An introduction to TensorFlow!

Chip Huyen (chiphuyen@cs.stanford.edu)
CS224N
1/25/2018
Agenda

Why TensorFlow
Graphs and Sessions
Linear Regression
tf.data
word2vec
Structuring your model
Managing experiments
Why TensorFlow?

- Flexibility + Scalability
- Popularity

![Stars and Repositories Chart]

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Stars</th>
<th>Repositories</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Keras</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>Caffe</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>PyTorch</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Torch</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Theano</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
import tensorflow as tf
Graphs and Sessions
Data Flow Graphs

TensorFlow separates definition of computations from their execution
Data Flow Graphs

Phase 1: assemble a graph

Phase 2: use a session to execute operations in the graph.
Data Flow Graphs

Phase 1: assemble a graph

Phase 2: use a session to execute operations in the graph.

This might change in the future with eager mode!!

Graph from TensorFlow for Machine Intelligence
What’s a tensor?
What’s a tensor?

An n-dimensional array

0-d tensor: scalar (number)

1-d tensor: vector

2-d tensor: matrix

and so on
import tensorflow as tf
a = tf.add(3, 5)
import tensorflow as tf
a = tf.add(3, 5)

Why x, y?

TF automatically names the nodes when you don’t explicitly name them.
x = 3
y = 5
import tensorflow as tf
a = tf.add(3, 5)

Nodes: operators, variables, and constants
Edges: tensors

Tensors are data. TensorFlow = tensor + flow = data + flow (I know, mind=blown)
import tensorflow as tf
a = tf.add(3, 5)
print(a)

>> Tensor("Add:0", shape=(), dtype=int32)
(Not 8)
How to get the value of a?

Create a `session`, assign it to variable `sess` so we can call it later.

Within the session, evaluate the graph to fetch the value of a.
How to get the value of a?

Create a **session**, assign it to variable `sess` so we can call it later

Within the session, evaluate the graph to fetch the value of a

```python
import tensorflow as tf
a = tf.add(3, 5)
sess = tf.Session()
print(sess.run(a))
sess.close()
```

The session will look at the graph, trying to think: hmm, how can I get the value of a, then it computes all the nodes that leads to a.
How to get the value of a?

Create a **session**, assign it to variable `sess` so we can call it later

Within the session, evaluate the graph to fetch the value of a

```python
import tensorflow as tf
a = tf.add(3, 5)
sess = tf.Session()
print(sess.run(a))
sess.close()
```

The session will look at the graph, trying to think: hmm, how can I get the value of a, then it computes all the nodes that leads to a.
How to get the value of a?

Create a `session`, assign it to variable `sess` so we can call it later

Within the session, evaluate the graph to fetch the value of a

```python
import tensorflow as tf
a = tf.add(3, 5)
sess = tf.Session()
with tf.Session() as sess:
    print(sess.run(a))
sess.close()
```
tf.Session()

A Session object encapsulates the environment in which Operation objects are executed, and Tensor objects are evaluated.
tf.Session()

A Session object encapsulates the environment in which Operation objects are executed, and Tensor objects are evaluated.

Session will also allocate memory to store the current values of variables.
More graph

Visualized by TensorBoard

```python
x = 2
y = 3
op1 = tf.add(x, y)
op2 = tf.multiply(x, y)
op3 = tf.pow(op2, op1)
with tf.Session() as sess:
    op3 = sess.run(op3)
```
Subgraphs

Because we only want the value of pow_op and pow_op doesn’t depend on useless, session won’t compute value of useless → save computation

x = 2
y = 3
add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)
useless = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
    z = sess.run(pow_op)
Subgraphs

Possible to break graphs into several chunks and run them parallelly across multiple CPUs, GPUs, TPUs, or other devices

Example: AlexNet

Graph from *Hands-On Machine Learning with Scikit-Learn and TensorFlow*
Distributed Computation

To put part of a graph on a specific CPU or GPU:

```python
# Creates a graph.
with tf.device('/gpu:2'):
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], name='a')
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], name='b')
    c = tf.multiply(a, b)

# Creates a session with log_device_placement set to True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))

# Runs the op.
print(sess.run(c))
```
y tho
Why graphs

1. Save computation. Only run subgraphs that lead to the values you want to fetch.
Why graphs

1. Save computation. Only run subgraphs that lead to the values you want to fetch.
2. Break computation into small, differential pieces to facilitate auto-differentiation
Why graphs

1. Save computation. Only run subgraphs that lead to the values you want to fetch.
2. Break computation into small, differential pieces to facilitate auto-differentiation
3. Facilitate distributed computation, spread the work across multiple CPUs, GPUs, TPUs, or other devices
Why graphs

1. Save computation. Only run subgraphs that lead to the values you want to fetch.
2. Break computation into small, differential pieces to facilitate auto-differentiation.
3. Facilitate distributed computation, spread the work across multiple CPUs, GPUs, TPUs, or other devices.
4. Many common machine learning models are taught and visualized as directed graphs.

Figure 3: This image captures how multiple sigmoid units are stacked on the right, all of which receive the same input $x$.

A neural net graph from Stanford’s CS224N course.
TensorBoard
import tensorflow as tf

a = tf.constant(2, name='a')
b = tf.constant(3, name='b')
x = tf.add(a, b, name='add')

with tf.Session() as sess:
    print(sess.run(x))
import tensorflow as tf

a = tf.constant(2, name='a')
b = tf.constant(3, name='b')
x = tf.add(a, b, name='add')

writer = tf.summary.FileWriter('./graphs', tf.get_default_graph())

with tf.Session() as sess:
    writer = tf.summary.FileWriter('./graphs', sess.graph)
    print(sess.run(x))

writer.close() # close the writer when you’re done using it

Create the summary writer after graph definition and before running your session

‘graphs’ or any location where you want to keep your event files
Run it

Go to terminal, run:

$ python [yourprogram].py
$ tensorboard --logdir="./graphs" --port 6006  6006 or any port you want

Then open your browser and go to: http://localhost:6006/
Main Graph

Auxiliary Nodes

add

Graph

Namespace*
OpNode*
Unconnected series*
Connected series*
Constant
Summary
Dataflow edge
Control dependency edge
Reference edge
Constants, Sequences, Variables, Ops
import tensorflow as tf

a = tf.constant([2, 2], name='a')
b = tf.constant([[0, 1], [2, 3]], name='b')
x = tf.multiply(a, b, name='mul')

with tf.Session() as sess:
    print(sess.run(x))

#  >>  [[0 2]
#  [4 6]]
Tensors filled with a specific value

```
tf.zeros([2, 3], tf.int32) ==> [[0, 0, 0], [0, 0, 0]]

# input_tensor is [[0, 1], [2, 3], [4, 5]]

tf.zeros_like(input_tensor) ==> [[0, 0], [0, 0], [0, 0]]

similar to NumPy

tf.fill([2, 3], 8) ==> [[8, 8, 8], [8, 8, 8]]
```
Constants as sequences

\[
\text{tf.lin_space(start, stop, num, name=None)}
\]
\[
\text{tf.lin_space(10.0, 13.0, 4)} \implies [10. 11. 12. 13.]
\]

\[
\text{tf.range(start, limit=None, delta=1, dtype=None, name='range')}
\]
\[
\text{tf.range(3, 18, 3)} \implies [3 6 9 12 15]
\]
\[
\text{tf.range(5)} \implies [0 1 2 3 4]
\]

NOT THE SAME AS NUMPY SEQUENCES

Tensor objects are not iterable

\[
\text{for \_ in tf.range(4): \# TypeError}
\]
Randomly Generated Constants

tf.random_normal
tf.truncated_normal
tf.random_uniform
tf.random_shuffle
tf.random_crop
tf.multinomial
tf.random_gamma
Randomly Generated Constants

tf.set_random_seed(seed)
TF vs NP Data Types

TensorFlow integrates seamlessly with NumPy

tf.int32 == np.int32  # ⇒ True

Can pass numpy types to TensorFlow ops

tf.ones([2, 2], np.float32)  # ⇒ [[1.0 1.0], [1.0 1.0]]

For tf.Session.run(fetches): if the requested fetch is a Tensor, output will be a NumPy ndarray.

sess = tf.Session()
a = tf.zeros([2, 3], np.int32)
print(type(a))  # ⇒ <class 'tensorflow.python.framework.ops.Tensor'>
a_out = sess.run(a)
print(type(a))  # ⇒ <class 'numpy.ndarray'>
Use TF DType when possible

- Python native types: TensorFlow has to infer Python type
Use TF DType when possible

- Python native types: TensorFlow has to infer Python type
- NumPy arrays: NumPy is not GPU compatible
What’s wrong with constants?

Not trainable
my_const = tf.constant([1.0, 2.0], name="my_const")

with tf.Session() as sess:
  print(sess.graph.as_graph_def())
Constants are stored in graph definition

This makes loading graphs expensive when constants are big
Constants are stored in graph definition

This makes loading graphs expensive when constants are big

Only use constants for primitive types.

Use variables or readers for more data that requires more memory
Variables

# create variables with tf.Variable
s = tf.Variable(2, name="scalar")
m = tf.Variable([[0, 1], [2, 3]], name="matrix")
W = tf.Variable(tf.zeros([784,10]))

# create variables with tf.get_variable
s = tf.get_variable("scalar", initializer=tf.constant(2))
m = tf.get_variable("matrix", initializer=tf.constant([[0, 1], [2, 3]]))
W = tf.get_variable("big_matrix", shape=(784, 10), initializer=tf.zeros_initializer())
You have to **initialize** your variables

The easiest way is initializing all variables at once:

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

Initializer is an op. You need to execute it within the context of a session.
You have to **initialize** your variables

The easiest way is initializing all variables at once:

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

Initialize only a subset of variables:

```python
with tf.Session() as sess:
    sess.run(tf.variables_initializer([a, b]))
```
You have to **initialize** your variables

The easiest way is initializing all variables at once:

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

Initialize only a subset of variables:

```python
with tf.Session() as sess:
    sess.run(tf.variables_initializer([a, b]))
```

Initialize a single variable

```python
W = tf.Variable(tf.zeros([784,10]))
with tf.Session() as sess:
    sess.run(W.initializer)
```
Eval() a variable

# W is a random 700 x 100 variable object
W = tf.Variable(tf.truncated_normal([700, 10]))
with tf.Session() as sess:
    sess.run(W.initializer)
    print(W)

>> Tensor("Variable/read:0", shape=(700, 10), dtype=float32)
W = tf.Variable(10)
W.assign(100)
with tf.Session() as sess:
    sess.run(W.initializer)
    print(W.eval())  # >> ????
tf.Variable.assign()

W = tf.Variable(10)
W.assign(100)

with tf.Session() as sess:
    sess.run(W.initializer)
    print(W.eval())  # >> 10

Ugh, why?
W = tf.Variable(10)
W.assign(100)

with tf.Session() as sess:
    sess.run(W.initializer)
    print(W.eval())  # >> 10

W.assign(100) creates an assign op.
That op needs to be executed in a session
to take effect.
tf.Variable.assign()

W = tf.Variable(10)
W.assign(100)
with tf.Session() as sess:
    sess.run(W.initializer)
    print(W.eval())  # >> 10

--------

W = tf.Variable(10)
assign_op = W.assign(100)
with tf.Session() as sess:
    sess.run(W.initializer)
    sess.run(assign_op)
    print(W.eval())  # >> 100
A quick reminder

A TF program often has 2 phases:
1. Assemble a graph
2. Use a session to execute operations in the graph.
A TF program often has 2 phases:

1. Assemble a graph
2. Use a session to execute operations in the graph.

⇒ Assemble the graph first without knowing the values needed for computation
A TF program often has 2 phases:
1. Assemble a graph
2. Use a session to execute operations in the graph.

⇒ Assemble the graph first without knowing the values needed for computation

**Analogy:**
Define the function $f(x, y) = 2 \times x + y$ without knowing value of $x$ or $y$. $x, y$ are placeholders for the actual values.
Why placeholders?

We, or our clients, can later supply their own data when they need to execute the computation.
Placeholders

tf.placeholder(dtype, shape=None, name=None)

# create a placeholder for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])

b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b  # short for tf.add(a, b)

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

```
tf.placeholder(dtype, shape=None, name=None)

# create a placeholder for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])

b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b  # short for tf.add(a, b)

with tf.Session() as sess:
    print(sess.run(c))  # >> InvalidArgumentError: a doesn't an actual value
```
Supplement the values to placeholders using a dictionary
tf.placeholder(dtype, shape=None, name=None)

# create a placeholder for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])

b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b  # short for tf.add(a, b)

with tf.Session() as sess:
    print(sess.run(c, feed_dict={a: [1, 2, 3]}))  # the tensor a is the key, not the string ‘a’
    # >> [6, 7, 8]
Placeholders

\( \text{tf.placeholder(dtype, \text{shape=None}, \text{name=None})} \)

# create a placeholder for a vector of 3 elements, type tf.float32
a = \text{tf.placeholder(tf.float32, shape=[3])}

b = \text{tf.constant([5, 5, 5], tf.float32)}

# use the placeholder as you would a constant or a variable
c = a + b  # short for tf.add(a, b)

with \text{tf.Session()}\ as \text{sess:}
    \text{print(sess.run(c, feed_dict={a: [1, 2, 3]}))}

# >> [6, 7, 8]

\textbf{Quirk:}
shape=\text{None} means that tensor of any shape will be accepted as value for placeholder.

shape=\text{None} is easy to construct graphs and great when you have different batch sizes, but nightmarish for debugging
Placeholders

\textbf{tf.placeholder(dtype, shape=None, name=None)}

# create a placeholder of type float 32-bit, shape is a vector of 3 elements
a = tf.placeholder(tf.float32, shape=[3])

# create a constant of type float 32-bit, shape is a vector of 3 elements
b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b  # Short for tf.add(a, b)

with tf.Session() as sess:
    print(sess.run(c, {a: [1, 2, 3]}))

# >> [6, 7, 8]

\textbf{Quirk:}
shape=None also breaks all following shape inference, which makes many ops not work because they expect certain rank.
Placeholders are valid ops

```python
tf.placeholder(dtype, shape=None, name=None)

# create a placeholder of type float 32-bit, shape is a vector of 3 elements
a = tf.placeholder(tf.float32, shape=[3])

# create a constant of type float 32-bit, shape is a vector of 3 elements
b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b  # Short for tf.add(a, b)

with tf.Session() as sess:
    print(sess.run(c, {a: [1, 2, 3]}))

# >> [6, 7, 8]
```
What if want to feed multiple data points in?

You have to do it one at a time

```python
with tf.Session() as sess:
    for a_value in list_of_values_for_a:
        print(sess.run(c, {a: a_value}))
```
Extremely helpful for testing
Feed in dummy values to test parts of a large graph
Linear Regression in TensorFlow
Model the linear relationship between:
- dependent variable Y
- explanatory variables X
World Development Indicators dataset

X: birth rate
Y: life expectancy
190 countries
Want

Find a linear relationship between X and Y to predict Y from X
Model

Inference: $Y_{\text{predicted}} = w \times X + b$

Mean squared error: $E[(y - Y_{\text{predicted}})^2]$
Interactive Coding

birth_life_2010.txt
Interactive Coding

linreg_starter.py

birth_life_2010.txt
Phase 1: Assemble our graph
Step 1: Read in data

I already did that for you
Step 2: Create placeholders for inputs and labels

tf.placeholder(dtype, shape=None, name=None)
Step 3: Create weight and bias

tf.get_variable(
    name,
    shape=None,
    dtype=None,
    initializer=None,
    ...)

No need to specify shape if using constant initializer
Step 4: Inference

\[ Y_{predicted} = w \times X + b \]
Step 5: Specify loss function

\[ \text{loss} = \text{tf.square}(Y - Y_{\text{predicted}}, \text{name}='\text{loss}') \]
Step 6: Create optimizer

```python
opt = tf.train.GradientDescentOptimizer(learning_rate=0.001)
optimizer = opt.minimize(loss)
```
Phase 2: Train our model

Step 1: Initialize variables

Step 2: Run optimizer

(use a feed_dict to feed data into X and Y placeholders)
Write log files using a FileWriter

```python
writer = tf.summary.FileWriter('./graphs/linear_reg', sess.graph)
```
See it on TensorBoard

Step 1: $ python linreg_starter.py

Step 2: $ tensorboard --logdir='./graphs'
tf.data
Pro: put the data processing outside TensorFlow, making it easy to do in Python

Cons: users often end up processing their data in a single thread and creating data bottleneck that slows execution down.
data, n_samples = utils.read_birth_life_data(DATA_FILE)

X = tf.placeholder(tf.float32, name='X')
Y = tf.placeholder(tf.float32, name='Y')

...  
with tf.Session() as sess:
  ...
    # Step 8: train the model
    for i in range(100):  # run 100 epochs
        for x, y in data:
            # Session runs train_op to minimize loss
            sess.run(optimizer, feed_dict={X: x, Y:y})
tf.data

Instead of doing inference with placeholders and feeding in data later, do inference directly with data
tf.data

tf.data.Dataset

tf.data.Iterator
Store data in tf.data.Dataset

- `tf.data.Dataset.from_tensor_slices((features, labels))`
- `tf.data.Dataset.from_generator(gen, output_types, output_shapes)`
Store data in tf.data.Dataset

tf.data.Dataset.from_tensor_slices((features, labels))

dataset = tf.data.Dataset.from_tensor_slices((data[:,0], data[:,1]))
Store data in tf.data.Dataset

tf.data.Dataset.from_tensor_slices((features, labels))
dataset = tf.data.Dataset.from_tensor_slices((data[:,0], data[:,1]))
print(dataset.output_types)  # >> (tf.float32, tf.float32)
print(dataset.output_shapes)  # >> (TensorShape([]), TensorShape([]))
Can also create Dataset from files

- `tf.data.TextLineDataset(filenames)`
- `tf.data.FixedLengthRecordDataset(filenames)`
- `tf.data.TFRecordDataset(filenames)`
tf.data.Iterator

Create an iterator to iterate through samples in Dataset
tf.data.Iterator

- `iterator = dataset.make_one_shot_iterator()`
- `iterator = dataset.make_initializable_iterator()`
tf.data.Iterator

- **iterator = dataset.make_one_shot_iterator()**
  Iterates through the dataset exactly once. No need to initialization.

- **iterator = dataset.make_initializable_iterator()**
  Iterates through the dataset as many times as we want. Need to initialize with each epoch.
tf.data.Iterator

```
iterator = dataset.make_one_shot_iterator()
X, Y = iterator.get_next()  # X is the birth rate, Y is the life expectancy

with tf.Session() as sess:
    print(sess.run([X, Y]))  # >> [1.822, 74.82825]
    print(sess.run([X, Y]))  # >> [3.869, 70.81949]
    print(sess.run([X, Y]))  # >> [3.911, 72.15066]
```
tf.data.Iterator

```
iterator = dataset.make_initializable_iterator()

...

for i in range(100):
    sess.run(iterator.initializer)
    total_loss = 0
    try:
        while True:
            sess.run([optimizer])
            sess.run([optimizer])
    except tf.errors.OutOfRangeError:
        pass
```
Handling data in TensorFlow

dataset = dataset.shuffle(1000)

dataset = dataset.repeat(100)

dataset = dataset.batch(128)

dataset = dataset.map(lambda x: tf.one_hot(x, 10))
# convert each element of dataset to one_hot vector
Does tf.data really perform better?
Does tf.data really perform better?

With placeholder: 9.05271519 seconds

With tf.data: 6.12285947 seconds
Should we always use tf.data?

- For prototyping, feed dict can be faster and easier to write (pythonic)
- tf.data is tricky to use when you have complicated preprocessing or multiple data sources
- NLP data is normally just a sequence of integers. In this case, transferring the data over to GPU is pretty quick, so the speedup of tf.data isn't that large
How does TensorFlow know what variables to update?
Optimizers
Optimizer

```python
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(loss)

_, l = sess.run([optimizer, loss], feed_dict={X: x, Y:y})
```
Optimizer

```python
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(loss)
_, l = sess.run([optimizer, loss], feed_dict={X: x, Y: y})
```

Session looks at all trainable variables that loss depends on and update them
Optimizer

Session looks at all **trainable** variables that optimizer depends on and update them
Trainable variables

tf.Variable(initial_value=None, trainable=True,...)

Specify if a variable should be trained or not
By default, all variables are trainable
List of optimizers in TF

tf.train.GradientDescentOptimizer

tf.train.AdagradOptimizer

tf.train.MomentumOptimizer

tf.train.AdamOptimizer

tf.train.FtrlOptimizer

tf.train.RMSPropOptimizer

...
word2vec skip-gram in TensorFlow
Embedding Lookup

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

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

```
import tensorflow as tf

params = tf.constant([17, 24, 1, 23, 5, 7, 4, 6, 13, 10, 12, 19, 11, 18, 25])

ids = tf.constant([1])

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

- Negative sampling is a simplified model of Noise Contrastive Estimation (NCE)
- NCE guarantees approximation to softmax. Negative sampling doesn’t
NCE Loss

tf.nn.nce_loss(
    weights,
    biases,
    labels,
    inputs,
    num_sampled,
    num_classes,
    ...)

Interactive Coding

word2vec_utils.py

word2vec_starter.py
Embedding visualization
Interactive Coding

word2vec_visualize.py
Visualize vector representation of anything
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

# ...


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='h2_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

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

What will happen if we make these two calls?
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()

```python
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)
```
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?
```python
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()
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)
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)
Layer ‘em up
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

```python
# 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)
```
# 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`

- checkpoint: 265 bytes
- skip-gram-1000.data-00000-of-00001: 51.4 MB
- skip-gram-1000.index: 261 bytes
- skip-gram-1000.meta: 87 KB
- skip-gram-2000.data-00000-of-00001: 51.4 MB
- skip-gram-2000.index: 261 bytes
- skip-gram-2000.meta: 87 KB
- skip-gram-3000.data-00000-of-00001: 51.4 MB
- skip-gram-3000.index: 261 bytes
- skip-gram-3000.meta: 87 KB
- skip-gram-4000.data-00000-of-00001: 51.4 MB
- skip-gram-4000.index: 261 bytes
- skip-gram-4000.meta: 87 KB
tf.train.Saver

Only save variables, not graph

Checkpoints map variable names to tensors
Can also choose to save certain variables

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

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

You can save your variables in one of three ways:

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

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

Still need to first build graph
# 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
with tf.name_scope("summar...):  
    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
Questions?

Feedback: chiphuyen@cs.stanford.edu

Thanks!