Vector Representations of Words with TensorFlow

Harvey Alférez, Ph.D.

Distributed Representations of Words and Phrases and their Compositionality

Tomas Mikolov
Google Inc.
Mountain View
mikolov@google.com

Ilya Sutskever
Google Inc.
Mountain View
ilyasu@google.com

Kai Chen
Google Inc.
Mountain View
kai@google.com

Greg Corrado
Google Inc.
Mountain View
gcorrado@google.com

Jeffrey Dean
Google Inc.
Mountain View
jeff@google.com

Abstract

The recently introduced continuous Skip-gram model is an efficient method for learning high-quality distributed vector representations that capture a large num-

Introduction

- In this tutorial we look at the word2vec model by Mikolov et al.

- This model is used for learning vector representations of words, called "word embeddings”.

- https://www.tensorflow.org/tutorials/word2vec

- https://www.tensorflow.org/code/tensorflow/examples/tutorials/word2vec/word2vec_basic.py

Highlights

- This tutorial is meant to highlight the interesting, substantive parts of building a **word2vec** model in TensorFlow.

- We start by giving the motivation for why we would want to represent words as vectors.

- We look at the intuition behind the model and how it is trained.

- We also show a simple implementation of the model in TensorFlow.

Motivation: Why Learn Word Embeddings?

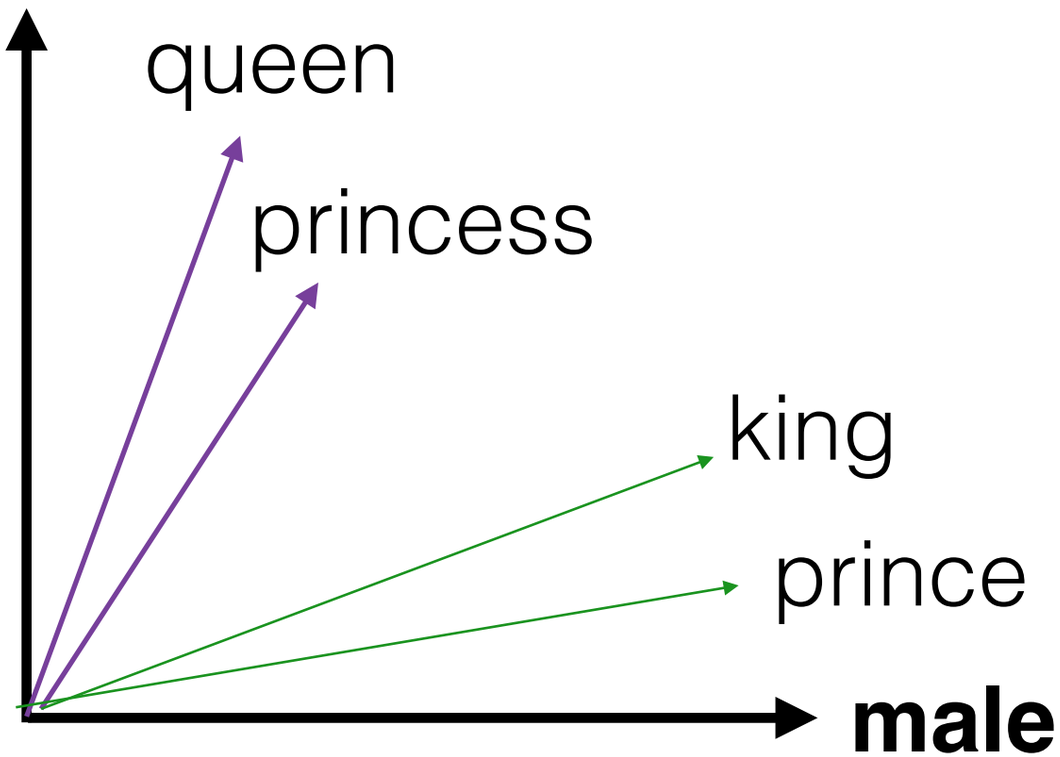
- Natural language processing systems **traditionally** treat **words** as **discrete atomic symbols**.
  - 'cat' may be represented as Id537 and 'dog' as Id143.
    - These encodings are arbitrary, and **provide no useful information to the system regarding the relationships that may exist between the individual symbols.**

Motivation: Why Learn Word Embeddings?

- **Vector space models (VSMs) represent (embed) words** in a **continuous vector space** where semantically similar words are mapped to nearby points ('are embedded nearby each other').

- Words that appear in the **same contexts share semantic meaning.**

Motivation: Why Learn Word Embeddings?



- Represent words as vectors.
- Learn from a lot of data
- Use machine learning (e.g., deep learning algorithms)
- A word vector is built using surrounding words

Motivation: Why Learn Word Embeddings?

- **Word2vec** is a particularly computationally-efficient **predictive model** (*directly tries to predict a word from its neighbors in terms of learned small, dense embedding vectors*) for learning word embeddings from raw text. It comes in two flavors:

  - **The Continuous Bag-of-Words model (CBOW)**

    - **Predicts target words** (e.g. 'mat') **from source context words** ('the cat sits on the')

  - **The Skip-Gram model**

    - It does the **inverse** and predicts source context-words from the target words.

Motivation: Why Learn Word Embeddings?

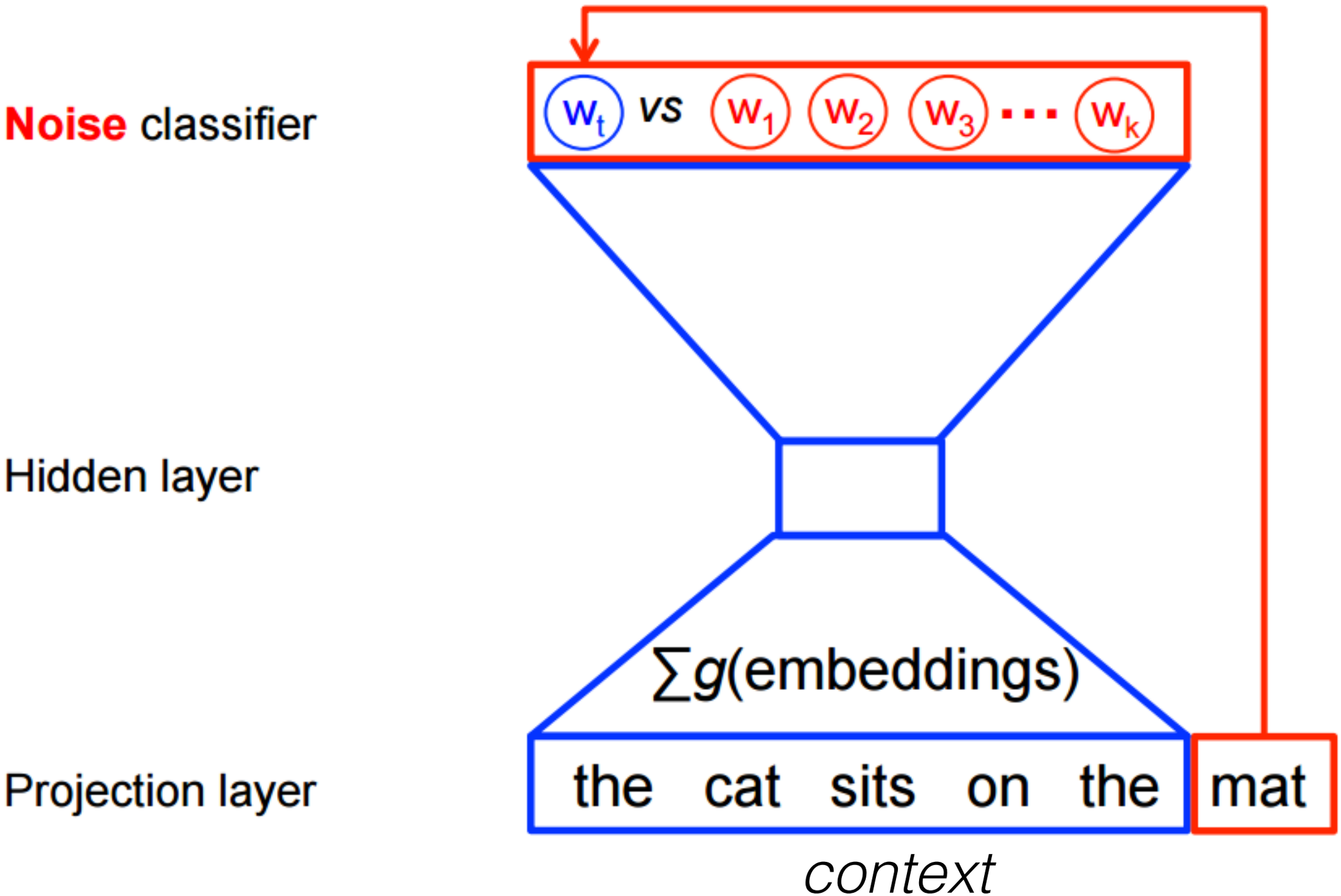
- The **CBOW** and **skip-gram** models are trained using a **binary classification objective** (logistic regression) to discriminate the real target words w_t from k imaginary (noise) words $\overline{w}$, in the same context.

- We illustrate this below for a **CBOW** model.

  - **Predicts target words** (e.g. 'mat') **from source context words** ('the cat sits on the')

  - For **skip-gram** the direction is simply inverted.

Motivation: Why Learn Word Embeddings?



The Skip-gram Model

- As an example, let's consider the dataset
  - *the quick brown fox jumped over the lazy dog*
- Let's define '**context**' as the window of words to the left and to the right of a target word. Using a window size of 1, we then have the dataset:
  - ([the, brown], quick), ([quick, fox], brown), ([brown, jumped], fox), ...
    - **([context], target) pairs.**

The Skip-gram Model

- Recall that **skip-gram** inverts contexts and targets, and tries to predict each context word from its target word, so the task becomes to predict 'the' and 'brown' from 'quick', 'quick' and 'fox' from 'brown', etc. Therefore our dataset becomes:

- (quick, the), (quick, brown), (brown, quick), (brown, fox), …

  - **of (input, output) pairs**

The Skip-gram Model

- **Objective**: 'move' the embedding vectors around for each word until the model is successful at discriminating real words from noise words.

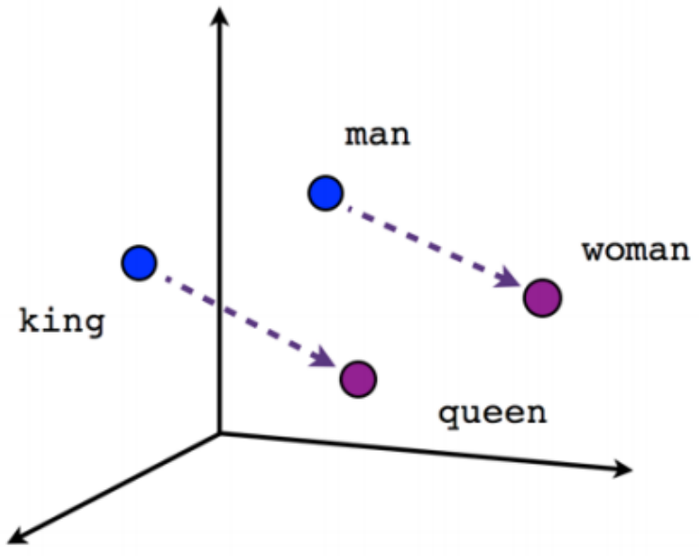
- **the, quick** vs. **the, sheep (sheep = noise)**

The Skip-gram Model

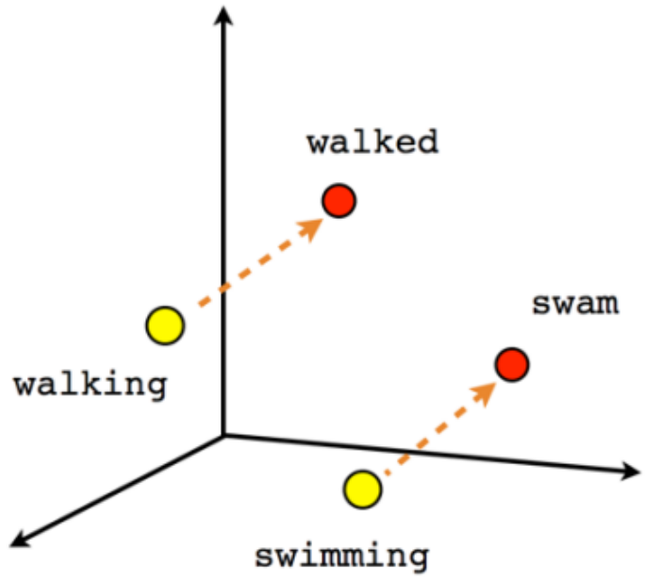
- We can **visualize** the learned vectors by projecting them down to 2 dimensions.

- The vectors capture some general, and in fact quite useful, **semantic information** about **words** and their **relationships** to one another.

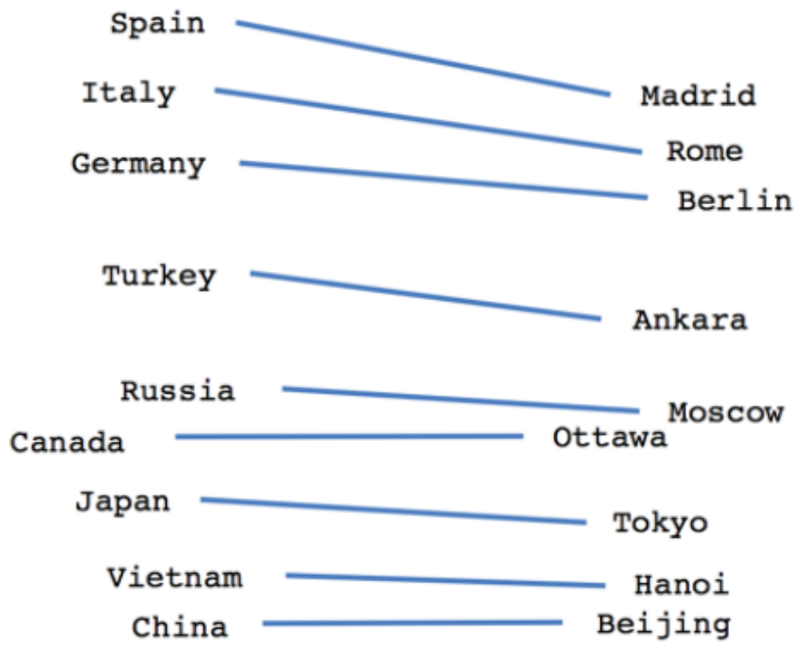
The Skip-gram Model



Male-Female

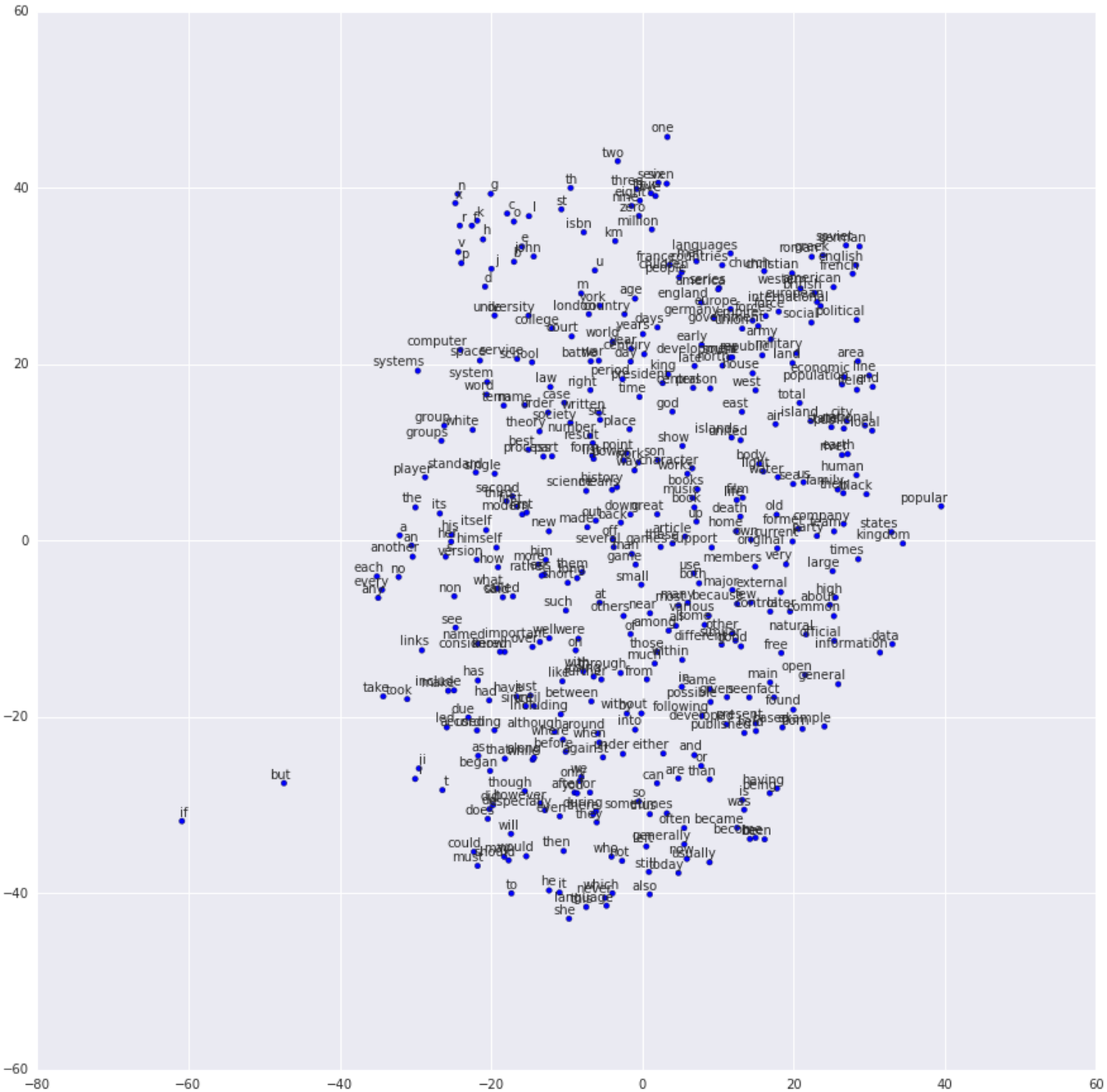


Verb tense



Country-Capital

Visualizing the Learned Embedding



Building the Graph

- Let's define our **embedding matrix**.

- This is just a big random matrix to start. We'll initialize the values to be uniform in the unit cube.

```
embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
```

Building the Graph

- The noise-contrastive estimation loss is defined in terms of a logistic regression model.
- For this, we need to define the weights and biases for each word in the vocabulary (also called the output weights as opposed to the input embeddings). So let's define that.

```
nce_weights = tf.Variable(tf.truncated_normal([vocabulary_size, embedding_size], stddev=1.0 / math.sqrt(embedding_size)))
nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
```

Building the Graph

- The skip-gram model takes two inputs. One is a batch full of integers representing the source context words, the other is for the target words.

```
# Placeholders for inputs
train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
```

- Now what we need to do is look up the vector for each of the source words in the batch. TensorFlow has handy helpers that make this easy.

```
embed = tf.nn.embedding_lookup(embeddings, train_inputs)
```

Building the Graph

- Ok, now that we have the embeddings for each word, we'd like to try to predict the target word using the noise-contrastive training objective.

```
# Compute the NCE loss, using a sample of the negative labels each time.
loss = tf.reduce_mean(tf.nn.nce_loss(nce_weights, nce_biases, embed, train_labels,
num_sampled, vocabulary_size))
```

- Now that we have a loss node, we need to add the nodes required to compute gradients and update the parameters, etc. For this we will use stochastic gradient descent, and TensorFlow has handy helpers to make this easy as well.

```
# We use the SGD optimizer.
optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0).minimize(loss)
```

Training the Model

- Training the model is then as simple as using a feed_dict to push data into the placeholders and calling tf.Session.run with this new data in a loop.

```
for inputs, labels in generate_batch(...):
  feed_dict = {training_inputs: inputs, training_labels: labels}
  _, cur_loss = session.run([optimizer, loss], feed_dict=feed_dict)
```