Lecture note 4: Eager execution and interface

CS 20: TensorFlow for Deep Learning Research
cs20.stanford.edu
Prepared by Chip Huyen and Akshay Agrawal
Contact: chiphuyen@cs.stanford.edu, akshayka@{cs.stanford.edu, google.com}

Up until this point, we've implemented two simple models in TensorFlow: linear regression to predict life expectancy from birth rate , and logistic regression to do an Optical Character Recognition task on the MNIST dataset. We've learned that a TensorFlow program often has two phases: assembling the computation graph and executing that graph. But what if you could execute TensorFlow operations imperatively, directly from Python? This can make debugging our TensorFlow models a lot less intimidating.

In this lecture, we introduce eager execution, rewriting our linear regression model with eager. We will also cover word2vec and the concept of a model base.

Eager execution

Eager execution is (1) a NumPy-like library for numerical computation with support for GPU acceleration and automatic differentiation, and (2) a flexible platform for machine learning research and experimentation. It's available as `tf.contrib.eager`, starting with version 1.50 of TensorFlow.

- Motivation:
  - TensorFlow today: Construct a graph and execute it.
    - This is *declarative* programming. Its benefits include performance and easy translation to other platforms; drawbacks include that declarative programming is non-Pythonic and difficult to debug. Maybe include a
  - What if you could execute operations directly?
    - Eager execution offers just that: it is an *imperative* front-end to TensorFlow.
- Key advantages: Eager execution ...
  - is compatible with Python debugging tools
    - pdb.set_trace() to your heart's content!
  - provides immediate error reporting
  - permits use of Python data structures
    - e.g., for structured input
  - enables you to use and differentiate through Python control flow
- Enabling eager execution requires two lines of code

```
import tensorflow as tf
import tensorflow.contrib.eager as tfe
tfe.enable_eager_execution() # Call this at program start-up
```

and lets you write code that you can easily execute in a REPL, like this

```
x = [[2.]]  # No need for placeholders!
m = tf.matmul(x, x)

print(m)  # No sessions!
# tf.Tensor([[4.]], shape=(1, 1), dtype=float32)
```

For more details, check out lecture slides 04.

Word2vec

Most of you are probably already familiar with word embedding and understand the importance of a model like word2vec. For those who aren't, Stanford CS 224N's lecture on word vectors is a great resource. When you're at it, it might be a good idea to check out the following two papers:
Distributed Representations of Words and Phrases and their Compositionality (Mikolov et al., 2013)
Efficient Estimation of Word Representations in Vector Space (Mikolov et al., 2013)

At a high level, we need the find an efficient way to represent textual data (in this case, words) so that we can use this representation to solve natural language tasks. Word embeddings form the backbone in the solutions to many tasks such as language modeling, machine translation, sentiment analysis, etc.

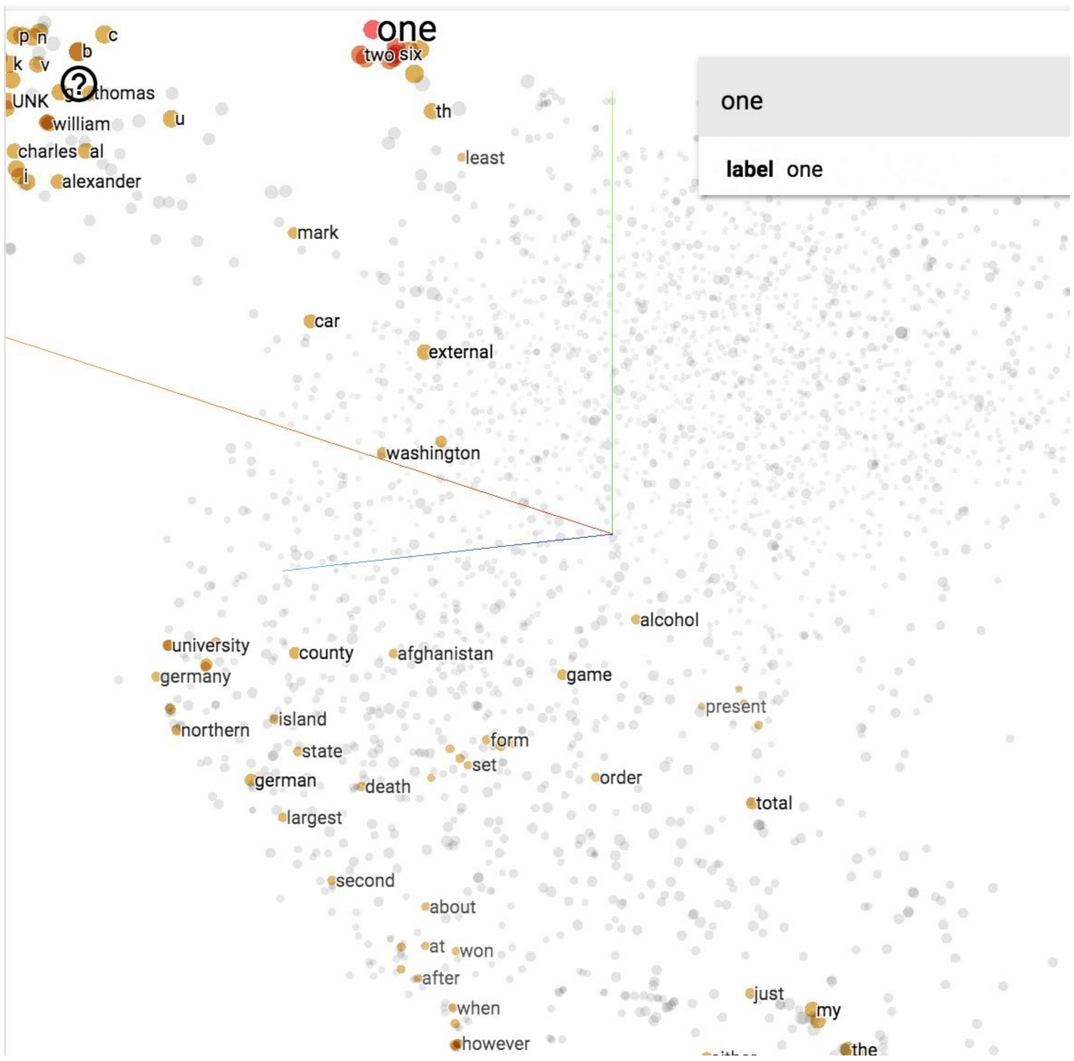
Created by a team of researchers led by Tomas Mikolov, word2vec is a group of models that are used to produce word embeddings. There are two main models used in word2vec: skip-gram and CBOW.

```
Skip-gram vs CBOW (Continuous Bag-of-Words)

Algorithmically, these models are similar, except that CBOW predicts center words from
context words, while the skip-gram does the inverse and predicts source context-words from
the center words. For example, if we have the sentence: ""The quick brown fox jumps"", then
CBOW tries to predict ""brown"" from ""the"", ""quick"", ""fox"", and ""jumps"", while
skip-gram tries to predict ""the"", ""quick"", ""fox"", and ""jumps"" from ""brown"".

Statistically it has the effect that CBOW smoothes over a lot of the distributional
information (by treating an entire context as one observation). For the most part, this
turns out to be a useful thing for smaller datasets. However, skip-gram treats each
context-target pair as a new observation, and this tends to do better when we have larger
datasets.
```

In this lecture, we will build word2vec, the skip-gram model. In this model, to get the vector representations of words, we train a simple neural network with a single hidden layer to perform a certain task, but then we don't use that neural network for the task we train it on. Instead, we care about the weights of the hidden layer. These weights are actually the "word vectors", or "embedding matrix" that we're trying to learn.

The certain, fake task we're going to train our model on is predicting the neighboring (context) words given the center word. Given a specific word in a sentence (the center word), look at the words nearby and pick one at random. The network is going to tell us the probability for every word in our vocabulary of being a neighbor to a specific word. Chris McCormick wrote a tutorial that explains the skip-gram model wonderfully if you want more details.

Vector representations of words visualized with t-SNE projected on a 3D space, using TensorBoard.



Softmax, Negative Sampling, and Noise Contrastive Estimation

To get the distribution of the possible neighboring words, in theory, we often use softmax. Softmax maps arbitrary values x_i to a probability distribution p_i. In this case, $softmax(x_i)$ is the probability that x_i is a neighboring word of a specific word we are considering.

$$softmax(x_i) = exp(x_i) / \sum_i exp(x_i)$$

However, the normalization term in the denominator requires us to perform exp on all words in the dictionary and sum the results up, which could be millions of words. Even if you disregard uncommon words, a natural language model doesn't perform well unless you consider at least tens of thousands of the most common words. The normalization term causes softmax to be computationally prohibitive.

There are two main approaches to circumvent this bottleneck: hierarchical softmax and sample-based softmax. Mikolov et al. have shown in their paper *Distributed Representations of Words and Phrases and their Compositionality* that for training the skip-gram model, negative sample results in faster training and better vector representations for frequent words, compared to more complex hierarchical softmax.

Negative sampling, as the name suggests, belongs to the family of sample-based approaches. This family also includes importance sampling and target sampling. Negative sampling is actually a simplified model of an approach called Noise Contrastive Estimation (NCE), e.g. negative sampling makes certain assumption about the number of noise samples to generate -- let's call it k -- and the distribution of noise samples -- let's call it Q -- such that $kQ(w) = 1$ to simplify computation. For more details, please see Sebastian Rudder's *On word embeddings - Part 2: Approximating the Softmax* and Chris Dyer's *Notes on Noise Contrastive Estimation and Negative Sampling*.

While negative sampling is useful for the learning word embeddings, it doesn't have the theoretical guarantee that its derivative tends towards the gradient of the softmax function. NCE, on the contrary, offers this guarantee as the number of noise samples increases. Mnih and Teh (2012) reported that 25 noise samples are sufficient to match the performance of the regular softmax, with an expected speed-up factor of about 45. For this reason, in this example, we will be using NCE.

Note that sample-based approaches, whether it's negative sampling or NCE, are only useful at training time -- during inference, the full softmax still needs to be computed to obtain a normalized probability.

Dataset

text8 is the first 100 MB of cleaned text of the English Wikipedia dump on Mar. 3, 2006 (whose link is no longer available). We use text that has already been pre-processed because it takes a lot of time to process the raw text and we'd rather use the time in this class to focus on TensorFlow. We can download the dataset here. The file word_utils.py on our GitHub repo has a script to download and read the text

100MB is not enough to train really good word embeddings, but enough to see some interesting relations. There are 17,005,207 tokens if you count tokens by splitting the text by blank space. For better results, you should use the dataset fil9 of the first 10^9 bytes of the Wikipedia dump, as described on Matt Mahoney's website.

Implementing word2vec

In this example, we implement skip-gram without eager execution. For example with eager execution, please see examples/04_word2vec_eager.py. If you want to give it a try first, use examples/04_word2vec_eager_starter.py.

Phase 1: Assemble the graph

1. Create dataset and generate samples from them
Input is the center word and output is the neighboring (context) word. Instead of feeding words into our model, we create a dictionary of the most common words, and feed the indices of those words. For example, if the center word is the 1000^{th} word in the vocabulary, we input the number 1001.

Each sample input is a scalar, so BATCH_SIZE of sample inputs with have shape [BATCH_SIZE] Similarly, BATCH_SIZE of sample outputs with have shape [BATCH_SIZE, 1].

```
dataset = tf.data.Dataset.from_generator(gen,
                            (tf.int32, tf.int32),
                            (tf.TensorShape([BATCH_SIZE]), tf.TensorShape([BATCH_SIZE, 1])))
iterator = dataset.make_initializable_iterator()
center_words, target_words = iterator.get_next()
```

2. Define the weight (in this case, embedding matrix)
Each row corresponds to the representation vector of one word. If one word is represented with a vector of size EMBED_SIZE, then the embedding matrix will have shape [VOCAB_SIZE, EMBED_SIZE]. We initialize the embedding matrix to value from a random distribution. In this case, let's choose uniform distribution.

```
embed_matrix = tf.get_variable('embed_matrix',
                                shape=[VOCAB_SIZE, EMBED_SIZE],
                                initializer=tf.random_uniform_initializer())
```

3. Inference (compute the forward path of the graph)

Our goal is to get the vector representations of words in our dictionary. Remember that the embed_matrix has dimension VOCAB_SIZE x EMBED_SIZE, with each row of the embedding matrix corresponds to the vector representation of the word at that index. So to get the representation of all the center words in the batch, we get the slice of all corresponding rows in the embedding matrix. TensorFlow provides a convenient method to do so.

```
tf.nn.embedding_lookup(
    params,
    ids,
    partition_strategy='mod',
    name=None,
    validate_indices=True,
    max_norm=None
)
```

This method is really useful when it comes to matrix multiplication with one-hot vectors because it saves us from doing a bunch of unnecessary computation that will return 0 anyway. An illustration from Chris McCormick for multiplication of a one-hot vector with a matrix.

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

So, to get the embedding (or vector representation) of the input center words, we use this:

```
embed = tf.nn.embedding_lookup(embed_matrix, center_words, name='embed')
```

4. Define the loss function

While NCE is cumbersome to implement in pure Python, TensorFlow already implemented it for us.

```
tf.nn.nce_loss(
    weights,
    biases,
    labels,
    inputs,
    num_sampled,
    num_classes,
    num_true=1,
    sampled_values=None,
    remove_accidental_hits=False,
    partition_strategy='mod',
    name='nce_loss'
)
```

Note that by the way the function is implemented, the third argument is actually inputs, and the fourth is labels. This ambiguity can be quite troubling sometimes, but keep in mind that TensorFlow is still new and growing and therefore might not be perfect. Nce_loss source code can be found here.

For nce_loss, we need weights and biases for the hidden layer to calculate NCE loss. They will be updated by optimizer during training. After sampling, the final output score will be computed by:

```
tf.matmul(embed, tf.transpose(nce_weight)) + nce_bias
```

```
nce_weight = tf.get_variable('nce_weight',
       shape=[VOCAB_SIZE, EMBED_SIZE],
       initializer=tf.truncated_normal_initializer(stddev=1.0 / (EMBED_SIZE ** 0.5)))
nce_bias = tf.get_variable('nce_bias', initializer=tf.zeros([VOCAB_SIZE]))
```

Then we define loss:

```
loss = tf.reduce_mean(tf.nn.nce_loss(weights=nce_weight,
                                     biases=nce_bias,
                                     labels=target_words,
                                     inputs=embed,
                                     num_sampled=NUM_SAMPLED,
                                     num_classes=VOCAB_SIZE))
```

5. Define optimizer
We will use the good old gradient descent.

```
optimizer = tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(loss)
```

Phase 2: Execute the computation

We will create a good old session to run the optimizer to minimize the loss, and report the loss value back to us. Don't forget to reinitialize your iterator!

```
with tf.Session() as sess:
        sess.run(iterator.initializer)
        sess.run(tf.global_variables_initializer())
```

```
    writer = tf.summary.FileWriter('graphs/word2vec_simple', sess.graph)

    for index in range(NUM_TRAIN_STEPS):
        try:
            loss_batch, _ = sess.run([loss, optimizer])
        except tf.errors.OutOfRangeError:
            sess.run(iterator.initializer)
    writer.close()
```

You can see the full basic model on the class's GitHub repo under the name `word2vec.py`.

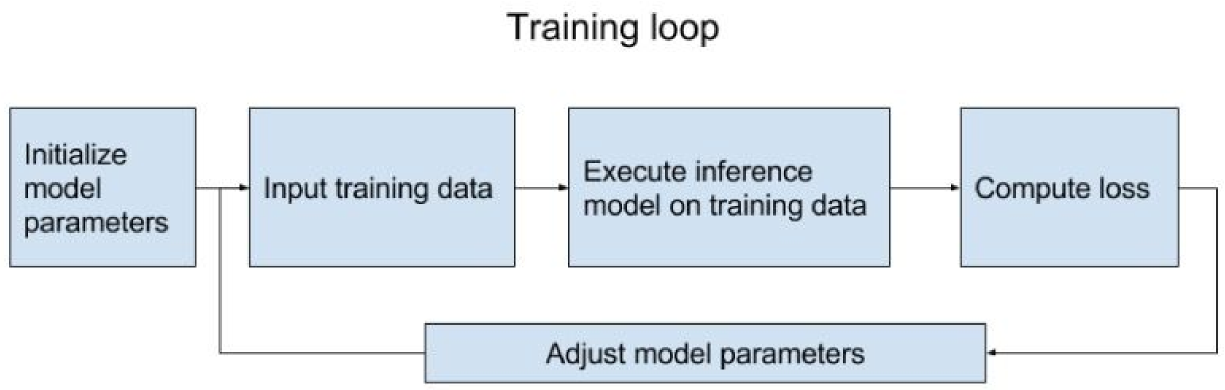
Interface: How to structure your TensorFlow model

All models we've built so far have more or less have the same structure:

Phase 1: assemble your graph
1. Import data (either with tf.data or with placeholders)
2. Define the weights
3. Define the inference model
4. Define loss function
5. Define optimizer

Phase 2: execute the computation
Which is basically training your model. There are a few steps:
1. Initialize all model variables for the first time.
2. Initialize iterator / feed in the training data.
3. Execute the inference model on the training data, so it calculates for each training input example the output with the current model parameters.
4. Compute the cost
5. Adjust the model parameters to minimize/maximize the cost depending on the model.

Here is a visualization of training loop from the book *TensorFlow for Machine Intelligence* (Abrahams et al., 2016).


Training loop
Initialize model parameters
Input training data
Execute inference model on training data
Compute loss
Adjust model parameters

It took us about 20 lines of code to build a word embedding model! It's fast but ... "whatever happened to decomposition?" After we've spent an ungodly amount of time building a model, we'd like to use it more than once.

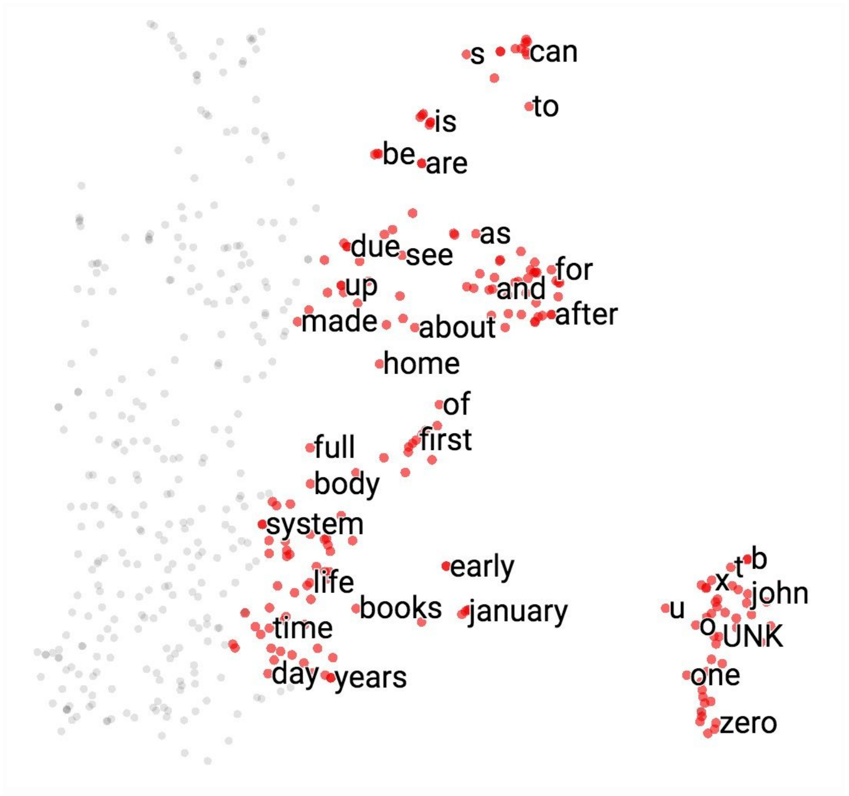
Question: how do we make our model most easy to reuse?
Hint: take advantage of Python's object-oriented-ness.
Answer: build our model as a class!

Our model base class should follow the interface. We combined step 3 and 4 because we want to put embed under the name scope of "NCE loss".

```
class SkipGramModel:
    """ Build the graph for word2vec model """
    def __init__(self, params):
        pass

    def _import_data(self):
        """ Step 1: import data """
        pass

    def _create_embedding(self):
        """ Step 2: in word2vec, it's actually the weights that we care about """
        pass

    def _create_loss(self):
        """ Step 3 + 4: define the inference + the loss function """
        pass

    def _create_optimizer(self):
        """ Step 5: define optimizer """
        pass
```

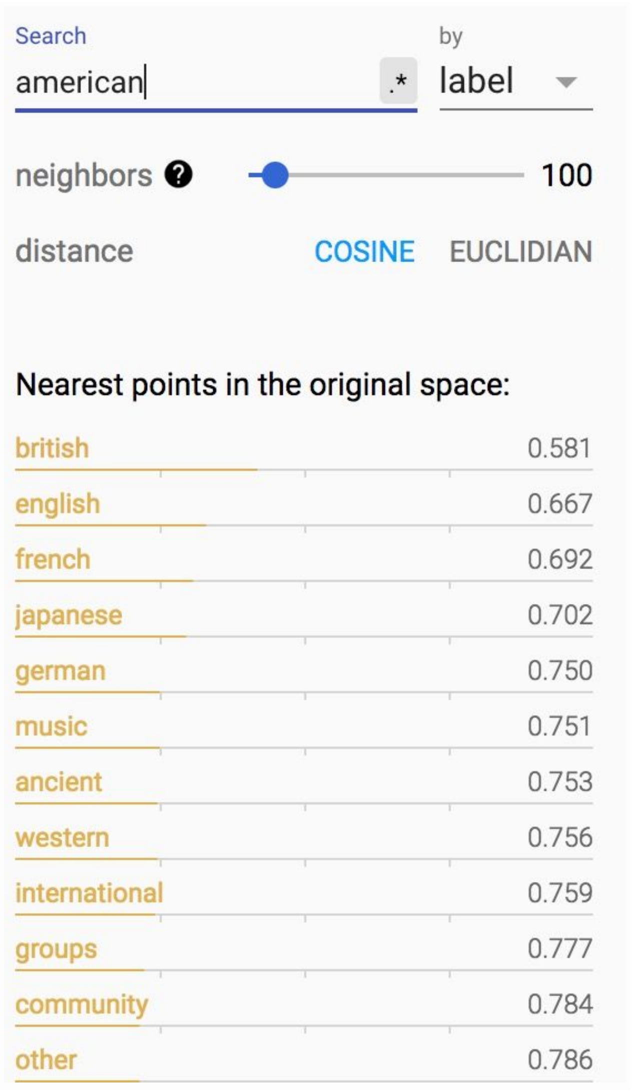
Visualize embeddings

Now let's see what our model finds after training it for 100,000 epochs. If we visualize our embedding with t-SNE, we will see something like this.



It's hard to see in 2D, but we'll see in class in 3D that all the number (one, two, ..., zero) are grouped in a line on the bottom right, next to all the alphabet (a, b, ..., z) and names (john, james, david, and such). All the months are grouped together. "Do", "does", "did" are also grouped together and so on.

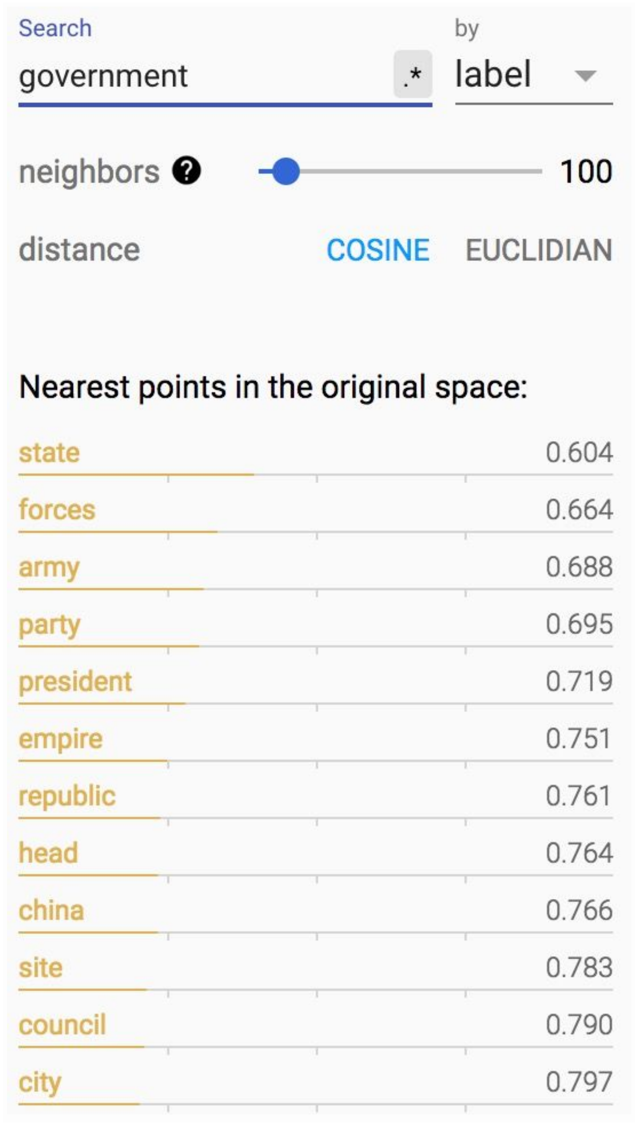
If you print out the closest words to 'american', you will find its closest cosine neighbors are 'british' and 'english'. Fair enough.



Nearest points in the original space:

british	0.581
english	0.667
french	0.692
japanese	0.702
german	0.750
music	0.751
ancient	0.753
western	0.756
international	0.759
groups	0.777
community	0.784
other	0.786

How about words closest to ‘government’?



Search government by label

neighbors 100

distance COSINE EUCLIDIAN

Nearest points in the original space:

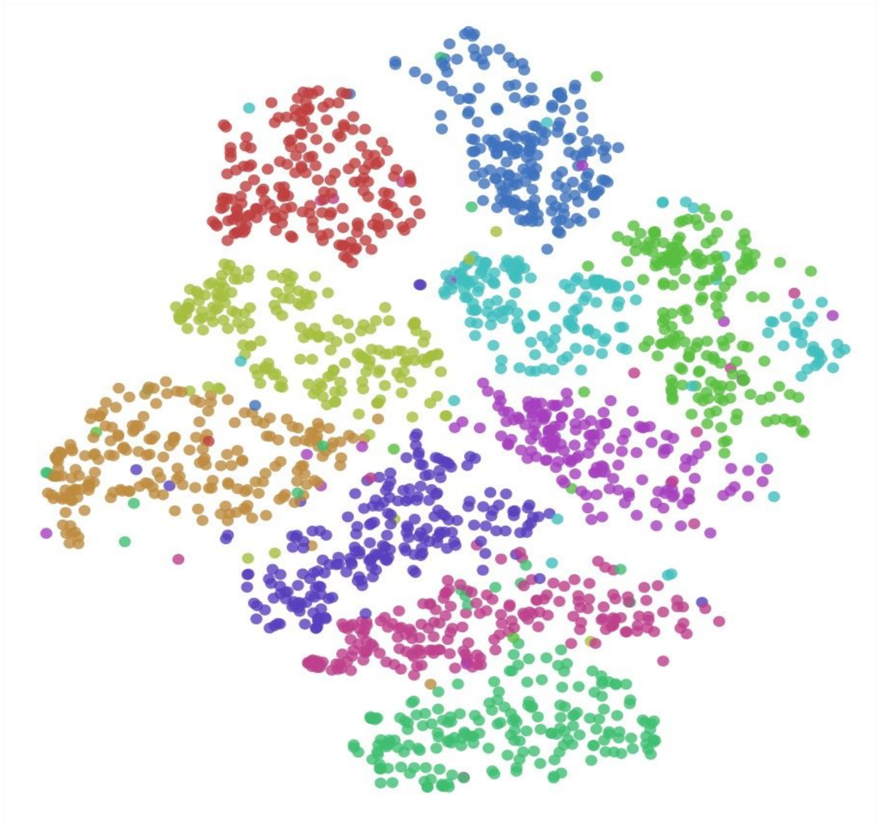
state	0.604
forces	0.664
army	0.688
party	0.695
president	0.719
empire	0.751
republic	0.761
head	0.764
china	0.766
site	0.783
council	0.790
city	0.797

```
t-SNE (from Wikipedia)

t-distributed stochastic neighbor embedding (t-SNE) is a machine learning algorithm for
dimensionality reduction developed by Geoffrey Hinton and Laurens van der Maaten. It is a
nonlinear dimensionality reduction technique that is particularly well-suited for embedding
high-dimensional data into a space of two or three dimensions, which can then be visualized
in a scatter plot. Specifically, it models each high-dimensional object by a two- or
three-dimensional point in such a way that similar objects are modeled by nearby points and
dissimilar objects are modeled by distant points.

The t-SNE algorithm comprises two main stages. First, t-SNE constructs a probability
distribution over pairs of high-dimensional objects in such a way that similar objects have
```

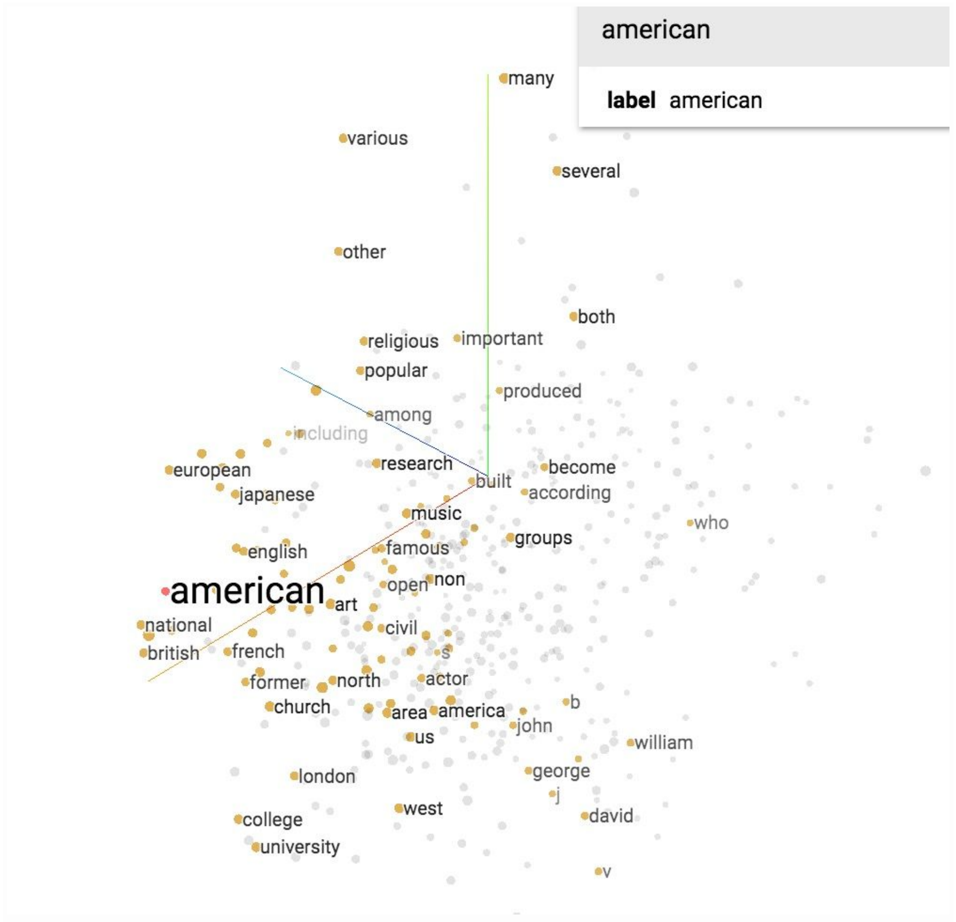
```
a high probability of being picked, whilst dissimilar points have an extremely small
probability of being picked. Second, t-SNE defines a similar probability distribution over
the points in the low-dimensional map, and it minimizes the Kullback–Leibler divergence
between the two distributions with respect to the locations of the points in the map. Note
that whilst the original algorithm uses the Euclidean distance between objects as the base
of its similarity metric, this should be changed as appropriate.
```

If you haven't used t-SNE, you should start using it! It's super cool.

You can visualize more than word embeddings, aka, you can visualize any vector representations of anything! Have you read Chris Olah's blog post about visualizing MNIST? t-SNE made MNIST desirable! Image below is from Olah's blog.


A t-SNE plot of MNIST

We can also visualize our embeddings using PCA too.



And we did all that visualization with less than 10 lines of code using TensorFlow projector with TensorBoard! The tool is super useful, albeit a bit finicky to use. The visualization will be stored in visualization folder. To see it, run `"'tensorboard --logdir='visualization'".`

```
from tensorflow.contrib.tensorboard.plugins import projector

def visualize(self, visual_fld, num_visualize):
        # create the list of num_variable most common words to visualize
        word2vec_utils.most_common_words(visual_fld, num_visualize)

        saver = tf.train.Saver()
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            ckpt = tf.train.get_checkpoint_state(os.path.dirname('checkpoints/checkpoint'))

            # if that checkpoint exists, restore from checkpoint
            if ckpt and ckpt.model_checkpoint_path:
                saver.restore(sess, ckpt.model_checkpoint_path)

            final_embed_matrix = sess.run(self.embed_matrix)

            # you have to store embeddings in a new variable
            embedding_var = tf.Variable(final_embed_matrix[:num_visualize], name='embeded')
            sess.run(embedding_var.initializer)

            config = projector.ProjectorConfig()
            summary_writer = tf.summary.FileWriter(visual_fld)

            # add embedding to the config file
            embedding = config.embeddings.add()
            embedding.tensor_name = embedding_var.name

            # link this tensor to the file with the first NUM_VISUALIZE words of vocab
            embedding.metadata_path = os.path.join(visual_fld,[file_of_most_common_words])

            # saves a configuration file that TensorBoard will read during startup.
            projector.visualize_embeddings(summary_writer, config)
            saver_embed = tf.train.Saver([embedding_var])
            saver_embed.save(sess, os.path.join(visual_fld, 'model.ckpt'), 1)
```

To see the full code, please see `examples/04_word2vec_visualize.py` on the class's GitHub repo.