

# Natural Language Processing

## Module 1.2: Word Representations

# Learning Objectives

1. How do we represent language to an algorithm?
2. Word embeddings intuition
3. Learning word embeddings
4. What can word embeddings do?
5. Sub-unit tokenization

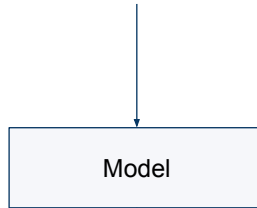
# Contents

1. Motivation
2. One-hot encoding & drawbacks
3. Intuition of vectors
4. Obtaining word embeddings
  - a. Skipgram
  - b. Negative sampling
5. What can we do with embeddings
6. Tokenization & BPE

**How do we represent language to an  
algorithm?**

# The Problem

`[“this movie had a brilliant storyline with great action”,  
“some parts were not so great, but overall pretty ok”,  
“even my dog went to sleep watching this 🤡 trash”]`



`[5, 3, 1]`

5

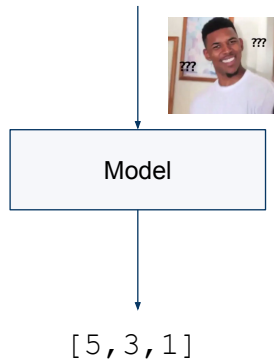
[SPEAK SLOWLY]

So, here's our starting point and motivation.

We have a corpus of documents (here, 3 documents). And we want to classify each document/review out of 5.

# The Problem

`[“this movie had a brilliant storyline with great action”,  
“some parts were not so great, but overall pretty ok”,  
“even my dog went to sleep watching this 🤡 trash”]`



6

[SPEAK SLOWLY]

Over the course of this module, we'll look at the different types of models we can use to solve this problem.

However, right now we're concerned with: "how can we represent our documents to a model"?

# How?

7

[SPEAK SLOWLY]

And that's a question for all of you. Spend a minute or 2 talking about what you already know about ML to discuss how you might represent these documents to our model. And what issues do you think these forms of representation might have?

By the way, the point of this exercise is so we can *derive*, both in intuition and necessity, vectors. So don't answer 'vectors' and be done with it.

# One hot encoding

8

[SPEAK SLOWLY]

So, outside of vectors, there are a couple of different ways we could ‘technically’ represent the document to a model.

However, pretty much all of them will essentially boil down to one-hot encoding at one point (this is also true for vectors, as we’ll see shortly)

Let’s take a look at the simple case of encoding a corpus using one-hot encoding. We’ll then dive further into the drawbacks of representing language to an algo in this way.



# One hot encoding - Corpus

**Corpus:** A collection of documents. I.e. Our entire dataset

```
[“this movie had a brilliant storyline with great action”,  
“some parts were not so great, but overall pretty ok”,  
“even my dog went to sleep watching this 🤡 trash”]
```

9

[SPEAK SLOWLY]

Let's start with some definitions.

First up, we have corpus. A corpus is simply the linguist's way of referring to a collection of text data. It is essentially the language dataset we are working with (and in modern NLP the terms are basically interchangeable).

In our motivating example, our corpus contains the different reviews. If we were working with a different type of task, our corpus might just be, for example, the whole of wikipedia

# One hot encoding - Document

**Document:** One item of our corpus

```
["this movie had a brilliant storyline with great action"]
```

Documents are typically **sequences**

10

[SPEAK SLOWLY]

Next up is document.

A document simply refers to one item of our corpus. In NLP, the documents will typically be a **sequence**. A sequence is just a string of words (e.g. a phrase, sentence, paragraph etc)

# One hot encoding - Token

**Token:** The atomic unit of a sequence (e.g. a word)

```
this  
movie  
had  
a  
...
```

Nowadays tokens are typically sub-words. We'll look at this later. But just consider a token as a word for now

11

[SPEAK SLOWLY]

Then, token.

A token is the atomic unit of a sequence. Pre-2018 (and what's shown on here), tokens were mostly considered as words or punctuation. However, since the introduction of BPE (which we'll look at later, tokens now refer to sub-word units)

# One hot encoding - Vocabulary

**Vocabulary:** The unique tokens across our entire corpus

aardvark

aarhus

...

this

...

zyzzyva

```
vocabulary = []  
for unique_word in corpus:  
    vocabulary.append(unique_word)
```

12

[SPEAK SLOWLY]

Finally, vocabulary.

The vocabulary are the unique tokens across our entire corpus. (Note that here, for examples sake, I've added tokens that weren't in our corpus)

# One hot encoding - Recap

**Corpus:** A collection of documents. I.e. Our entire dataset

**Document:** One item of our corpus (e.g. a sequence)

**Token:** The atomic unit of a sequence (e.g. a word)

**Vocabulary:** The unique tokens across our entire corpus

[SPEAK SLOWLY]

Let's recap.

# Building a one-hot representation

## Vocabulary

i	Token
1	aardvark
2	aarhus
...	...
7431	this
...	...
10000	zyzzyva

14

[SPEAK SLOWLY]

Ok, we'll start with our vocabulary, which we've enumerated with indexes.

Note that in practise, the vocabulary does not need to be sorted, but it's a nice QoL for teaching

# Building a one-hot representation


## Vocabulary

i	Token
1	aardvark
2	aarhus
...	...
7431	this
...	...
10000	zyzzyva

## Document

this movie had a brilliant story line with great action  
1 2 3 4 5 6 7 8 9 10

seq\_len



[SPEAK SLOWLY]

We then take one of the document/sequences we have. We will enumerate each token in this sequence, and call the length of this sequence `sequence_length`

# Building a one-hot representation

## Vocabulary

i	Token
1	aardvark
2	aarhus
...	...
7431	this
...	...
10000	zyzzyva

## Document

this movie had a brilliant story line with great action  
1 2 3 4 5 6 7 8 9 10

## One-hot

	<i>aardvark</i>	<i>arrhus</i>	...	<i>movie</i>	...	<i>this</i>	...	<i>zyzzyva</i>
this	0	0	...	0	...	1	...	0
movie	0	0	...	1	...	0	...	0
...	...	...	...	...	...	...	...	...

16

[SPEAK SLOWLY]

Now, we will build our one-hot matrix.

To do this, we are going to find the vocabulary index for each token in our document. Then we build a one-hot representation using these indices.

Intuitively, that looks like this: [Run through the one-hot table]


As we see, for each token in our document, we have a “1” at the location where that token appears in the vocabulary. So for the word “this” we have a 1 at index 7431, and 0 everywhere else.



# Building a one-hot representation

## One-hot

	<i>aardvark</i>	<i>arrhus</i>	...	<i>movie</i>	...	<i>this</i>	...	<i>zyzzyva</i>
this	0	0	...	0	...	1	...	0
movie	0	0	...	1	...	0	...	0
...	...	...	...	...	...	...	...	...



```
[  
  [0, 0, ..., 0, ..., 1, ..., 0],  
  [0, 0, ..., 1, ..., 0, ..., 0]  
  ...  
]
```

17

[SPEAK SLOWLY]

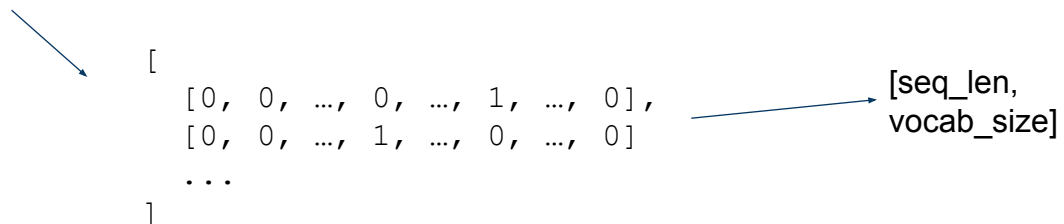
So here I've shown the one-hot matrix representation.

**What is the shape of this matrix?**

# Building a one-hot representation

## One-hot

	<i>aardvark</i>	<i>arrhus</i>	...	<i>movie</i>	...	<i>this</i>	...	<i>zyzzyva</i>
this	0	0	...	0	...	1	...	0
movie	0	0	...	1	...	0	...	0
...	...	...	...	...	...	...	...	...



```
[  
  [0, 0, ..., 0, ..., 1, ..., 0],  
  [0, 0, ..., 1, ..., 0, ..., 0]  
  ...  
]
```

[seq\_len,  
vocab\_size]

18

[SPEAK SLOWLY]

Seq\_len number of rows, and vocab\_size number of columns.

In other words, we have as many rows as tokens in our document

And as many columns as the size of our vocabulary

# Building a one-hot representation

`["this movie had a brilliant storyline with great action"]`

↓

```
[  
  [0, 0, ..., 0, ..., 1, ..., 0],  
  [0, 0, ..., 1, ..., 0, ..., 0]  
  ...  
]
```

↓

Model



19

[SPEAK SLOWLY]

Done, right?

# Building a one-hot representation

`["this movie had a brilliant storyline with great action"]`

↓

```
[  
  [0, 0, ..., 0, ..., 1, ..., 0],  
  [0, 0, ..., 1, ..., 0, ..., 0]  
  ...  
]
```

↓

Model



20

[SPEAK SLOWLY]

Done, right?

# The issues with one hot encoding

21

[SPEAK SLOWLY]

I know I asked you to think through the issues before, but let's try it once more armed with this knowledge.

There are 3 core issues with representing language as one-hot encoding. What do you think they are?

Take a minute to discuss this with someone next to you.

# The issues with one hot encoding

1. Orthogonal definitions
2. High dimensionality (and sparse)
3. Fixed vocabulary

22

[SPEAK SLOWLY]

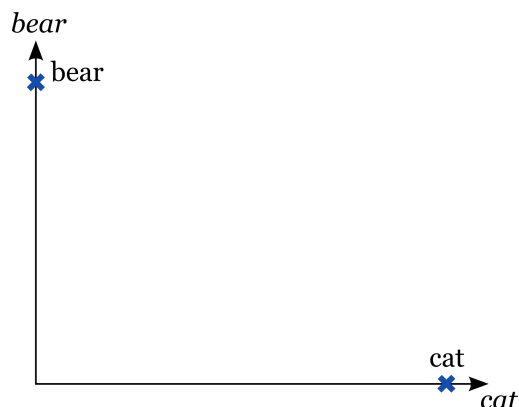
I know I asked you to think through the issues before, but let's try it once more armed with this knowledge.

There are 3 core issues with representing language as one-hot encoding. What do you think they are?

Take a minute to discuss this with someone next to you.

# Orthogonal definitions

Vectors are orthogonal → Every word is equidistant from every other word



23

[SPEAK SLOWLY]

Let's start with orthogonal definitions.

Basically, with one-hot encoding, every token is equidistant from every other token. This is because each representation is orthogonal to each other representation (as the 'hot' token is represented as a discrete 1 or 0).

In other words, one-hot encodings do not capture or leverage the similarity between words. In one-hot encoding, cat and kitten are as similar to each other as cat and airplane

## High dimensionality (and sparse)

The vector dimensionality for each token is the size of our vocabulary!

AND, all but one of the elements are 0's!

24

[SPEAK SLOWLY]

One-hot encoding leads to extremely high-dimensional vectors, especially for large vocabularies common in natural languages. Each token in the vocabulary gets a unique vector, and this vector's size is equal to the size of the vocabulary. This high dimensionality can lead to significant computational inefficiency and increased memory usage, making the model less scalable and harder to train.

On top of that, the vectors are extremely sparse, meaning that most of their elements are zeros. This sparsity is inefficient for computational resources, as it involves operations on large vectors where most of the elements are not contributing any information.



# Fixed vocabulary

What do we do with out of vocabulary (OOV) tokens?

25

[SPEAK SLOWLY]

During inference, what happens when we see a token which was not in our original training vocabulary?

These OOV tokens cannot be represented accurately, and there is no straightforward way to represent these words, which can be a substantial problem. Especially in applications dealing with dynamic and evolving languages, like social media text.

Admittedly, this problem goes beyond one-hot encoding, and this drawback is also suffered by the embedding methodology we'll look at next. Though, thankfully, newer tokenization schemes (which will look at later in this lecture) account for this.

# WordNet

We could map words to broader concepts. For example, using WordNet:

<http://wordnetweb.princeton.edu/perl/webwn>

<https://www.nltk.org/howto/wordnet.html>

“cat”, “kitten” → “feline mammal”

“London”, “Paris”, “Tallinn” → “national capital”

“yellow”, “green” → “colour”

	<i>feline mammal</i>	<i>national capital</i>	<i>colour</i>
<b>Tallinn</b>	0	1	0
<b>London</b>	0	1	0
<b>yellow</b>	0	0	1

26

[SPEAK SLOWLY]

During inference, what happens when we see a token which was not in our original training vocabulary?

These OOV tokens cannot be represented accurately, and there is no straightforward way to represent these words, which can be a substantial problem. Especially in applications dealing with dynamic and evolving languages, like social media text.

Admittedly, this problem goes beyond one-hot encoding, and this drawback is also suffered by the embedding methodology we'll look at next. Though, thankfully, newer tokenization schemes (which will look at later in this lecture) account for this.

# WordNet

## This helps

- Maps some similar words together
- Reduces vector size

## But

- Relies on the completeness of a manually curated database
- Misses out on rare and new meanings
- Vectors are still orthogonal and equally distant from each other
- Disambiguating word meanings is difficult

“mouse” → “rodent mammal”

or

“mouse” → “computer device” ?

27

[SPEAK SLOWLY]

During inference, what happens when we see a token which was not in our original training vocabulary?

These OOV tokens cannot be represented accurately, and there is no straightforward way to represent these words, which can be a substantial problem. Especially in applications dealing with dynamic and evolving languages, like social media text.

Admittedly, this problem goes beyond one-hot encoding, and this drawback is also suffered by the embedding methodology we'll look at next. Though, thankfully, newer tokenization schemes (which will look at later in this lecture) account for this.

# Word Embedding Intuition

# Distributional hypothesis

Words which are similar in meaning occur in similar contexts.

(Harris, 1954)

You shall know a word by the company it keeps

(Firth, 1957)

He is reading a **magazine**

I was reading a **newspaper**

This **magazine** published my story

The **newspaper** published an article

She buys a **magazine** every month

He buys this **newspaper** every day

29

[SPEAK SLOWLY]

Most of modern NLP is founded on this concept of the distributional hypothesis that was introduced by Harris, and later J.R Firth.

The hypothesis essentially states that the meaning of word can be understood by the context the word appears in.

For example, magazine and newspaper typically appear in similar contexts. And therefore, they are similar in concept.

## Motivating Example

Animal	Cuteness	Size
Lion	80	50
Elephant	85	95
Hyena	10	30
Mouse	60	8
Pig	40	30
Horse	50	65
Dolphin	90	85
Wasp	2	1
Baby goat	90	15

30

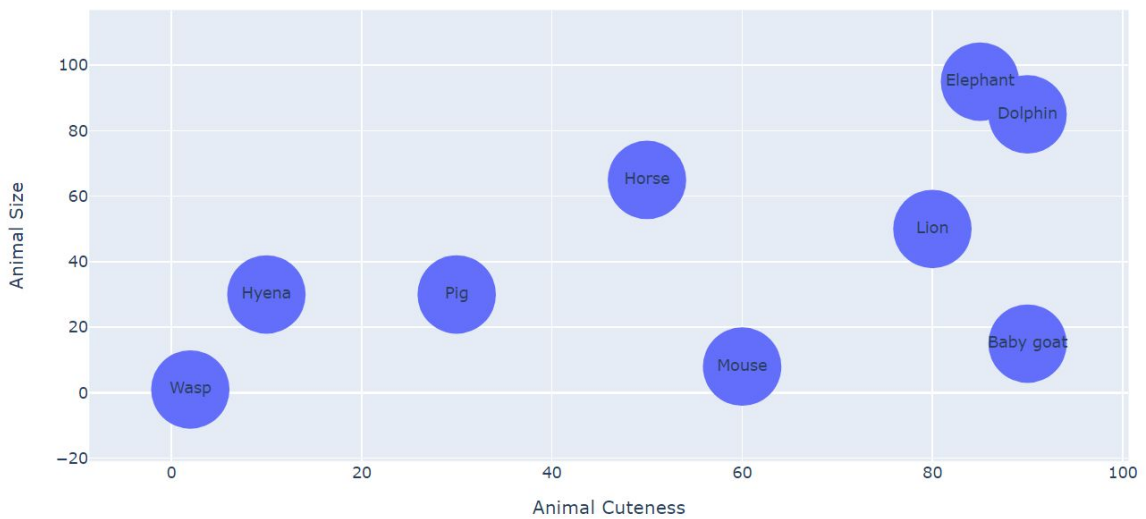
[SPEAK SLOWLY]

Now, before we look at how to obtain distributed representations (word vectors) or words themselves, let's break down the intuition of why vector representations are inherently useful.

We're going to be working with a motivating example with some animals. We'll start with our 'definition' of a each animal being 2d.

Which animals are most similar to each other?

Animal Cuteness vs Animal Size



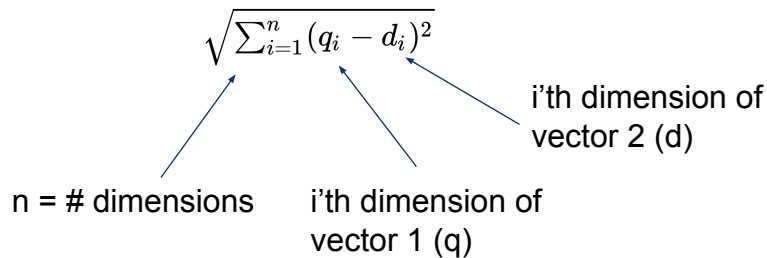
[SPEAK SLOWLY]

When represented visually, we can see that Elephant and Dolphin are very similar to each other.

We might also start to see how we can quantify what 'distance' is.

Who remembers how we can calculate the distance between 2 points?

# Euclidean Distance

$$\sqrt{\sum_{i=1}^n (q_i - d_i)^2}$$


The diagram illustrates the components of the Euclidean distance formula. Three arrows point from descriptive text to parts of the formula: one from 'n = # dimensions' to the upper limit of the summation, one from 'i'th dimension of vector 1 (q)' to the  $q_i$  term, and one from 'i'th dimension of vector 2 (d)' to the  $d_i$  term.

32

[SPEAK SLOWLY]

So you've probably come across euclidean distance before (in 2d it is basically Pythagorous), but worth quickly breaking it down anyway.

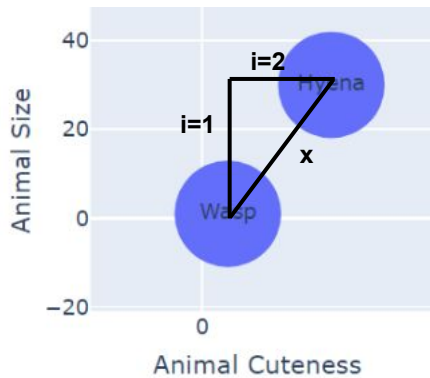
Basically, we're finding the distance between 2 vectors (represented here as q and d).

If we ignore the square root and square term for a second, this distance is basically just the sum of the differences between the each vector dimension.

The square and square root terms then ensure the distance is a positive value and at scale with the original units. (There are also some mathematical properties of the euclidean distance that make using it nicer than taking the absolute value of the differences of the vector dimensions instead of square and square-rooting, but that's not particularly relevant for this lecture).



# Euclidean Distance



Animal	Cuteness	Size
Wasp	2	1
Hyena	10	30

$$\begin{aligned}i=1: 10 - 2 &= 8 \\i=2: 30 - 1 &= 29 \\x &= \sqrt{8^2 + 29^2} = 30\end{aligned}$$

$$\sqrt{\sum_{i=1}^n (q_i - d_i)^2}$$

33

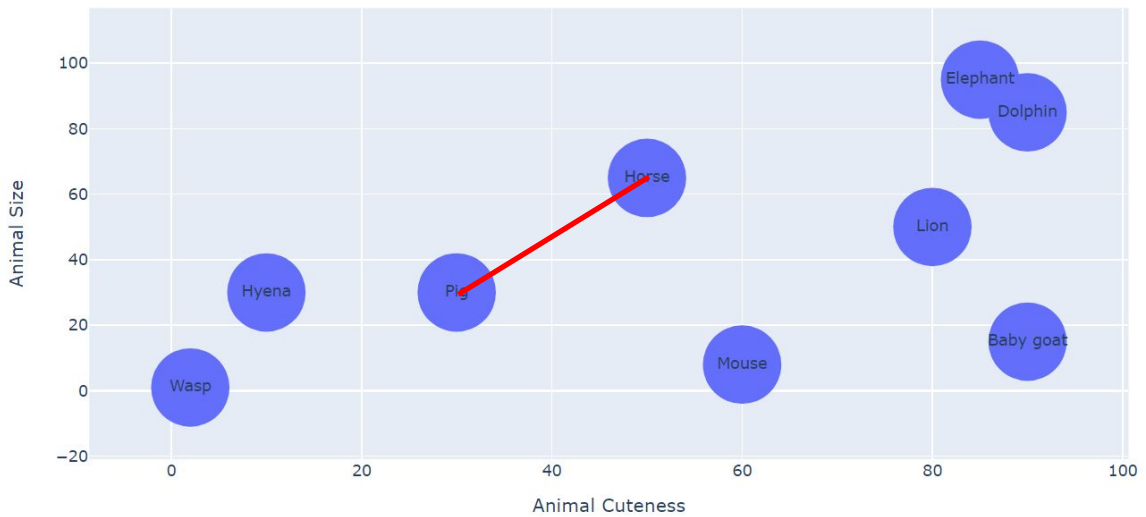
[SPEAK SLOWLY]

Now, before we look at how to obtain distributed representations (word vectors) or words themselves, let's break down the intuition of why vector representations are inherently useful.

We're going to be working with a motivating example with some animals. We'll start with our 'definition' of a each animal being 2d.

Which animals are most similar to each other?

Animal Cuteness vs Animal Size

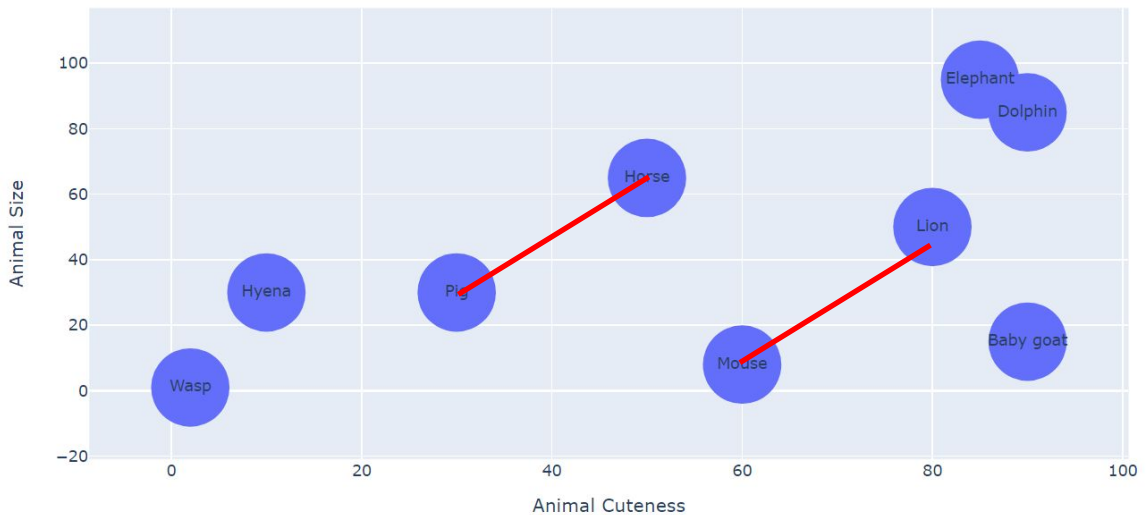


[SPEAK SLOWLY]

Armed with this notion of “maths on vectors”, we can start to do some cool things. For example, we can do analogies.

Pig is to horse as mouse is to ?

Animal Cuteness vs Animal Size



[SPEAK SLOWLY]

Lion.

So, how do we do this? We're using the absolute differences of each vector index (i.e. size and cuteness), and then translating those over to "mouse")

Mathematically, we do this by working out 2 distances:

1. The (absolute) cuteness difference between pig and horse
2. The (absolute) size difference between pig and horse

Then, we take our mouse vector, and add the respective differences for each vector index. That is, mouse cuteness + cuteness difference, and mouse size + size difference

Finally, we loop over all of our vectors and find the smallest euclidean distance to the point where we end up (in this case, Lion)

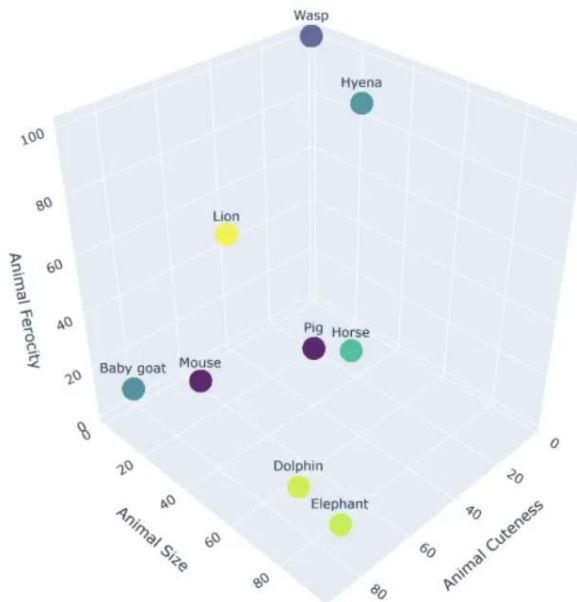
## It works in higher dimensions!

Animal	Cuteness	Size	Ferocity
Lion	80	50	85
Elephant	85	95	20
Hyena	10	30	90
Mouse	60	8	1
Pig	40	30	10
Horse	50	65	40
Dolphin	90	85	30
Wasp	2	1	100
Baby goat	90	15	20

36

[SPEAK SLOWLY]

The cool thing is, is that these concepts generalise to higher dimensions too. Let's take a look at our animals in 3d.



Distance between...

Dolphin and Elephant:	15
Dolphin and Lion:	65.4
Dolphin and Pig:	77

[SPEAK SLOWLY]

Visualising a 3d graph when you don't have a mouse to play around with is a little bit challenging... so bear with me here.

There are 2 key things to take away here:

- the same way that we plotted our animals in 2d, we can do it in 3d.
- Our distance formula extends to dimensions higher above 2! So based on our new definition of an animal (i.e. adding ferocity), we can quantify which animals are most similar to each other

# Cosine Distance

similarity = 1 - distance

**Measures the angle between 2 points**

38

[SPEAK SLOWLY]

So, last thing to do before we start looking at word vectors. Cosine distance (or similarity).

Cosine distance is another distance metric that is commonly used when comparing vectors together. Typically it works better for higher dimensional space, but in practise it is worth validating which distance metric works better for application.

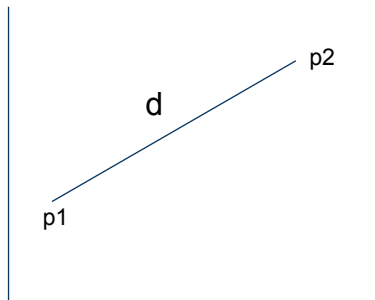
Let's break down how it works.

So, whereas Euclidean distance measures the magnitude between 2 points, Cosine distance measures the angle between 2 points as a way of capturing similarity/distance

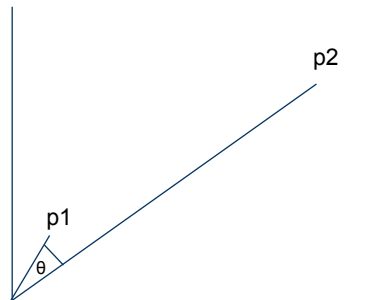
# Cosine Distance

similarity = 1 - distance

**Measures the angle between 2 points**



**Euclidean Distance**



**Cosine Distance**

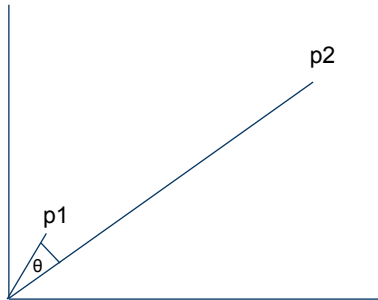
[SPEAK SLOWLY]

Visually, this looks like the right hand diagram – we aim to calculate the angle between the 2 points

# Cosine Distance

similarity = 1 - distance

**Measures the angle between 2 points**



**Cosine Distance**

similarity =  $\cos(\theta)$  =

$$\frac{p_1 \cdot p_2}{||p_1|| \times ||p_2||}$$

If vectors are already normalized... then denominator is 1.  
Therefore cosine similarity = dot product

40

[SPEAK SLOWLY]

The cool thing about this function is that the cosine distance is the dot product when the vectors are normalized!



## Recap

- Continuous representations allow us to group similar things together
- We can apply functions over vectors
- We looked at 2 kinds of distances: Euclidean and Cosine

[SPEAK SLOWLY]

# Words as Vectors

# Word Vectors

Numerical definition of a word

We don't know what each dimension means

King	0.50451	0.68607	-0.59517	-0.0228	0.60046	-0.13498	-0.08813	0.47377	...
Queen	0.37854	1.8233	-1.2648	-0.1043	0.35829	0.60029	-0.17538	0.83767	...

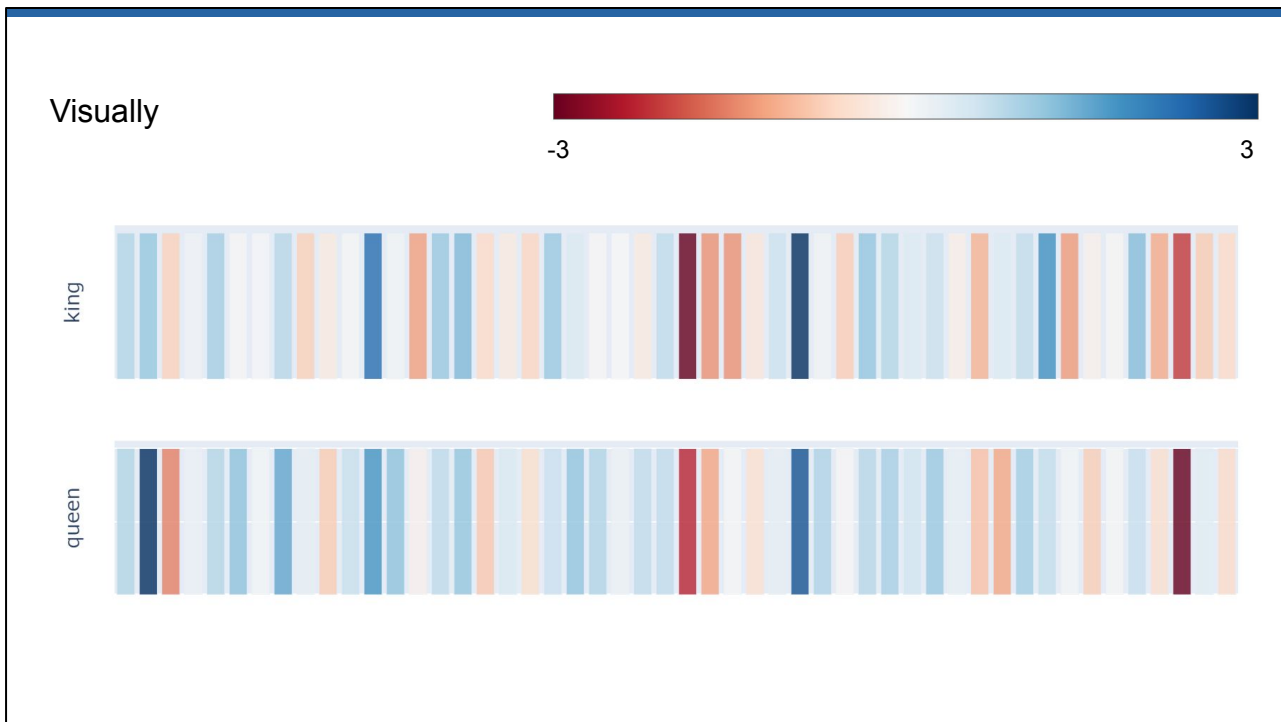
[SPEAK SLOWLY]

Ok, so similar to how our animal representation was a numerical definition for each animal, our word vector representations are a numerical definition of a word.

However, the features are something that our ML algo itself will learn. Therefore, we don't actually know what each dimension means!

Here you're directly at the word vector for king and queen.

We can try and apply some intuition to it (which I'll cover on the next slide), but before diving into how we calculate this, I wanted to show you how AI algorithms think of the definition of a word



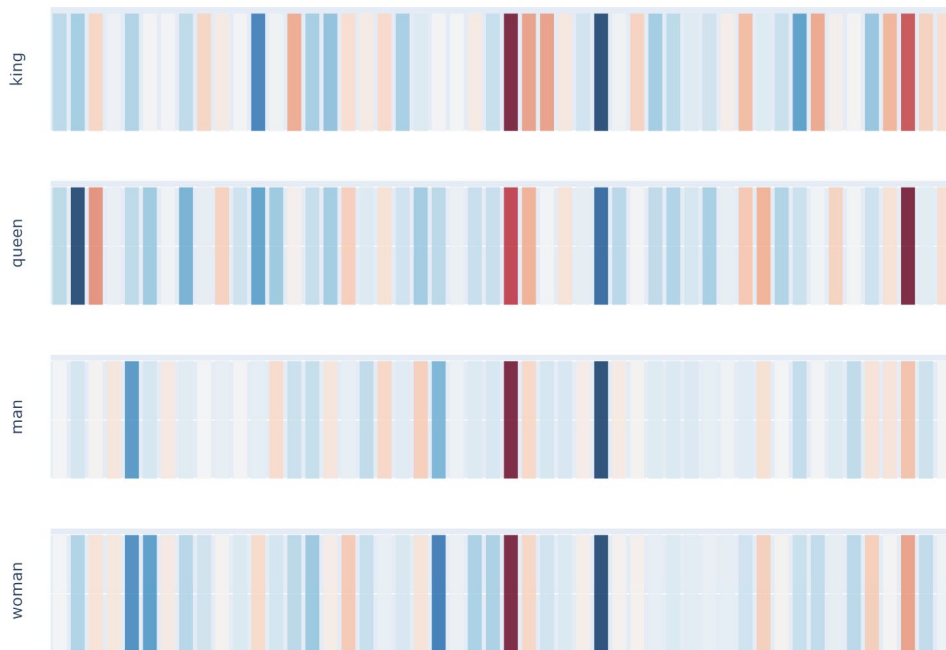
[SPEAK SLOWLY]

Ok, so what I've done here is taken the vectors for King and Queen and plotted the intensity of each one of its dimensions. Note that here I've only shown 50 dimensions (i.e. there are 50 columns). A few years ago this level was the norm... However, over recent years the number of dimensions we've used to represent words have increased. And holy shit LLMs have gotten HUGE. For example, GPT-3 represents its words as almost 13,000 dimensions.

Let's interpret the diagram as follows: if a dimension is close to -3, then it's colour will be red. If a dimension is close to 3, then it's colour will be blue.

That means the 2nd dimension on Queen has a value close to 2, and this dimension (~30th) on King has a value close to -2.

Ok, now that you know what you're looking at, let's look at the intuition

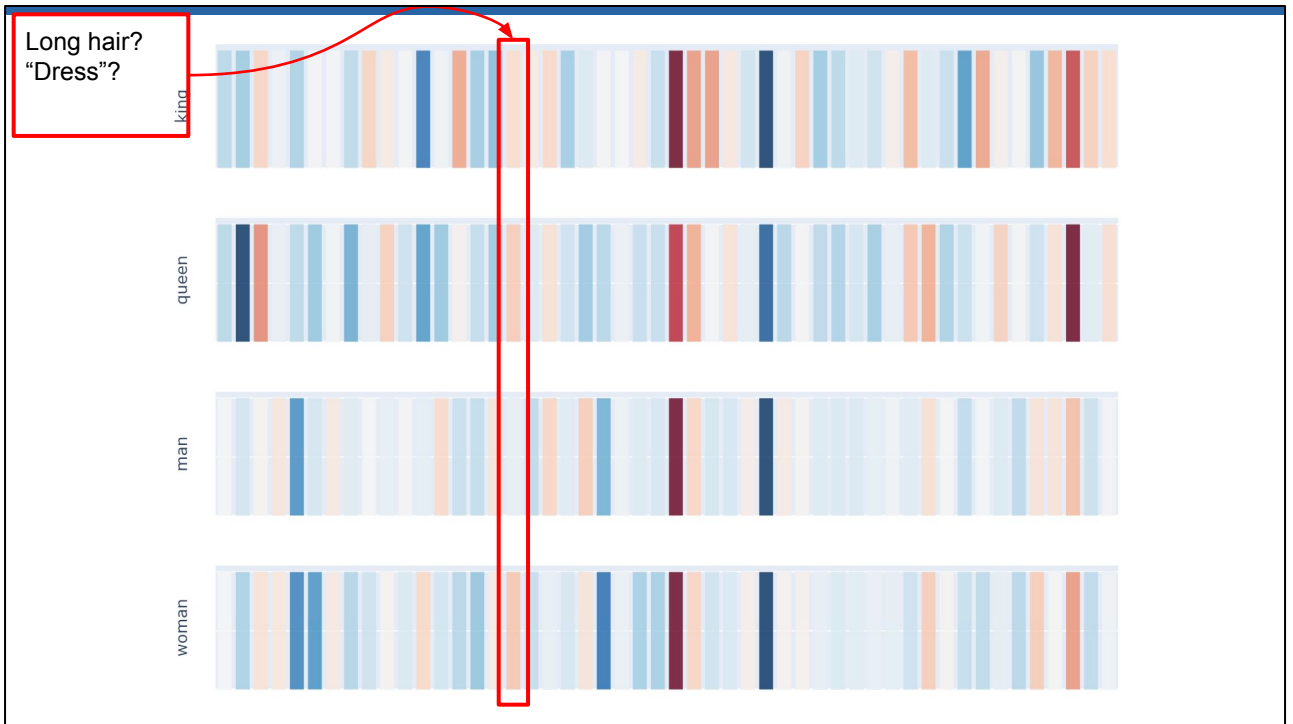


[SPEAK SLOWLY]

Ok. Here I've shown the vectors for King, Queen, Man and Woman.

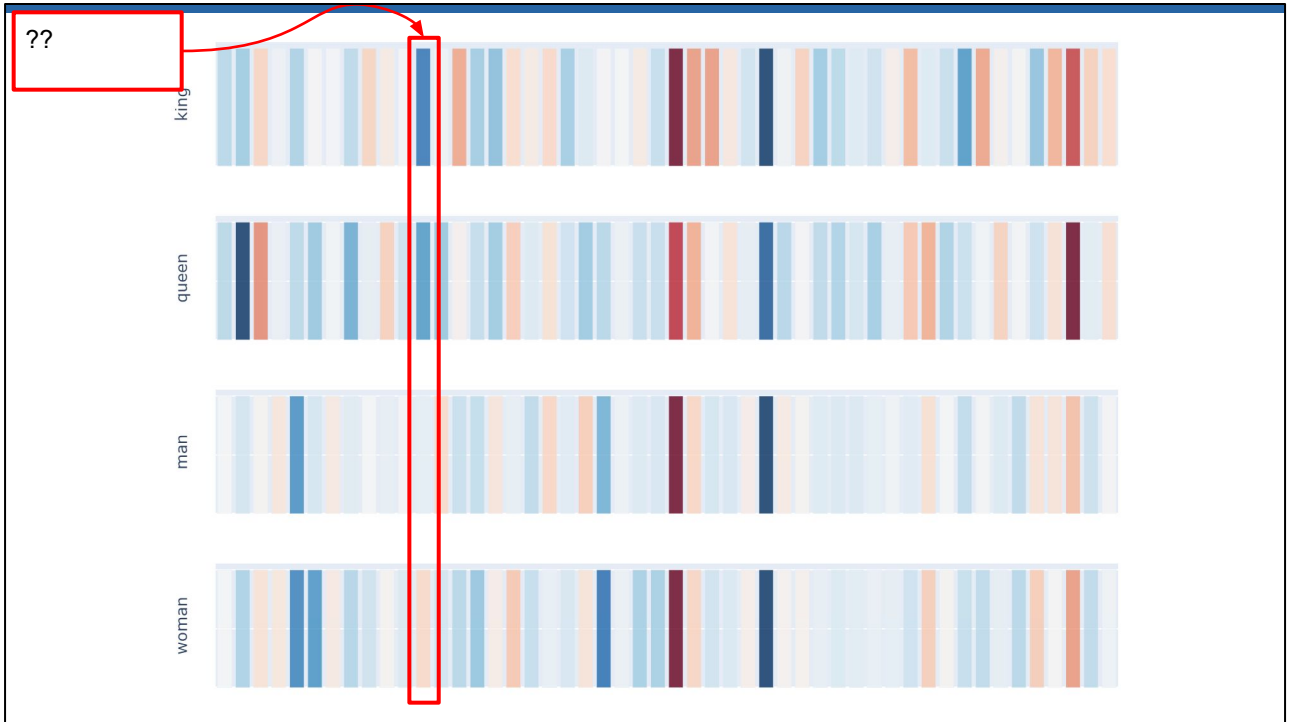
As I mentioned on a previous slide, we do not know what each dimension means.

However, once we have these vectors, we can try to make a best guess for each of these dimensions. Note that this exercise is actually mostly useless in practise, but it is great for teaching the theory of why word vectors might work. Let's try do this here.



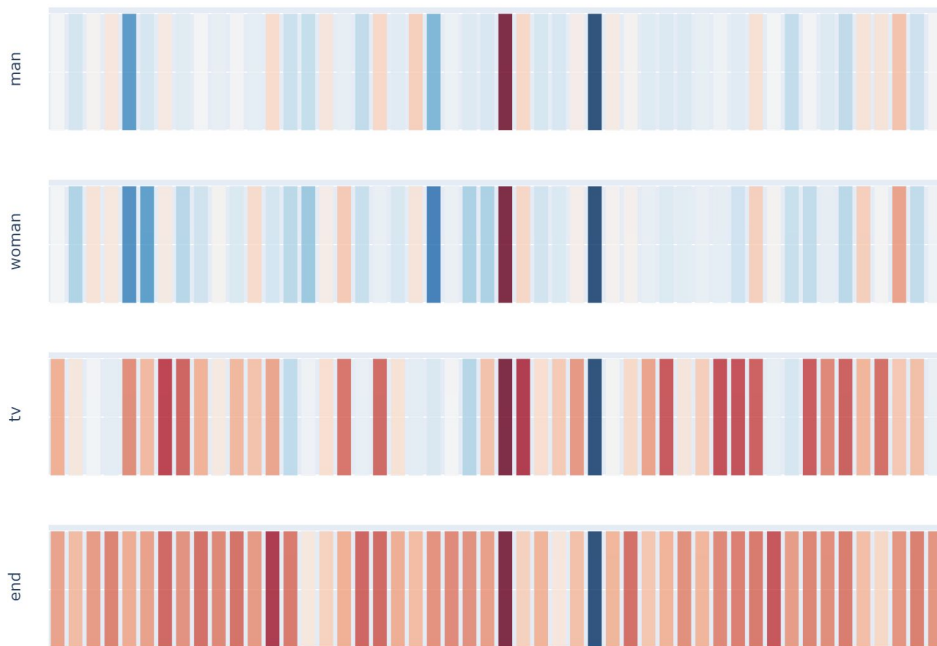
[SPEAK SLOWLY]

In this example, we see these dimensions/numbers are more intense for Queen and Woman. This means that this dimension could have some meaning about the length of their hair or the context of dressing.



[SPEAK SLOWLY]

In this example, we see values more intense for King and Queen. What are your suggestions on the attribute of this dimension?



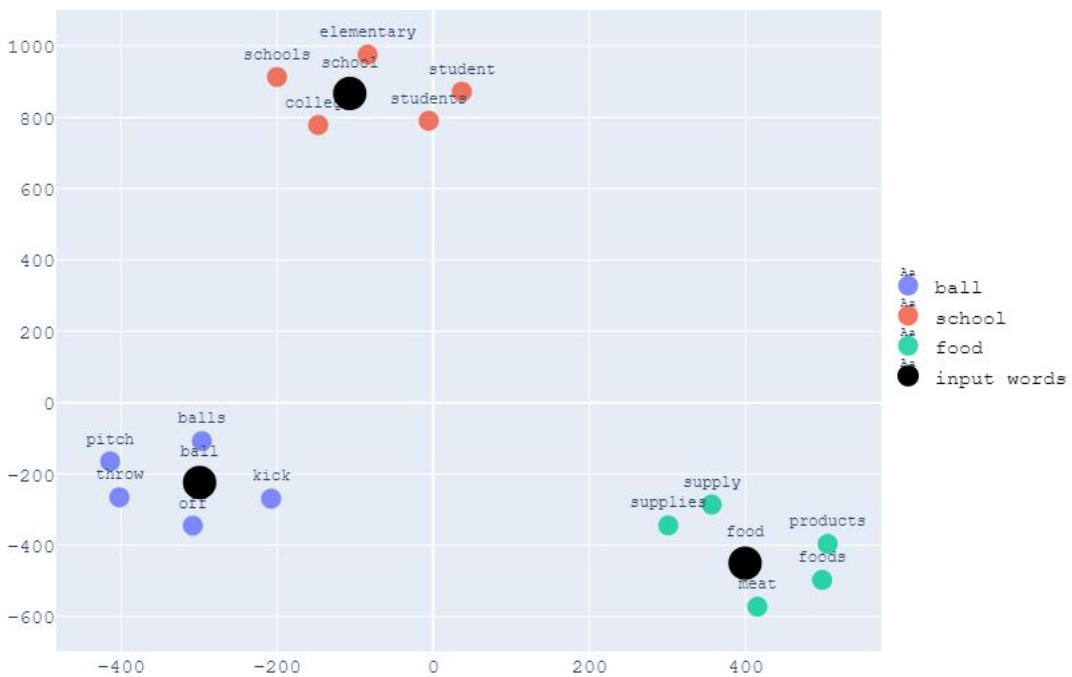
[SPEAK SLOWLY]

Here, I'm just showing words which are largely unrelated to each other.

The key takeaways from here are:

- Intuitively, we know that Man and Woman are similar to each other (i.e. contexts we use the word “man” in, we can use the word “woman” in). And here we can see that the word vectors for man and woman are very similar.
- Man, Woman and TV are all exclusively nouns. By that virtue, they are more similar to each other than “end”. Thus, we can see there being a higher discrepancy between these words.





[SPEAK SLOWLY]

Ok – so let's bring it back to reality now.

The KEY takeaway from everything I've shown you above is that **similar words have similar vectors**

Here I've used some dimensionality reduction to visualise these 50d vectors on a graph. What you can see is that words which are similar to each other are close together. When the words are not similar to each other (e.g. pitch and foods), we can see they are far away on the graph.

Ok, let's look at how to get these vectors now.

# Obtaining Word Embeddings

# Word Embeddings

Fixed dimensional dense representation learned by our neural network algorithm.

[SPEAK SLOWLY]

Ok, so as you've seen in the previous few slides, a word embedding is a fixed dimensional dense representation learned by our neural network algorithm.

In other words, we're going to allocate a number of parameters for each word and allow a neural network to automatically learn what the useful values should be.

These are often referred to as "word embeddings", as we are embedding the words into a real-valued low-dimensional space.

The resulting vectors are orders of magnitude smaller than the vocabulary (typically, ~300-1K dimensions) and dense (non-zero).

# Word2Vec

Not the only way of obtaining word embeddings

Word embeddings enable transfer learning

Word2Vec is a “family” of NN based algorithms to obtain embeddings:

- Architectures: CBOW and **Skip-gram**
- Training strategy: Hierarchical Softmax and **Negative Sampling**

[SPEAK SLOWLY]

Here, we'll be looking at the seminal work of word2vec, which was formulated by Thomas Mikolov in 2013. Note that it is not the only method for obtaining word embeddings. There is GloVe, FastText, and more recently, pre-trained language models.

We'll be looking at pre-trained models later in the module.

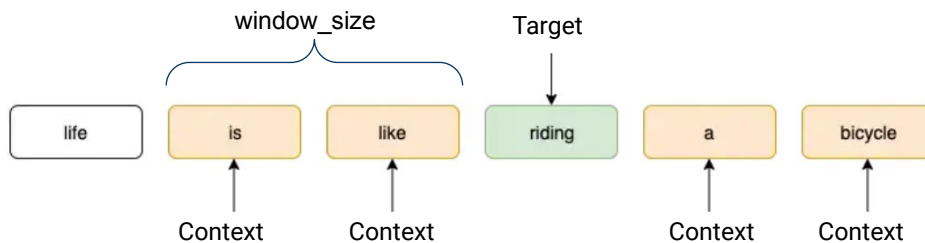
The reason Word2Vec is focused on is because it provides a simple intuition behind how we can use neural networks to reduce dimensionality and then use the lower-dimensional vectors in downstream tasks. Using parameters from one model in another is also part of a paradigm known as transfer learning.

Word2Vec refers to a family of neural network based algorithms which obtain these word vectors/embeddings. One part of this family consists of two different model architectures: Continuous bag-of-words and Skip-gram. The other part of this family consists of two different approaches of dealing with a vocabulary in the order of millions: Hierarchical Softmax and Negative Sampling.

We'll be focusing our efforts on Skip-gram, and also briefly discussing negative sampling later in this lecture.

However, let's start with a quick comparison between CBOW and Skip-gram

# Representing a sequence



<https://medium.com/analytics-vidhya/word2vec-skipgram-explained-f7b3af90f02e>

53

[SPEAK SLOWLY]

One thing that'll be useful before we look at these architectures is understanding how we "represent" our sequence.

We have a notion of a 'target' word and 'context' words.

A context word is a word that is "window\_size" away from a target word. In this example, window\_size is 2. That means the context words are the 2 words preceding the target word, and the 2 words proceeding the target word.

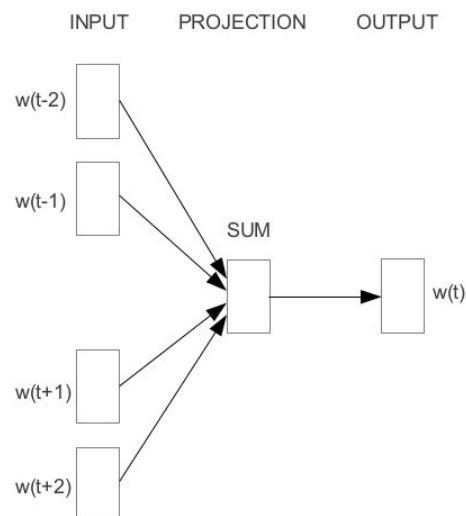
If "like" was our target word, then "life, is, riding, a" would be our context words.

# Continuous Bag-of-Words (CBOW)

Predict the target word  $w_t$  based on the surrounding context words

$w_{t-j} \dots w_{t-2} w_{t-1}$  ?  $w_{t+1} w_{t+2} \dots w_{t+j}$

Mikolov et. al. 2013. Efficient Estimation of Word Representations in Vector Space.



[SPEAK SLOWLY]

So, firstly we have CBOW. This is the simpler variant.

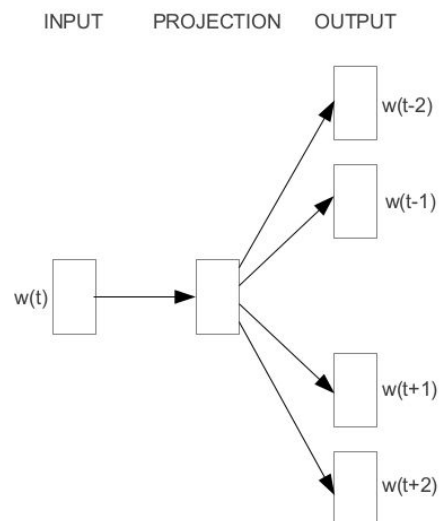
In CBOW, we feed our context words as input to our model. Our model is then optimized to predict what the target word actually is

# Skip-gram

Predict the **context words** based on the  
**target word**  $w_t$

? ? ?  $w_t$  ? ? ?

Mikolov et. al. 2013. Efficient Estimation of Word Representations in Vector Space.



55

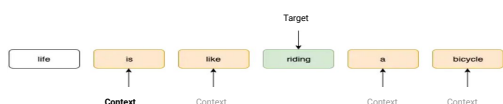
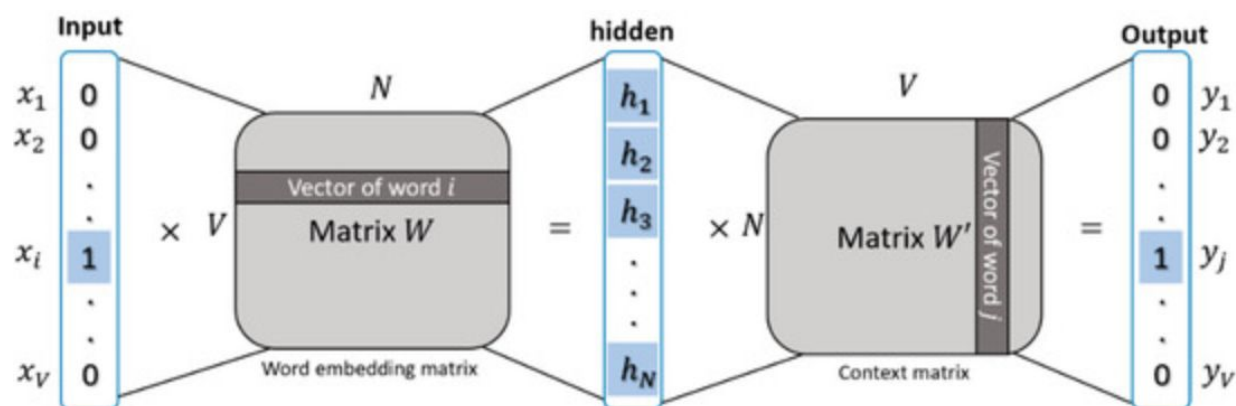
[SPEAK SLOWLY]

In Skip-gram, we do the opposite. We feed in our target word as input, and the model aims to predict what the context words are!

Wild, right? How does this work? Does the training signal not get confused by being optimized against different values?

Our loss function should be able to shine light on these questions. But before that, let's look at the forward pass algorithm in more detail.

# Skip-gram



<https://www.mdpi.com/1999-5903/11/5/114>

56

[SPEAK SLOWLY]

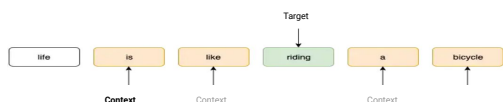
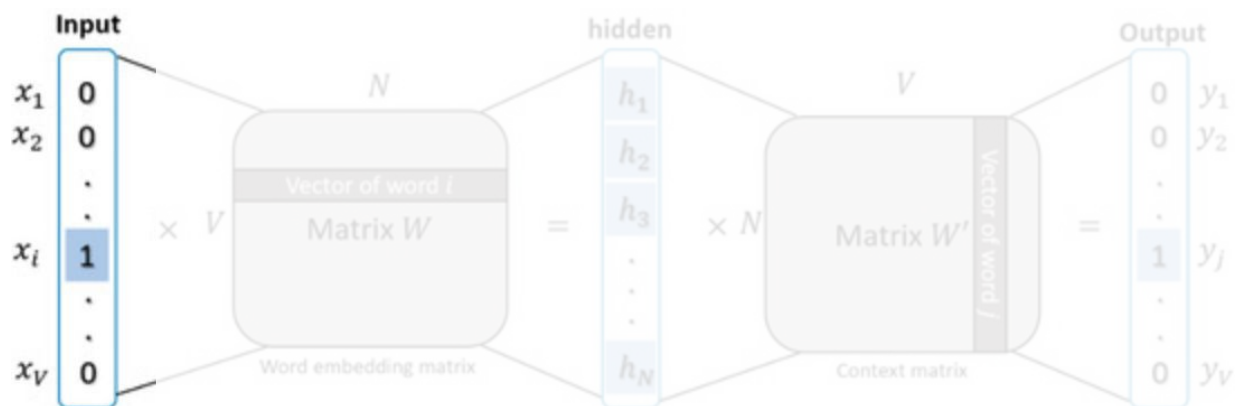
So this is our entire architecture. It is a 2 layered neural network without any non-linearities in between them. We've called the parameters of the first layer "weight embedding matrix", and the parameters of the second layer "context matrix".

And though it might look like a lot right now, it's actually incredibly simple once we understand it.

Let's run through it one piece at a time.



# Skip-gram



<https://www.mdpi.com/1999-5903/11/5/114>

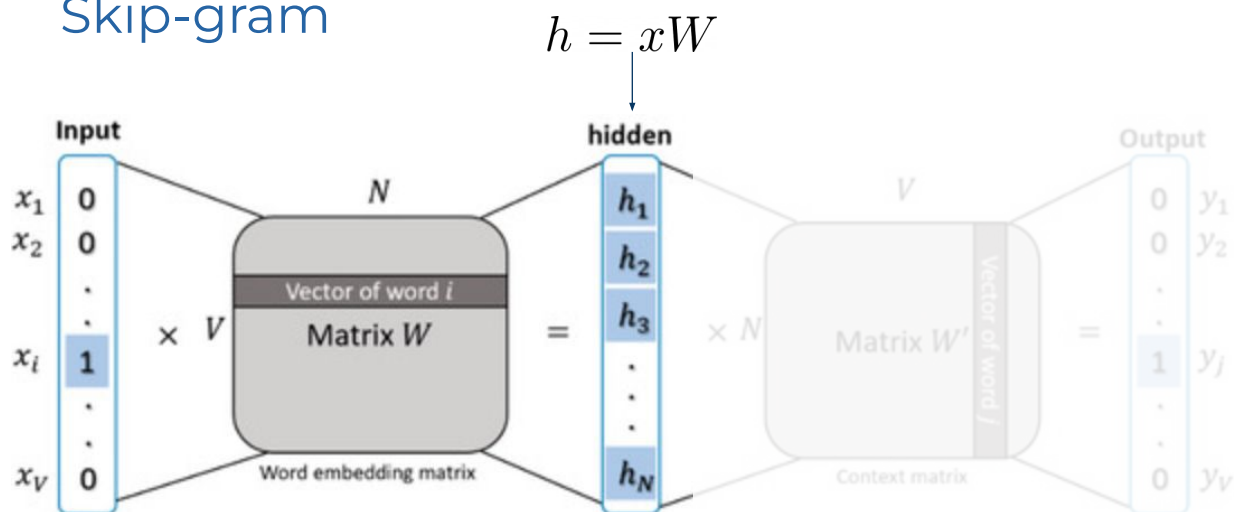
57

[SPEAK SLOWLY]

We'll start with our input.

Our input is simply the one-hot encoding of the target word. In this case, we have a one-hot encoding representing the word "riding"

# Skip-gram



58

<https://www.mdpi.com/1999-5903/11/5/114>

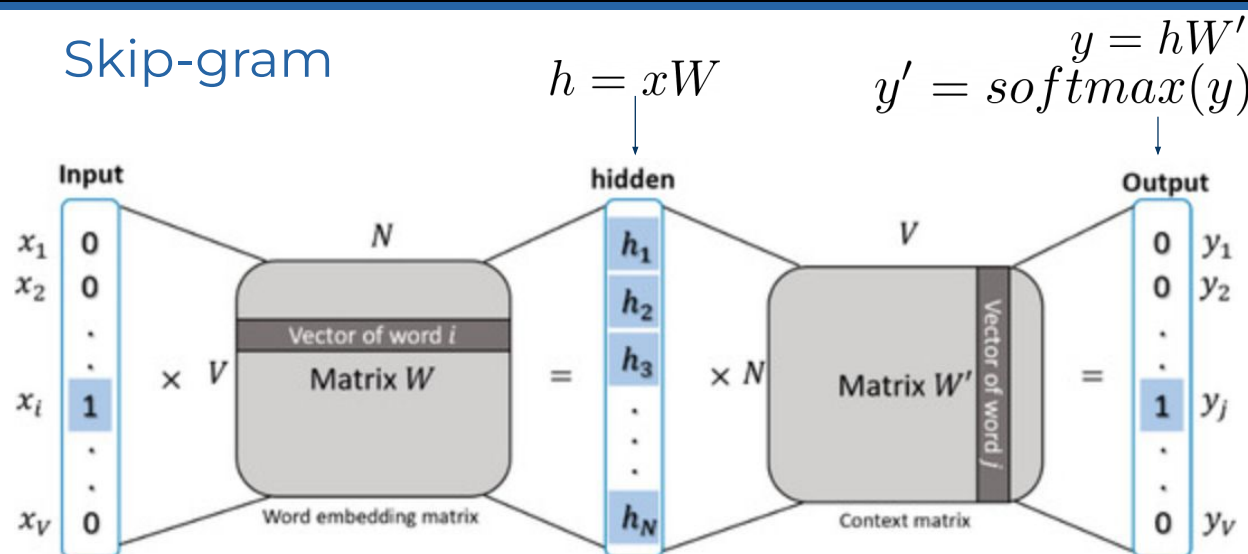
[SPEAK SLOWLY]

Now, we map that to the embedding of that target word, using weight matrix  $W$ . In other words,  $h = xW$

Once again, we take our one-hot representation. Our one-hot representation is  $V$ -dimensional.

We then multiply it by our weights matrix. Our weight matrix is  $V \times N$ . What's happening when we perform this multiplication is that we're basically extracting the  $i$ -th row from our weights matrix.  $N$  is the dimensionality of our word vectors, so once this multiplication has finished, we have our  $N$ -dimensional EMBEDDING.

# Skip-gram



59

<https://www.mdpi.com/1999-5903/11/5/114>

[SPEAK SLOWLY]

Then we perform our prediction. This happens in 2 parts.

This first part is that we map the embedding of the target word to the possible context words, using context weight matrix  $W'$ . This is done by  $y=hW'$ .

Here, the  $N$ -dimensional embedding gets transformed into **logits** by being matrix multiplied with the context matrix.

Now the second part. We then make our prediction by applying a softmax over our  $y$ . This transforms the logits into a probability distribution, and allows us to use this prediction in our loss function.

# Skip-gram

Give 1-hot vector of the target word as input  $x$

Map that to the **embedding of that target word**, using weight matrix  $W$ :

$$h = xW$$

Map the embedding of the target word to the **possible context words**, using weight matrix  $W'$ :

$$y = hW'$$

$y$  has length of whole vocabulary. Apply **softmax** to make  $y$  into a probability distribution over the whole vocabulary.

$$y' = \text{softmax}(y)$$

# Skip-gram

Two matrices of embeddings without a non-linearity.

Directly optimizing the target embedding from  $W$  to be similar to context embedding from  $W'$

61

[SPEAK SLOWLY]

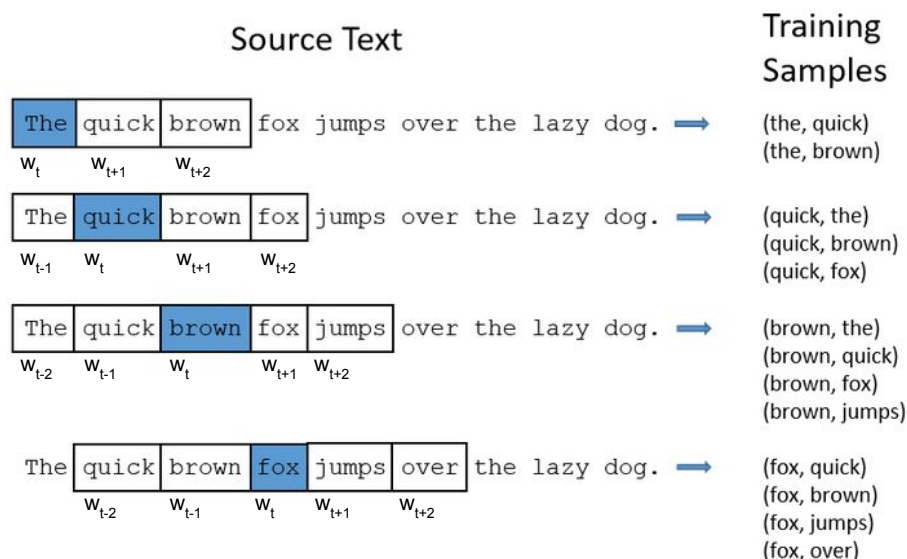
Before we dive into the loss function, it is worth spending 2 slides clarifying the above. Firstly, the whole model is essentially just two matrices of embeddings.

Directly optimizing for the embedding of the target word from  $W$  to be similar to the embedding of the context word from  $W'$ .

There is sort-of a hidden layer step, but we're not applying a non-linear activation function there.

Now, those astute among you will realise that we actually have MULTIPLE context words. But softmax would mean that we can only optimise against one word?

# Skip-gram



62

[SPEAK SLOWLY]

Well yes, for one given training sample, that is actually what is happening. However, we are building MULTIPLE training samples for each target word we have.

More specifically, for each target word, we will normally have  $c*2$  training samples (where  $c$  is our `window_size`).

Take a look at this diagram. For the target word “brown”, and  $c=2$ , we have 4 training samples.

Therefore we will be optimising the model 4 times for the word brown, each time against a different context word.

In this way, the representation for our word “brown” essentially becomes an “average” of the context words that surround it.

I want to emphasise this point. The word brown (or a target word in general) is optimised against multiple context words.

After the first sample, the word “brown” will be closer to the word “the” in vector space. After the second sample, the word “brown” gets pushed closer to the word “quick” in vector space, and so forth.

Ok, let's look at the loss function now.

## Skip-gram model

$$\frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j} | w_t; \theta)$$

63

[SPEAK SLOWLY]

So, when I was a masters student, the maths kinda scared me. I think this is a great example of when I would look at something and be confused at how to interpret it.

So I want you to spend a minute thinking about what this function means.

[ask if anyone feels comfortable enough to explain it]



## Skip-gram model

$$p(w_{t+j} | w_t) \quad \text{e.g. } p(\text{the} | \text{brown})$$

64

[SPEAK SLOWLY]

Ok so I was gonna run you through it as is. But then I realised this might be way more intuitive if we actually derived it.

We'll start with what we want to do. Maximise the probability of a context word being correctly predicted for a target word.

## Skip-gram model

$$p(w_{t+j} | w_t) \quad \text{e.g. } p(\text{the} | \text{brown})$$

$$\max \prod_j p(w_{t+j} | w_t)$$

65

[SPEAK SLOWLY]

We then want this maximised for all  $w_{\{t+j\}}$

That is, maximise the probability of predicting all the context words given the window word. (I've omitted  $j=0$  from the equation so it's easier to read, but in reality we do not compute a prediction for  $j=0$ )

## Skip-gram model

$p(w_{t+j}|w_t)$  e.g. p(the | brown)

$$\max \prod_j p(w_{t+j}|w_t)$$

$$\max \prod_t \prod_j p(w_{t+j}|w_t)$$

[SPEAK SLOWLY]

And for all t. I.e. for all words in our document

## Skip-gram model

$p(w_{t+j}|w_t)$  e.g.  $p(\text{the} | \text{brown})$

$$\max \prod_j p(w_{t+j}|w_t)$$

$$\max \prod_t \prod_j p(w_{t+j}|w_t)$$

$$= \min_{\theta} - \log \prod_t \prod_j p(w_{t+j}|w_t; \theta)$$

$$= \min_{\theta} - \sum_t \sum_j \log p(w_{t+j}|w_t; \theta)$$

67

[SPEAK SLOWLY]

As you know, we like minimising loss functions instead of maximising them. And also, multiplying values leads to vanishing gradients, so we'd like to sum them instead.

So we apply negative log likelihood on our loss function.

## Skip-gram model

$$\frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j} | w_t; \theta)$$

$$p(w_{t+j} | w_t) = \frac{\exp(u_{w_{t+j}}^\top h_{w_t})}{\sum_{w'=1}^W \exp(u_{w'}^\top h_{w_t})}$$

68

[SPEAK SLOWLY]

The only differences between this and the derived formula is that:

1. This explicitly shows that we don't calculate a loss over  $j=0$
2. The loss for one document is the average loss for each word in that document

And obviously the loss for the corpus will be this applied over every document in our corpus (e.g we'll have another  $1/N$ ,  $\sum_N$  on the left hand side)

As mentioned before, the probability distribution  $p(w_{t+j} | w_t)$  is computed just using a softmax.

All in all, this is actually just a cross-entropy loss function for every target-context training pair we have.

# Using Word Embeddings

Use the learned word embedding matrix for downstream tasks

69

[SPEAK SLOWLY]

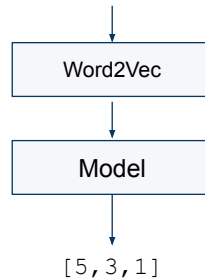
Ok. We've trained our model. But now what? How do we use our word embeddings?

Basically, the first thing to note is that we can separate out the training of the word embeddings and the downstream task we wanted to do.

# Using Word Embeddings

Use the learned word embedding matrix for downstream tasks

```
[“this movie had a brilliant storyline with great action”,  
“some parts were not so great, but overall pretty ok”,  
“even my dog went to sleep watching this 🍌 trash”]
```



70

[SPEAK SLOWLY]

What this means is that we can take our text corpus and train a word2vec algo on it.

We can then use one of the learned weights embedding matrix (next slide) to extract out a vector for each word.

Using techniques that we'll look at over the next few lectures, we can then feed these numerical representations into our model for it to perform the classification.

# Embedding matrices

Two ways of thinking about an embedding matrix.

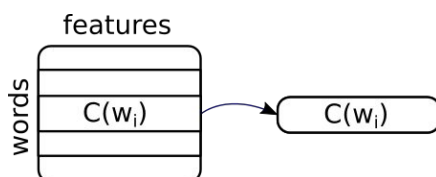
It is a normal **weight matrix**, multiplied with a 1-hot input vector



The diagram illustrates the multiplication of a 1-hot input vector with a weight matrix. On the left, a horizontal vector of zeros with a '1' at index  $i$  is shown, labeled 'all words'. This is multiplied by a vertical matrix labeled  $C(w_i)$ . The result is a single row from the matrix, also labeled  $C(w_i)$ .

$$\begin{bmatrix} 0 & 0 & 0 & \dots & 1 & \dots & 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = \begin{bmatrix} 10 & 12 & 19 \end{bmatrix}$$

This means **each row** contains a word embedding, which we can extract



<http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/>

71

[SPEAK SLOWLY]

Ok, so to re-iterate, using ONE of the learned matrices from the NN architecture we looked at a few slides ago (empirically it doesn't really matter which one we use), we can utilise this to extract out an embedding for each of our words.

Using it is actually very simple.

The two things I've shown here are essentially equivalent.

Basically, we have our weights matrix (here,  $C$ ). Let's say we have a one-hot encoding of a word we want an embedding for. We multiply our one-hot encoding with our weights matrix, which just extracts the row in that weights matrix where the index is one.

So if the one-hot encoding is hot at  $i=10$ , then we're simply extracting the 10th row from our word embedding matrix



# Pre-training and fine-tuning

72

[SPEAK SLOWLY]

Who knows what this is?

# A problem

Vocabularies could be 100k+ in size...

$$p(w_{t+j}|w_t) = \frac{\exp(u_{w_{t+j}}^\top h_{w_t})}{\sum_{w'=1}^W \exp(u_{w'}^\top h_{w_t})}$$

Expensive!

73

[SPEAK SLOWLY]

[Take questions first]

In order to compute a single forward pass of the model, we have to sum across the entire corpus vocabulary (in the softmax)

This gets inefficient with large vocabularies and large embeddings

E.g. for 300-dimensional embeddings and  $V = 10K$  vocabulary, we have to do 3 million feature-weight multiplications, with each embedding matrix (6M total).

# Negative Sampling

Try to approximate the softmax instead

74

[SPEAK SLOWLY]

As I mentioned when introducing Word2Vec, there were 2 strategies which allowed us to perform training more efficiently. Negative sampling is one of these strategies which attempts to approximate the softmax instead of explicitly computing it.

# Negative Sampling

Try to approximate the softmax instead

Use binary classification to distinguish **real** context words from **noise** context words

75

[SPEAK SLOWLY]

With negative sampling, we train a vocabulary number of binary classifiers. That is, each item in the vocabulary has a binary classifier which attempts to predict whether a target word appears in the context of a context word or not.

# Negative Sampling

Try to approximate the softmax instead

Use binary classification to distinguish **real** context words from **noise** context words

$$\log p(D = 1|w_t, w_{t+1}) + k \mathbb{E}_{\tilde{c} \sim P_{noise}} [\log p(D = 0|w_t, \tilde{c})]$$

where  $p(D = 1|w_t, w_{t+1})$  is binary logistic regression probability of seeing the word  $w_t$  in the context  $w_{t+1}$ .

76

[SPEAK SLOWLY]

We'll spend the next few slides discussing this loss function, but showing it right now provides good intuition for what's to come.

So, this loss function has 2 terms. The left hand side ( $D=1$ ), and the right hand side ( $D=0$ ).

Remember, we're now working with a binary classification task. So our goal is to predict whether a target word and a context word appear in context of each other or not.

For the left hand side, what we're saying is that we want to maximise the probability of correctly predicting that a target word and a TRUE context word appear in context of each other (i.e. predicting a 1 with our binary classifier)

And the right hand side is saying that we want to maximise the probability of correctly predicting that a target word and a noise word do NOT appear in context of each other (i.e. predicting a 0 with our binary classifier).

# Negative Sampling

Try to approximate the softmax instead

Use binary classification to distinguish **real** context words from **noise** context words

$$\log p(D = 1|w_t, w_{t+1}) + k \mathbb{E}_{\tilde{c} \sim P_{noise}} [\log p(D = 0|w_t, \tilde{c})]$$

where  $p(D = 1|w_t, w_{t+1})$  is binary logistic regression probability of seeing the word  $w_t$  in the context  $w_{t+1}$ .

Approximate expectation by drawing k contrastive context words from a noise distribution  $\tilde{c}$

77

[SPEAK SLOWLY]

These noise words,  $\tilde{c}$  come from a noise distribution. The noise distribution could be random sampling or frequency based sampling. There are k noise words.

Let's look at an example to make this clearer.

# Skip-gram model - negative sampling

The quick brown fox jumps over the lazy dog.

To predict quick from the, select  $k$  noisy (contrastive) words which are NOT in the context window of the

- For example,  $k = 1$ , and randomly selected noisy word = sheep
- Compute loss function for observed and contrastive pairs. The objective, at this time step becomes

$$\log p(D = 1 | \text{the}, \text{quick}) + \log(p(D = 0 | \text{the}, \text{sheep}))$$

78

[SPEAK SLOWLY]

So, let's take this example "The quick brown fox" etc.

As we looked at in the previous slide, we're going to have 2 parts of our loss term. The first part is a true context word.

In this case, our true context word is "quick".

We're also going to have  $k$  noise words. Here, let's say  $k=1$ . So we're going to sample the rest of our vocabulary and find a random word which does not appear in the true context of this target word. In this case, "sheep"

Our loss function is now the probability of correctly predicting that "the" and "quick" appear of context of each other, and correctly predicting that "the" and "sheep" do not appear in context of each other.

Note, "sheep" *can* be a true context word of target word "the". But for sequence we have been given, it is not a context word of "the". Therefore, it is a valid sample from our noise distribution (as the noise distribution only considers the current window around our target word)

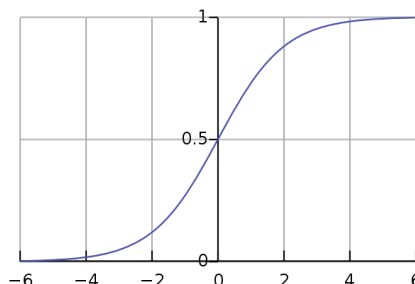
# Skip-gram model - negative sampling

Probability that was defined using the **softmax function**

$$p(w_{t+j}|w_t) = \frac{\exp(u_{w_{t+j}}^\top h_{w_t})}{\sum_{w'=1}^W \exp(u_{w'}^\top h_{w_t})}$$

is now defined by **sigmoid function**

$$p(D = 1|w_t, w_{t+1}) = \frac{1}{1 + \exp(-u_{w_{t+1}}^\top h_{w_t})}$$



The model now has  $|V|$  binary classifiers, one for each possible context word

79

[SPEAK SLOWLY]

Ok, so as I mentioned, we're doing this all to overcome the costly sum operation in softmax.

Instead of a softmax, we're now using a sigmoid loss function, which simply involves us computing the probability for each context & noise embedding. More on the computational efficiency of this in a second.



# Skip-gram model - negative sampling

How many **k**?

- 5-20 negative words works well for smaller datasets
- 2-5 negative words

80

[SPEAK SLOWLY]

So what is the ideal number for k?

Well, it depends on the size of the dataset. Smaller datasets need a larger k, whereas smaller k works better for larger datasets (e.g. 2-5).

Now, let's revisit the computational aspect of it. Let's say we had k=5:

For a model with weight matrix 300x10K, need to update weights for 1 positive word, plus 5 negative samples.

That's 6 output neurons. So  $6 * 300 = 1800$  weight parameters need to be updated.

That is only 0.06% of the 3M weights in the output layer!

# Skip-gram model - negative sampling

How many **k**?

- 5-20 negative words works well for smaller datasets
- 2-5 negative words

## Noise Distribution

Frequency based > random sampling

$$p(w_i) = \frac{1}{|V|} \quad p(w_i) = \frac{f(w_i)^{3/4}}{\sum_{w'} f(w')^{3/4}}$$

81

[SPEAK SLOWLY]

Regarding our noise distribution, I know I've been mentioning "random sampling" throughout the last few slides.

Yes, this works and you can do it, but it was shown that frequency based sampling tends to outperform pure random sampling.

With frequency based sampling, we up-weight more frequently occurring words

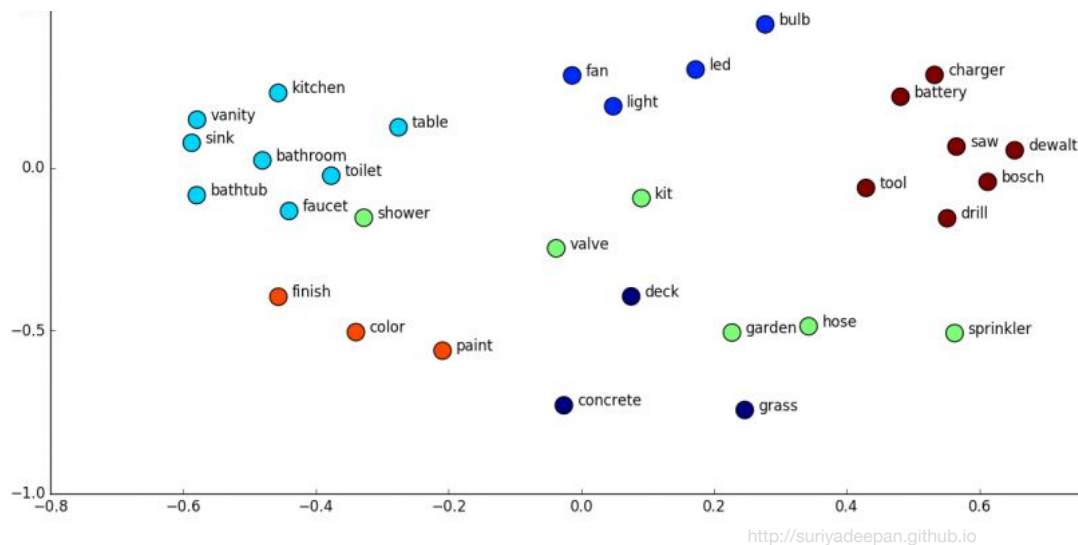
**What can we do with vectors?**

# Examples of what vectors capture

We can use them to find the most similar other words, given an input word

FRANCE	JESUS	XBOX	REDDISH	SCRATCHED	MEGABITS
AUSTRIA	GOD	AMIGA	GREENISH	NAILED	OCTETS
BELGIUM	SATI	PLAYSTATION	BLUISH	SMASHED	MB/S
GERMANY	CHRIST	MSX	PINKISH	PUNCHED	BIT/S
ITALY	SATAN	IPOD	PURPLISH	POPPED	BAUD
GREECE	KALI	SEGA	BROWNISH	CRIMPED	CARATS
SWEDEN	INDRA	psNUMBER	GREYISH	SCRAPED	KBIT/S
NORWAY	VISHNU	HD	GRAYISH	SCREWED	MEGAHERTZ
EUROPE	ANANDA	DREAMCAST	WHITISH	SECTIONED	MEGAPIXELS
HUNGARY	PARVATI	GEFORCE	SILVERY	SLASHED	GBIT/S
SWITZERLAND	GRACE	CAPCOM	YELLOWISH	RIPPED	AMPERES

# Examples of what vectors capture



# Analogy Recovery

## Task:

a is to b as c is to d

## For example:

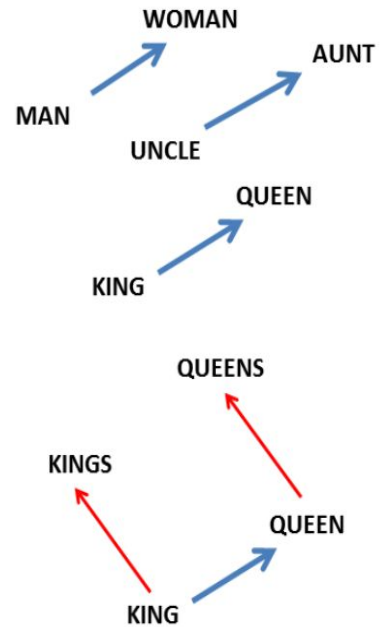
'apple' is to 'apples' as 'car' is to 'cars'

**Idea:** The offset of the vectors should reflect their relation.

$$a - b \approx c - d$$

$$d \approx c - a + b$$

$$queen \approx king - man + woman$$



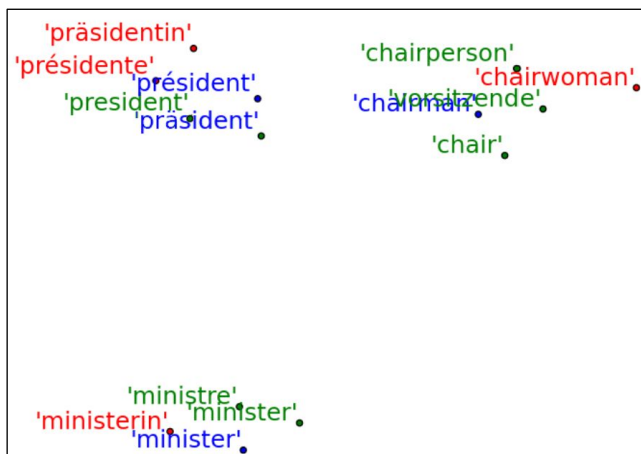
# Analogy Recovery

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

# Multilingual word embeddings

The words from English, German and French are mapped into clusters based on meaning.

The colours indicate gender:  
blue=male  
red=female  
green=neutral





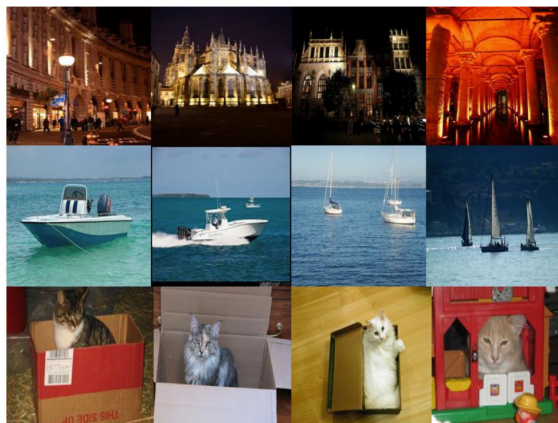
# Multimodal word embeddings



- day + night =

- flying + sailing =

- bowl + box =



[Kiros et. al. 2014 "Unifying Visual-Semantic Embeddings with Multimodal Neural Language Models"](#)

## Things to consider

- Word embeddings require only plain text - which we have a lot
- They are very easy to use
- Already pretrained vectors also available (trained on 100B words)
- They will help your models start from an informed position

but.....

- Require large amounts of data to train
- Low quality for rare words
- No coverage for unseen words
- Antonyms tend to have similar distributions: e.g. "good" and "bad"
- Does not consider morphological similarity: "car" and "cars" are independent
- Does not differentiate between different meanings of a word

# Other types of word embeddings

## Context matters



<http://jalammar.github.io/illustrated-bert/>

Instead of a fixed embedding for each word, ELMo looks at the entire sentence before assigning each word in it an embedding

# Pre-trained word embeddings

## Glove

Homepage: <https://nlp.stanford.edu/projects/glove/>

Vector dimensionality: 50 / 100 / 200 / 300

Vocabulary size: 400,000 - 2.2 million

Training data: Wikipedia + GigaWord / CommonCrawl / Twitter

## Fast-text

Homepage: <https://github.com/facebookresearch/fastText/>

Vector dimensionality: 300

Training data: Common Crawl and Wikipedia

Trained for 157 different languages

## Other types of word embeddings

- CBOW: Continuous Bag of Words (Mikolov et al., 2013)
  - Glove (Pennington, Socher and Manning, 2014)
  - Fasttext (Joulin et al., 2016)
  - Contextualised word embeddings
    - ELMo (Peters et al., 2018)
    - BERT (Devlin et al., 2018)
    - RoBERTa (Liu et al., 2019)
    - ALBERT (Lan et al., 2019)
    - XLNet (Yang et al., 2019)
    - GPT-2 (Radford et al., 2019)
    - ELECTRA (Clark et al., 2020)
    - DistilBERT (Sanh et al., 2020)
    - DeBERTa (He et al., 2020)
    - GPT-3 (Brown et al., 2020)
- We'll come back to these



# Byte Pair Encoding

# Byte Pair Encoding

Instead of manually specifying rules for lemmatisation or stemming, let's learn from data which character sequences occur together frequently (and are therefore more likely to be meaningful).

Can do this using Byte Pair Encoding (Sennrich et al. 2015)

Originally a compression technique.

For example, given an input file: 'aaabdaaabc'

1. Find byte pair that occurs most often: "aa"
2. Replace that with "Z" → ZabdZabac
3. Repeat the process, replacing byte pair "ab" with Y → ZYdZYac
4. ...
5. Keep a list of replacements to uncompress later.

# Byte Pair Encoding: Training algorithm

1. Start with a vocabulary of all individual characters:  
`{"a", "b", "c", "d", "e", "f", "g", ....}`
1. Split the words in your training corpus also into individual characters  
`"Table" -> "t", "a", "b", "l", "e", "_"` ("`_`" indicates the end-of-word)
1. Find which two vocabulary items occur together most frequently in the training corpus. (For example, `"ab"`).
2. Add that combination (`"ab"`) as a new vocabulary item.  
`{"a", "b", "c", "d", "e", "f", "g", ....., "ab"}`
3. Merge all occurrences of that combination in your corpus.  
`"t", "a", "b", "l", "e", "_" -> "t", "ab", "l", "e", "_"`
4. Repeat until a desired number of merges has been performed



# Byte Pair Encoding: Training example

1. Let's start with a corpus of words with the following frequencies:

("low": 5), ("lower": 2), ("newest": 6), ("widest": 3)

2. Break into characters, add end-of-word markers:

("l" "o" "w" " ": 5), ("l" "o" "w" "e" "r" " ": 2), ("n" "e" "w" "e" "s" "t" " ": 6), ("w" "i" "d" "e" "s" "t" " ": 3)

3. Find most frequent bigram of characters

"t " occurred  $6 + 3 = 9$  times ("e s" or "s t" are also equal candidates)

4. Merge these two and update vocabulary

("l" "o" "w" " ": 5), ("l" "o" "w" "e" "r" " ": 2), ("n" "e" "w" "e" "s" "t\_": 6), ("w" "i" "d" "e" "s" "t\_": 3)

5. Repeat from step 3 for number of 'merge' operations (hyperparameter)

# Byte Pair Encoding: Training example

## Iteration 1

new merge: ('t', '\_')

vocab: {'new est\_': 6, 'low er\_': 2, 'low \_': 5, 'wid est\_': 3}

## Iteration 2

new merge: ('s', 't\_')

vocab: {'low er\_': 2, 'new est\_': 6, 'wid est\_': 3, 'low \_': 5}

## Iteration 3

new merge: ('e', 'st\_')

vocab: {'low er\_': 2, 'low \_': 5, 'new est\_': 6, 'wid est\_': 3}

## Iteration 4

new merge: ('l', 'o')

vocab: {'wid est\_': 3, 'low \_': 5, 'new est\_': 6, 'low er\_': 2}

## Iteration 5

new merge: ('lo', 'w')

vocab: {'low \_': 5, 'low er\_': 2, 'new est\_': 6, 'wid est\_': 3}

Keep track of the merge operations in order:

"t", "\_" → "t\_"

"s", "t\_" → "st\_"

"e", "st\_" → "est\_"

"l", "o" → "lo"

"lo", "w" → "low"

[http://ufal.mff.cuni.cz/~helcl/courses/npl116/python/byte\\_pair\\_encoding.html](http://ufal.mff.cuni.cz/~helcl/courses/npl116/python/byte_pair_encoding.html)

# Byte Pair Encoding: Training example

## Iteration 6

new merge: ('e', 'w')

vocab: {'w i d est\_': 3, 'low \_': 5, 'low e r \_': 2, 'n ew est\_': 6}

## Iteration 7

new merge: ('ew', 'est\_')

vocab: {'n ewest\_': 6, 'low \_': 5, 'low e r \_': 2, 'w i d est\_': 3}

## Iteration 8

new merge: ('n', 'ewest\_')

vocab: {'newest\_': 6, 'low \_': 5, 'low e r \_': 2, 'w i d est\_': 3}

## Iteration 9

new merge: ('low', '\_')

vocab: {'newest\_': 6, 'w i d est\_': 3, 'low \_': 5, 'low e r \_': 2}

## Iteration 10

new merge: ('i', 'd')

vocab: {'low e r \_': 2, 'newest\_': 6, 'w i d est\_': 3, 'low \_': 5}

Keep track of the merge operations in order:

"t", "\_" → "t\_"

"s", "t\_" → "st\_"

"e", "st\_" → "est\_"

"l", "o" → "lo"

"lo", "w" → "low"

"e", "w" → "ew"

"ew", "est\_" → "ewest\_"

"n", "ewest\_" → "newest\_"

"low", "\_" → "low\_"

"i", "d" → "id"

[http://ufal.mff.cuni.cz/~helcl/courses/npl116/ipython/byte\\_pair\\_encoding.html](http://ufal.mff.cuni.cz/~helcl/courses/npl116/ipython/byte_pair_encoding.html)

# Byte Pair Encoding

The end-of-word symbol “\_”

Not actually an underscore (there are underscores in normal text).  
Could be any symbol or character that is not otherwise used in the normal vocabulary.

Why do we have it?

1. It distinguishes suffixes.  
“st” in “star” vs “st\_” in “widest\_”
1. It tells us where to insert spaces when putting words back together.  
“th” “is” “is” “a” “sen” “te” “nce” → “thisisasentence”  
“th” “is\_” “is\_” “a\_” “sen” “te” “nce\_” → “this is a sentence ”

# Byte Pair Encoding: Inference Algorithm

Once the model is trained, we get a new word which we didn't see during training.

What do we do?

1. Split word into characters
2. Get all symbol bigrams of the word
3. Find a symbol pair that appeared first among the symbol merges dictionary
4. Apply the merge on the word
5. Repeat from step 3 until no more merge operations apply

# Byte Pair Encoding: Inference Example

New word at inference time: 'lowest'

First split into characters: ('l', 'o', 'w', 'e', 's', 't', '')

## Iteration 1:

bigrams in the word: {('l', 'o'), ('o', 'w'), ('w', 'e'), ('e', 's'), ('s', 't'), ('t', '\_')}

candidate for merging: ('t', '\_')

word after merging: ('l', 'o', 'w', 'e', 's', 't\_')

## Iteration 2:

bigrams in the word: {('l', 'o'), ('o', 'w'), ('w', 'e'), ('e', 's'), ('s', 't\_')}

candidate for merging: ('s', 't\_')

word after merging: ('l', 'o', 'w', 'e', 'st\_')

## Iteration 3:

bigrams in the word: {('l', 'o'), ('o', 'w'), ('w', 'e'), ('e', 'st\_')}

candidate for merging: ('e', 'st\_')

word after merging: ('l', 'o', 'w', 'est\_')

## Merge operations:

"t", "\_" → "t\_"

"s", "t\_" → "st\_"

"e", "st\_" → "est\_"

"l", "o" → "lo"

"lo", "w" → "low"

"e", "w" → "ew"

"ew", "est\_" → "ewest\_"

"n", "ewest\_" → "newest\_"

"low", "\_" → "low\_"

"i", "d" → "id"

# Byte Pair Encoding: Inference Example

## Iteration 4:

bigrams in the word: {'l', 'o'}, ('o', 'w'), ('w', 'est\_')

candidate for merging: ('l', 'o')

word after merging: ('lo', 'w', 'est\_')

## Iteration 5:

bigrams in the word: {'lo', 'w'}, ('w', 'est\_')

candidate for merging: ('lo', 'w')

word after merging: ('low', 'est\_')

## Iteration 6:

bigrams in the word: {'low', 'est\_'}

No more merge operations can be applied, algorithm stops.

Resulting BPE split: ('low', 'est\_')

## Merge operations:

"t", " " → "t\_ "

"s", "t\_ " → "st\_ "

"e", "st\_ " → "est\_ "

"l", "o" → "lo"

"lo", "w" → "low"

"e", "w" → "ew"

"ew", "est\_ " → "ewest\_ "

"n", "ewest\_ " → "newest\_ "

"low", " " → "low\_ "

"i", "d" → "id"

# Byte Pair Encoding

The individual characters are always in the vocabulary, so we can map even “hyzxixmsg” to “h”, “y”, “z”, “x”, “i”, “x”, “m”, “s”, “g”

However, when dealing with Unicode characters, we may encounter characters that we haven't seen during training.

A variant of BPE works on sequences of bytes instead of characters. The base vocabulary size is then only 256 and we don't have to deal with unseen characters.

BPE is successfully used by GPT, GPT-2, GPT-3, RoBERTa, BART, and DeBERTa.

More info and an example: <https://huggingface.co/course/chapter6/5>





## TF-IDF

**Problem:** Common words (“a”, “the”, “it”, “this”, ...) dominate the counts but provide almost no additional information.

Can weight the vectors using TF-IDF:  $\text{TF-IDF}_{w,d,D} = \text{TF}_{w,d} \text{IDF}_{w,D}$

**d** - the item that we are creating a vector for. E.g., “magazine” or “newspaper” from the previous example. Could also be a full document that we want to create a vector for.

**w** - the context word that we want to weight. E.g., “reading” or “a” from the previous example.

**D** - A collection of documents (or sentences) that we are using for creating these vectors

# TF-IDF

## Term Frequency (TF):

Upweights words  $w$  that are more important to  $d$

Frequency of  $w$  occurring together with  $d$

$$TF_{w,d} = \frac{\text{count}(w, d)}{\sum_{w'} \text{count}(w', d)}$$

Size of document collection

## Inverse Document Frequency (IDF):

Downweights words that appear everywhere

$$IDF_{w,D} = \log \frac{|D|}{|\{d \in D : w \in d\}|}$$

Number of documents in  $D$  that contain  $w$

$$TF\text{-}IDF_{w,d,D} = TF_{w,d} IDF_{w,D}$$