



**DATA  
SCIENCE**

word2vec and friends

<https://github.com/bmtgoncalves/word2vec-and-friends/>

*Bruno Gonçalves*

[www.bgoncalves.com](http://www.bgoncalves.com)



# Teaching machines to read!

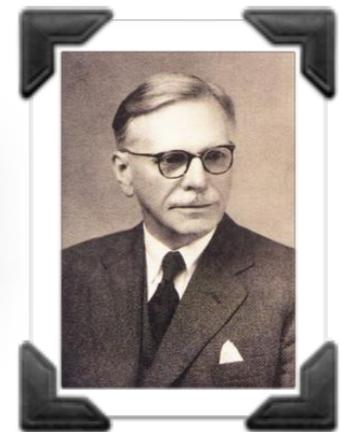
- Computers are really good at crunching numbers but not so much when it comes to words.
- Perhaps can we represent words numerically?

$$v_{\text{after}} = (0, 0, 0, 1, 0, 0, \dots)^T$$
$$v_{\text{above}} = (0, 0, 1, 0, 0, 0, \dots)^T$$

One-hot  
encoding

- Can we do it in a way that preserves **semantic** information?

“You shall know a word by the company it keeps”  
(J. R. Firth)



- **Words** that have similar **meanings** are used in similar **contexts** and the context in which a word is used helps us understand its meaning.

The red **house** is beautiful.

The blue **house** is old.

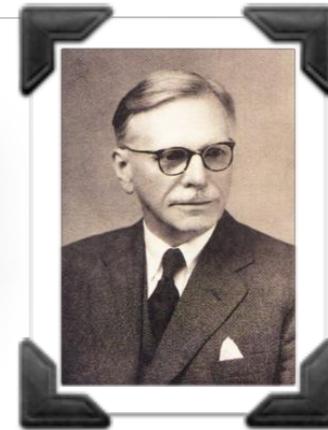
The red **car** is beautiful.

The blue **car** is old.

a	1
about	2
above	3
after	4
again	5
against	6
all	7
am	8
an	9
and	10
any	11
are	12
aren't	13
as	14
...	...

# Teaching machines to read!

"You shall know a word by the company it keeps"  
(J. R. Firth)



- Words with similar meanings should have similar representations.
- From a word we can get some idea about the context where it might appear

\_\_\_\_\_ house \_\_\_\_\_  
\_\_\_\_\_ car \_\_\_\_\_

$$\max p(C|w)$$

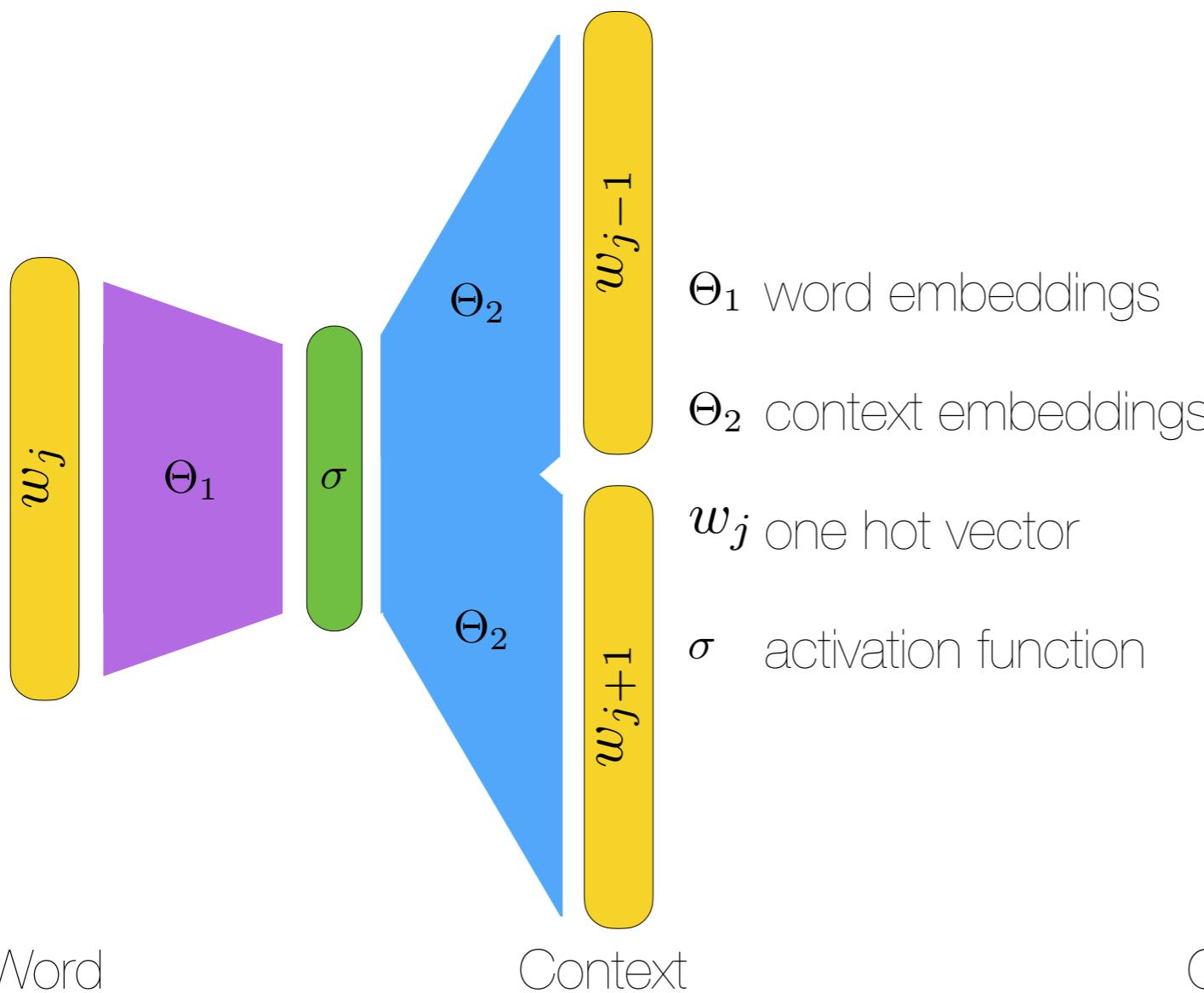
- And from the context we have some idea about possible words

The red \_\_\_\_\_ is beautiful.  
The blue \_\_\_\_\_ is old.

$$\max p(w|C)$$

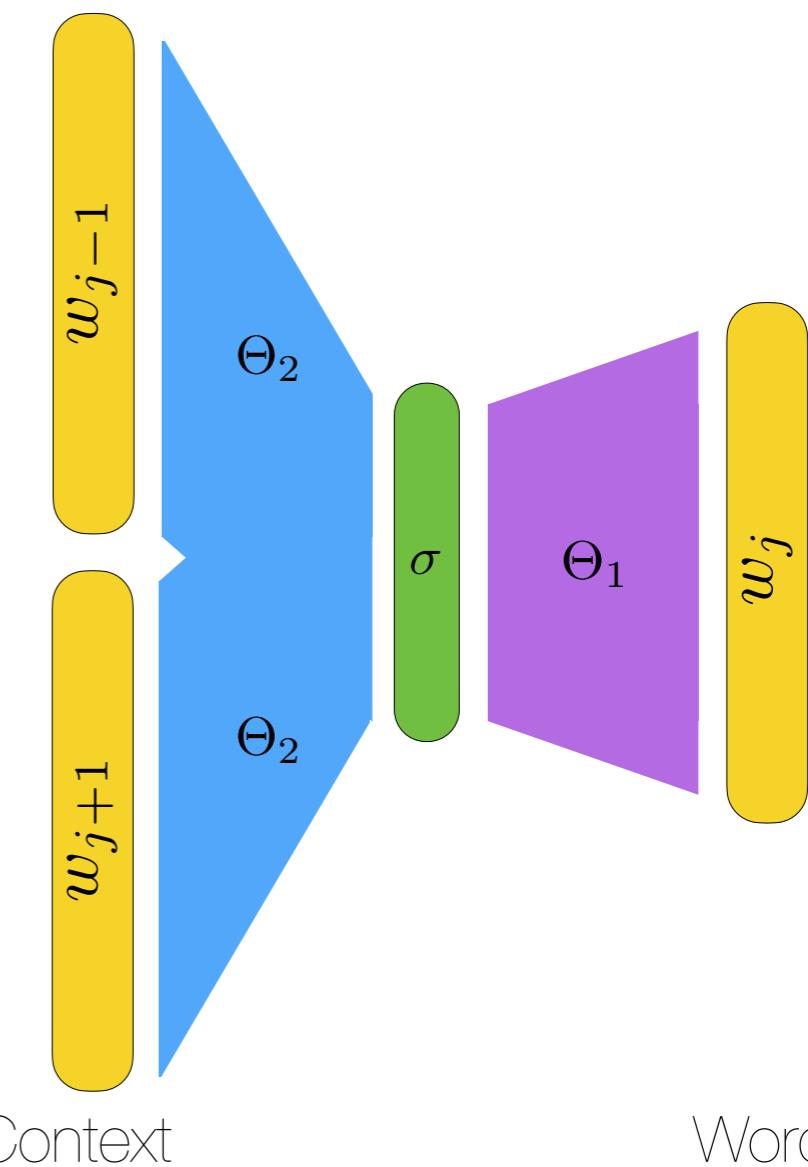
Skipgram

$$\max p(C|w)$$



Continuous Bag of Words

$$\max p(w|C)$$



# Skipgram

---

- Let us take a better look at a simplified case with a single context word
- Words are one-hot encoded vectors  $w_j = (0, 0, 1, 0, 0, 0, \dots)^T$  of length  $V$

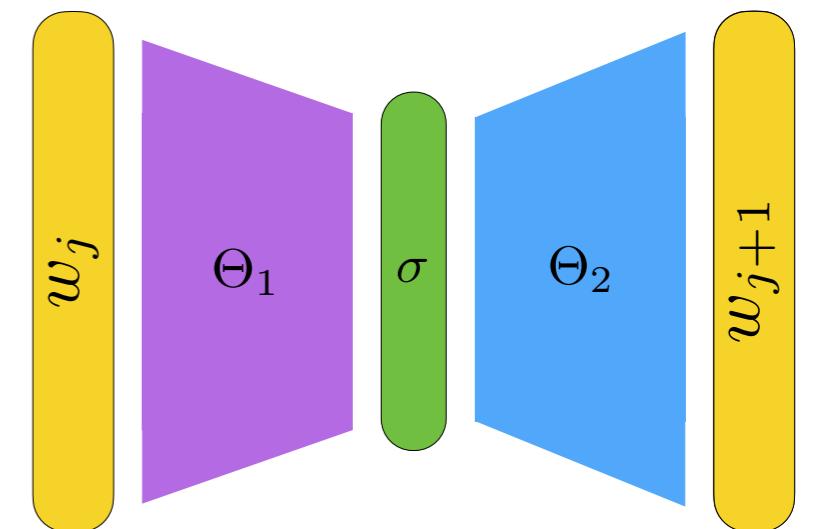
- $\Theta_1$  is an  $(M \times V)$  matrix so that when we take the product:

$$\Theta_1 \cdot w_j$$

- We are effectively selecting the  $j$ 'th column of  $\Theta_1$ :

$$v_j = \Theta_1 \cdot w_j$$

- The **linear** activation function simply passes this value along which is then multiplied by  $\Theta_2$ , a  $(V \times M)$  matrix.



- Each element  $k$  of the output layer its then given by:

$$u_k^T \cdot v_j$$

- We convert these values to a normalized probability distribution by using the **softmax**

# Softmax

---

- A standard way of converting a set of numbers to a normalized probability distribution:

$$\text{softmax}(x) = \frac{\exp(x_j)}{\sum_l \exp(x_l)}$$

- With this final ingredient we obtain:

$$p(w_k|w_j) \equiv \text{softmax}(u_k^T \cdot v_j) = \frac{\exp(u_k^T \cdot v_j)}{\sum_l \exp(u_l^T \cdot v_j)}$$

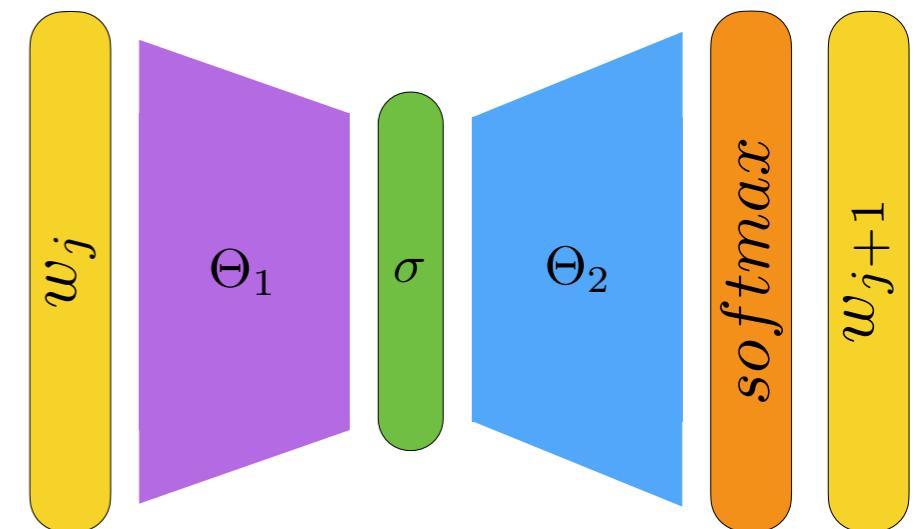
- Our goal is then to learn:

$$\Theta_1 \quad \Theta_2$$

- so that we can predict what the next word is likely to be using:

$$p(w_{j+1}|w_j)$$

- But how can we quantify how far we are from the correct answer? Our error measure shouldn't be just binary (right or wrong)...



# Cross-Entropy

---

- First we have to recall that what we are, in effect, comparing two probability distributions:

$$p(w_k|w_j)$$

- and the one-hot encoding of the context:

$$w_{j+1} = (0, 0, 0, 1, 0, 0, \dots)^T$$

- The Cross Entropy measures the distance, in number of bits, between two probability distributions  $\mathbf{p}$  and  $\mathbf{q}$ :

$$H(p, q) = - \sum_k p_k \log q_k$$

- In our case, this becomes:

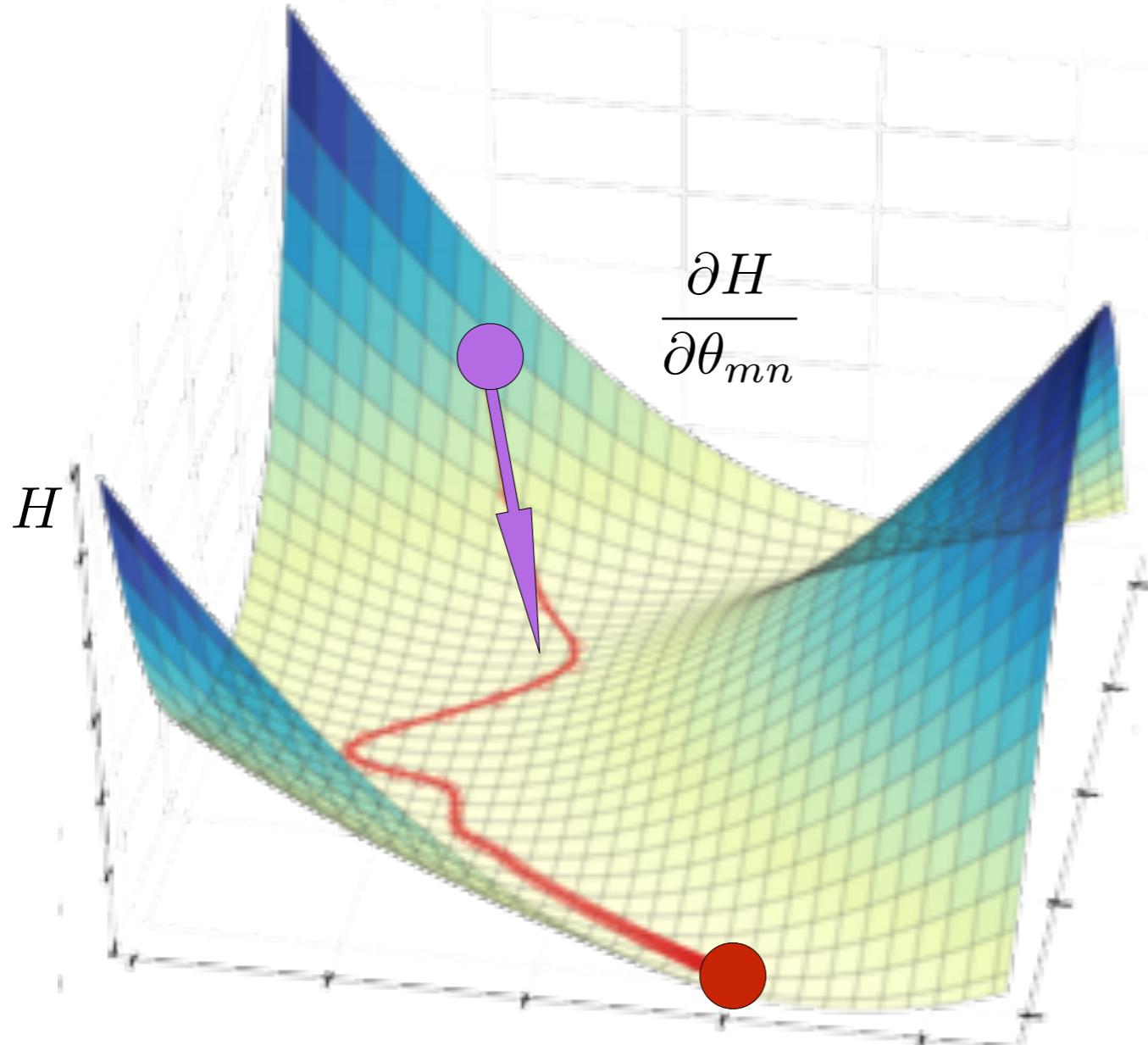
$$H[w_{j+1}, p(w_k|w_j)] = - \sum_k w_{j+1}^k \log p(w_k|w_j)$$

- So it's clear that the only non zero term is the one that corresponds to the "hot" element of  $w_{j+1}$

$$H = - \log p(w_{j+1}|w_j)$$

- This is our Error function. But how can we use this to update the values of  $\Theta_1$  and  $\Theta_2$ ?

# Gradient Descent



- Find the gradient for each training batch
  - Take a step **downhill** along the direction of the gradient
- $$\theta_{mn} \leftarrow \theta_{mn} - \alpha \frac{\partial H}{\partial \theta_{mn}}$$
- where  $\alpha$  is the step size.
  - Repeat until "convergence".

# Chain-rule

---

- How can we calculate

$$\frac{\partial H}{\partial \theta_{mn}} = \frac{\partial}{\partial \theta_{mn}} \log p(w_{j+1} | w_j) \quad \theta_{mn} = \left\{ \theta_{mn}^{(1)}, \theta_{mn}^{(2)} \right\}$$

- we rewrite:

$$\frac{\partial H}{\partial \theta_{mn}} = \frac{\partial}{\partial \theta_{mn}} \log \frac{\exp(u_k^T \cdot v_j)}{\sum_l \exp(u_l^T \cdot v_j)}$$

- and expand:

$$\frac{\partial H}{\partial \theta_{mn}} = \frac{\partial}{\partial \theta_{mn}} \left[ u_k^T \cdot v_j - \log \sum_l \exp(u_l^T \cdot v_j) \right]$$

- Then we can rewrite:

$$u_k^T \cdot v_j = \sum_q \theta_{kq}^{(2)} \theta_{qj}^{(1)}$$

- and apply the chain rule:

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x}$$

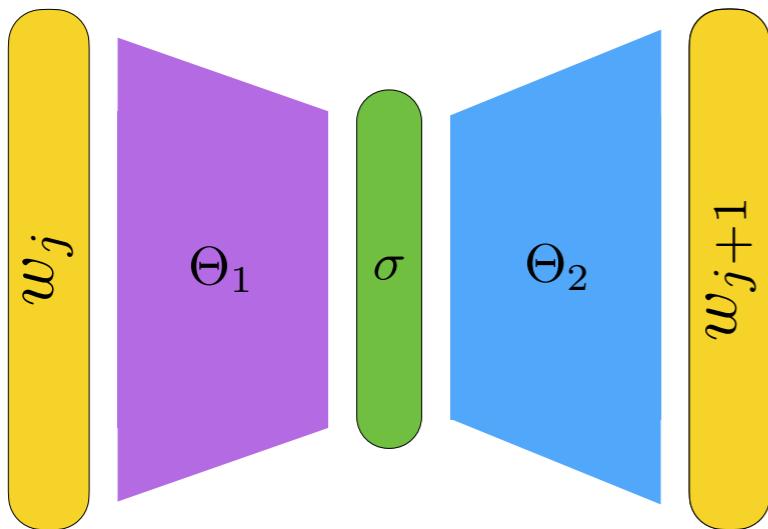
# Training procedures

---

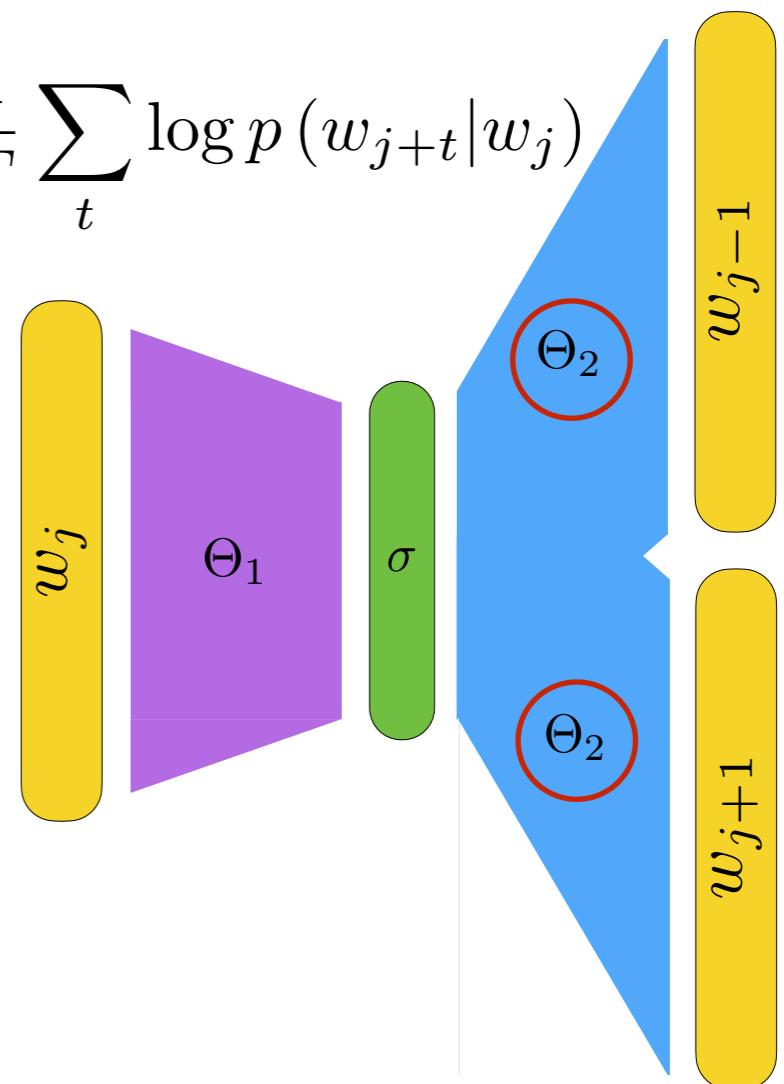
- **online learning** - update weights after **each** case
  - might be useful to update model as new data is obtained
  - subject to fluctuations
- **mini-batch** - update weights after a “**small**” number of cases
  - batches should be balanced
  - if dataset is redundant, the gradient estimated using only a fraction of the data is a good approximation to the full gradient.
- **momentum** - let gradient change the **velocity** of weight change instead of the value directly
- **rmsprop** - divide learning rate for each weight by a **running average** of “recent” gradients
- **learning rate** - vary over the course of the training procedure and use different learning rates for each weight

# SkipGram with Larger Contexts

$$H = -\log p(w_{j+1}|w_j)$$



$$H = -\frac{1}{T} \sum_t \log p(w_{j+t}|w_j)$$

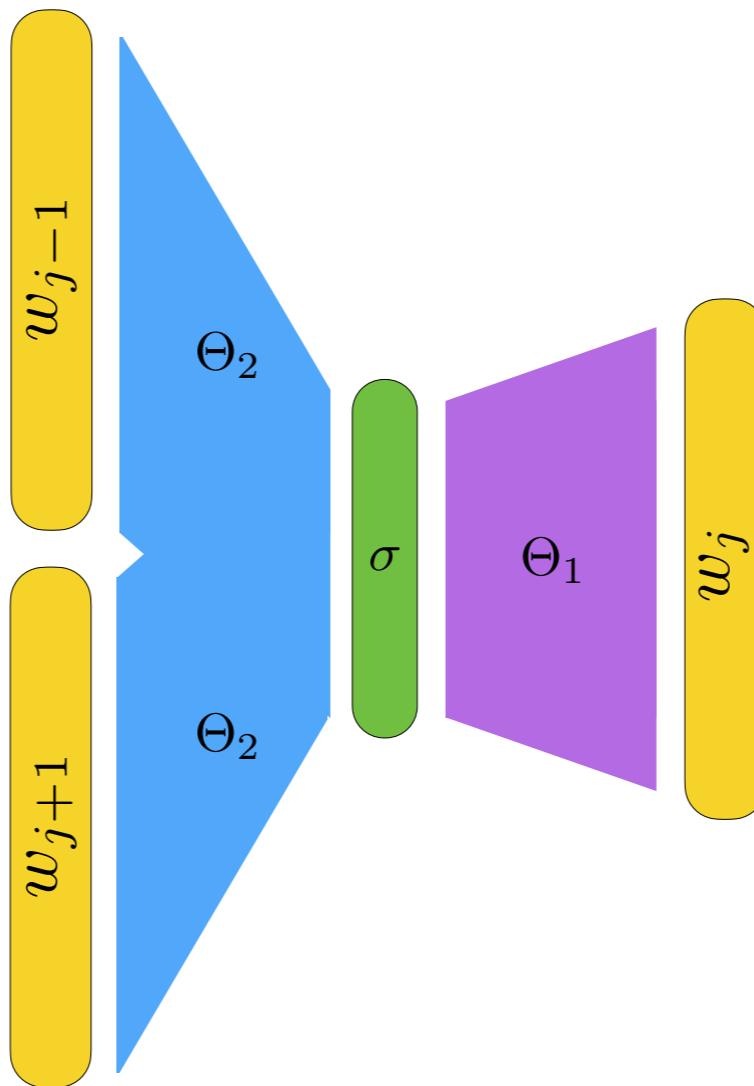


- Use the same  $\Theta_2$  for all context words.
- Use the average of cross entropy.
- word order is not important (the average does not change)
- Can essentially be trained one context word at a time..

# Continuous Bag of Words

---

- The process is essentially the same



# Variations

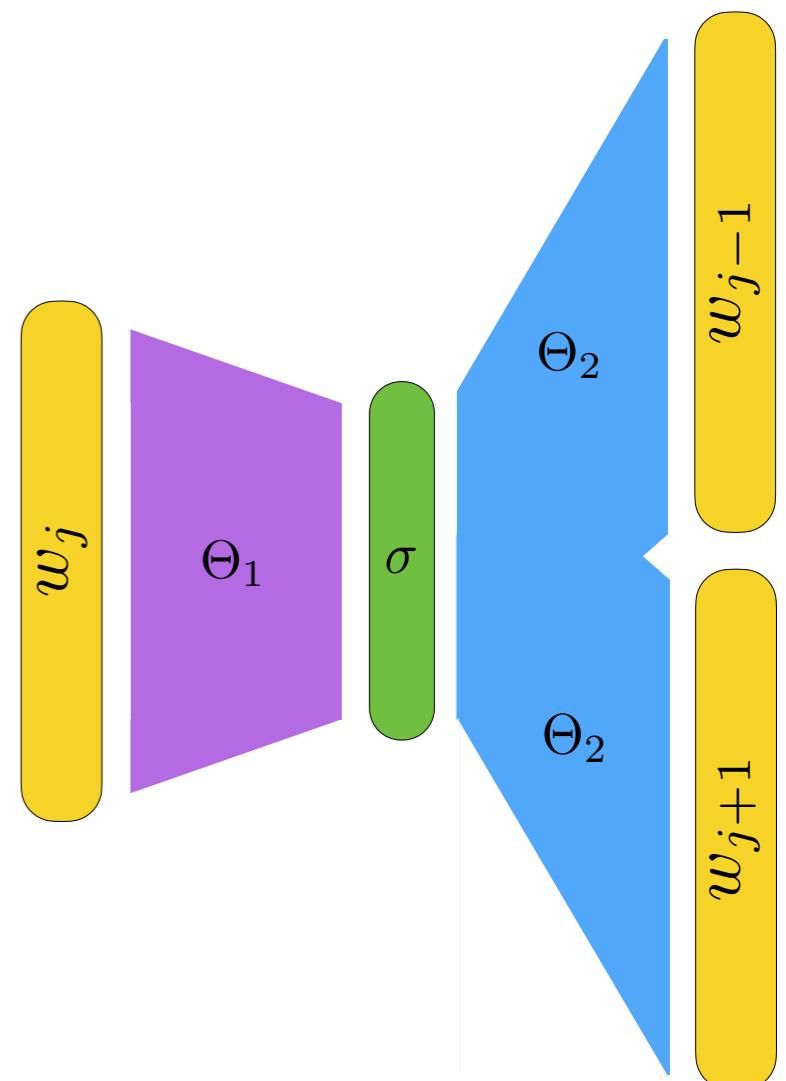
---

- Hierarchical Softmax:
  - Approximate the softmax using a binary tree
  - Reduce the number of calculations per training example from  $V$  to  $\log_2 V$  and increase performance by orders of magnitude.
- Negative Sampling:
  - Under sample the most frequent words by removing them from the text before generating the contexts
  - Similar idea to removing stop-words — very frequent words are less informative.
  - Effectively makes the window larger, increasing the amount of information available for context

# Comments

---

- **word2vec**, even in its original formulation is actually a family of algorithms using various combinations of:
  - Skip-gram, CBOW
  - Hierarchical Softmax, Negative Sampling
- The output of this neural network is deterministic:
  - If two words appear in the same context ("blue" vs "red", for e.g.), they will have similar internal representations in  $\Theta_1$  and  $\Theta_2$
  - $\Theta_1$  and  $\Theta_2$  are vector embeddings of the input words and the context words respectively
- Words that are too rare are also removed.
- The original implementation had a dynamic window size:
  - for each word in the corpus a window size  $k'$  is sampled uniformly between 1 and  $k$



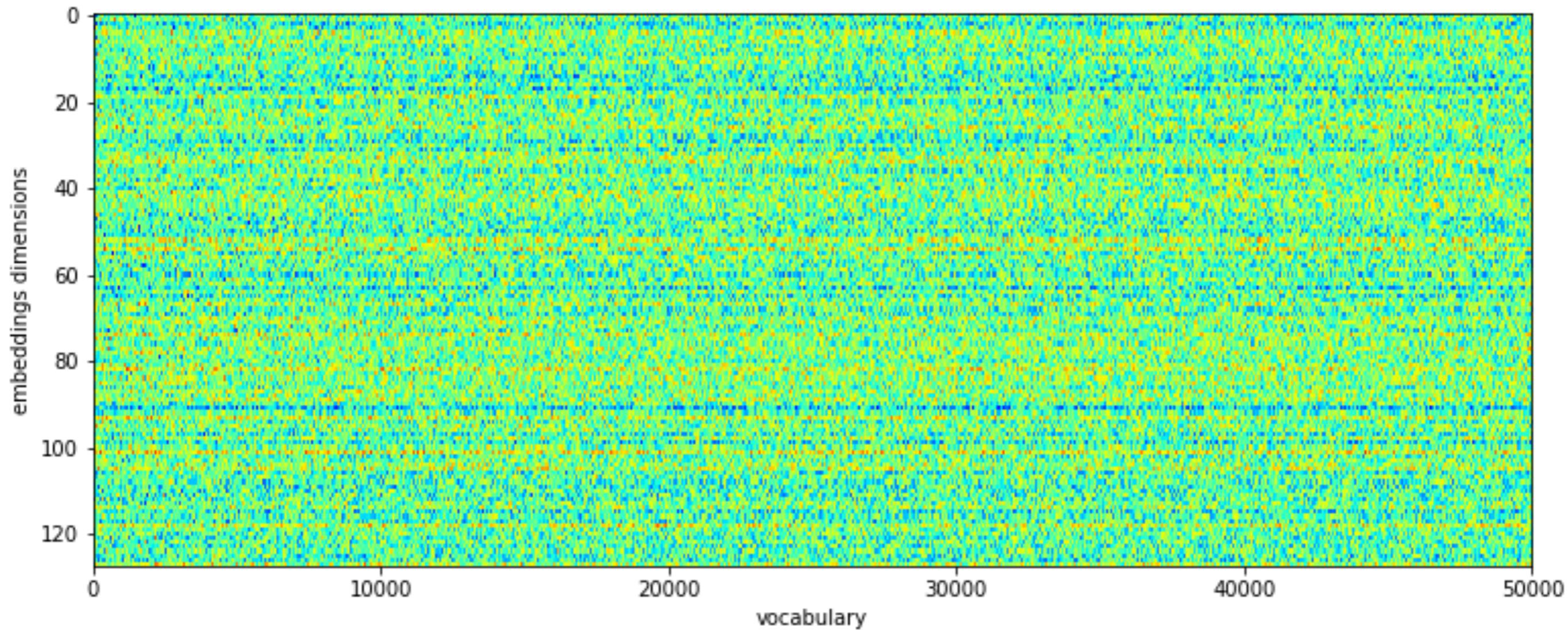
# Online resources

---

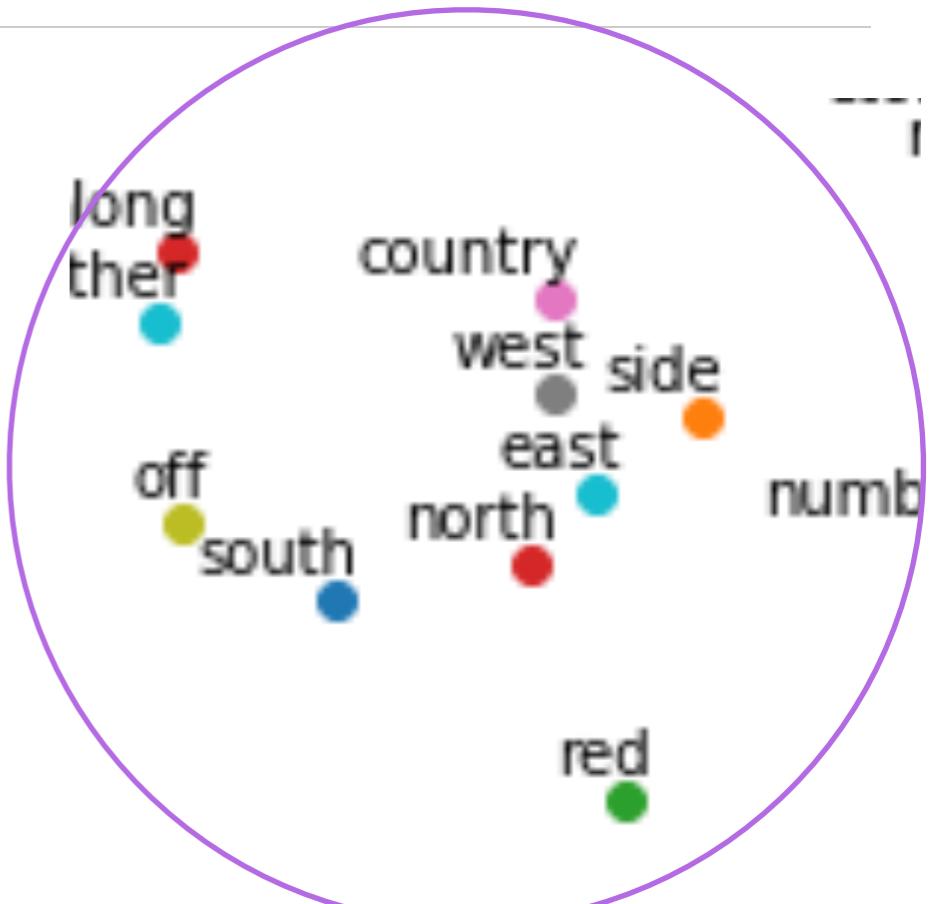
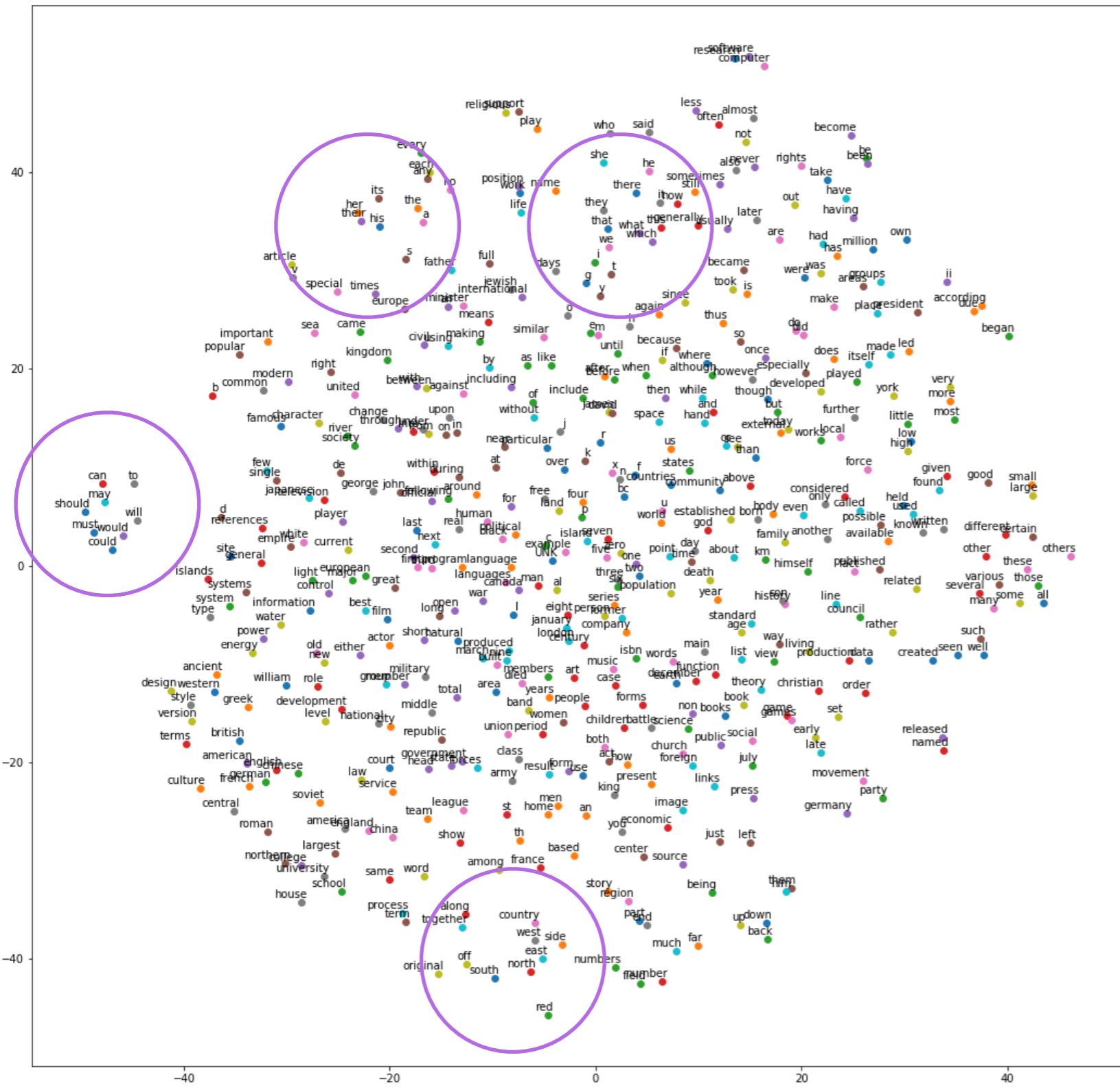
- C - <https://code.google.com/archive/p/word2vec/> (the original one)
- Python/tensorflow - <https://www.tensorflow.org/tutorials/word2vec>
  - Both a minimalist and an efficient versions are available in the tutorial
- Python/gensim - <https://radimrehurek.com/gensim/models/word2vec.html>
- Pretrained embeddings:
  - 30+ languages, <https://github.com/Kyubyong/wordvectors>
  - 100+ languages trained using wikipedia: <https://sites.google.com/site/rmyeid/projects/polyglot>

# Visualization

---



# Visualization





"You shall know a word by the company it keeps"  
(J. R. Firth)

# Analogies

- The embedding of each word is a function of the context it appears in:

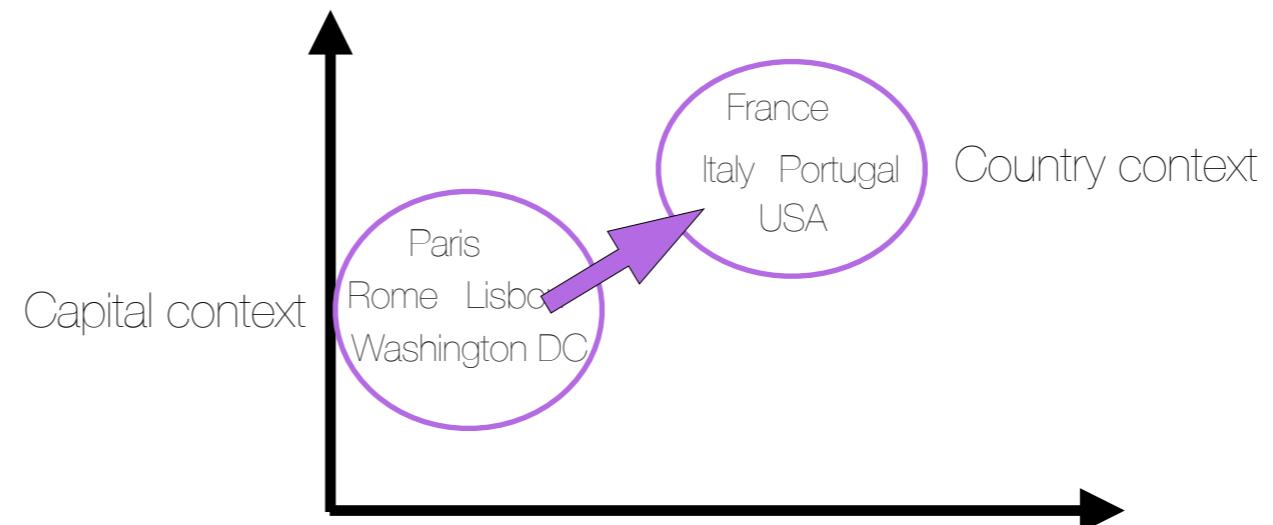
$$\sigma(\text{red}) = f(\text{context}(\text{red}))$$

- words that appear in similar contexts will have similar embeddings:

$$\text{context}(\text{red}) \approx \text{context}(\text{blue}) \implies \sigma(\text{red}) \approx \sigma(\text{blue})$$

- "Distributional hypothesis" in linguistics

Geometrical relations  
between contexts imply  
semantic relations  
between words!



$$\sigma(\text{France}) - \sigma(\text{Paris}) + \sigma(\text{Rome}) = \sigma(\text{Italy})$$

$$\vec{b} - \vec{a} + \vec{c} = \vec{d}$$

# Analogies

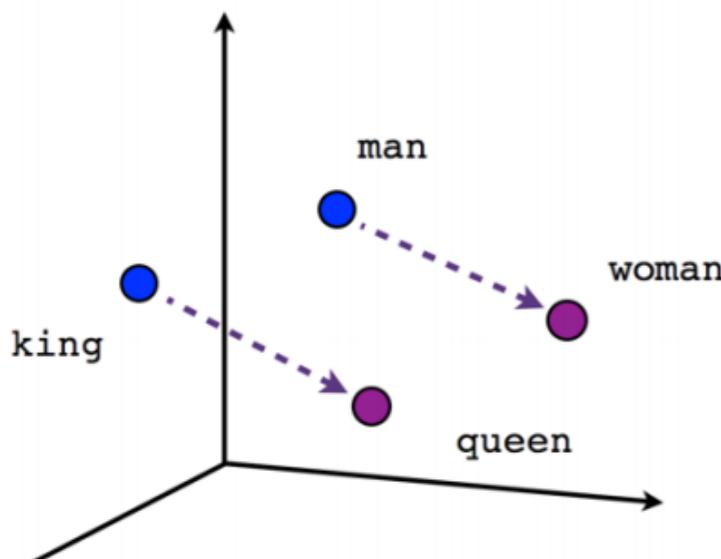
<https://www.tensorflow.org/tutorials/word2vec>

$$\vec{b} - \vec{a} + \vec{c} = \vec{d}$$

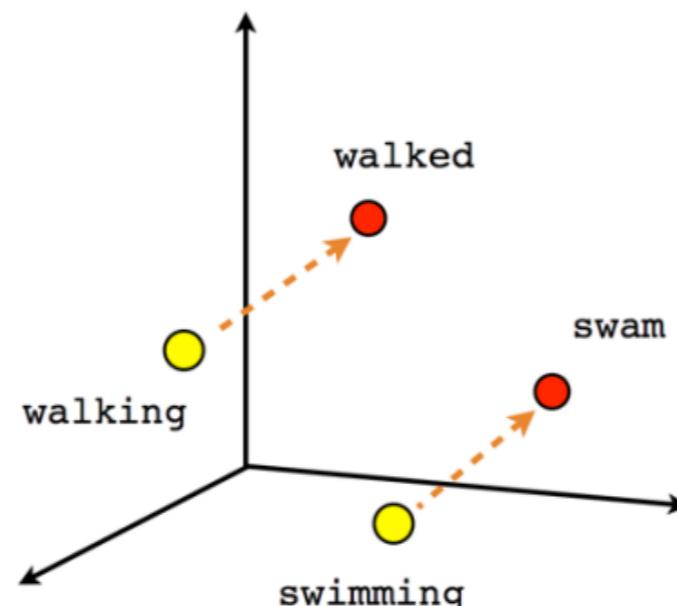
$$d^\dagger = \operatorname{argmax}_x \frac{\left( \vec{b} - \vec{a} + \vec{c} \right)^T}{\left\| \vec{b} - \vec{a} + \vec{c} \right\|} \vec{x}$$

What is the word **d** that is most **similar** to **b** and **c** and most **dissimilar** to **a**?

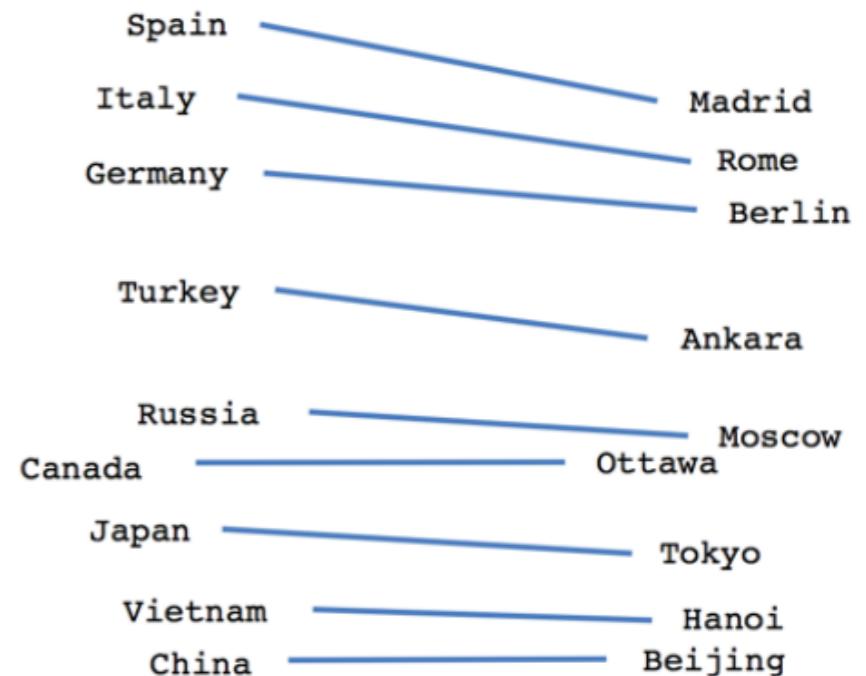
$$d^\dagger \sim \operatorname{argmax}_x \left( \vec{b}^T \vec{x} - \vec{a}^T \vec{x} + \vec{c}^T \vec{x} \right)$$



Male-Female



Verb tense



Country-Capital



# TensorFlow

<https://github.com/bmtgoncalves/word2vec-and-friends/>

## TensorFlow

From Wikipedia, the free encyclopedia

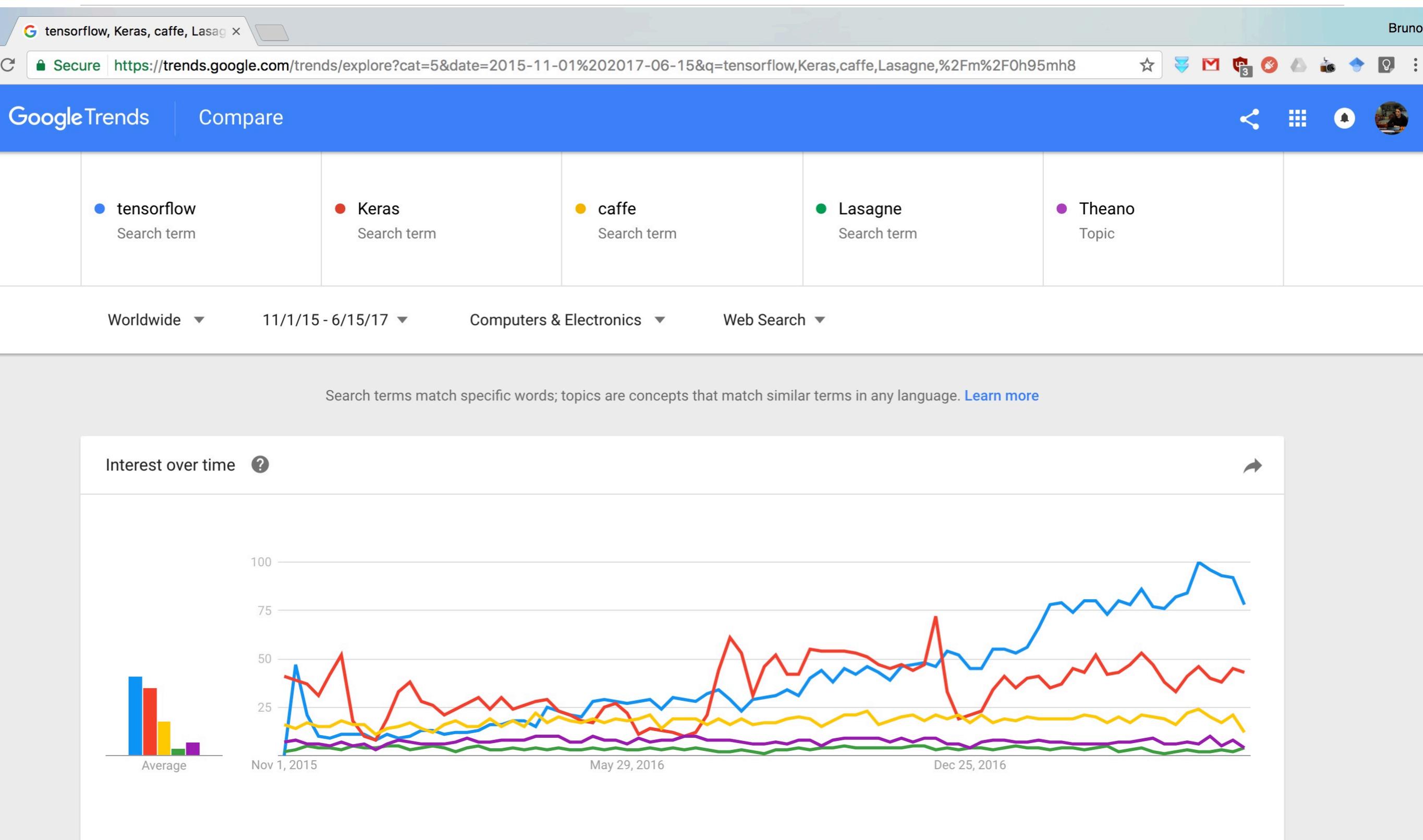
**TensorFlow** is an [open-source software library](#) for [dataflow](#) programming across a range of tasks. It is a symbolic math library, and also used for machine learning applications such as [neural networks](#).<sup>[3]</sup> It is used for both research and production at Google,<sup>[3]:min 0:15/2:17 [4]:p.2 [3]:0:26/2:17</sup> often replacing its closed-source predecessor, DistBelief.

TensorFlow was developed by the [Google Brain](#) team for internal Google use. It was released under the [Apache 2.0 open source license](#) on 9 November 2015.<sup>[1][5]</sup>



TensorFlow

# Tensorflow



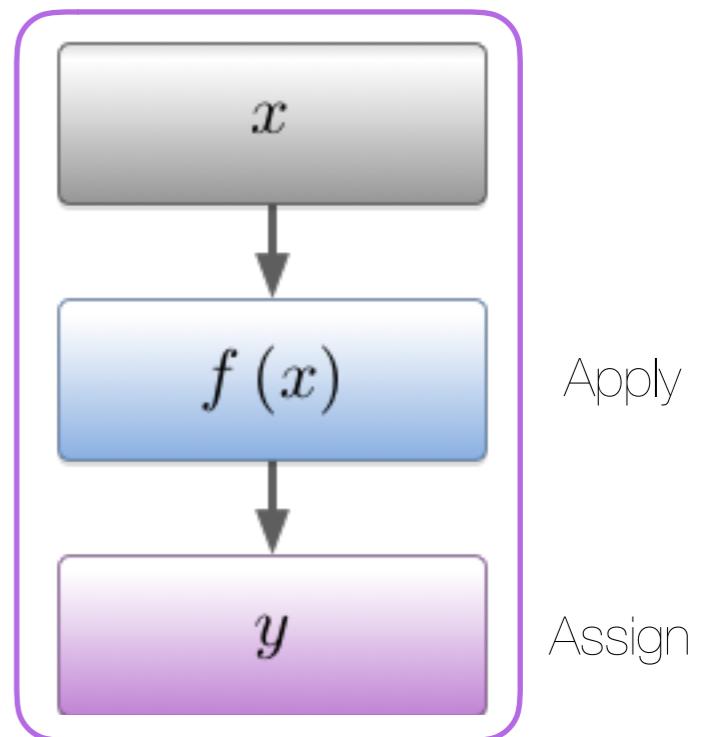
# A diversion... Data Flow

- Let's imagine I want to perform these calculations:

$$y = f(x)$$

$$z = g(y)$$

- for some given  $x$ .
- To calculate  $z$  we must follow a certain sequence of operations.
- Which can be shortened if we are interested in just the value of  $y$
- In **Tensorflow**, this is called a **Computational Graph** and it's the most fundamental concept to understand
- Data flows through the graph from inputs to outputs
- Tensorflow**, is, essentially, a way of defining arbitrary computational graphs in a way that can be automatically distributed and optimized.



# Computational Graphs

- If we use base functions, **Tensorflow** knows how to automatically calculate the respective gradients

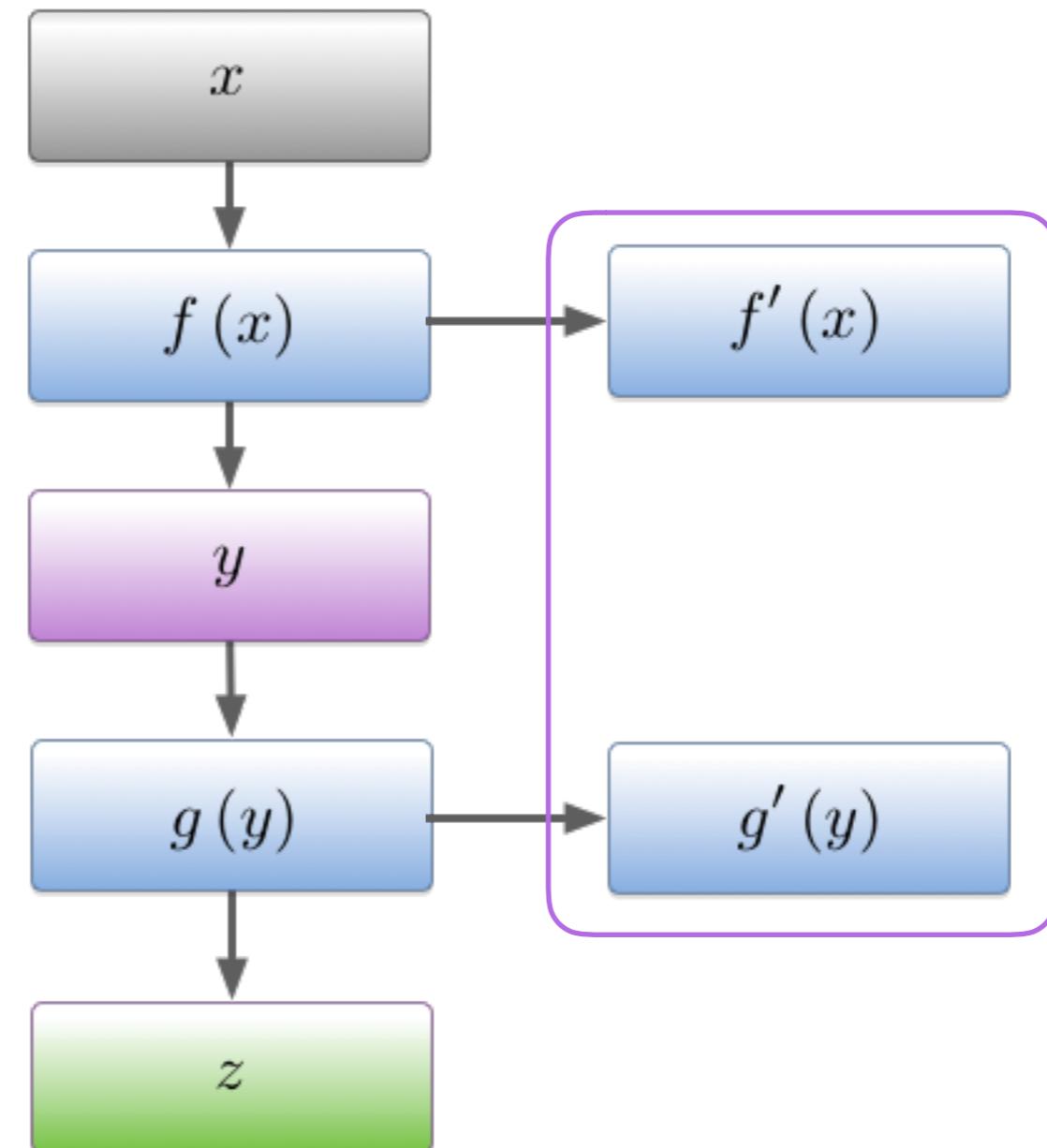
- Automatic BackProp

- Graphs can have multiple outputs

- Predictions

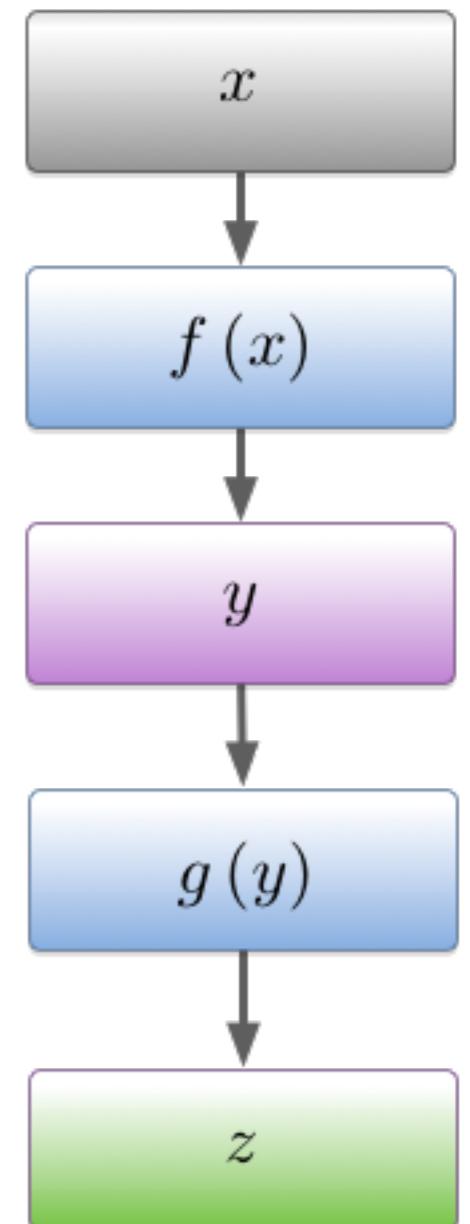
- Cost functions

- etc...



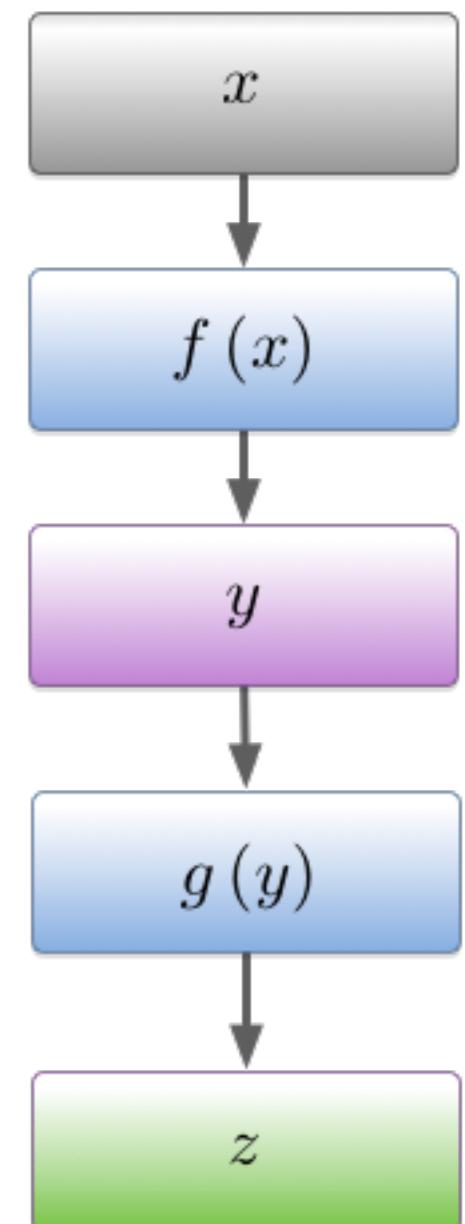
# Sessions

- After we have defined the computational graph, we can start using it to make calculations
- All computations must take place within a "session" that defines the values of all required input values

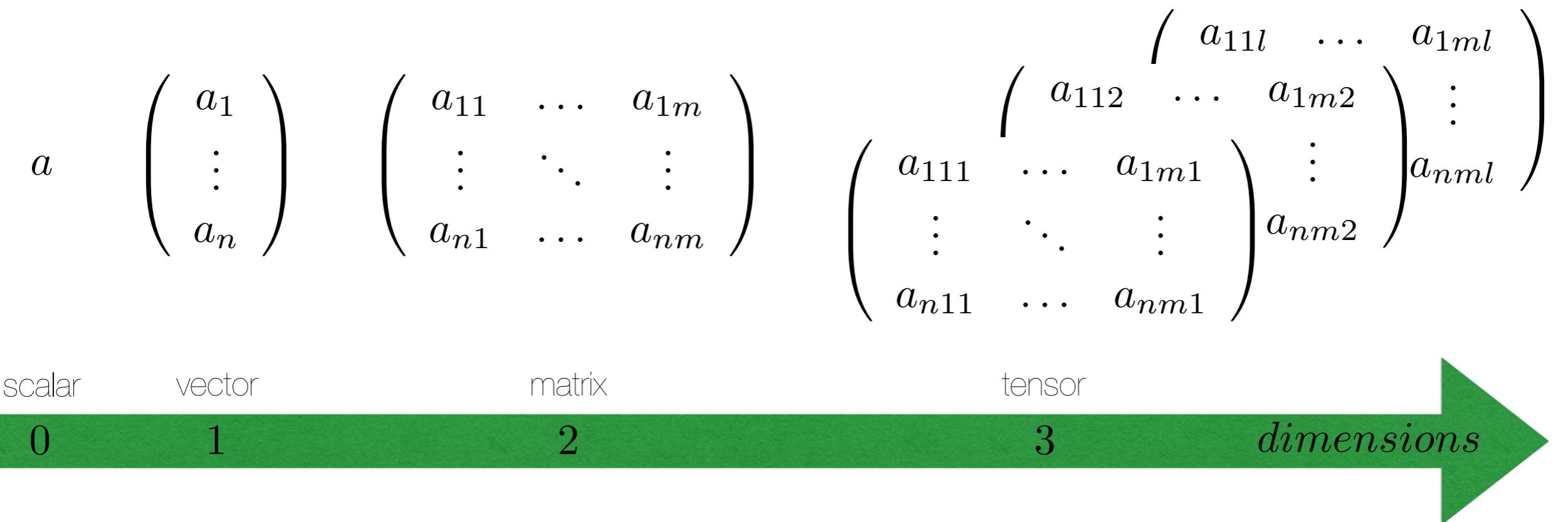


# Sessions

- After we have defined the computational graph, we can start using it to make calculations
- All computations must take place within a "session" that defines the values of all **required** input values
- Which values are required for a specific computation depend on what part of the graph is actually being executed.
- When you request the value of a specific output, **Tensorflow** determines what is the specific **subgraph** that must be executed and what are the required input values.
- For optimization purposes, it can also execute independent parts of the graph in different devices (CPUs, GPUs, TPUs, etc) at the same time.



# Data Flow? Tensor Flow?



Tensors are just  
multidimensional  
arrays of numbers

The number of **dimensions**  
is the number of  
**for loops** required

# Installation

<https://www.tensorflow.org/install/>

## Install TensorFlow

Assuming the prerequisite software is installed on your Mac, take the following steps:

1. Install TensorFlow by invoking **one** of the following commands:

```
$ pip install tensorflow      # Python 2.7; CPU support  
$ pip3 install tensorflow    # Python 3.n; CPU support
```

If the preceding command runs to completion, you should now [validate your installation](#).

# A basic Tensorflow program

$$z = c * (x + y)$$

```

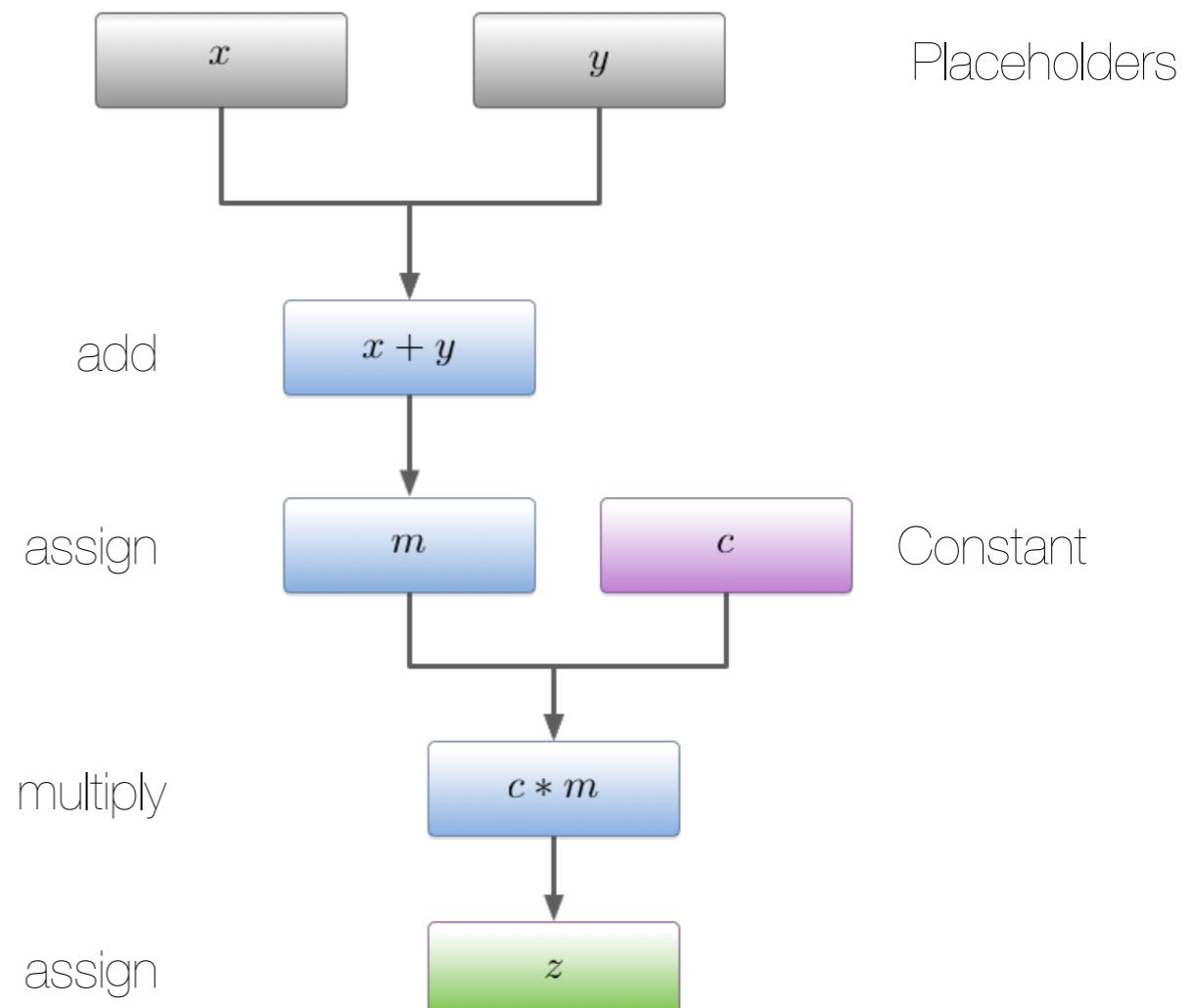
import tensorflow as tf

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
c = tf.constant(3.)

m = tf.add(x, y)
z = tf.multiply(m, c)

with tf.Session() as sess:
    output = sess.run(z, feed_dict={x: 1., y: 2.})
    print("Output value is:", output)

```



# A basic Tensorflow program

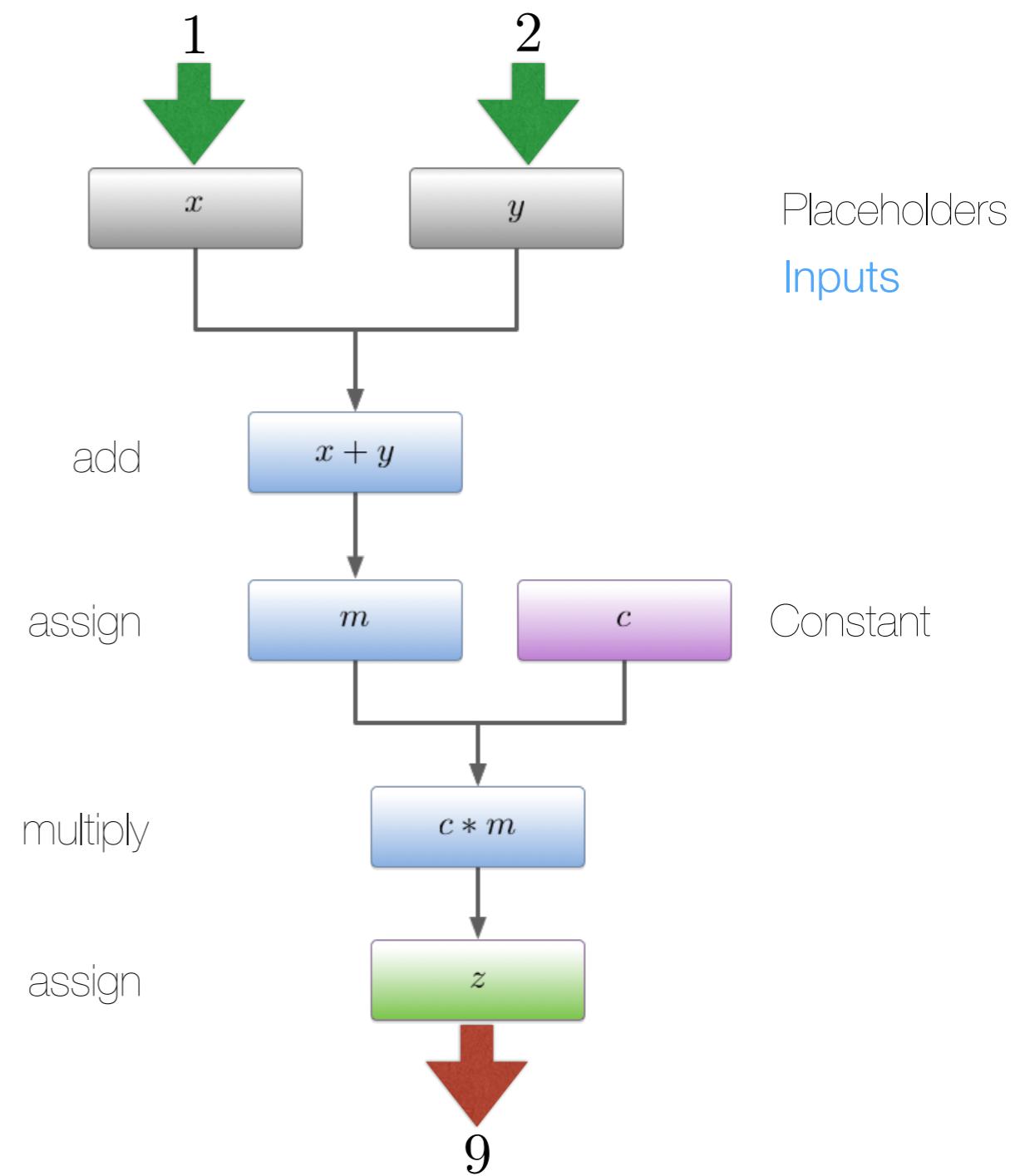
```

import tensorflow as tf

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
c = tf.constant(3.)

m = tf.add(x, y)
z = tf.multiply(m, c)

with tf.Session() as sess:
    output = sess.run(z, feed_dict={x: 1., y: 2.})
    print("Output value is:", output)
  
```



# Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

learning_rate = 0.01
N = 100
N_steps = 300

# Training Data
train_X = np.linspace(-10, 10, N)
train_Y = 2*train_X + 3 + np.random.random(N)

# Computational Graph
X = tf.placeholder("float")
Y = tf.placeholder("float")

W = tf.Variable(np.random.randn(), name="weight")
b = tf.Variable(np.random.randn(), name="bias")
y = tf.add(tf.multiply(X, W), b)

cost = tf.reduce_mean(tf.pow(y-Y, 2))
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for step in range(N_steps):
        sess.run(optimizer, feed_dict={X: train_X, Y: train_Y})
        cost_val, W_val, b_val = sess.run([cost, W, b], feed_dict={X: train_X, Y:train_Y})
        print("step", step, "cost", cost_val, "w", W_val, "b", b_val)
```

# Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

learning_rate = 0.01
N = 100
N_steps = 300

# Training Data
train_X = np.linspace(-10, 10, N)
train_Y = 2*train_X + 3 + np.random.random(N)

# Computational Graph
X = tf.placeholder("float")
Y = tf.placeholder("float")
W = tf.Variable(np.random.randn(), name="weight")
b = tf.Variable(np.random.randn(), name="bias")
y = tf.add(tf.multiply(X, W), b)
cost = tf.reduce_mean(tf.pow(y-Y, 2))
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for step in range(N_steps):
        sess.run(optimizer, feed_dict={X: train_X, Y: train_Y})
        cost_val, W_val, b_val = sess.run([cost, W, b], feed_dict={X: train_X, Y:train_Y})
        print("step", step, "cost", cost_val, "w", W_val, "b", b_val)
```

$$y = W * x + b$$

$$cost = \frac{1}{N} \sum_i (y_i - Y_i)^2$$

# Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

learning_rate = 0.01
N = 100
N_steps = 300

# Training Data
train_X = np.linspace(-10, 10, N)
train_Y = 2*train_X + 3 + np.random.random(N)

# Computational Graph
X = tf.placeholder("float")
Y = tf.placeholder("float")

W = tf.Variable(np.random.randn(), name="weight")
b = tf.Variable(np.random.randn(), name="bias")
y = tf.add(tf.multiply(X, W), b)

cost = tf.reduce_mean(tf.pow(y-Y, 2))
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for step in range(N_steps):
        sess.run(optimizer, feed_dict={X: train_X, Y: train_Y})
        cost_val, W_val, b_val = sess.run([cost, W, b], feed_dict={X: train_X, Y:train_Y})
        print("step", step, "cost", cost_val, "w", W_val, "b", b_val)
```

$$y = W * x + b$$

$$cost = \frac{1}{N} \sum_i (y_i - Y_i)^2$$

# Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

learning_rate = 0.01
N = 100
N_steps = 300

# Training Data
train_X = np.linspace(-10, 10, N)
train_Y = 2*train_X + 3 + np.random.random(N)

# Computational Graph
X = tf.placeholder("float")
Y = tf.placeholder("float")

W = tf.Variable(np.random.randn(), name="weight")
b = tf.Variable(np.random.randn(), name="bias")
y = tf.add(tf.multiply(X, W), b)

cost = tf.reduce_mean(tf.pow(y-Y, 2))
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
init = tf.global_variables_initializer()

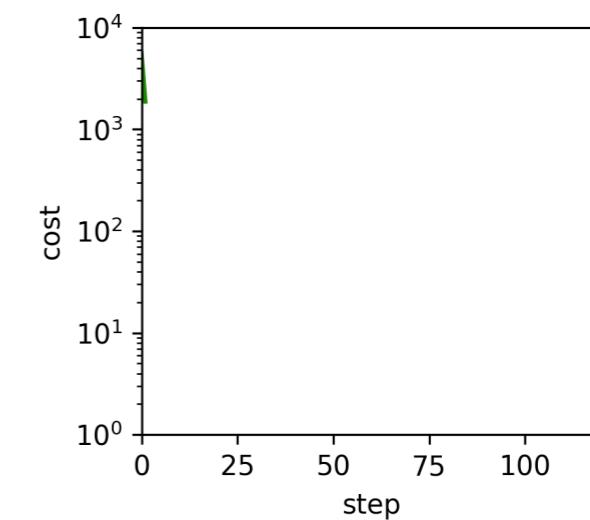
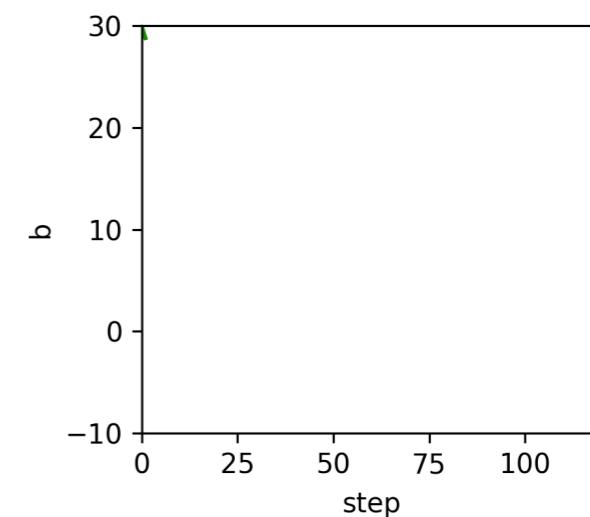
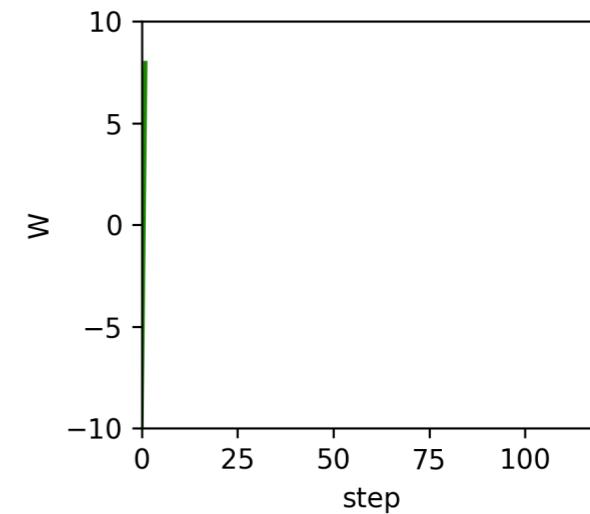
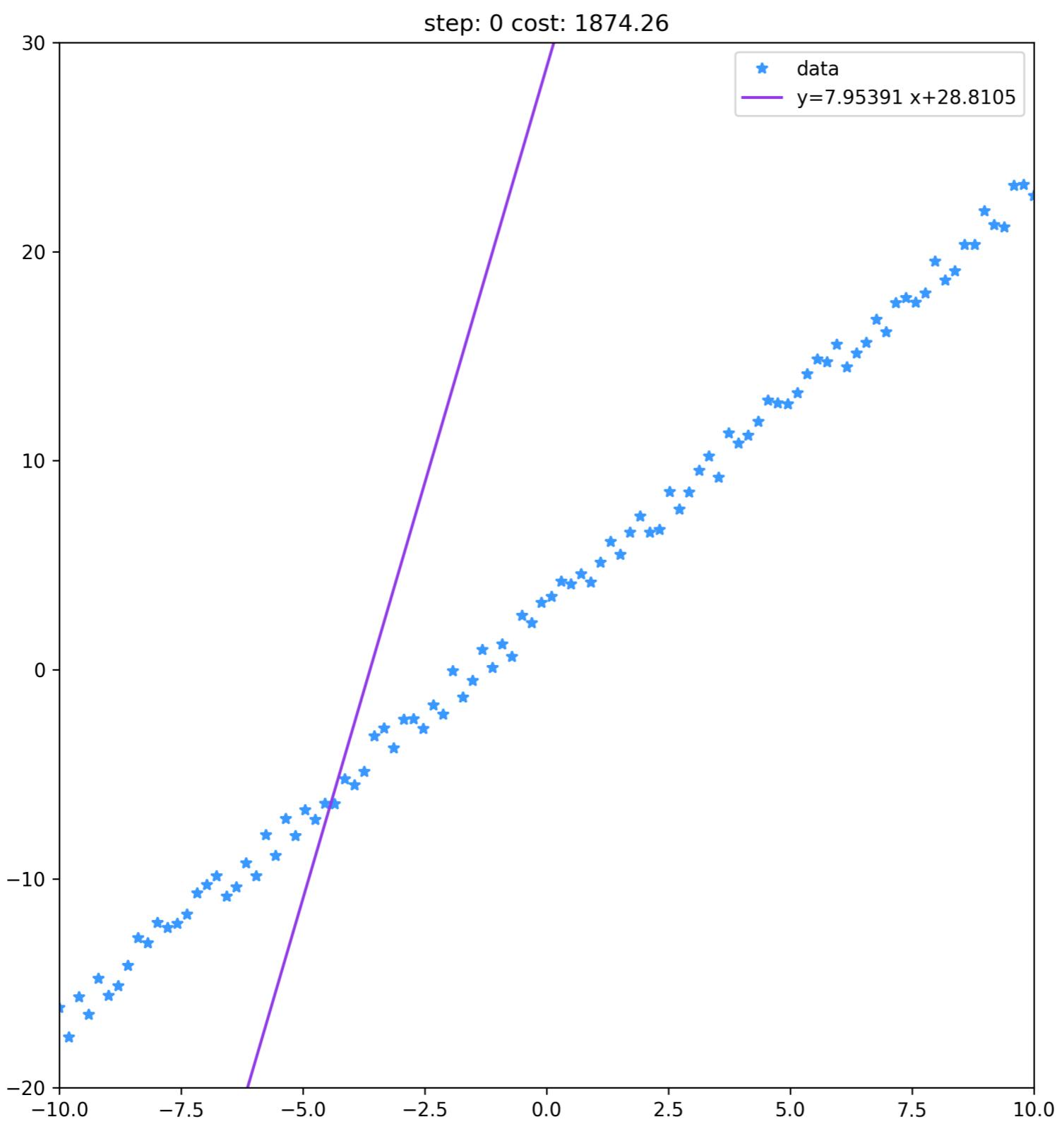
with tf.Session() as sess:
    sess.run(init)

    for step in range(N_steps):
        sess.run(optimizer, feed_dict={X: train_X, Y: train_Y})
        cost_val, W_val, b_val = sess.run([cost, W, b], feed_dict={X: train_X, Y:train_Y})
        print("step", step, "cost", cost_val, "w", W_val, "b", b_val)
```

$$y = W * x + b$$

$$cost = \frac{1}{N} \sum_i (y_i - Y_i)^2$$

# Linear Regression



Jupyter Notebook

## Statistically Significant Detection of Linguistic Change

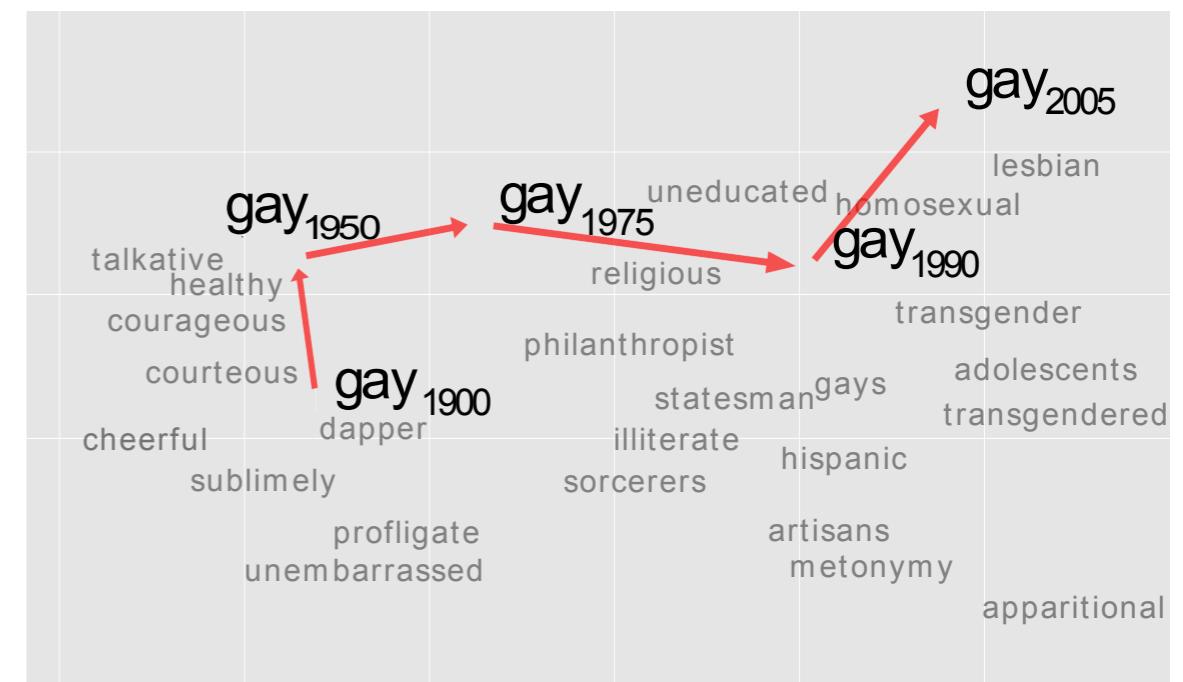
Vivek Kulkarni  
Stony Brook University, USA  
vvkulkarni@cs.stonybrook.edu

Bryan Perozzi  
Stony Brook University, USA  
bperozzi@cs.stonybrook.edu

Rami Al-Rfou  
Stony Brook University, USA  
ralrfou@cs.stonybrook.edu

Steven Skiena  
Stony Brook University, USA  
skiena@cs.stonybrook.edu

- Train word embeddings for different years using Google Books
- Independently trained embeddings differ by an arbitrary rotation
- Align the different embeddings for different years
- Track the way in which the meaning of words shifted over time!



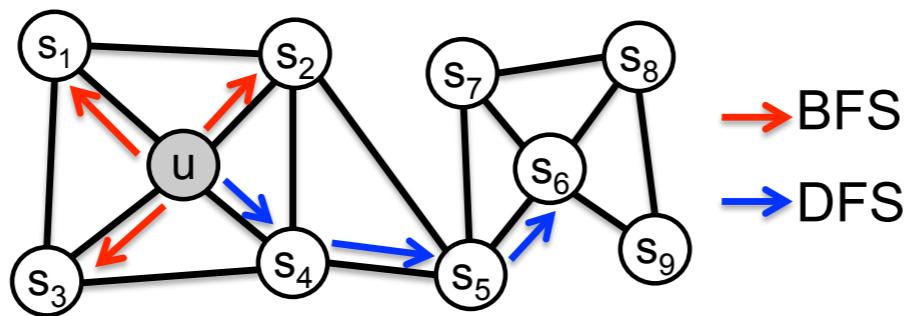
## node2vec: Scalable Feature Learning for Networks

Aditya Grover  
Stanford University  
adityag@cs.stanford.edu

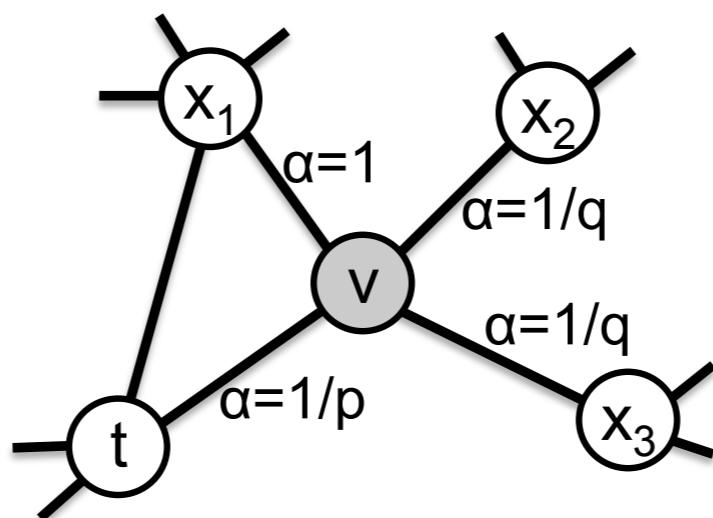
Jure Leskovec  
Stanford University  
jure@cs.stanford.edu

- You can generate a graph out of a sequence of words by assigning a node to each word and connecting the words within their neighbors through edges.
- With this representation, a piece of text is a walk through the network. Then perhaps we can invert the process? Use walks through a network to generate a sequence of nodes that can be used to train node embeddings?
- node embeddings should capture **features** of the network structure and allow for detection of similarities between nodes.

- The features depends strongly on the way in which the network is traversed
- Generate the contexts for each node using Breath First Search and Depth First Search



- Perform a biased Random Walk



- BFS - Explores only limited neighborhoods. Suitable for structural equivalences
- DFS - Freely explores neighborhoods and covers homophiles communities
- By modifying the parameter of the model it can interpolate between the BFS and DFS extremes

## dna2vec: Consistent vector representations of variable-length k-mers

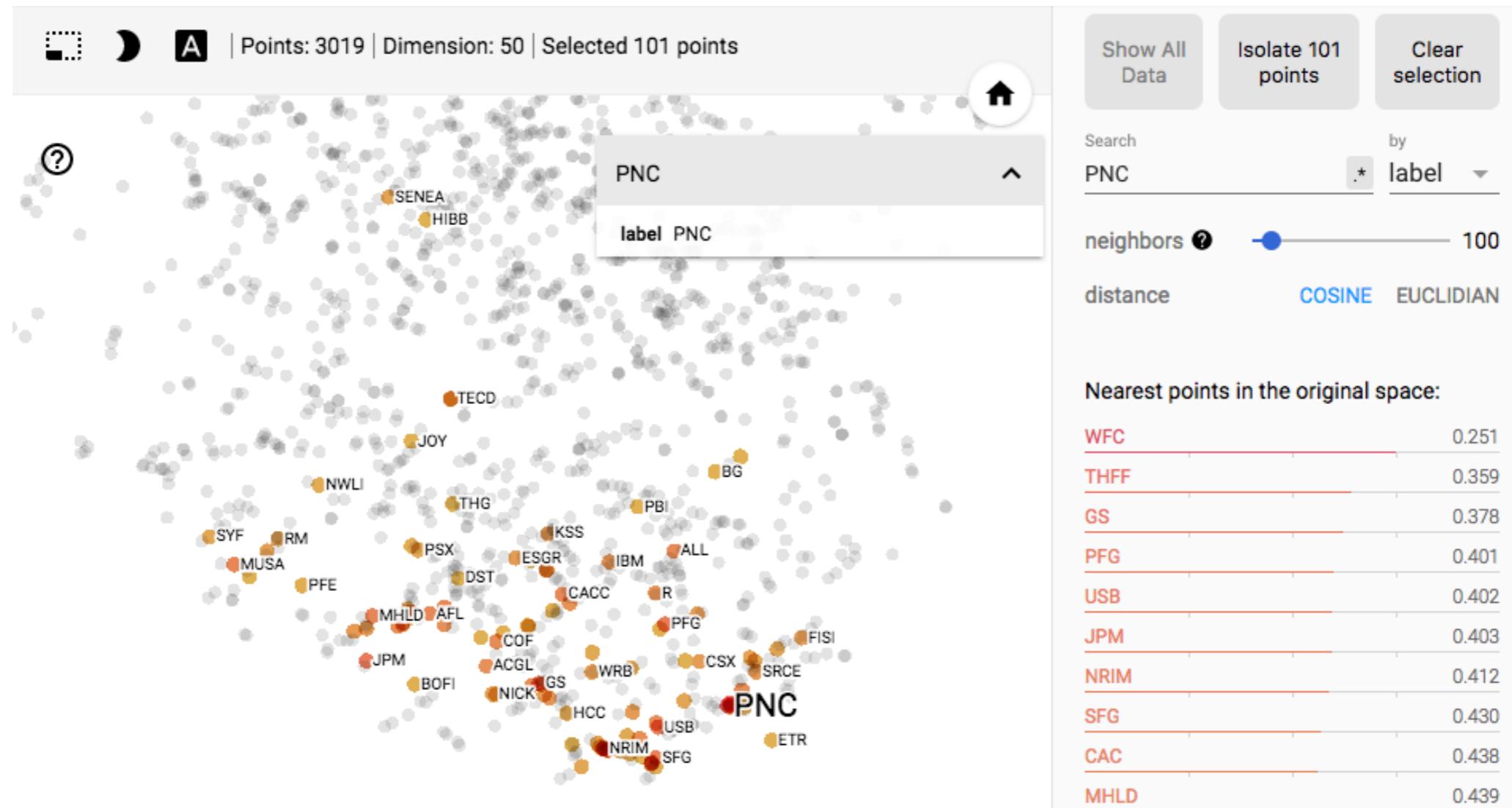
Patrick Ng  
ppn3@cs.cornell.edu

- Separate the genome into long non-overlapping DNA fragments.
- Convert long DNA fragments into overlapping **variable** length k-mers
- Train embeddings of each k-mer using **Gensim** implementation of SkipGram.
  - Summing embeddings is related to concatenating **k-mers**
  - Cosine similarity of k-mer embeddings reproduces a biologically motivated similarity score (Needleman-Wunsch) that is used to align nucleotides

# stock2vec

<https://medium.com/towards-data-science/stock2vec-from-ml-to-p-e-2e6ba407c24>

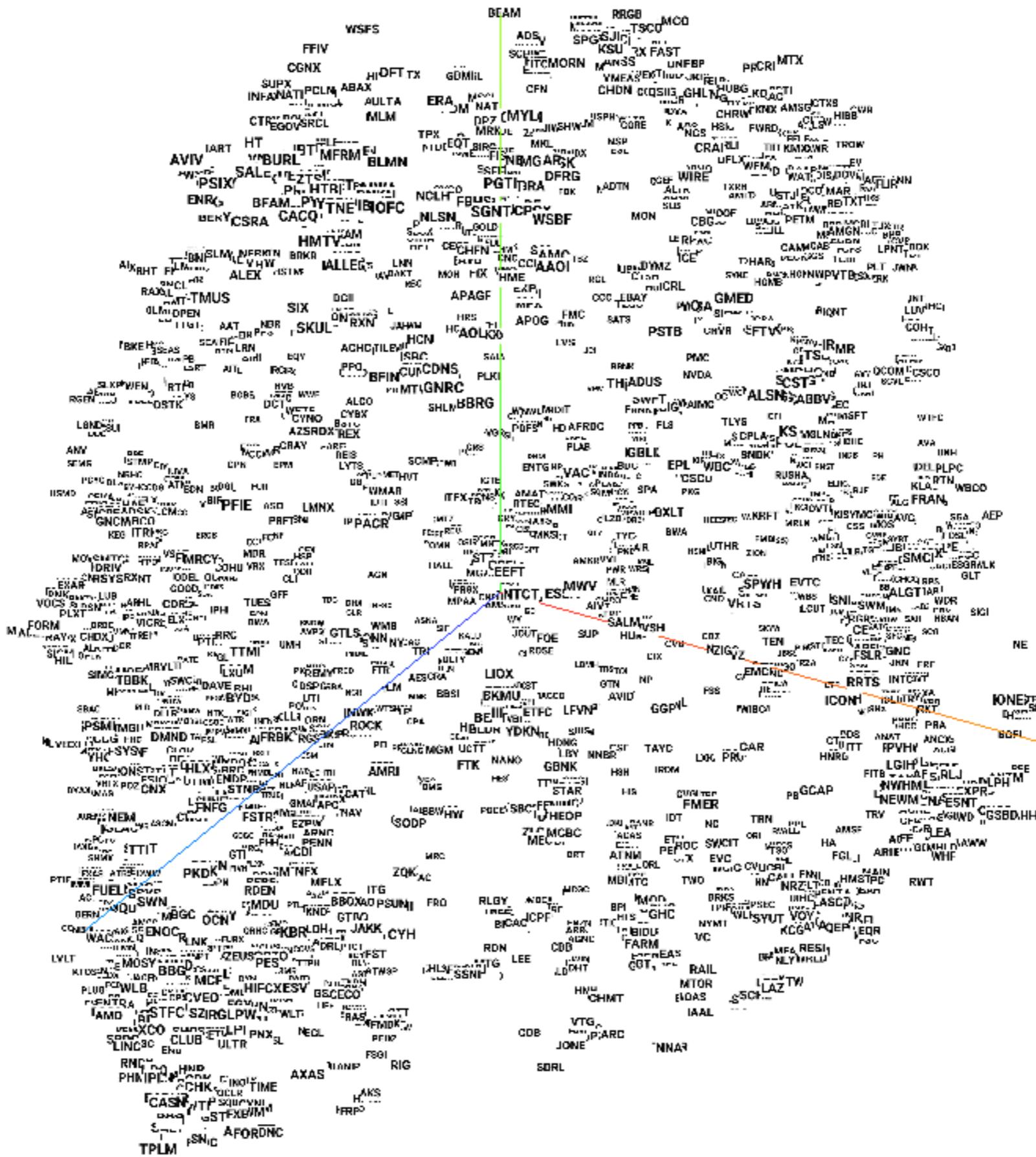
- Apply word2vec to the 40 years of stock market data
- Identify significant semantic similarities between companies working in the same area





stock2

ml-to-p-e-2e6ba407c24



# Deep Learning...

---

