

Lecture note 4: How to structure your model in TensorFlow

“CS 20SI: TensorFlow for Deep Learning Research” (cs20si.stanford.edu)

Prepared by Chip Huyen (huyenn@stanford.edu)

Reviewed by Danijar Hafner

Up until this point, we've implemented two simple models in TensorFlow: linear regression on the number of fire and theft in the city of Chicago, and logistic regression to do an Optical Character Recognition task on the MNIST dataset. With the tools we have, we can definitely build more complicated models. However, complex models would require better planning, otherwise our models would be pretty messy and hard to debug. In the next two lectures, we will discuss a way to efficiently structure our models. And we will be doing that through an example: word2vec.

I expect that most of you are already familiar with word embedding and understand the importance of a model like word2vec. For those who aren't familiar with this, you can read the CS 224N lecture slide about the motivation for and explanation of word2vec at [Simple Word Vector Representations](#).

The original papers by Mikolov et al.

[Distributed Representations of Words and Phrases and their Compositionality](#)

[Efficient Estimation of Word Representations in Vector Space](#)

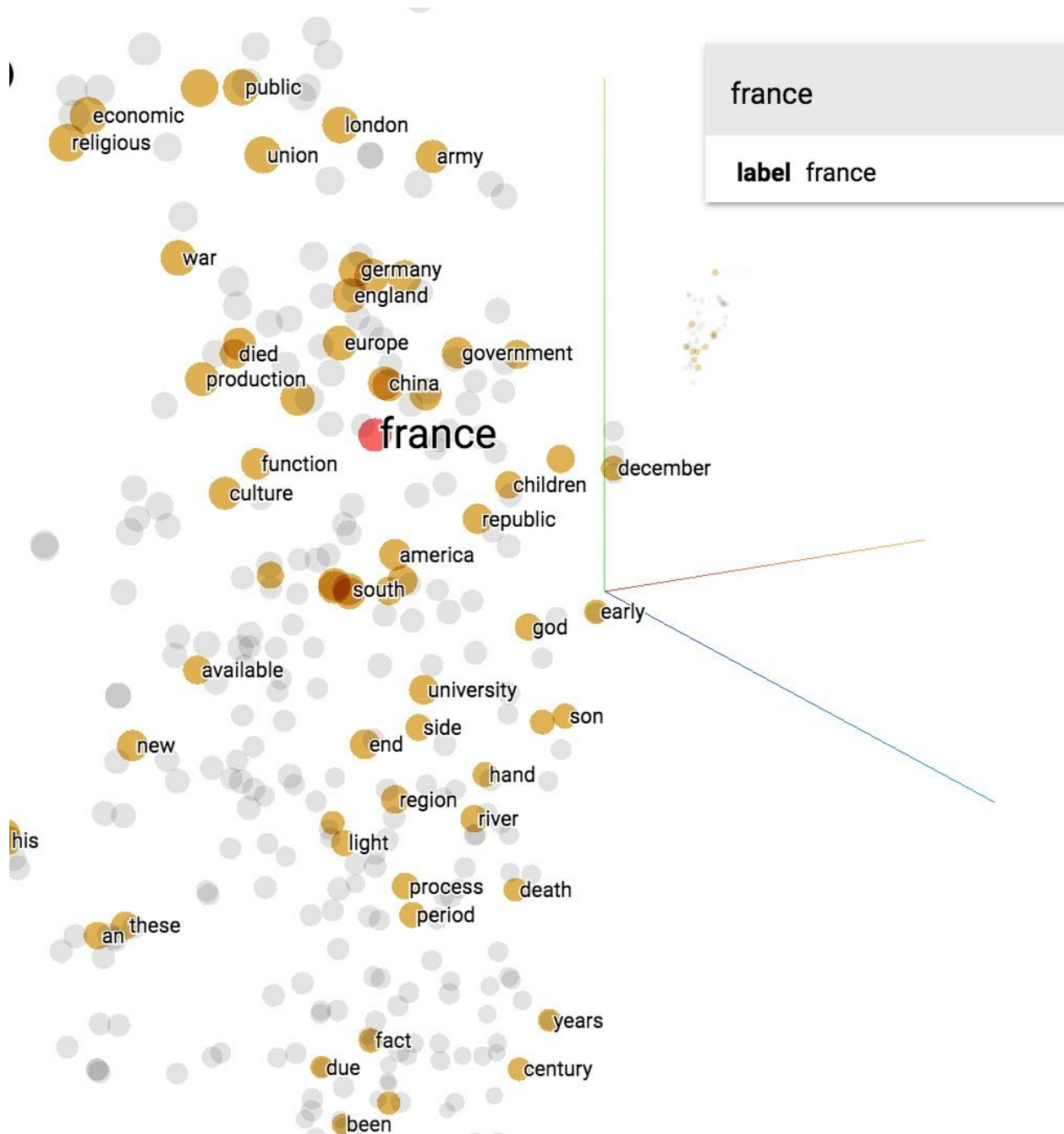
In short, we need a vector representation of words so that we can input them into our neural networks to do some magic tricks. Word vectors form the basis of many models studied in CS224N, as well as most language models in real life.

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.

Vector representations of words projected on a 3D space.



In this lecture, we will try to build word2vec, the skip-gram model. You can find an explanation/tutorial to the skip-gram model here.

[Word2Vec Tutorial - The Skip-Gram Model](#)

In the skip-gram 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 trained 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 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 the “nearby word” that we chose.

Softmax, Negative Sampling, and Noise Contrastive Estimation

In CS 224N, we learned about the two training methods: hierarchical softmax and negative sampling. We ruled out softmax because the normalization factor is too computationally expensive, and the students in CS 224N implemented the skip-gram model with negative sampling.

Negative sampling, as the name suggests, belongs to the family of sampling-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 (k) and the distribution of noise samples (Q) (negative sampling assumes that $kQ(w) = 1$) to simplify computation (read Sebastian Rudder’s [“On word embeddings - Part 2: Approximating the Softmax”](#) and Chris Dyer’s [“Notes on Noise Contrastive Estimation and Negative Sampling”](#)). Mikolov et al. have shown in their paper [“Distributed Representations of Words and Phrases and their Compositionality”](#) that training the Skip-gram model that results in faster training and better vector representations for frequent words, compared to more complex hierarchical softmax.

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, which makes it not so useful for language modelling.

NCE has this nice theoretical guarantees that negative sampling lacks 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.

In this example, we will be using NCE because of its nice theoretical guarantee.

Note that sampling-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.

About the 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 clean text 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.

100MB is not enough to train really good word embeddings, but enough to see some interesting relations. There are 17,005,207 tokens by simple splitting the text by blank space using `split()` function of python strings.

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](#).

Interface: How to structure your TensorFlow model

We've done only 2 models in the past, and they more or less have the same structure:

Phase 1: assemble your graph

1. Define placeholders for input and output
2. Define the weights
3. Define the inference model
4. Define loss function
5. Define optimizer

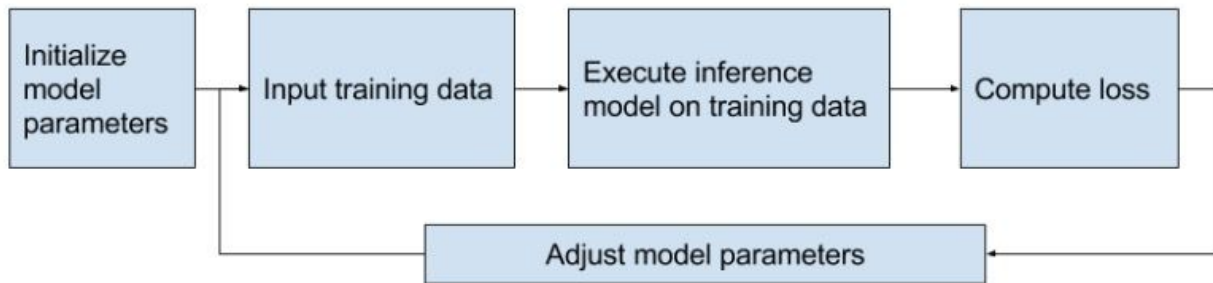
Phase 2: execute the computation

Which is basically training your model. There are a few steps:

1. Initialize all model variables for the first time.
2. Feed in the training data. Might involve randomizing the order of data samples.
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":

Training loop



Let's apply these steps to creating our word2vec, skip-gram model.

Phase 1: Assemble the graph

1. Define placeholders for input and output

Input is the center word and output is the target (context) word. Instead of using one-hot vectors, we input the index of those words directly. For example, if the center word is the 1001th word in the vocabulary, we input the number 1001.

Each sample input is a scalar, the placeholder for BATCH_SIZE sample inputs will have shape [BATCH_SIZE].

Similar, the placeholder for BATCH_SIZE sample outputs will have shape [BATCH_SIZE].

```
center_words = tf.placeholder(tf.int32, shape=[BATCH_SIZE])
target_words = tf.placeholder(tf.int32, shape=[BATCH_SIZE])
```

Note that our center_words and target_words being fed in are both scalars -- we feed in their corresponding indices in our vocabulary.

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.Variable(tf.random_uniform([VOCAB_SIZE, EMBED_SIZE], -1.0, 1.0))
```

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 called `tf.nn.embedding_lookup()`.

```
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)
```

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.

```
nce_weight = tf.Variable(tf.truncated_normal([VOCAB_SIZE, EMBED_SIZE],
stddev=1.0 / EMBED_SIZE ** 0.5))
nce_bias = tf.Variable(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 session then within the session, use the good old feed_dict to feed inputs and outputs into the placeholders, run the optimizer to minimize the loss, and fetch the loss value to report back to us.

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

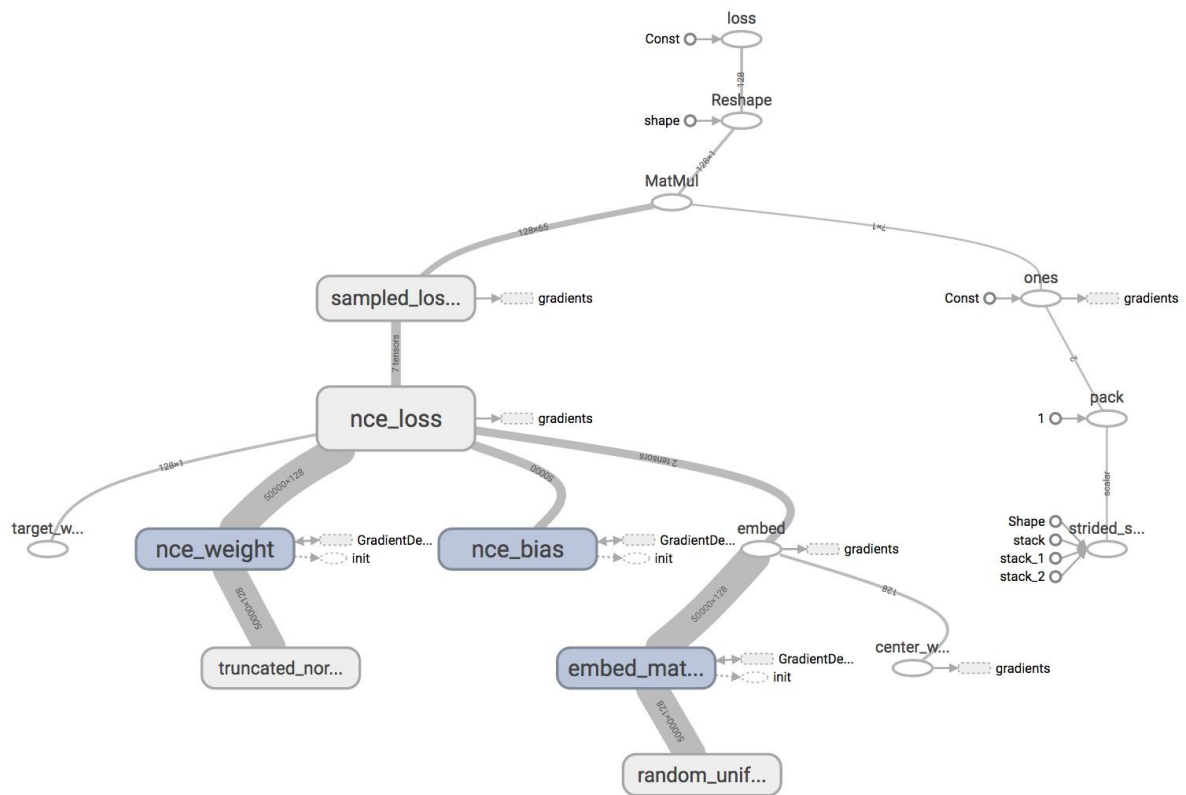
    average_loss = 0.0
    for index in xrange(NUM_TRAIN_STEPS):
        batch = batch_gen.next()
        loss_batch, _ = sess.run([loss, optimizer],
                                feed_dict={center_words: batch[0], target_words: batch[1]})
        average_loss += loss_batch
        if (index + 1) % 2000 == 0:
            print('Average loss at step {}: {:.5f}'.format(index + 1,
                                                            average_loss / (index + 1)))
```

You can see the full basic model on the class's GitHub repo under the name word2vec_no_frills.py

As you can see, the whole model takes less than 20 lines of code. If you've implemented word2vec without TensorFlow (as for the assignment 1 for CS224N), we know that this is really short. We've pretty much dumped everything into one giant function.

Name Scope

Let's give the tensors name and see how our model looks like in TensorBoard.



This doesn't look very readable, as you can see in the graph, the nodes are scattering all over. TensorBoard doesn't know which nodes are similar to which nodes and should be grouped together. This setback can grow to be extremely daunting when you build complex models with hundreds of ops.

Then, how can we tell TensorBoard to know which nodes should be grouped together? For example, we would like to group all ops related to input/output together, and all ops related to NCE loss together. Thankfully, TensorFlow lets us do that with name scope. You can just put all the ops that you want to group together under the block:

```
with tf.name_scope(name_of_that_scope):
    # declare op_1
    # declare op_2
    # ...
```

For example, our graph can have 3 op blocks: "Data", "embed", and "NCE_LOSS" like this:

```
with tf.name_scope('data'):
    center_words = tf.placeholder(tf.int32, shape=[BATCH_SIZE], name='center_words')
    target_words = tf.placeholder(tf.int32, shape=[BATCH_SIZE, 1], name='target_words')
```



```

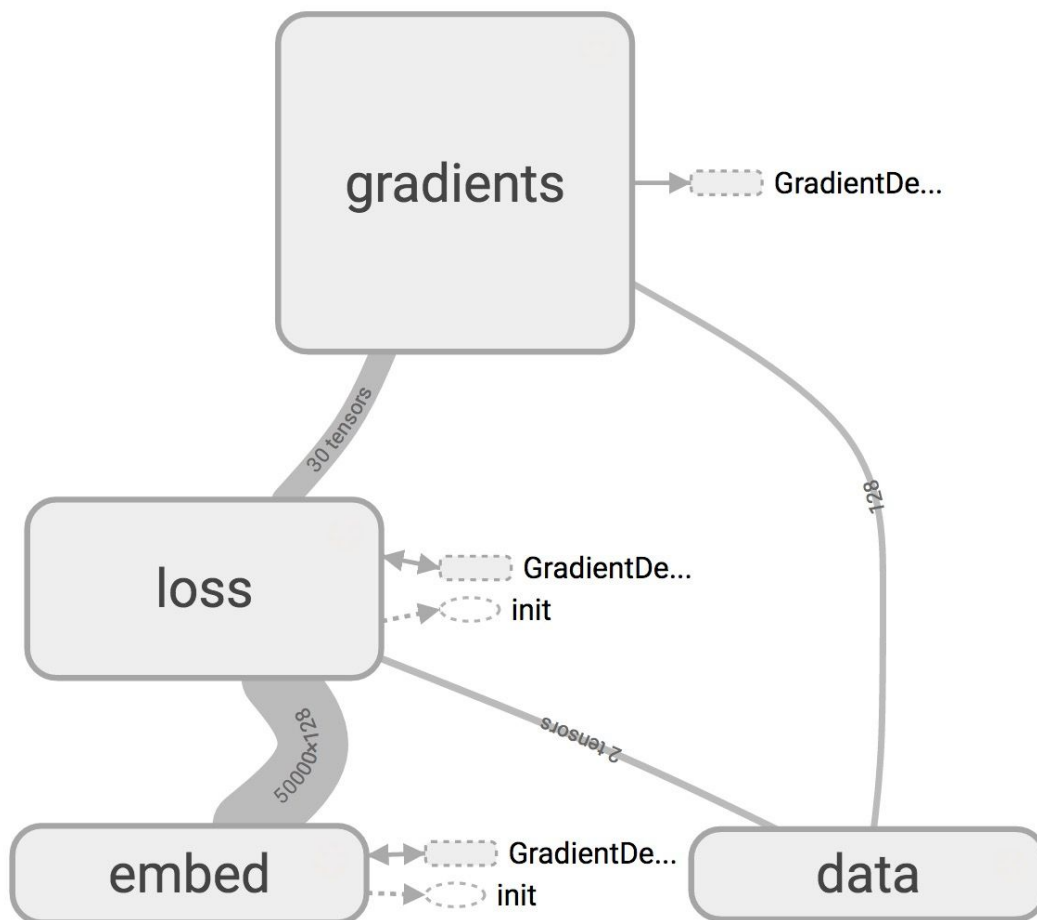
with tf.name_scope('embed'):
    embed_matrix = tf.Variable(tf.random_uniform([VOCAB_SIZE, EMBED_SIZE], -1.0, 1.0),
                              name='embed_matrix')

with tf.name_scope('loss'):
    embed = tf.nn.embedding_lookup(embed_matrix, center_words, name='embed')
    nce_weight = tf.Variable(tf.truncated_normal([VOCAB_SIZE, EMBED_SIZE],
                                                stddev=1.0 / math.sqrt(EMBED_SIZE)),
                             name='nce_weight')
    nce_bias = tf.Variable(tf.zeros([VOCAB_SIZE]), name='nce_bias')
    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),
                          name='loss')

```

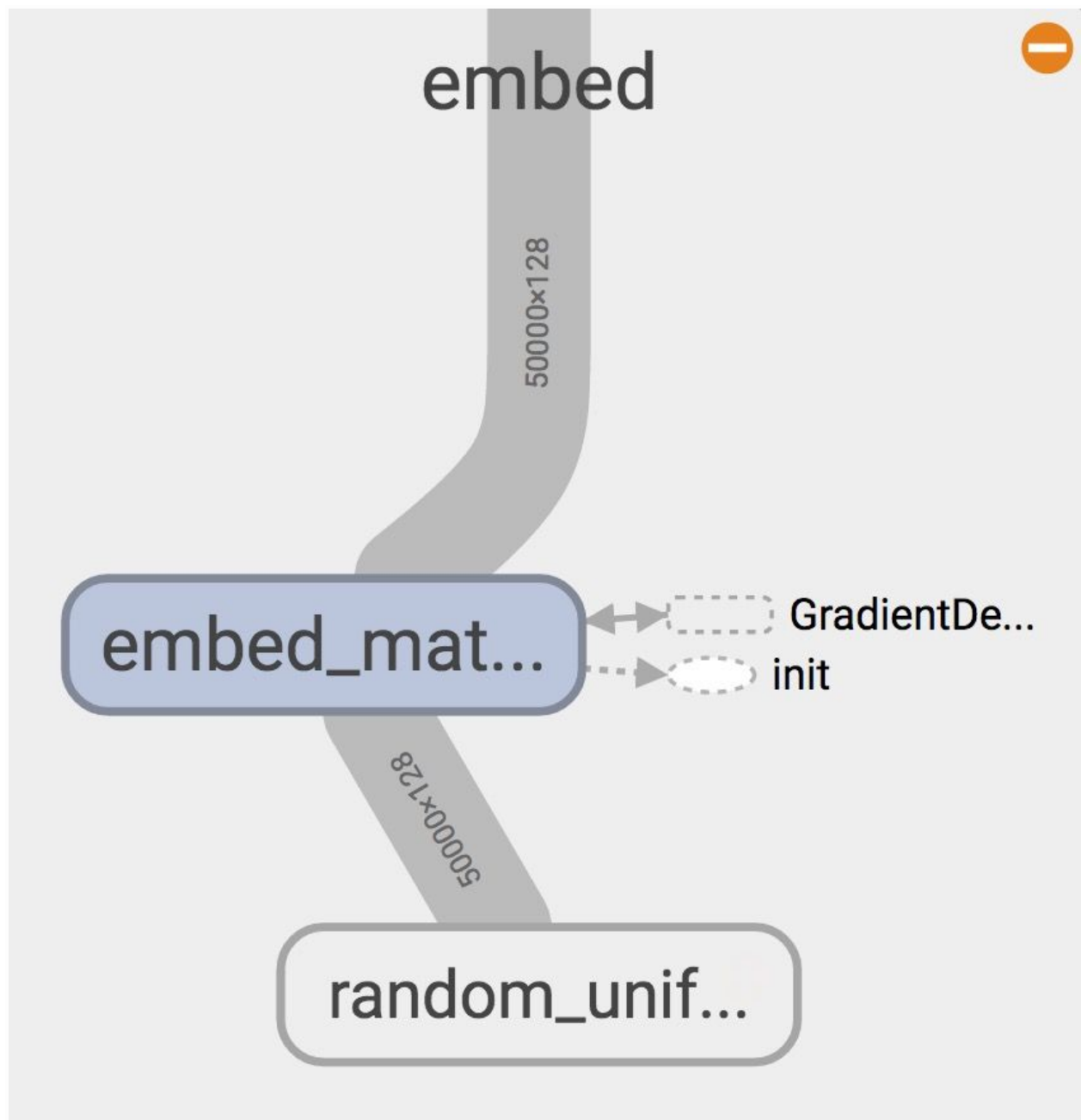
It seems like the namespace 'embed' has only one node and therefore it is useless to put it in a separate namespace. It, in fact, has two nodes: one for the `tf.Variable` and one for `tf.random_uniform`.

When you visualize that on TensorBoard, you will see your nodes are grouped into neat blocks:












You can click on the plus sign on top of each name scope block to see all the ops inside that block. I love graphs so I find this visualization fascinating. Take your time to play around with it.

You've probably noticed that TensorBoard has two kinds of edges: the solid lines and the dotted lines. The solid lines represent data flow edges. For example, the value of op `tf.add(x + y)` depends on the value of `x` and `y`. The dotted arrows represent control dependence edges. For example, a variable can only be used after being initialized, as you see variable `embed_matrix` depends on the op `init`. Control dependencies can also be declared using `tf.Graph.control_dependencies(control_inputs)` we talked about in lecture 2.



Here is the full legend of nodes in TensorBoard:

Graph (* = expandable)	
	Namespace*
	OpNode
	Unconnected series*
	Connected series*
	Constant
	Summary
	Dataflow edge
	Control dependency edge
	Reference edge

So now, our whole word2vec program looks more or less like this:

```
# Step 1: define the placeholders for input and output
with tf.name_scope("data"):
    center_words = tf.placeholder(tf.int32, shape=[BATCH_SIZE], name='center_words')
    target_words = tf.placeholder(tf.int32, shape=[BATCH_SIZE, 1], name='target_words')

# Assemble this part of the graph on the CPU. You can change it to GPU if you have GPU
with tf.device('/cpu:0'):
    with tf.name_scope("embed"):
        # Step 2: define weights. In word2vec, it's actually the weights that we care about
        embed_matrix = tf.Variable(tf.random_uniform([VOCAB_SIZE, EMBED_SIZE], -1.0, 1.0),
                                  name='embed_matrix')

    # Step 3 + 4: define the inference + the loss function
    with tf.name_scope("loss"):

        # Step 3: define the inference
        embed = tf.nn.embedding_lookup(embed_matrix, center_words, name='embed')

        # Step 4: construct variables for NCE loss
        nce_weight = tf.Variable(tf.truncated_normal([VOCAB_SIZE, EMBED_SIZE],
```

```

                                stddev=1.0 / math.sqrt(EMBED_SIZE)),
name='nce_weight')
    nce_bias = tf.Variable(tf.zeros([VOCAB_SIZE]), name='nce_bias')

    # define loss function to be NCE loss function
    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), name='loss')

    # Step 5: define optimizer
    optimizer = tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(loss)

```

If you've taken any CS106 class you'll know that this program will get check minus on styles because "whatever happened to decomposition?" You can't just dump everything into a giant function. Also, 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 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 _create_placeholders(self):
        """ Step 1: define the placeholders for input and output """
        pass

    def _create_embedding(self):
        """ Step 2: define weights. 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

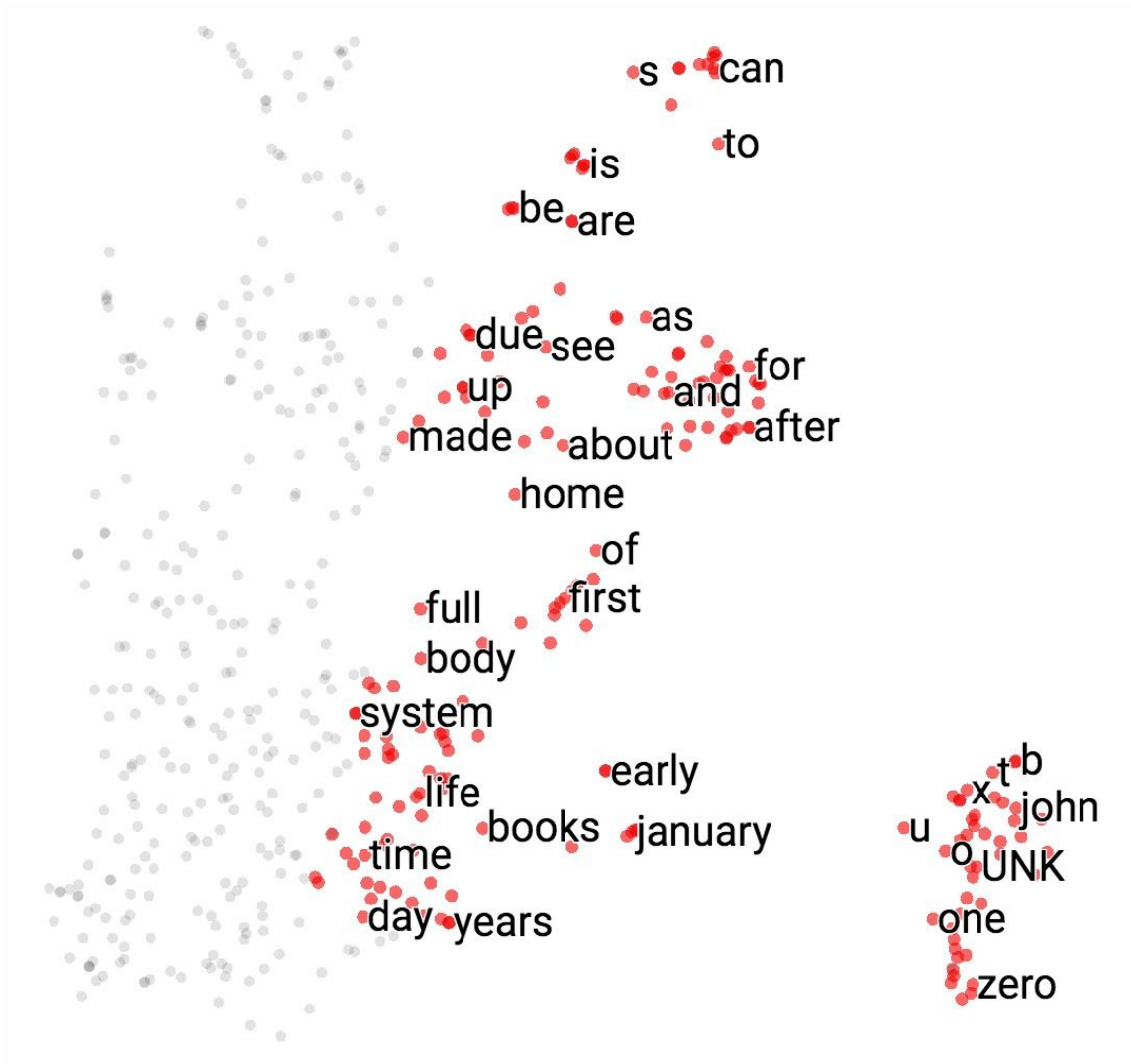
    def _create_optimizer(self):
        """ Step 5: define optimizer """
        pass

```

After 100,000 epochs, our loss went down to 10.0.

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 below. It's hard to visualize 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.

Search

american|

by

.*

label



neighbors ?



100

distance

COSINE

EUCLIDIAN


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

Search

government

bylabel

neighbors ?100

distanceCOSINE 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-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**

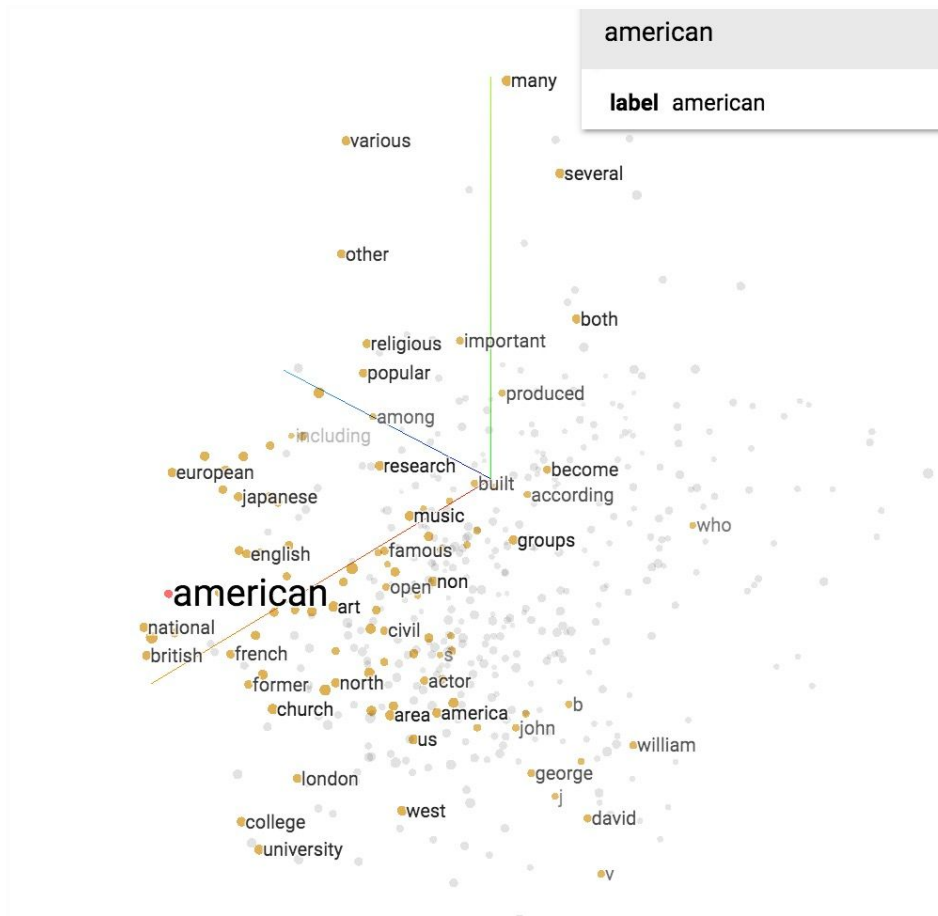
that whilst the original algorithm uses the **Euclidean** distance between objects as the base of its similarity metric, **this** should be changed as appropriate.

If you haven't used t-SNE, you should start using it! It's super cool. Have you read Chris Olah's blog post about [visualizing MNIST](#)? t-SNE made MNIST cool! Image below is from Olah's blog. You should head to his blog for the interactive version.



A t-SNE plot of MNIST

We can also visualize our embeddings using PCA too.



And I did all that visualization with less than 10 lines of code. TensorBoard provided a wonderful tool for doing so. Warning: the TensorFlow official guide is a bit ambiguous so you should follow this guide.

There are several steps.

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

# obtain the embedding_matrix after you've trained it
final_embed_matrix = sess.run(model.embed_matrix)

# create a variable to hold your embeddings. It has to be a variable. Constants
# don't work. You also can't just use the embed_matrix we defined earlier for our model. Why
# is that so? I don't know. I get the 500 most popular words.
embedding_var = tf.Variable(final_embed_matrix[:500], name='embedding')
sess.run(embedding_var.initializer)
config = projector.ProjectorConfig()
summary_writer = tf.summary.FileWriter(LOGDIR)

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

```
# link the embeddings to their metadata file. In this case, the file that contains
# the 500 most popular words in our vocabulary
embedding.metadata_path = LOGDIR + '/vocab_500.tsv'

# save a configuration file that TensorBoard will read during startup
projector.visualize_embeddings(summary_writer, config)

# save our embedding
saver_embed = tf.train.Saver([embedding_var])
saver_embed.save(sess, LOGDIR + '/skip-gram.ckpt', 1)
```

Now we run our model again, then again run tensorboard. If you go to <http://localhost:6006>, click on the Embeddings tab, you'll see all the visualization.

Cool, huh?

You can visualize more than word embeddings, aka, you can visualize any embeddings.

Why should we still learn gradients?

You've probably noticed that in all the models we've built so far, we haven't taken a single gradient. All we need to do is to build a forward pass and TensorFlow takes care of the backward path for us. So, the question is: why should we still learn to take gradient? Why are Chris Manning and Richard Socher making us take gradients of cross entropy and softmax? Shouldn't taking gradients by hands one day be as obsolete as trying to take square root by hands since the invention of calculator?

Well, maybe. But for now, TensorFlow can take gradients for us, but it can't give us intuition about what functions to use. It doesn't tell us if a function will suffer from exploding or vanishing gradients. We still need to know about gradients to get an understanding of why a model works while another doesn't.