Manage Experiments

CS 20: TensorFlow for Deep Learning Research
Lecture 5
1/26/2017

Agenda

word2vec

Embedding visualization

Structure your TensorFlow model

Variable sharing

Manage experiments

Autodiff

Word Embedding in TensorFlow

How do we represent words in an efficient way?

One-hot Representation

Each word is represented by one vector with a single 1 and the rest is 0

One-hot Representation

Each word is represented by one vector with a single 1 and the rest is 0

Example

Vocab: i, it, california, meh

i = [1 0 0 0]

it = [0 1 0 0]

california = [0 0 1 0]

meh = [0 0 0 1]

Problems with one-hot representation

- Vocabulary can be large

=> massive dimension, inefficient computation

- Can’t represent relationship between words

=> “anxious” and “nervous” are similar but would have completely different representations

Word Embedding

- Distributed representation
- Continuous values
- Low dimension
- Capture the semantic relationships between words

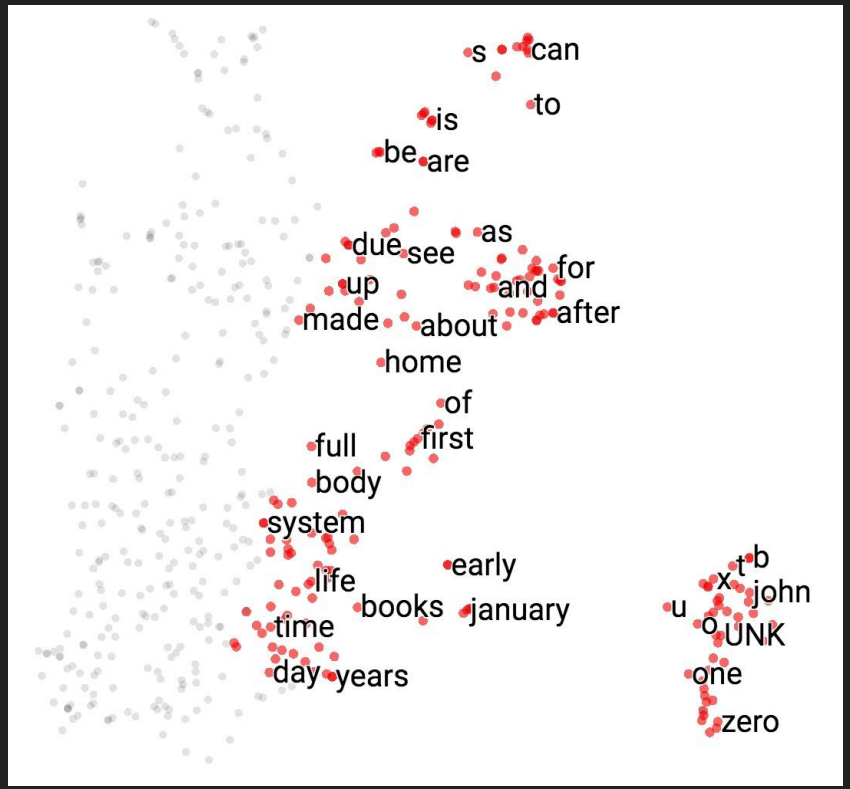
How?

Representing a word by means of its neighbors

"You shall know a word by the company it keeps."

- Firth, J. R. 1957:11

Word Embeddings



Live visualization

Count vs Predict

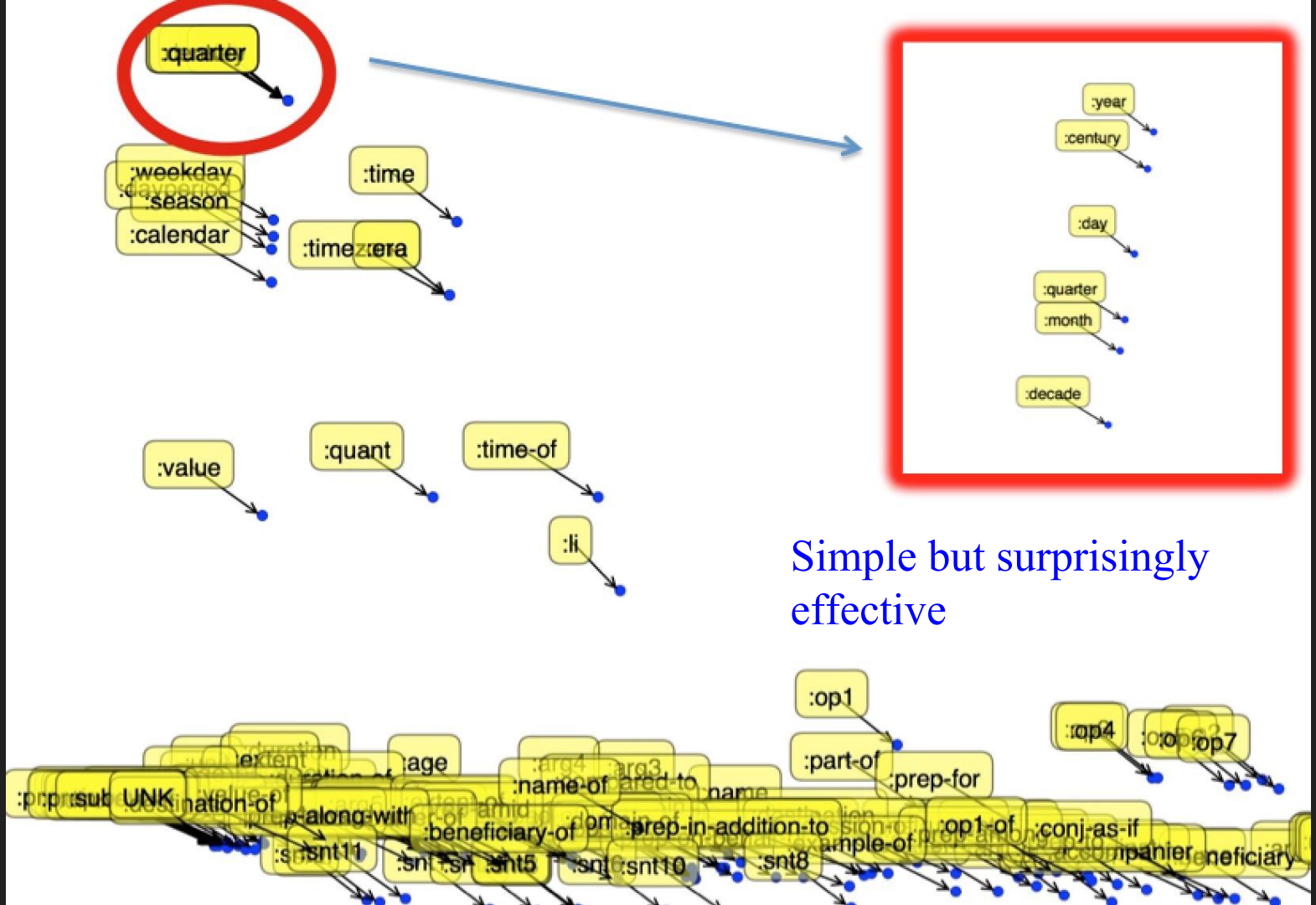
Counting

- Example corpus:
  - I like deep learning.
  - I like NLP.
  - I enjoy flying.

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

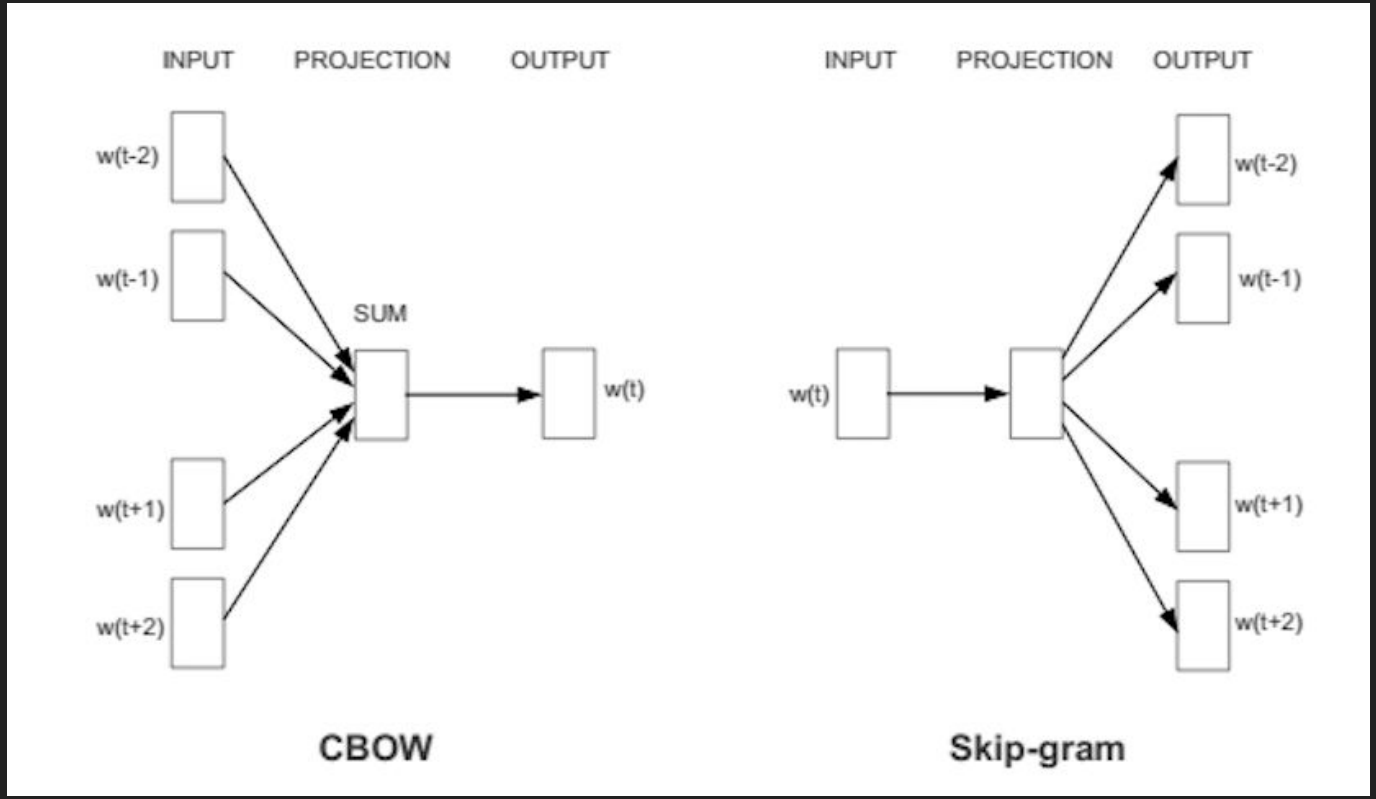
15

Graph by Richard Socher for CS224N



Simple but surprisingly effective

Predicting



Graph by deeplearning4j.org

Implementing word2vec skip-gram

Softmax vs Sample-based Approaches

Softmax

$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{V} \exp(u_w^T v_c)}$$

Computationally expensive

Handwritten formula from a slide for CS224N

Sample-based Approaches

Negative Sampling

is a simplified version of

Noise Contrastive Estimation

Sample-based Approaches

NCE guarantees approximation to softmax

Negative Sampling doesn't

For more information, see:

Sebastian Rudder's "On word embeddings - Part 2: Approximating the Softmax"

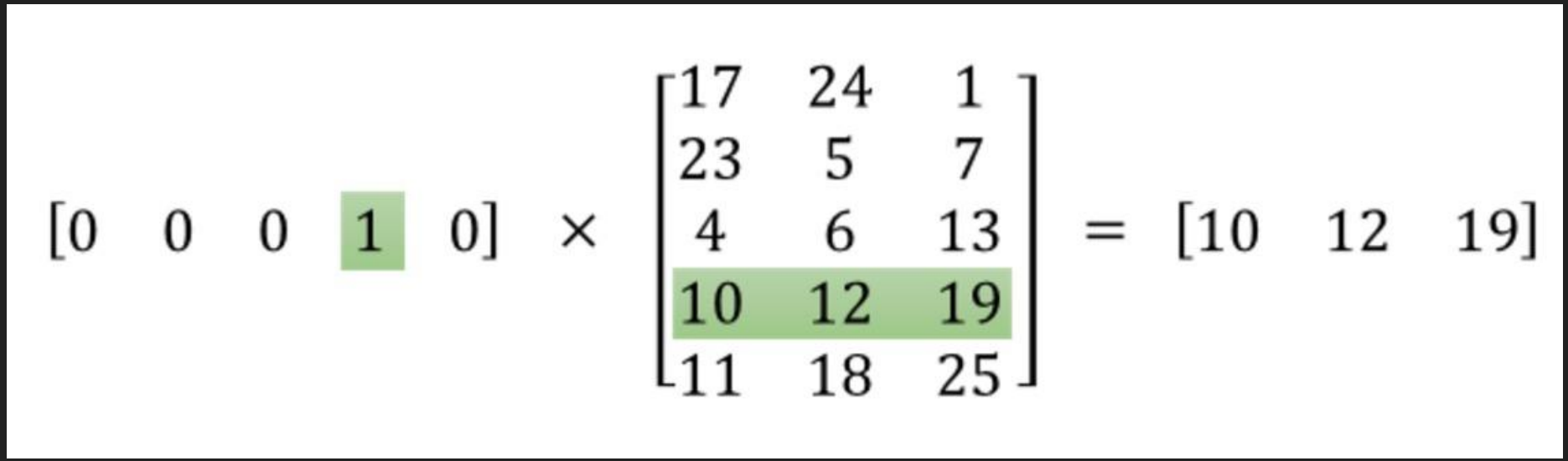
Chris Dyer's "Notes on Noise Contrastive Estimation and Negative Sampling"

Embedding Lookup

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

Illustration by Chris McCormick

Embedding Lookup




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


Illustration by Chris McCormick

NCE Loss

```
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'
)
```

Word2vec in TensorFlow

Interactive Coding

word2vec_utils.py

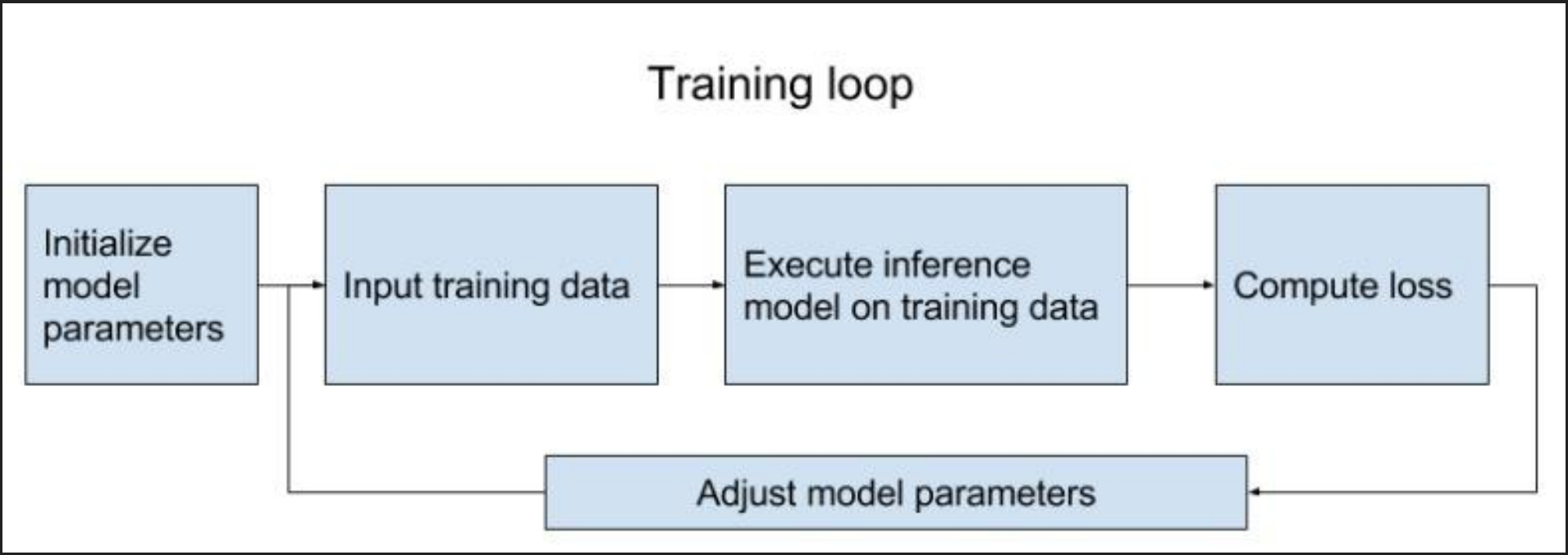
04_word2vec_eager_starter.py

Structure your TensorFlow model

Phase 1: Assemble graph

1. Import data (with tf.data or placeholders)
2. Define the weights
3. Define the inference model
4. Define loss function
5. Define optimizer

Phase 2: Compute



Graph from the book “TensorFlow for Machine Intelligence”

Need models to be reusable

Reusable models

- Define a class for your model
- Set up your model in a collection (e.g. map)

If you want to really reuse a model (without rebuilding it)

- For big models that take a long time to build, save the graph_def in a file and then load it

Model as a class

```
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: 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
```

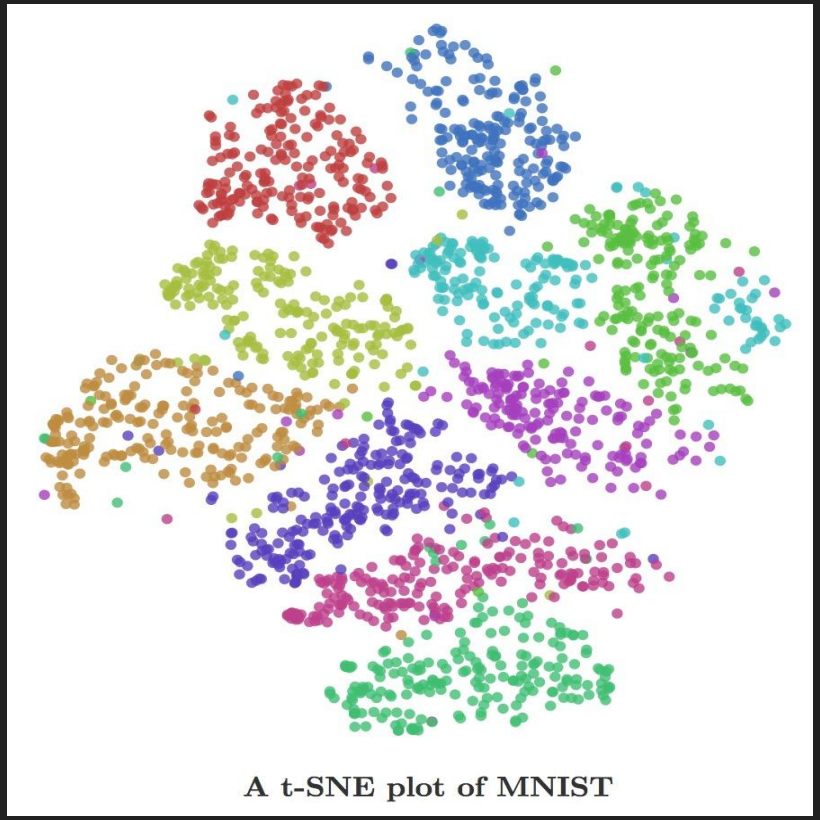
Yay, object oriented programming!!

Embedding visualization

Interactive Coding

`04_word2vec_visualize.py`

Visualize vector representation of anything

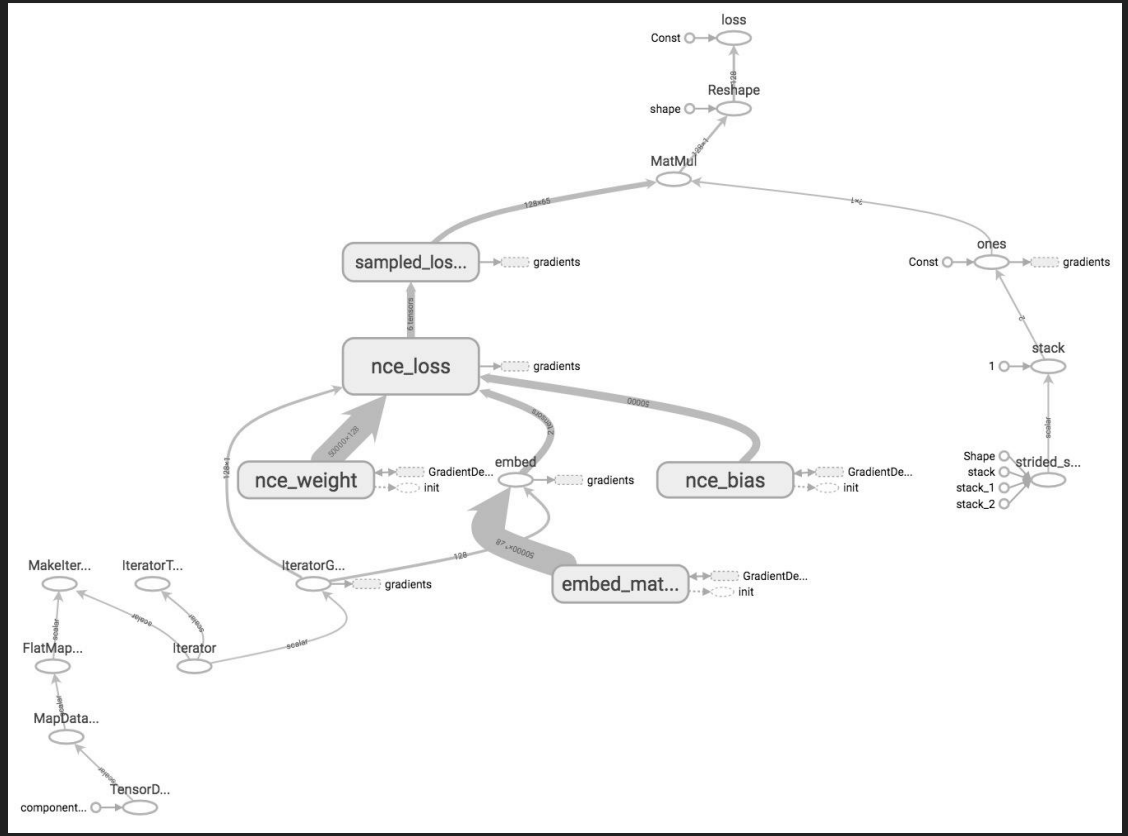


A t-SNE plot of MNIST

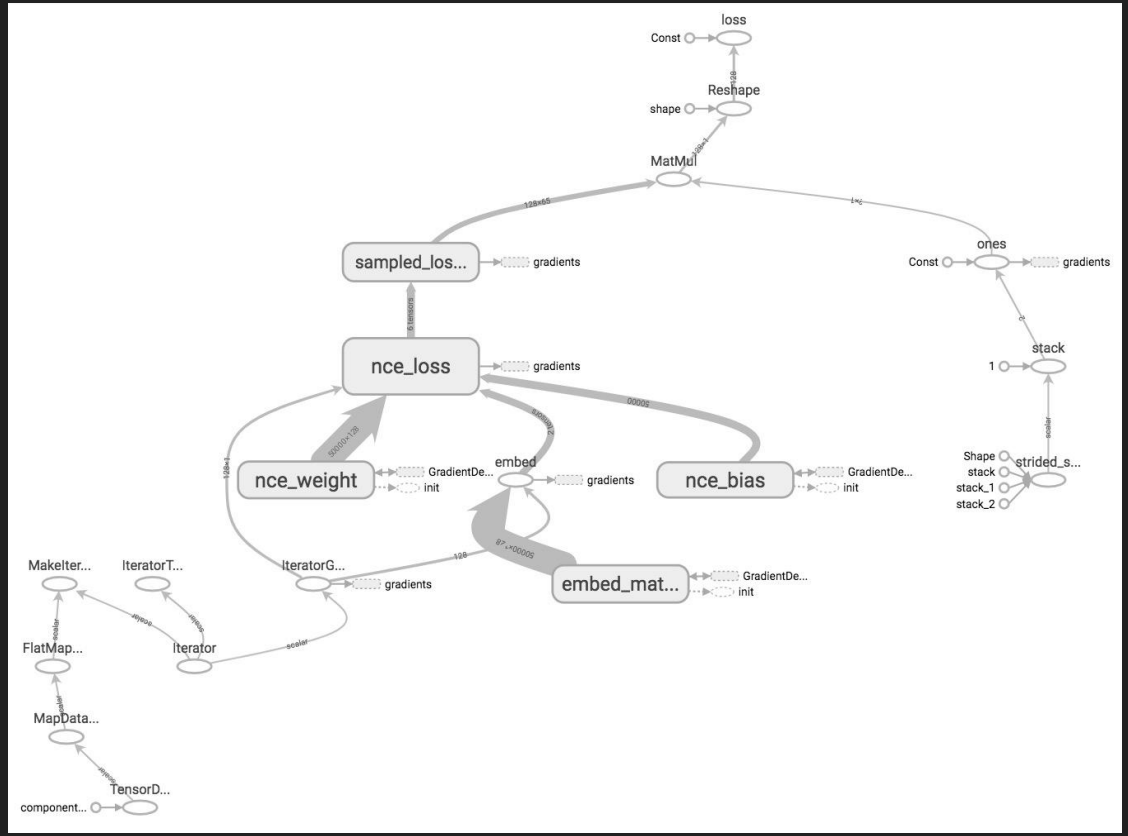
Visualization from Chris Olah's blog

Variable sharing

word2vec on TensorBoard



word2vec on TensorBoard



What if I have more complex models?

Name scope

TensorFlow doesn’t know what nodes should be grouped together, unless you tell it to

Name scope

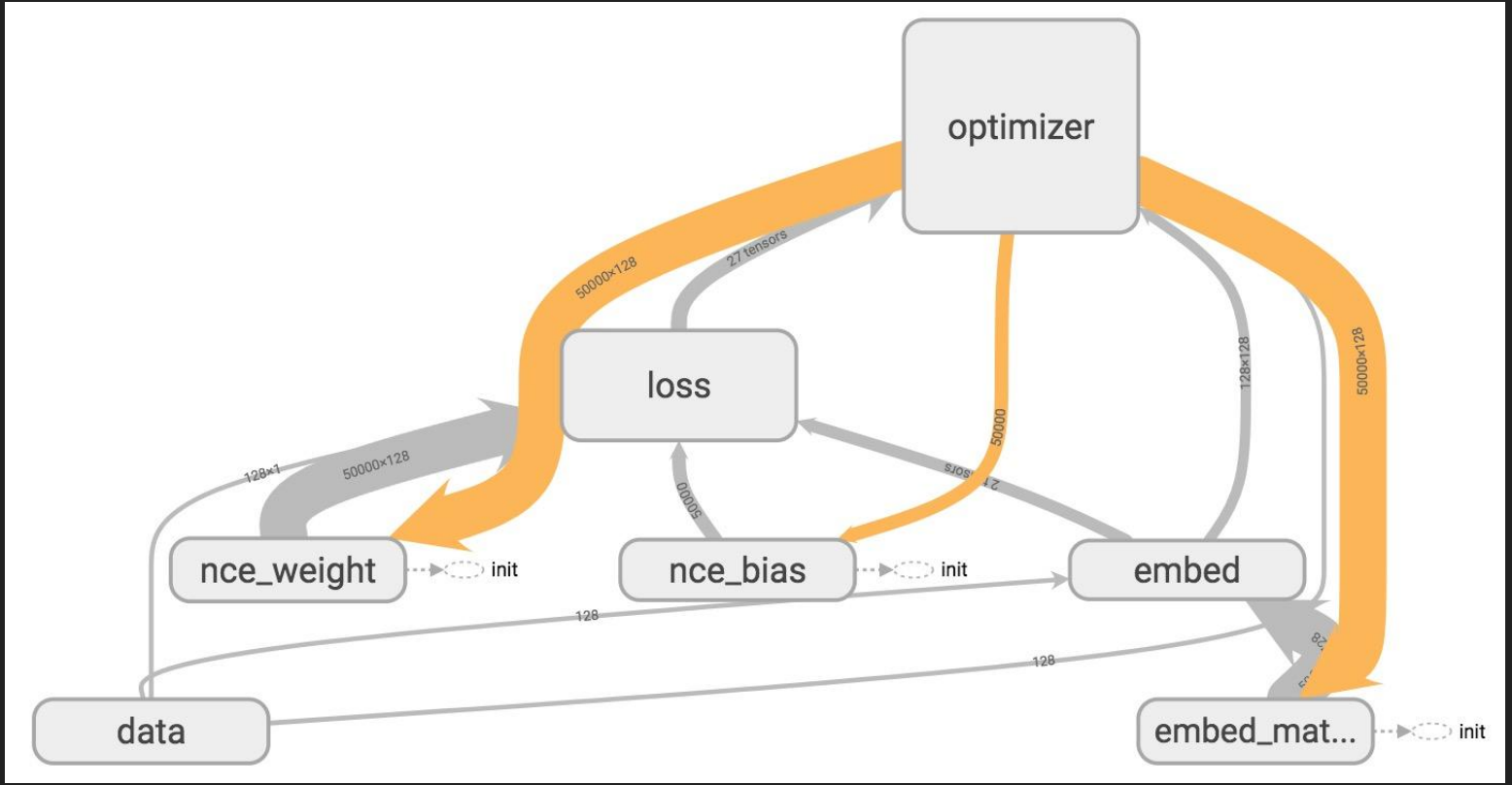
Group nodes together `with tf.name_scope(name)`

```
with tf.name_scope(name_of_that_scope):

    # declare op_1

    # declare op_2

    # ...
```

Name scope

```
with tf.name_scope('data'):
    iterator = dataset.make_initializable_iterator()
    center_words, target_words = iterator.get_next()

with tf.name_scope('embed'):
    embed_matrix = tf.get_variable('embed_matrix',
                                   shape=[VOCAB_SIZE, EMBED_SIZE], ...)
    embed = tf.nn.embedding_lookup(embed_matrix, center_words)

with tf.name_scope('loss'):
    nce_weight = tf.get_variable('nce_weight', shape=[VOCAB_SIZE, EMBED_SIZE], ...)
    nce_bias = tf.get_variable('nce_bias', initializer=tf.zeros([VOCAB_SIZE]))
    loss = tf.reduce_mean(tf.nn.nce_loss(weights=nce_weight, biases=nce_bias, ...)

with tf.name_scope('optimizer'):
    optimizer = tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(loss)
```

TensorBoard



Variable scope

Name scope vs variable scope

tf.name_scope() vs tf.variable_scope()

Variable scope

Name scope vs variable scope

Variable scope facilitates variable sharing

Variable sharing: The problem

```
def two_hidden_layers(x):
    w1 = tf.Variable(tf.random_normal([100, 50]), name='h1_weights')
    b1 = tf.Variable(tf.zeros([50]), name='h1_biases')
    h1 = tf.matmul(x, w1) + b1

    w2 = tf.Variable(tf.random_normal([50, 10]), name='h2_weights')
    b2 = tf.Variable(tf.zeros([10]), name='2_biases')
    logits = tf.matmul(h1, w2) + b2
    return logits
```

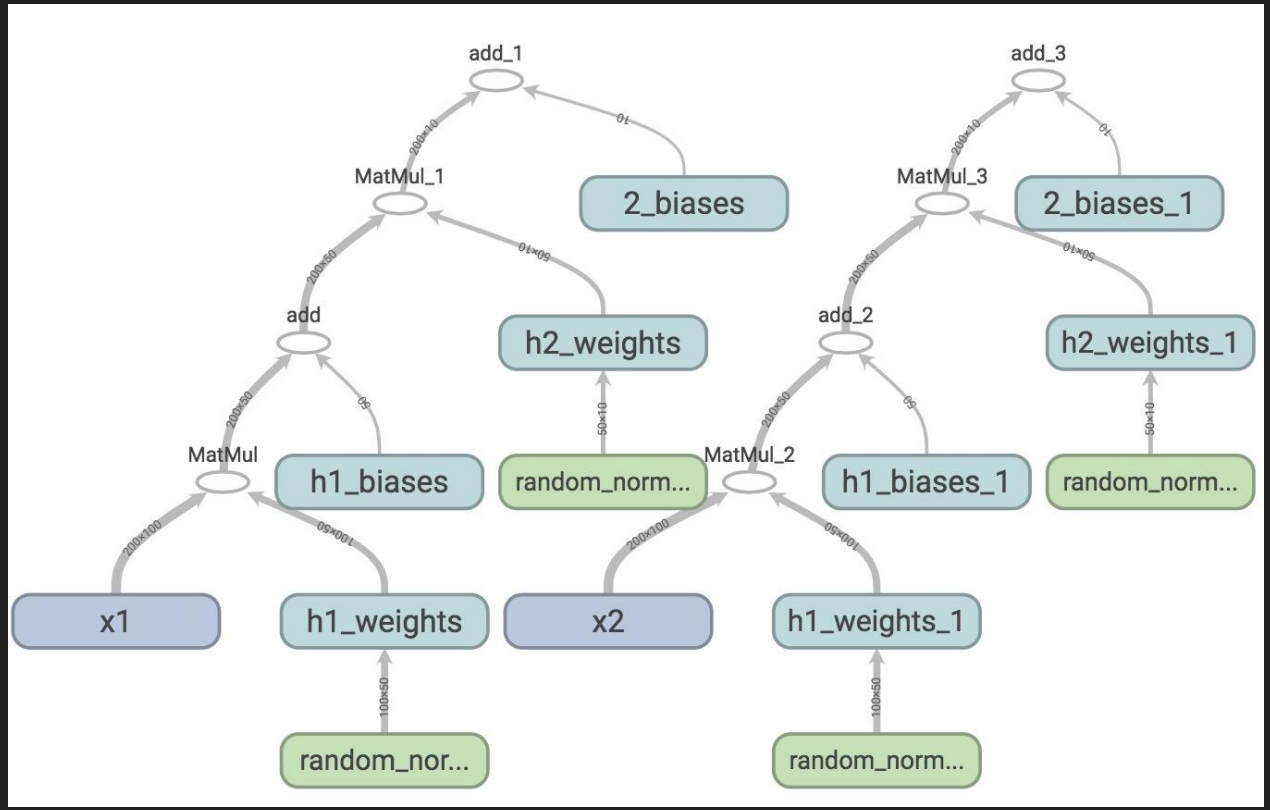
Variable sharing: The problem

```
def two_hidden_layers(x):
    w1 = tf.Variable(tf.random_normal([100, 50]), name='h1_weights')
    b1 = tf.Variable(tf.zeros([50]), name='h1_biases')
    h1 = tf.matmul(x, w1) + b1

    w2 = tf.Variable(tf.random_normal([50, 10]), name='h2_weights')
    b2 = tf.Variable(tf.zeros([10]), name='2_biases')
    logits = tf.matmul(h1, w2) + b2
    return logits
```

What will happen if we make these two calls?

```
logits1 = two_hidden_layers(x1)
logits2 = two_hidden_layers(x2)
```

Sharing Variable: The problem



Two sets of variables are created.

You want all your inputs to use the same weights and biases!

tf.get_variable()

```
tf.get_variable(<name>, <shape>, <initializer>)
```

If a variable with <name> already exists, reuse it

If not, initialize it with <shape> using <initializer>

tf.get_variable()

```
def two_hidden_layers(x):
    assert x.shape.as_list() == [200, 100]
    w1 = tf.get_variable("h1_weights", [100, 50], initializer=tf.random_normal_initializer())
    b1 = tf.get_variable("h1_biases", [50], initializer=tf.constant_initializer(0.0))
    h1 = tf.matmul(x, w1) + b1
    assert h1.shape.as_list() == [200, 50]
    w2 = tf.get_variable("h2_weights", [50, 10], initializer=tf.random_normal_initializer())
    b2 = tf.get_variable("h2_biases", [10], initializer=tf.constant_initializer(0.0))
    logits = tf.matmul(h1, w2) + b2
    return logits
logits1 = two_hidden_layers(x1)
logits2 = two_hidden_layers(x2)
```

tf.get_variable()

```
def two_hidden_layers(x):
    assert x.shape.as_list() == [200, 100]
    w1 = tf.get_variable("h1_weights", [100, 50], initializer=tf.random_normal_initializer())
    b1 = tf.get_variable("h1_biases", [50], initializer=tf.constant_initializer(0.0))
    h1 = tf.matmul(x, w1) + b1
    assert h1.shape.as_list() == [200, 50]
    w2 = tf.get_variable("h2_weights", [50, 10], initializer=tf.random_normal_initializer())
    b2 = tf.get_variable("h2_biases", [10], initializer=tf.constant_initializer(0.0))
    logits = tf.matmul(h1, w2) + b2
    return logits
logits1 = two_hidden_layers(x1)
logits2 = two_hidden_layers(x2)
```

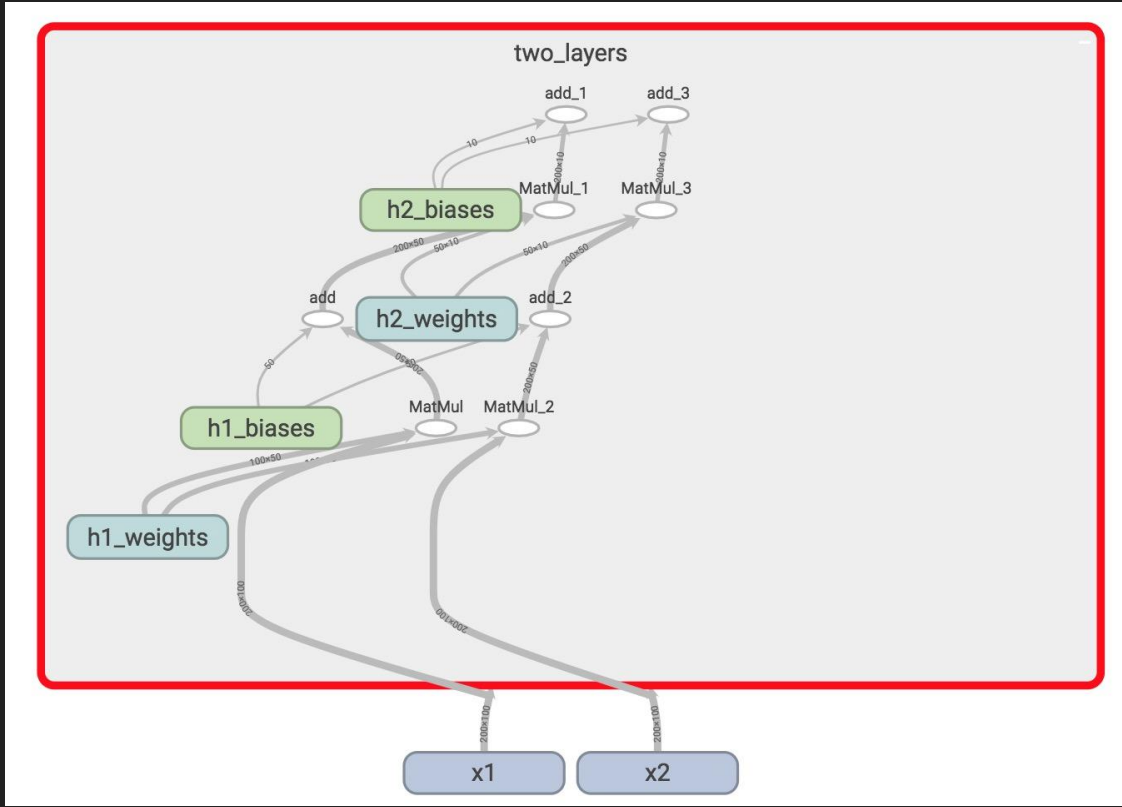
ValueError: Variable h1_weights already exists, disallowed. Did you mean to set reuse=True in VarScope?

tf.variable_scope()

```
def two_hidden_layers(x):
    assert x.shape.as_list() == [200, 100]
    w1 = tf.get_variable("h1_weights", [100, 50], initializer=tf.random_normal_initializer())
    b1 = tf.get_variable("h1_biases", [50], initializer=tf.constant_initializer(0.0))
    h1 = tf.matmul(x, w1) + b1
    assert h1.shape.as_list() == [200, 50]
    w2 = tf.get_variable("h2_weights", [50, 10], initializer=tf.random_normal_initializer())
    b2 = tf.get_variable("h2_biases", [10], initializer=tf.constant_initializer(0.0))
    logits = tf.matmul(h1, w2) + b2
    return logits
with tf.variable_scope('two_layers') as scope:
    logits1 = two_hidden_layers(x1)
    scope.reuse_variables()
    logits2 = two_hidden_layers(x2)
```

Put your variables within a scope and reuse all variables within that scope

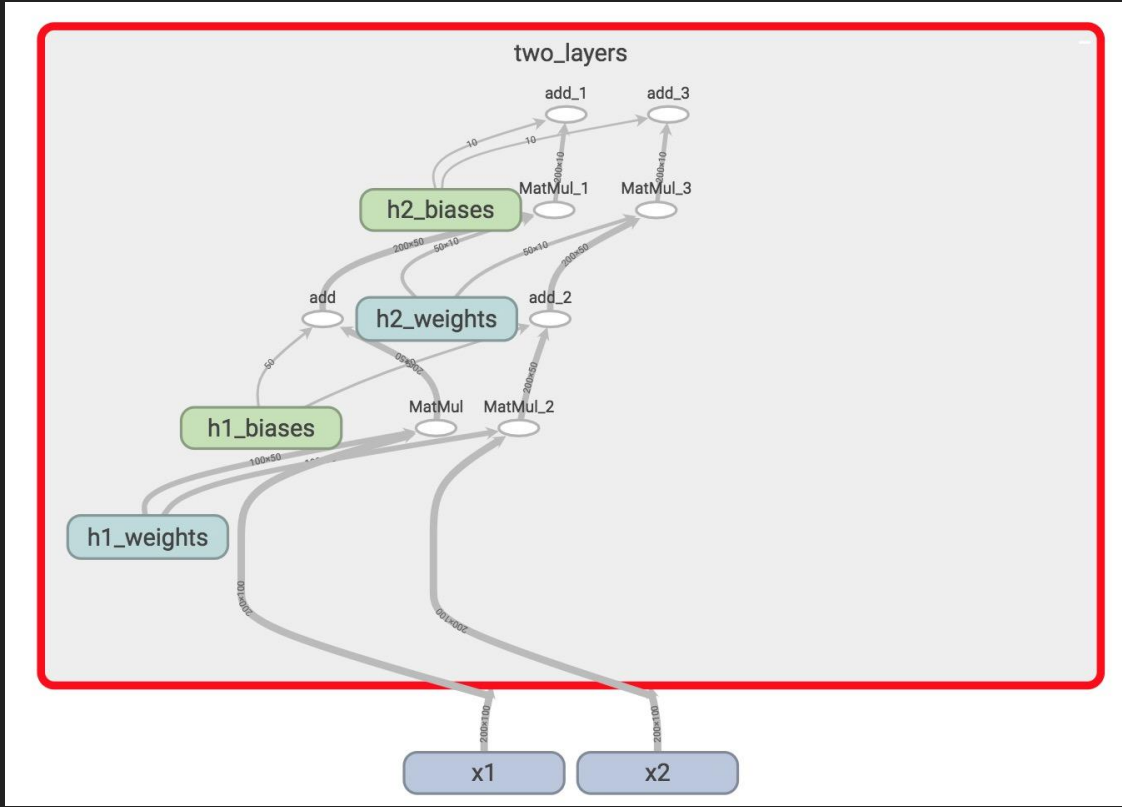
tf.variable_scope()



Only one set of variables, all within the variable scope “two_layers”

They take in two different inputs

tf.variable_scope()



tf.variable_scope implicitly creates a name scope

Reusable code?

```
def two_hidden_layers(x):
    assert x.shape.as_list() == [200, 100]
    w1 = tf.get_variable("h1_weights", [100, 50], initializer=tf.random_normal_initializer())
    b1 = tf.get_variable("h1_biases", [50], initializer=tf.constant_initializer(0.0))
    h1 = tf.matmul(x, w1) + b1
    assert h1.shape.as_list() == [200, 50]
    w2 = tf.get_variable("h2_weights", [50, 10], initializer=tf.random_normal_initializer())
    b2 = tf.get_variable("h2_biases", [10], initializer=tf.constant_initializer(0.0))
    logits = tf.matmul(h1, w2) + b2
    return logits
with tf.variable_scope('two_layers') as scope:
    logits1 = two_hidden_layers(x1)
    scope.reuse_variables()
    logits2 = two_hidden_layers(x2)
```

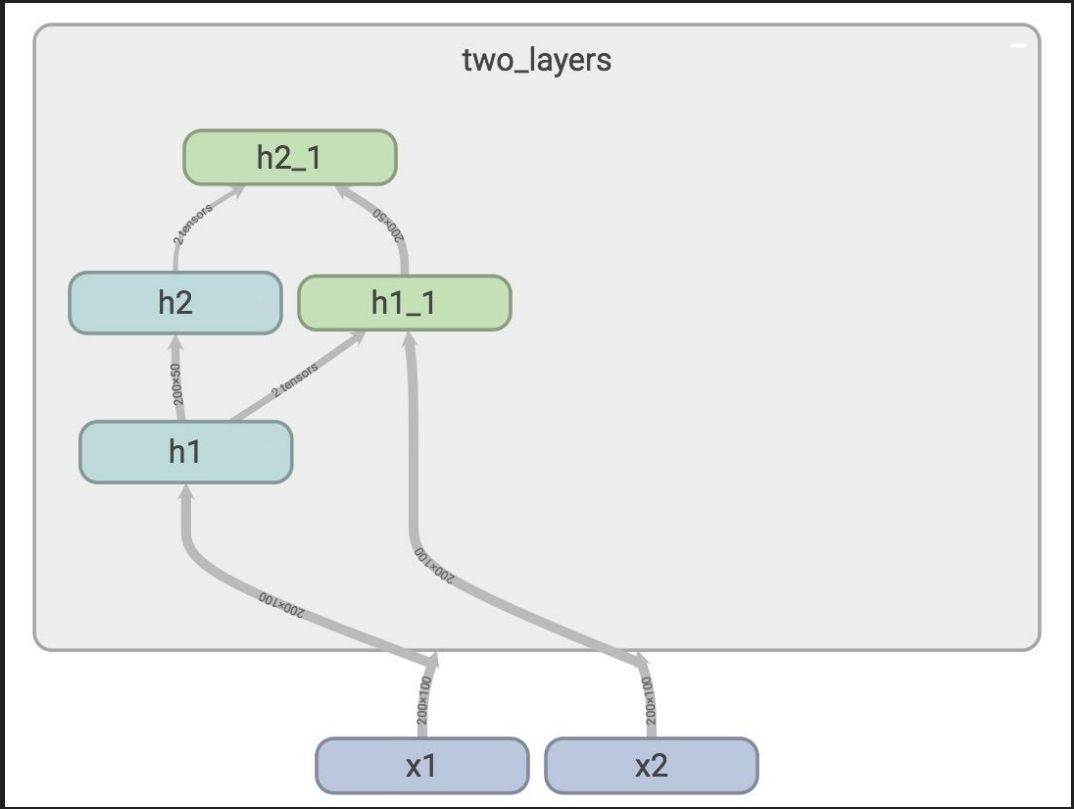
Layer ‘em up

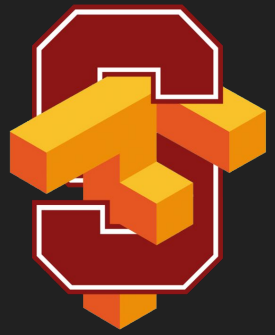
```
def fully_connected(x, output_dim, scope):
    with tf.variable_scope(scope, reuse=tf.AUTO_REUSE) as scope:
        w = tf.get_variable("weights", [x.shape[1], output_dim], initializer=tf.random_normal_initializer())
        b = tf.get_variable("biases", [output_dim], initializer=tf.constant_initializer(0.0))
        return tf.matmul(x, w) + b

def two_hidden_layers(x):
    h1 = fully_connected(x, 50, 'h1')
    h2 = fully_connected(h1, 10, 'h2')

with tf.variable_scope('two_layers') as scope:
    logits1 = two_hidden_layers(x1)
    logits2 = two_hidden_layers(x2)
```

Fetch variables if they already exist

Else, create them

Layer ‘em up


two_layers
h2_1
2 tensors
200×50
h2
h1_1
200×50
2 tensors
h1
200×100
200×100
200×100
200×100
x1
x2



Manage Experiments

tf.train.Saver

saves graph's variables in binary files

Saves sessions, not graphs!

```
tf.train.Saver.save(sess, save_path, global_step=None...)
tf.train.Saver.restore(sess, save_path)
```

Save parameters after 1000 steps

```
# define model
model = SkipGramModel(params)

# create a saver object
saver = tf.train.Saver()

with tf.Session() as sess:
     for step in range(training_steps):
          sess.run([optimizer])

          # save model every 1000 steps
          if (step + 1) % 1000 == 0:
               saver.save(sess,
                          'checkpoint_directory/model_name',
                          global_step=step)
```

Specify the step at which the model is saved

```
# define model
model = SkipGramModel(params)

# create a saver object
saver = tf.train.Saver()

with tf.Session() as sess:
     for step in range(training_steps):
          sess.run([optimizer])

          # save model every 1000 steps
          if (step + 1) % 1000 == 0:
               saver.save(sess,
                          'checkpoint_directory/model_name',
                          global_step=step)
```

Global step

```
global_step = tf.Variable(0, dtype=tf.int32, trainable=False, name='global_step')
```

Very common in
TensorFlow program

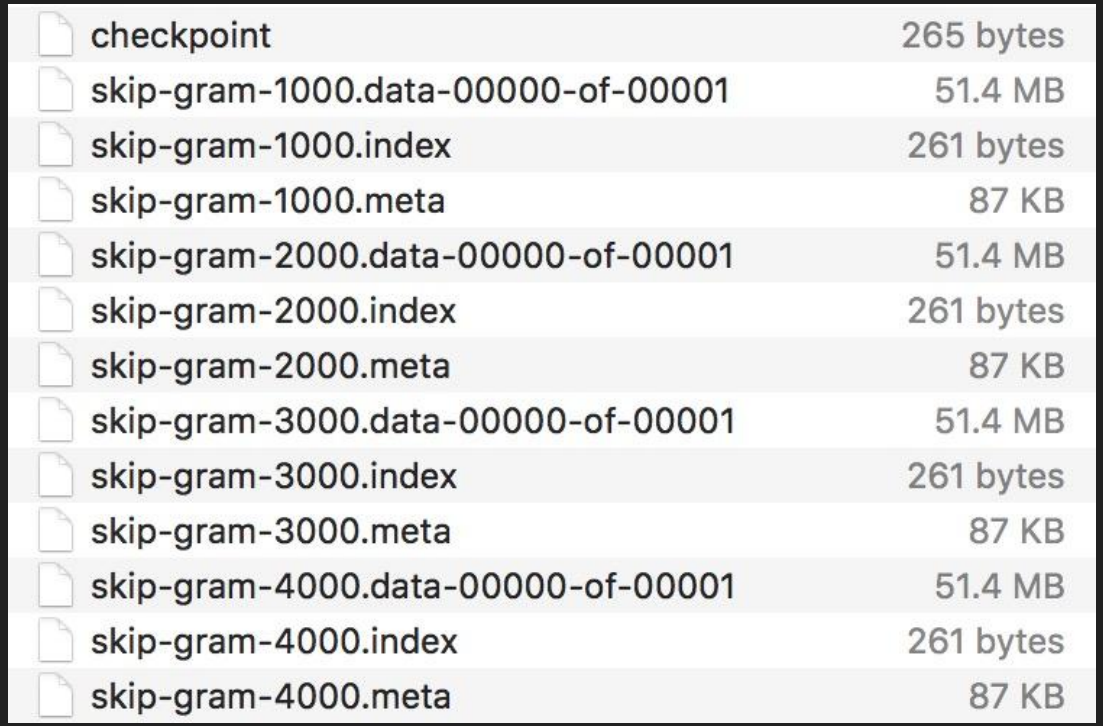
Global step

```
global_step = tf.Variable(0,
                          dtype=tf.int32,
                          trainable=False,
                           name='global_step')

optimizer = tf.train.AdamOptimizer(lr).minimize(loss, global_step=global_step)
```

Need to tell optimizer to increment global step

This can also help your optimizer know when to decay learning rate

Your checkpoints are saved in checkpoint_directory



tf.train.Saver

Only save variables, not graph

Checkpoints map variable names to tensors

tf.train.Saver

Can also choose to save certain variables

```
v1 = tf.Variable(..., name='v1')

v2 = tf.Variable(..., name='v2')
```

You can save your variables in one of three ways:

```
saver = tf.train.Saver({'v1': v1, 'v2': v2})

saver = tf.train.Saver([v1, v2])

saver = tf.train.Saver({v.op.name: v for v in [v1, v2]}) # similar to a dict
```

Restore variables

```
saver.restore(sess, 'checkpoints/name_of_the_checkpoint')

e.g. saver.restore(sess, 'checkpoints/skip-gram-99999')
```

Still need to first build graph

Restore the latest checkpoint

```
# check if there is checkpoint
ckpt = tf.train.get_checkpoint_state(os.path.dirname('checkpoints/checkpoint'))

# check if there is a valid checkpoint path
if ckpt and ckpt.model_checkpoint_path:
     saver.restore(sess, ckpt.model_checkpoint_path)
```

1. checkpoint file keeps track of the latest checkpoint
2. restore checkpoints only when there is a valid checkpoint path

tf.summary

Why matplotlib when you can summarize?

tf.summary

Visualize our summary statistics during our training

`tf.summary.scalar`
`tf.summary.histogram`
`tf.summary.image`

Step 1: create summaries

```
with tf.name_scope("summaries"):
    tf.summary.scalar("loss", self.loss)
    tf.summary.scalar("accuracy", self.accuracy)
    tf.summary.histogram("histogram loss", self.loss)
    summary_op = tf.summary.merge_all()
```

merge them all into one summary op to make managing them easier

Step 2: run them

```
loss_batch, _, summary = sess.run([loss,
                                   optimizer,
                                   summary_op])
```

Like everything else in TF, summaries are ops. For the summaries to be built, you have to run it in a session

Step 3: write summaries to file

```
writer.add_summary(summary, global_step=step)
```

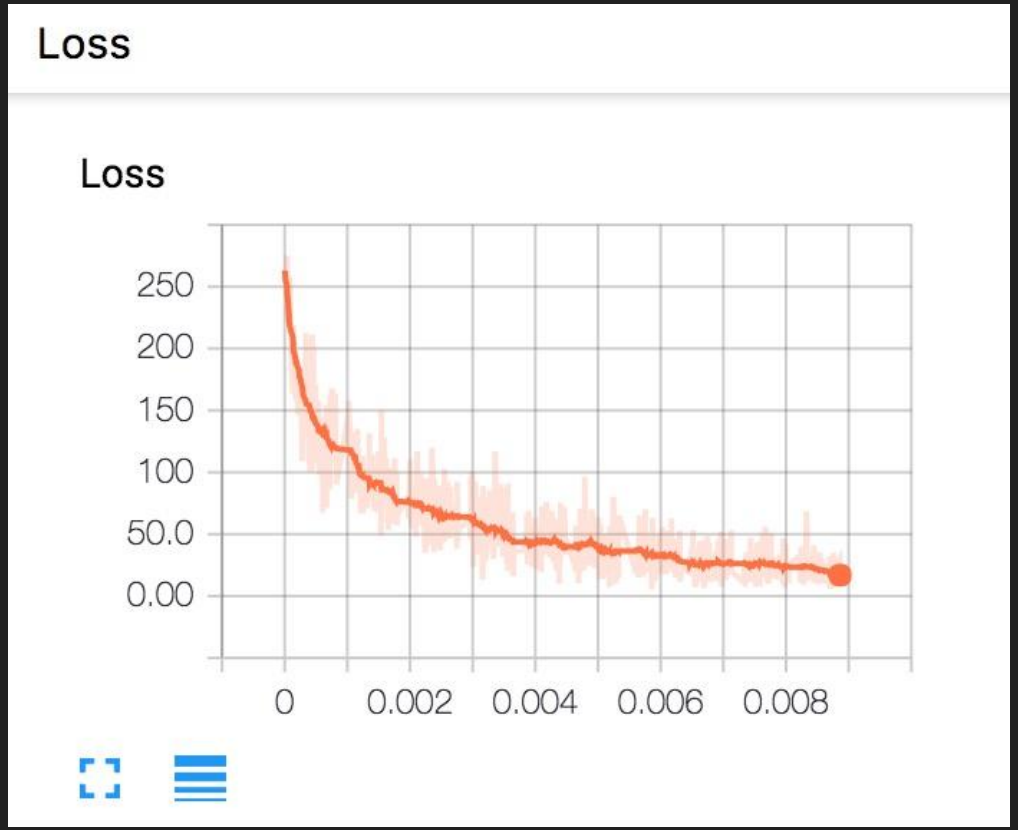
Need global step here so the model knows what summary corresponds to what step

Putting it together

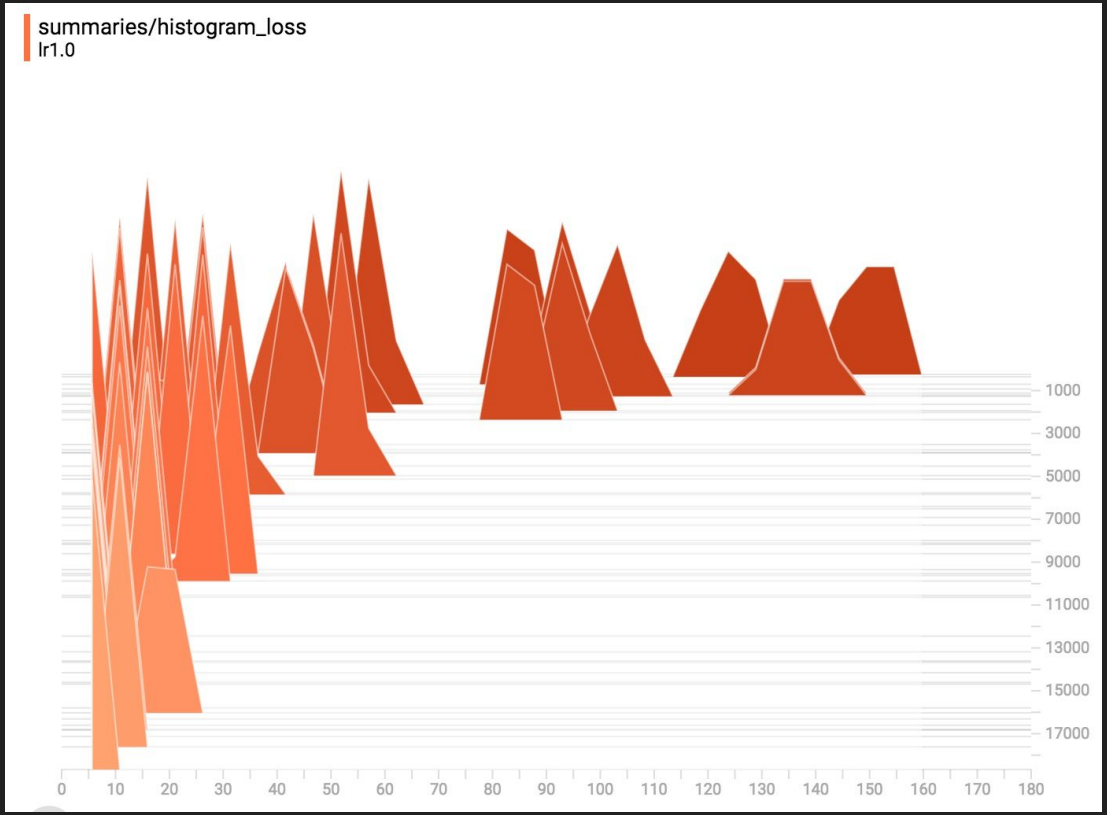
```
tf.summary.scalar("loss", self.loss)
tf.summary.histogram("histogram loss", self.loss)
summary_op = tf.summary.merge_all()

saver = tf.train.Saver() # defaults to saving all variables

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    ckpt = tf.train.get_checkpoint_state(os.path.dirname('checkpoints/checkpoint'))
    if ckpt and ckpt.model_checkpoint_path:
        saver.restore(sess, ckpt.model_checkpoint_path)

    writer = tf.summary.FileWriter('./graphs', sess.graph)
    for index in range(10000):
        ...
        loss_batch, _, summary = sess.run([loss, optimizer, summary_op])
        writer.add_summary(summary, global_step=index)

        if (index + 1) % 1000 == 0:
            saver.save(sess, 'checkpoints/skip-gram', index)
```

See summaries on TensorBoard

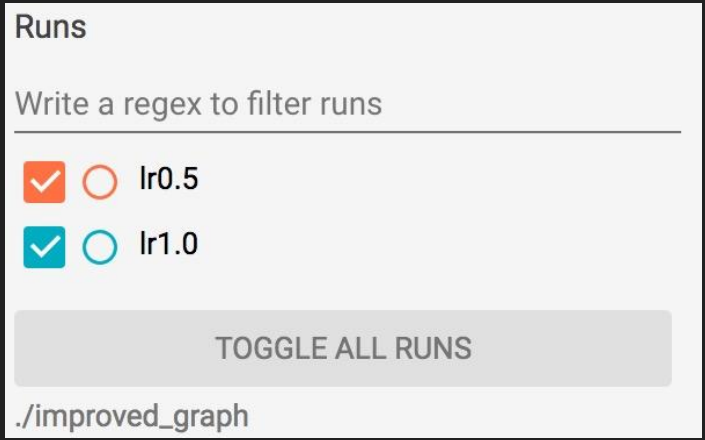
Scalar loss

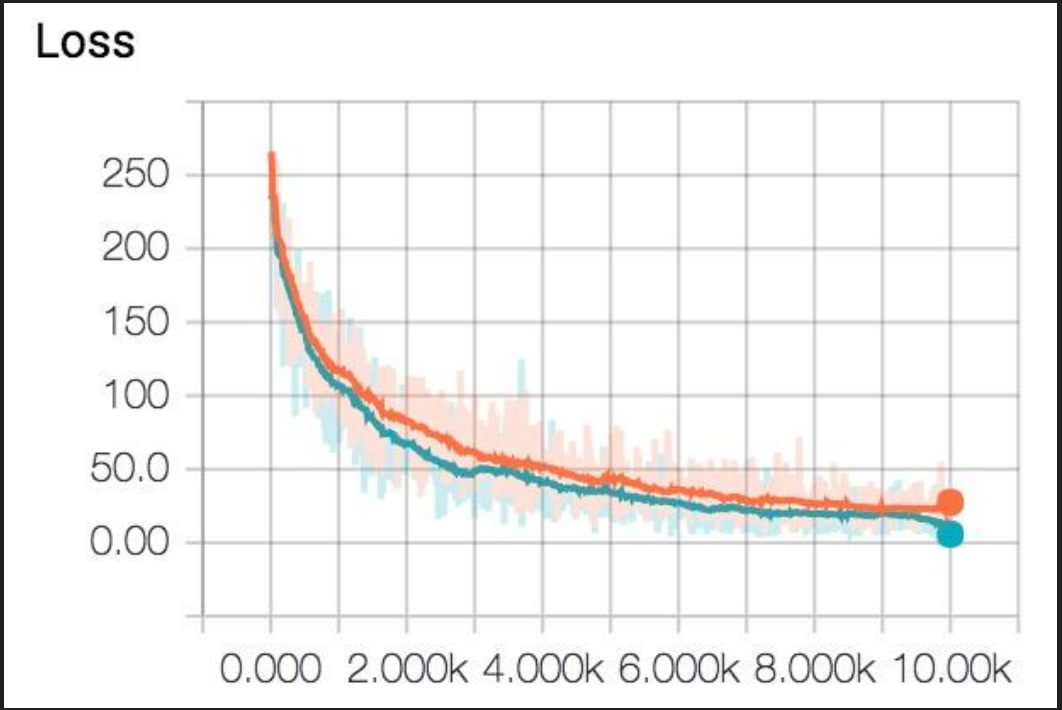


Histogram loss



Toggle run to compare experiments





Control Randomization

Op level random seed

```
my_var = tf.Variable(tf.truncated_normal((-1.0,1.0), stddev=0.1, seed=0))
```

Sessions keep track of random state

```
c = tf.random_uniform([], -10, 10, seed=2)

with tf.Session() as sess:
    print(sess.run(c)) # >> 3.57493
    print(sess.run(c)) # >> -5.97319

----
c = tf.random_uniform([], -10, 10, seed=2)

with tf.Session() as sess:
    print(sess.run(c)) # >> 3.57493

with tf.Session() as sess:
    print(sess.run(c)) # >> 3.57493
```

Each new session restarts the random state

Op level seed: each op keeps its own seed

```
c = tf.random_uniform([], -10, 10, seed=2)
d = tf.random_uniform([], -10, 10, seed=2)

with tf.Session() as sess:
    print(sess.run(c)) # >> 3.57493
    print(sess.run(d)) # >> 3.57493
```

Graph level seed

```
tf.set_random_seed(2)
c = tf.random_uniform([], -10, 10)
d = tf.random_uniform([], -10, 10)

with tf.Session() as sess:
    print(sess.run(c)) # >> -4.00752
    print(sess.run(d)) # >> -2.98339
```

Note that the result is different from op-level seed

Autodiff

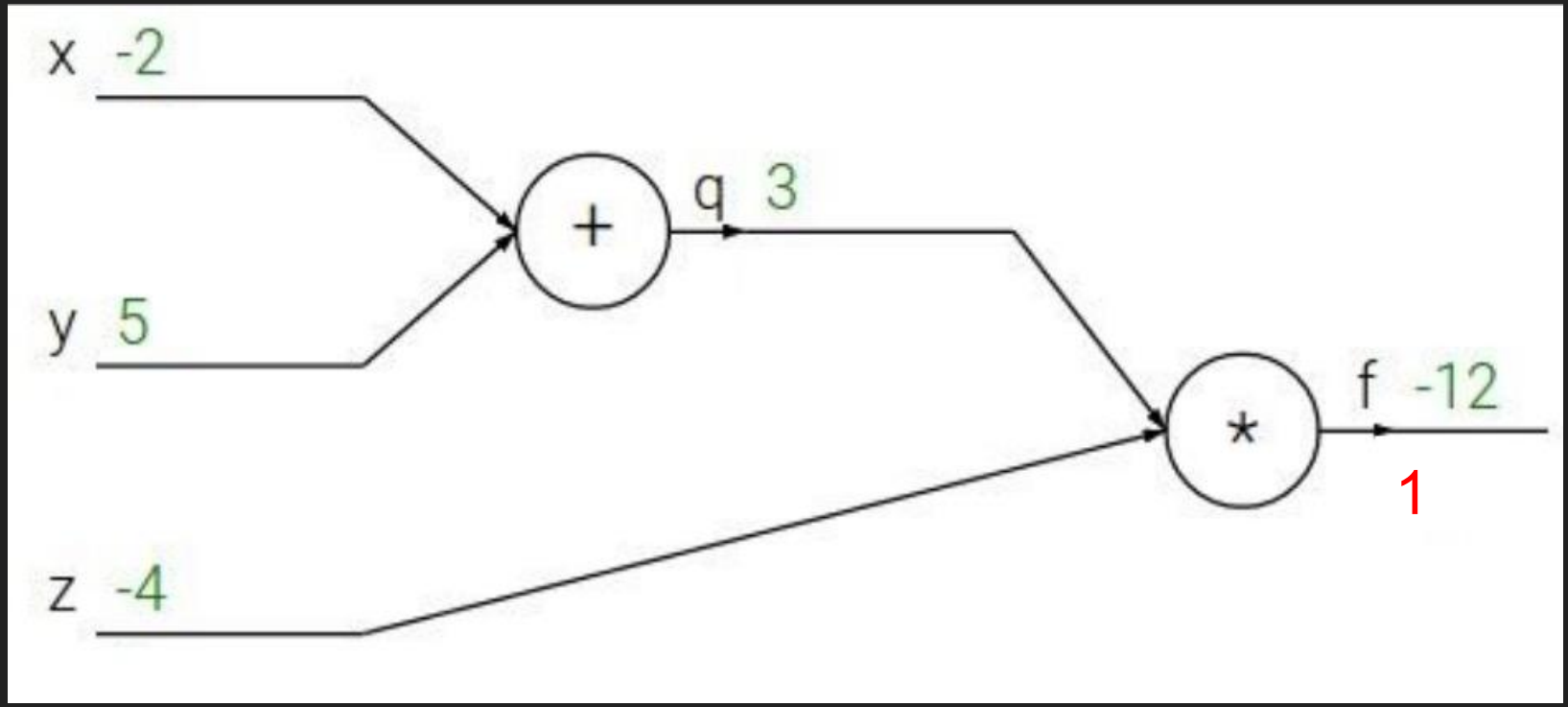
Where are the gradients?

TensorFlow builds the backward path for you!

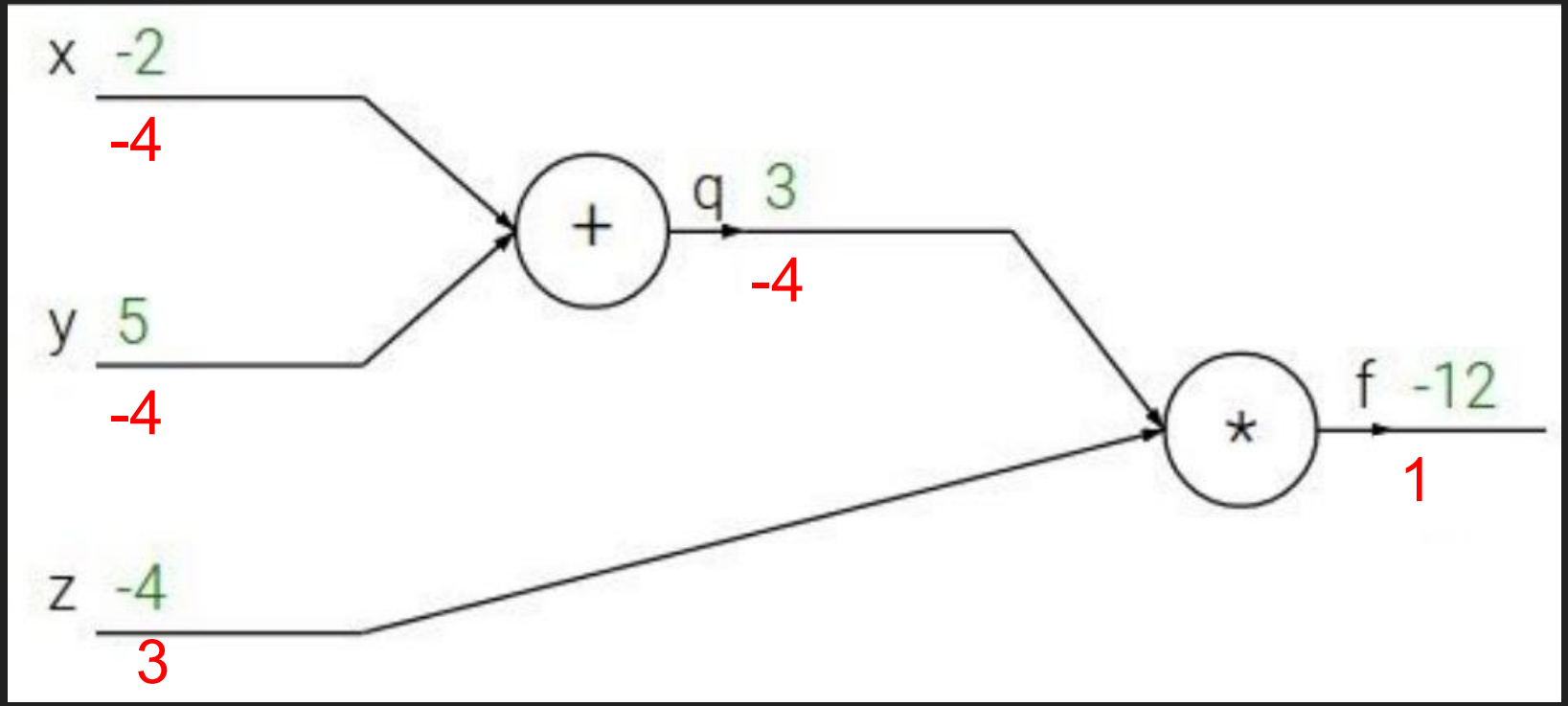
Reverse mode automatic differentiation

- The computation graph makes computing symbolic gradients straightforward
- Chain rule

Can you take gradients for this graph?



Reverse mode automatic differentiation



Reverse mode automatic differentiation

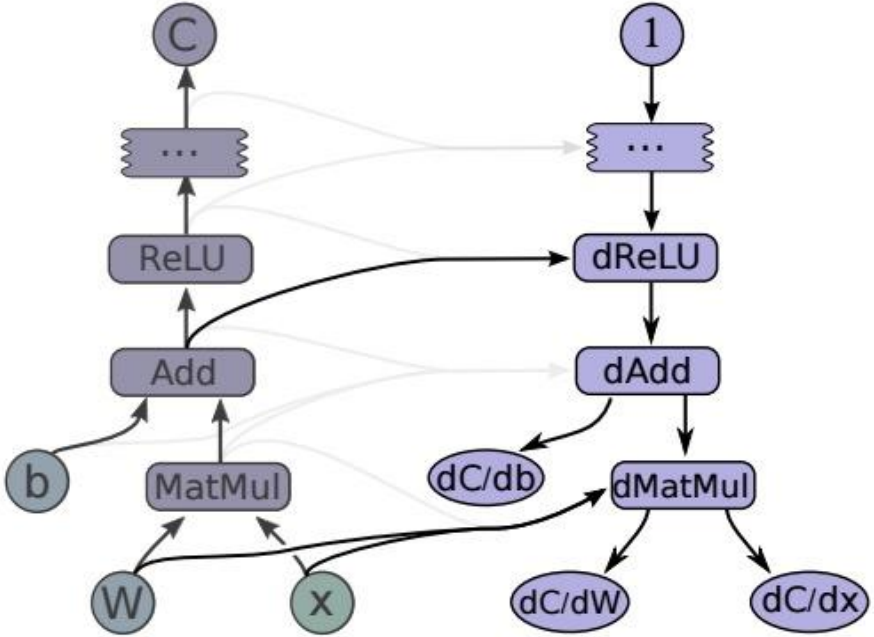


Figure 5: Gradients computed for graph in Figure 2

Reverse mode automatic differentiation

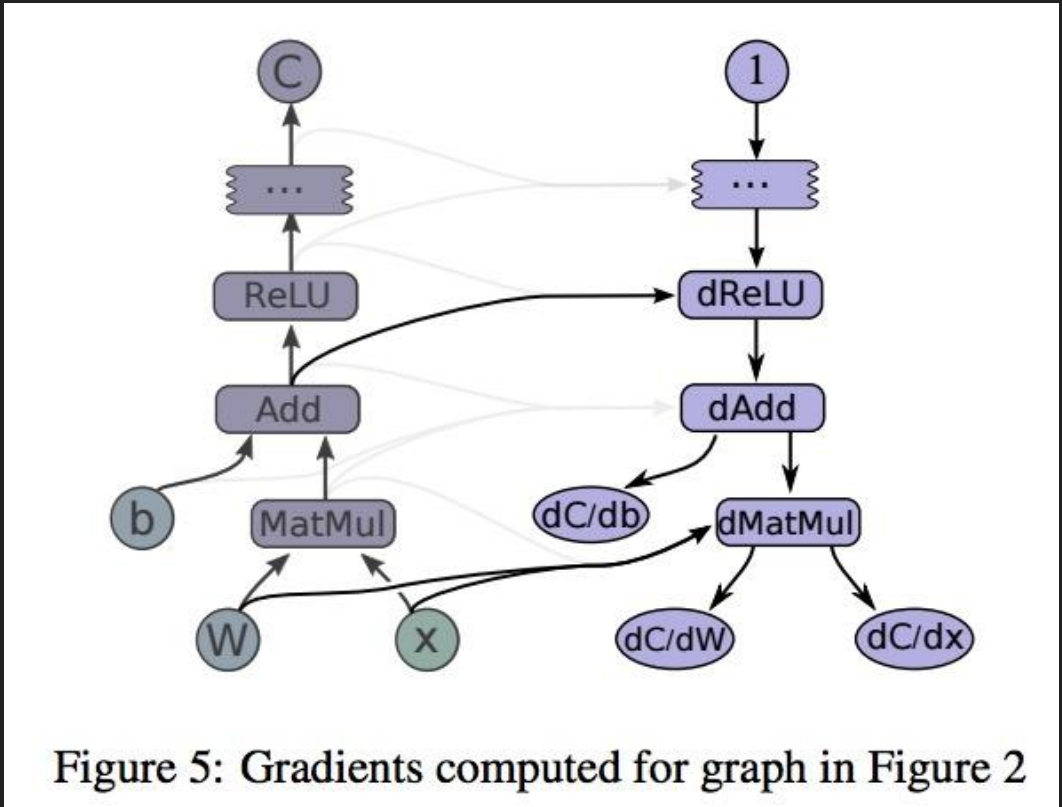


Figure 5: Gradients computed for graph in Figure 2

The backward path takes the same time as forward path

tf.gradients(y, [xs])

Take derivative of y with respect to each tensor

in the list [xs]

tf.gradients(y, [xs])

```
x = tf.Variable(2.0)

y = 2.0 * (x ** 3)

z = 3.0 + y ** 2

grad_z = tf.gradients(z, [x, y])

with tf.Session() as sess:

      sess.run(x.initializer)

      print(sess.run(grad_z)) # >> [768.0, 32.0]

# 768 is the gradient of z with respect to x, 32 with respect to y
```

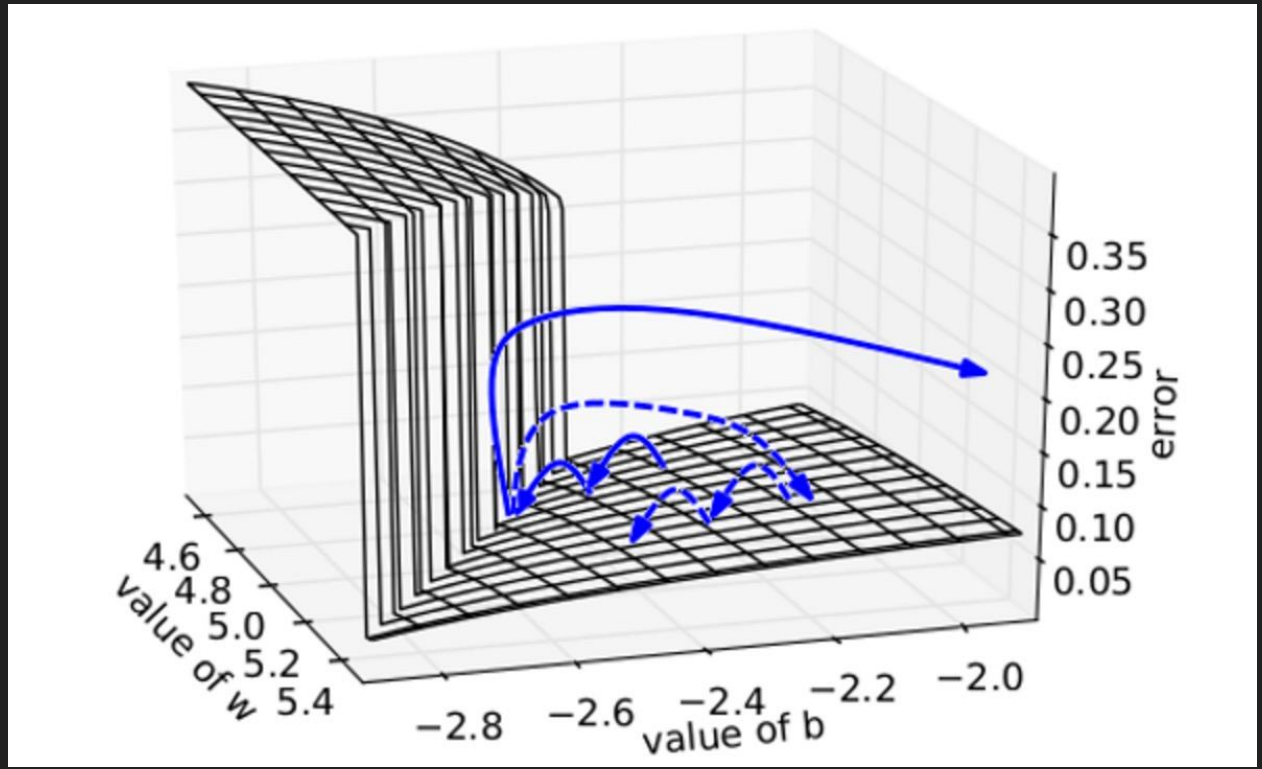
Gradient Computation

```
tf.gradients(ys, xs, grad_ys=None, ...)

tf.stop_gradient(input, name=None)

# prevents the contribution of its inputs to be taken into account

tf.clip_by_value(t, clip_value_min, clip_value_max, name=None)

tf.clip_by_norm(t, clip_norm, axes=None, name=None)
```

Should I still learn to take gradients?

Yes

Vanishing/exploding gradients



Plot by “Understanding the exploding gradient problem”, Pascanu et al. (2012)

Next class

Computer Vision

Convolution

Convnet

No class on Friday, 2/2

Feedback: huyenn@stanford.edu

Thanks!