it)

```
!pip3 install flair
from flair.embeddings import WordEmbeddings, DocumentRNNEmbeddings,
Sentence

glove_embedding = WordEmbeddings('glove')
document_embeddings = DocumentRNNEmbeddings([glove_embedding])

def rnn_pool(sentence):
  s = Sentence(sentence)
  document_embeddings.embed(s)
  return s.get_embedding().detach().numpy()
```



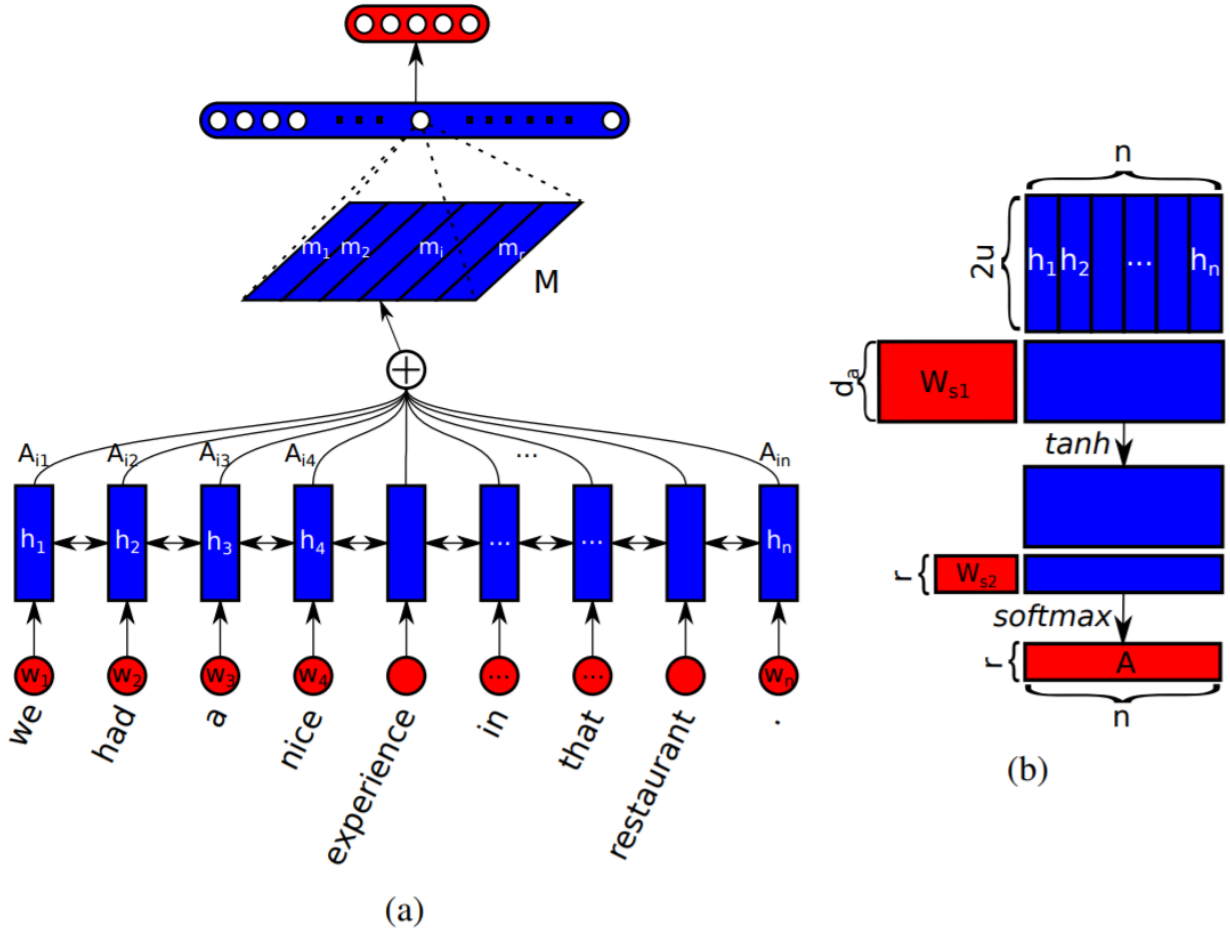


A similar, yet slightly more sophisticated method, is to use a bi-directional **Long Short-Term Memory** Network (bi-LSTM) and take the values for the final LSTM cells going in each direction, then concatenate the two outputs together to form your single, fixed-length paragraph vector



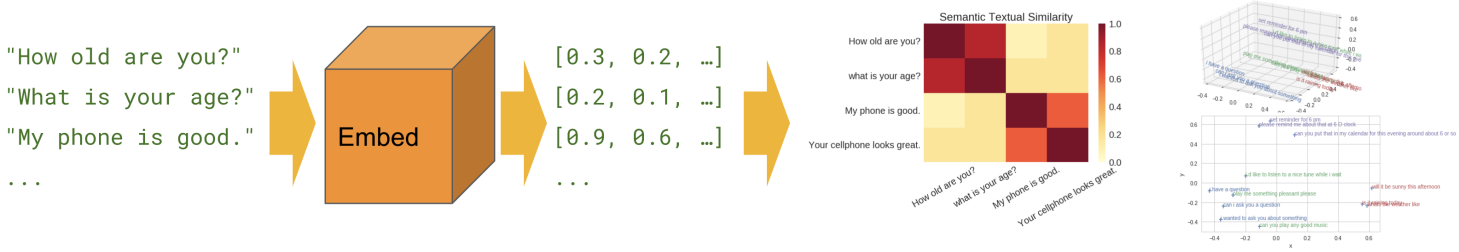
This method uses a CNN with an LSTM

Alternative merging methods include **Random Projections**, Stacked **AutoEncoders** and **Structured Self-Attentive** Sentence Embeddings.



Universal Encoder

There are also neural models specifically trained to produce fixed-length paragraph embeddings from variable length sentences, like the **Universal Sentence Encoder**



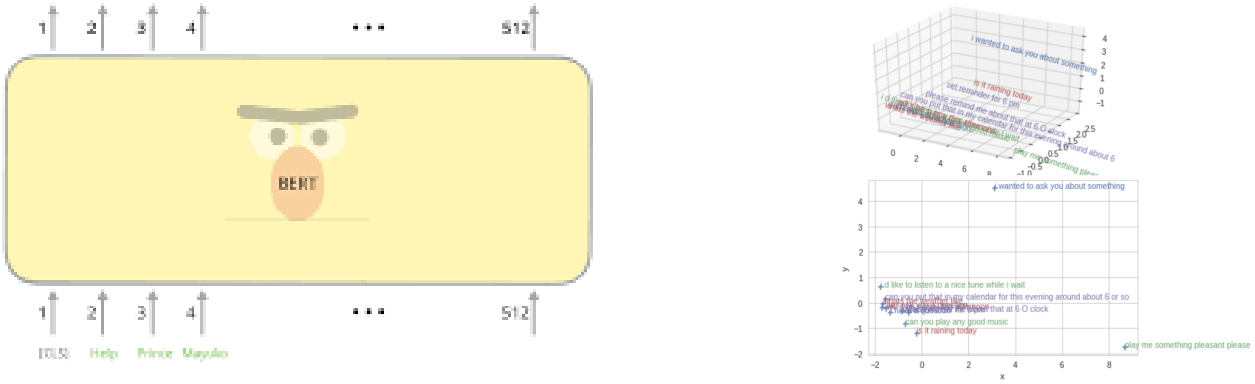
Universal Sentence Encoder

```
import tensorflow as tf
```

```
import tensorflow_hub as hub

tf.logging.set_verbosity(tf.logging.ERROR)

module_url = "https://tfhub.dev/google/universal-sentence-encoder-
large/3"
embed = hub.Module(module_url)

with tf.Session() as session:
  session.run([tf.global_variables_initializer(),
tf.tables_initializer()])
  paragraph_vector = session.run(embed([sentence]))
```

BERT

We can also extract paragraph embeddings from the fixed-length hidden layers of any pre-trained language models like **BERT**



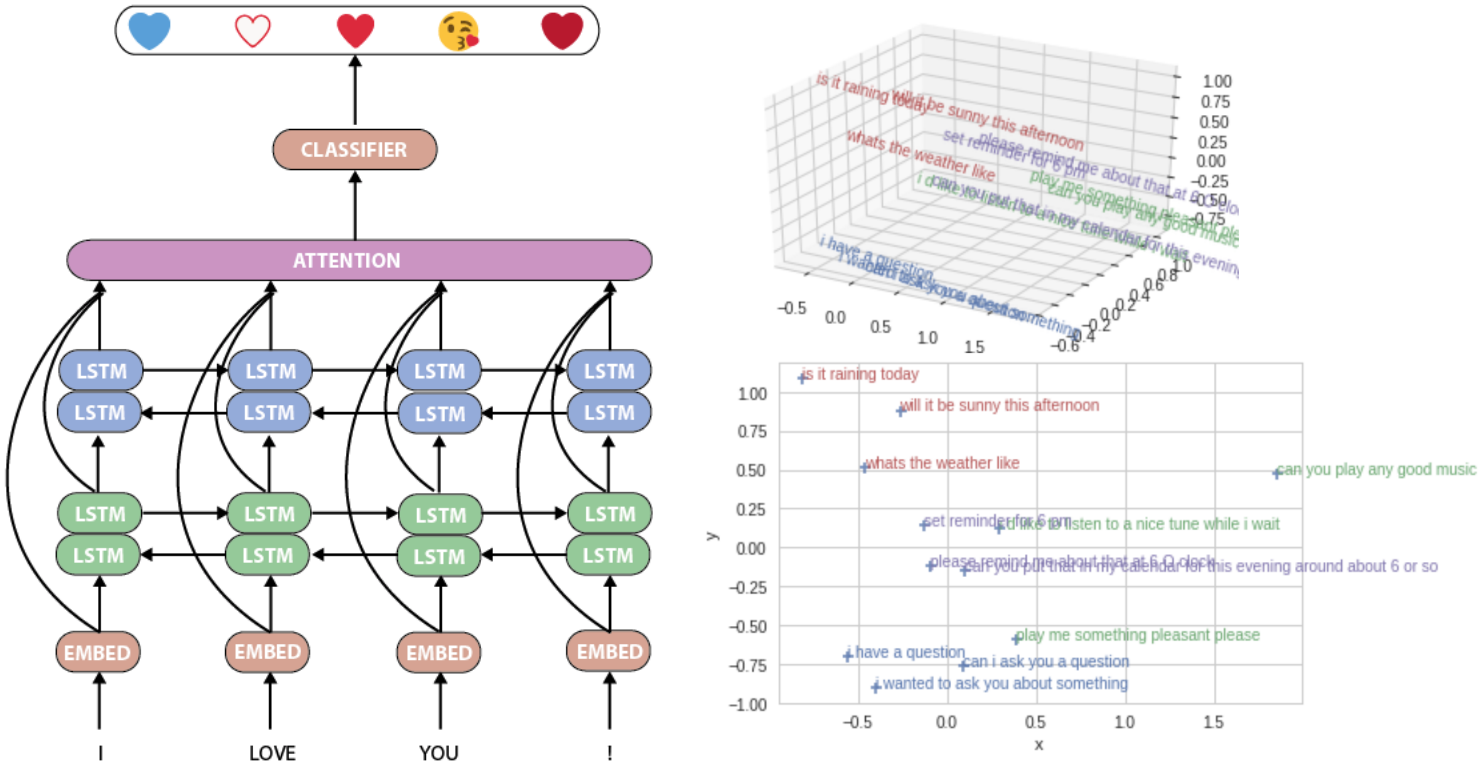
Paragraph embeddings using BERT

```
!pip3 install pytorch_pretrained_bert

from pytorch_pretrained_bert import BertTokenizer
tokeniser = BertTokenizer.from_pretrained('bert-large-uncased',
do_lower_case = True)

from pytorch_pretrained_bert import BertModel
bert = BertModel.from_pretrained('bert-large-uncased')

import torch
tokenised_sentence = ["[CLS]"] + tokeniser.tokenize(sentence) +  ["
[SEP]"]
tokens_tensor =
torch.tensor([tokeniser.convert_tokens_to_ids(tokenised_sentence)])
```

```
segments_tensor = torch.tensor([[0 for _ in
range(len(tokenised_sentence))]])
encoded_layers, _ = bert(tokens_tensor, segments_tensor)
paragraph_vector = encoded_layers[0][-1][0].detach().numpy()
```

DeepMoji

Or why not use some more unusual, neural models, like **DeepMoji** (which has learnt to identify various sentiment and emotions in a sentence as well as the semantics too)

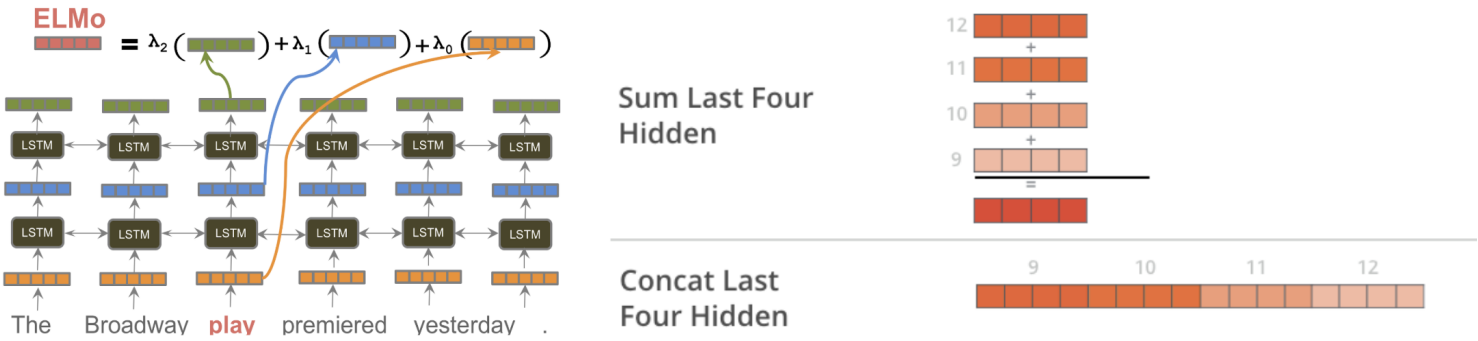


Paragraph embeddings using DeepMoji

```
!git clone https://github.com/huggingface/torchMoji
import os
os.chdir('torchMoji')
!pip3 install -e .
!python3 scripts/download_weights.py

import torchmoji, json
from torchmoji.model_def import torchmoji_feature_encoding
from torchmoji.sentence_tokenizer import SentenceTokenizer
from torchmoji.global_variables import PRETRAINED_PATH, VOCAB_PATH

deepmoji_model = torchmoji_feature_encoding(PRETRAINED_PATH)
with open(VOCAB_PATH, 'r') as f:
```

```
    vocabulary = json.load(f)
st = SentenceTokenizer(vocabulary, 30)

tokenized, _, _ = st.tokenize_sentences([sentence])
paragraph_vector = deepmoji_model(tokenized)
```

Or why not try Open AI's **GPT-2** Language Model, or Facebook AI's InferSent models.

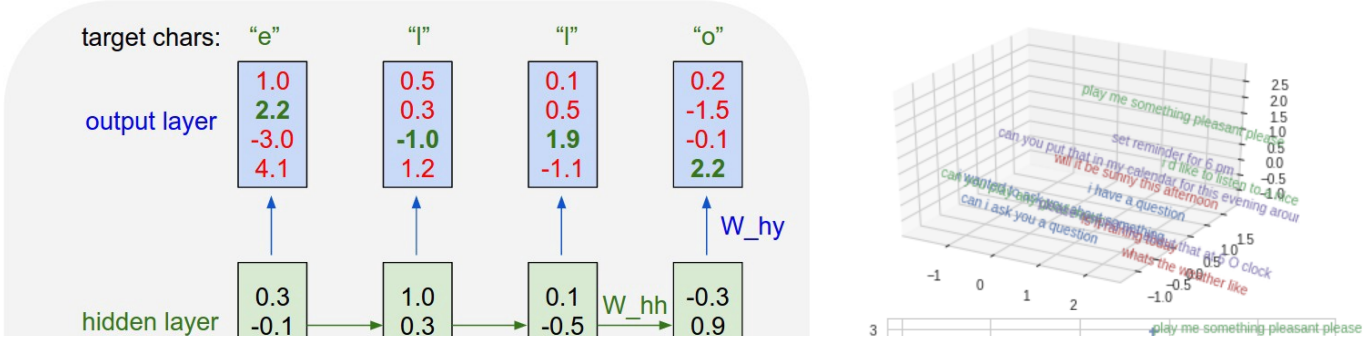


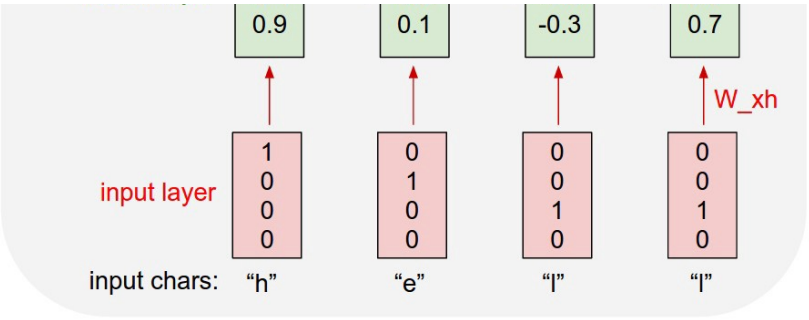
(left) ELMo takes a concatenation of each hidden layer (for contextual embeddings) and the word vectors. (right) The best F1 score for NER using BERT was to concatenate its final four hidden layers, while the second best was to sum its last four hidden layers.

There is also the question of which layer to take as the embeddings when a model has multiple layers? The last layer? Second to last layer? The average of all layers? All layers summed together? All layers concatenated together? (Each of the above are valid approaches!)

Char2Vec

There are even language models trained at the character-level (instead of the word-level) such as char2vec (although they work surprisingly well and are robust to things like typos and oov words, character level language models like this seem to capture the syntactical structure of the sentence better than they do the underlying semantics)



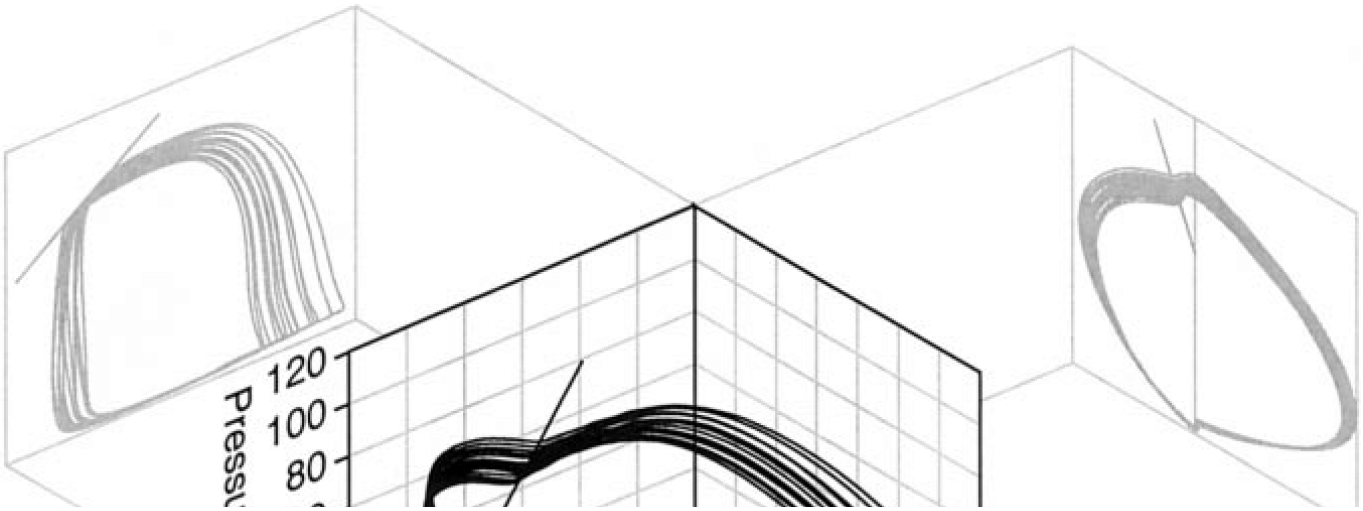


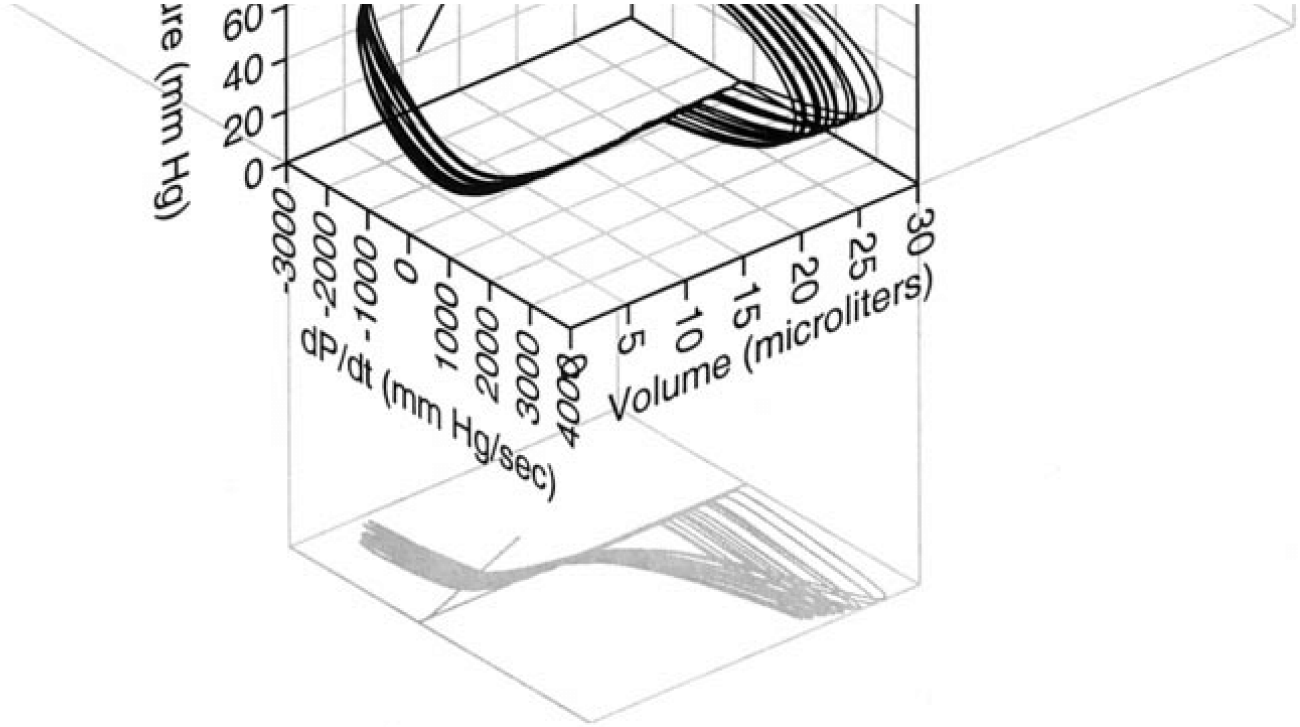
char-level RNN

```
!pip3 install chars2vec
import chars2vec
c2v_model = chars2vec.load_model('eng_150')
c2v_vectors = c2v_model.vectorize_words([sentence])
```

Comparing Vectors

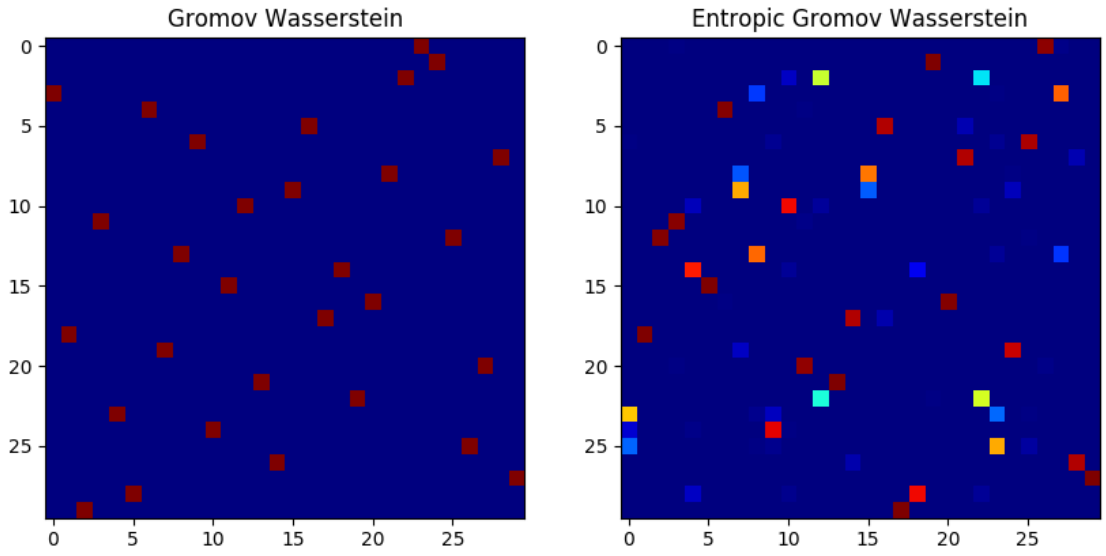
You may have wondered why we even need a fixed-length vector to represent a sentence or paragraph. Why not use the individual word vectors, for instance, and just stack them together (append them). The problem with this is, sentences can have variable lengths (different number of words) and so paragraph vectors composed of stacked word vectors would have different lengths (or different dimensions). This becomes a problem when we try to compare one vector to another, since most of the vector comparison techniques (e.g. cosine distance, euclidean distance, etc) require the vectors to be the same length to make sense (an N-dimensional vector and an M-dimensional vector are actually embedding things into two different spaces and thus cannot be easily compared — it would be like comparing a 2D square with a 3D cube).



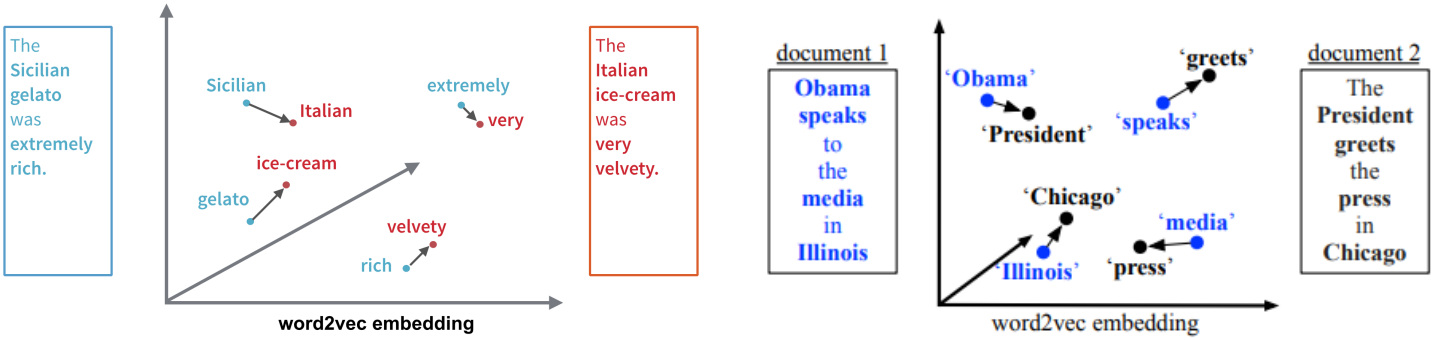


vectors of 2 values embed things into a 2d space. vectors of 3 values embed things into a 3d space. and so on.

There are, however, ways to compare different dimensional spaces (and thus vectors of different lengths), such as the **Gromov-Wasserstein** Distance (which actually compares the relations of items to one another within their respective hyperspace)



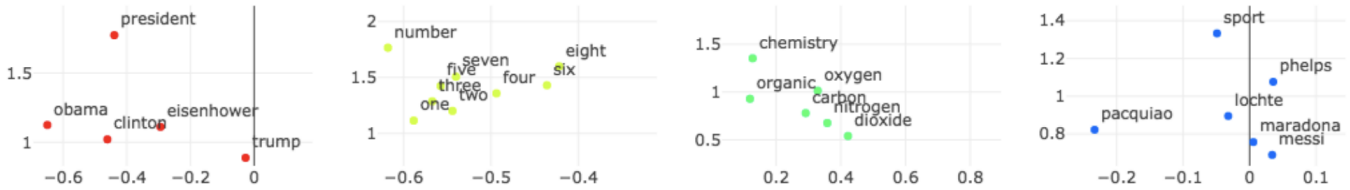
The **Word Mover Distance** (WMD) is a method which compares individual word vectors in a paragraph without requiring the word vectors to be merged at all.



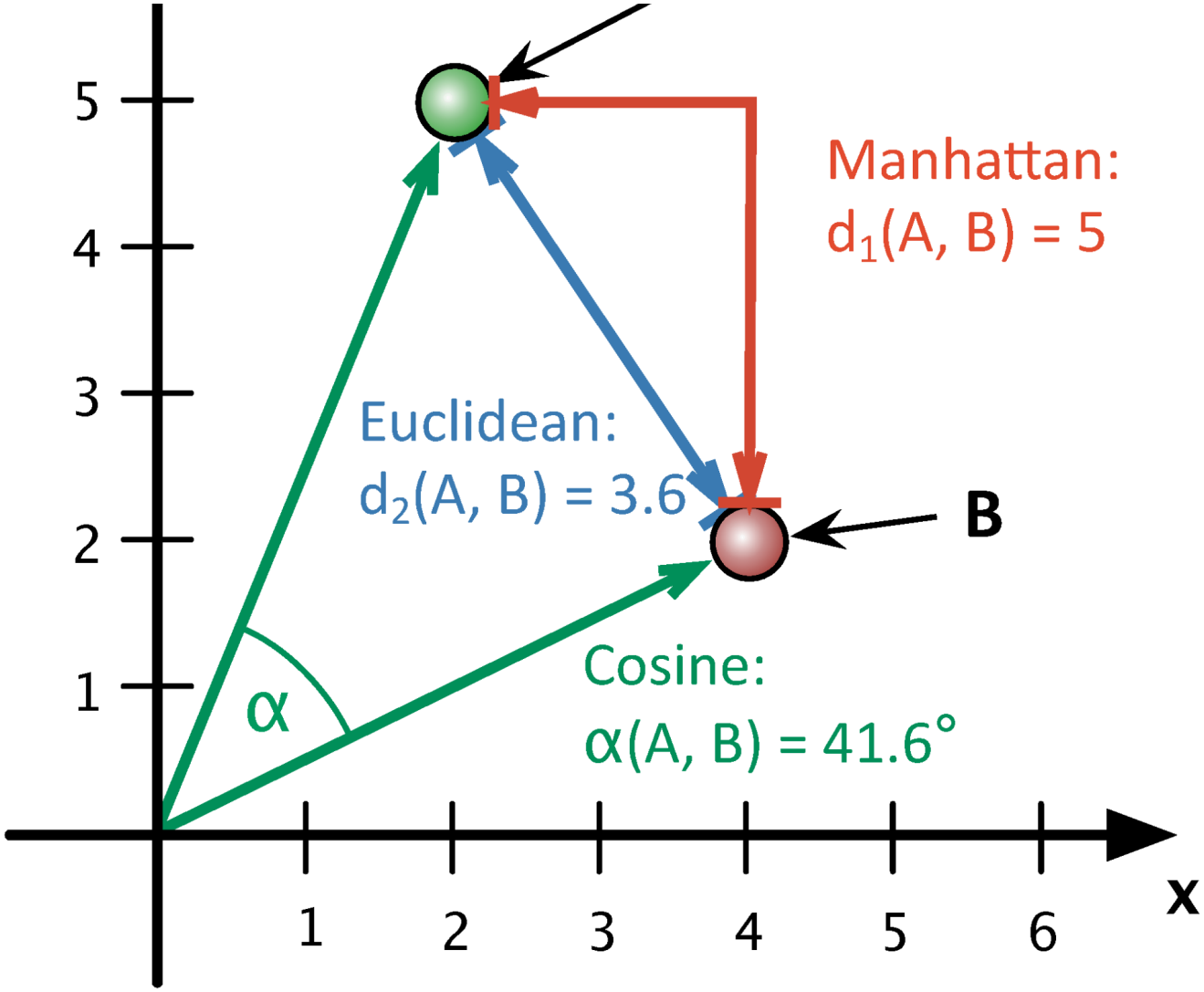
Here are two examples of using Word Mover Distances (WMD) to calculate the similarity of paragraphs using the individual word vectors (as opposed to a paragraph vector)

```
!wget https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-
negative300.bin.gz
from gensim.models import KeyedVectors
w2v = KeyedVectors.load_word2vec_format('GoogleNews-vectors-
negative300.bin.gz', binary=True)

w2v.wmdistance(sentence.lower().split(),
other_sentence.lower().split()
```

Of course, its always easier to compare vectors of the same length (same space)

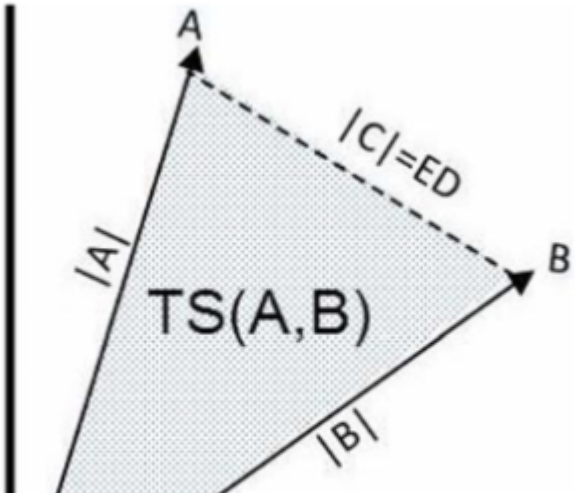


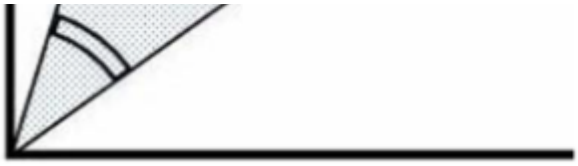
Some common distances to compare vectors include:



Different approaches to compare the similarity of two vectors (Cosine Similarity, Euclidean Distance, manhattan Distance)

There are even some more novel methods to compare two vectors, such as Triangle-Similarity (TS) or even Triangle-Similarity Sector's Area Similarity (TS-SS)





Triangle Similarity (TS)

Machine Learning Word Embeddings Neural Networks NLP Naturallanguageprocessing

About Help Legal