Applied Deep Learning

Lecture 6 • Feb 28th, 2019

Administrative stuff

Assignments

- Assignment #3 released, due 3/14

Projects

- Projects released, due **5/19** (please not later than this).

- Custom project proposal due 3/11.

Syllabus updated

Agenda

Concepts

- Dropout
- Embeddings and the embedding projector

TensorFlow

- TensorBoard (now works in Jupyter notebooks!)
- TF1 (for historical reasons) and TF2 (**tf.function and AutoGraph**)

Assignment 3 walkthrough

TensorBoard

Demo

- Docs (skip everything else, the other stuff is mostly for TF 1.0)

How to

- Visualize loss
- Results of experiments
- The graph
- Tips (how to start reset everything to a clean state / delete logs)

Best resources to start with

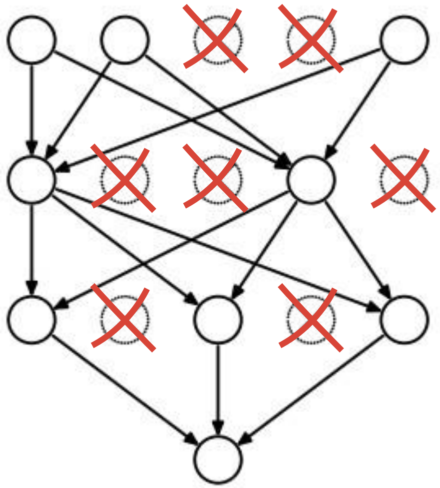
- [TensorBoard 2.0 docs]

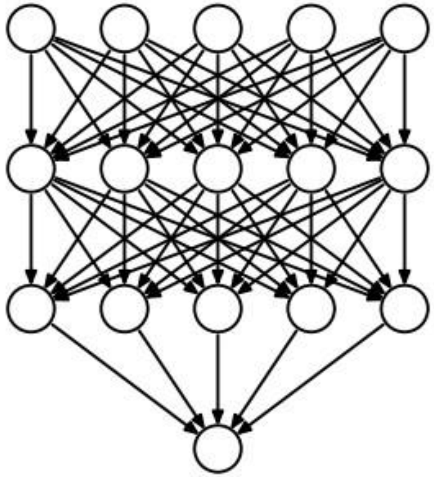
Important: skip anything else (this is new, most existing docs are for TF v1.0)

Dropout in a nutshell

Dropout

Note: used to drop neurons in hidden layers (not usually inputs to the network as shown in the diagram).





Dropout rate is the fraction of the activations that are zeroed out; it's usually set between 0.2 and 0.5 (left).

No activations are dropped at testing time (right).

Quick discussion: Does anyone who hasn't seen this before have an idea why it might work?

[Dropout: A Simple Way to Prevent Neural Networks from Overfitting]

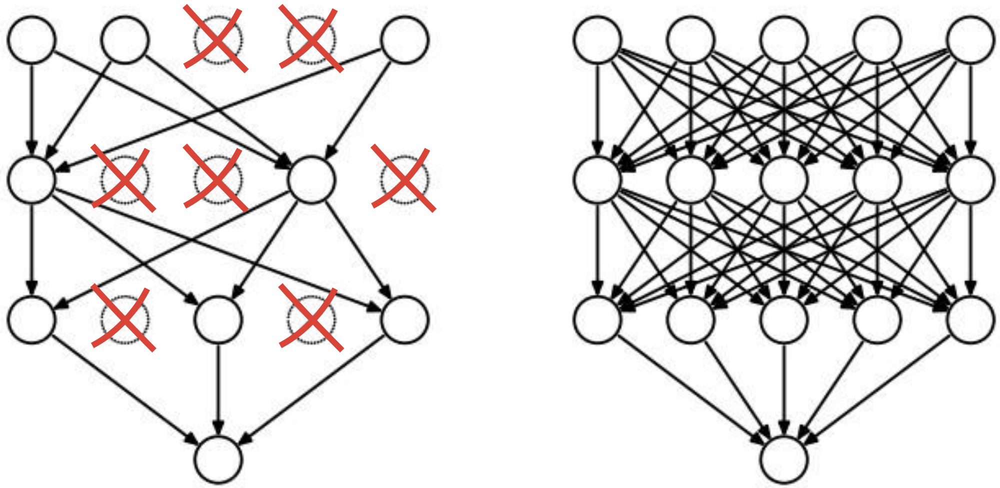
Reddit AMA with Hinton

"...I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting."

Reddit

Dropout

Note: used to drop neurons in hidden layers (not usually inputs to the network as shown in the diagram).



Dropout rate is the fraction of the activations that are zeroed out; it's usually set between 0.2 and 0.5 (left).

No activations are dropped at testing time (right).

Answer: a) Forces the model to learn redundant representations / reduces it's capacity / can only "remember" most important patterns. b) A little bit like ensembling (we're loosely training a different model on each iteration).

[Dropout: A Simple Way to Prevent Neural Networks from Overfitting]

Implementation (Sequential)

```
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(rate=0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(num_classes, activation='softmax'))
```

Common pattern: use following Dense or Flatten layers.

How many Dropout layers should you use? Hyperparameter.

Best bet: start with just one, right before the output layer.

[Keras Layers - Dropout](#)

Implementation (Subclassing)

```python
class MyModel(tf.keras.Model):
  def __init__(self):
    self.dense1 = layers.Dense(100)
    self.dropout = layers.Dropout(0.2)
    # ...

  def call(self, x, training=True):
    x = self.dense1(x)
    x = self.dropout(x, training=training)
    # ...
    return x

preds = model(data, training=False) # at test time
```

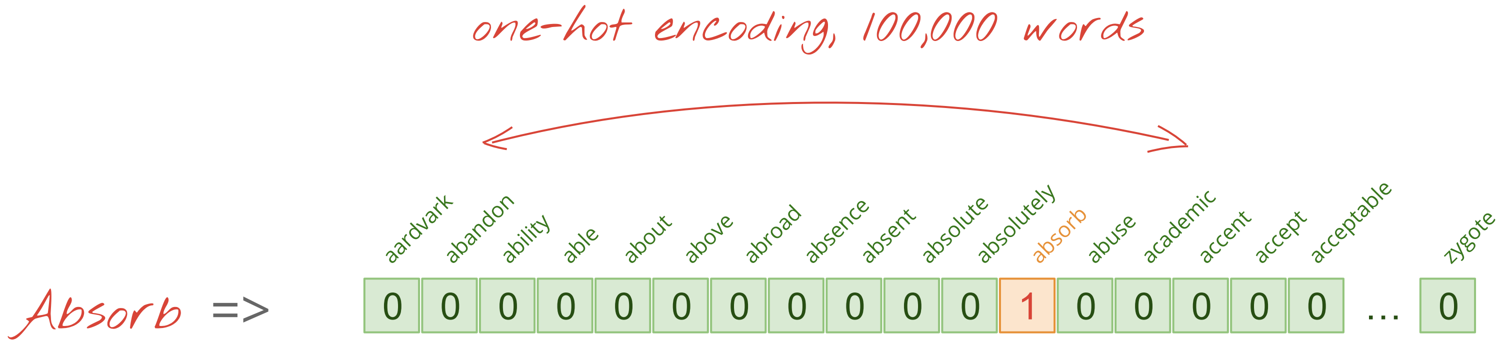
Note: If you're using Dropout with the Subclassing API, you will need to pass a parameter to let your model know whether it's train or test time.

[Keras Layers - Dropout]

Embeddings (and try the Projector)

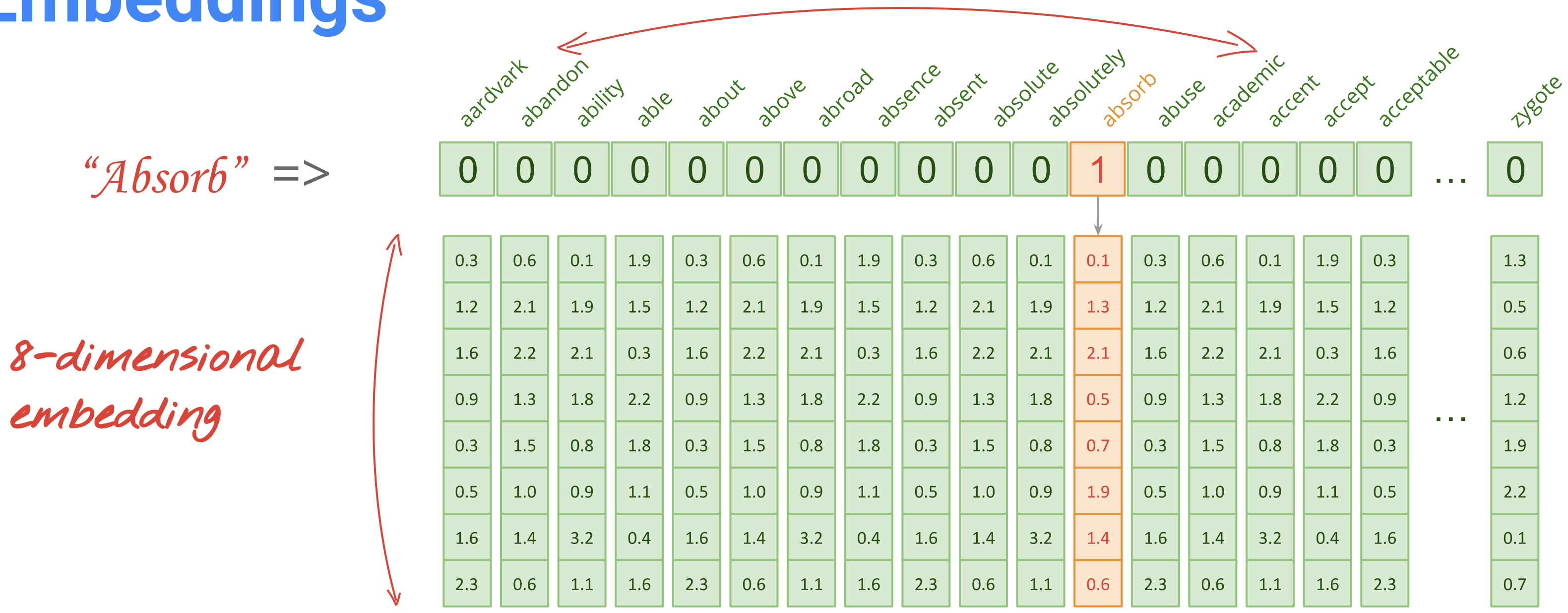
One-hot encodings

- Imagine our vocabulary is 100,000 words. A one-hot encoding results in **sparse**, **high-dimensional vector**.



Doesn't capture the relationships between words / cannot be adjusted by the classifier.

Embeddings



Notes: the embedding layer is basically a lookup table. Embedding weights begin randomly and are adjusted by the classifier by backprop (exactly as in a Dense layer). Size of the embedding is a hyperparameter. How is the lookup performed? Using a dictionary mapping from integer-encoded words -> embeddings. This is more efficient than an approach you may see written in math notation (where a one-hot vector is multiplied against a matrix to "select" a column).

Embeddings

Dense, lower-dimensional vectors learned from data.

Common sizes

- **8** (if training your a small amount of data) to **1024** (if training a large model for reuse down the road).

Parameters in an Embedding layer

```
from tensorflow.keras.layers import Embedding

vocab_size, embed_dim, = 1000, 64

model = Sequential()
model.add(Embedding(vocab_size, embed_dim))
model.summary()
```

Parameters in an Embedding layer

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_1 (Embedding)      (None, None, 64)          64000     
=================================================================
Total params: 64,000
Trainable params: 64,000
Non-trainable params: 0
_________________________________________________________________
```

Shape: batch size, sentence length (in words), embedding dimension

As you would expect: vocabulary_size * embeddding_dimension.

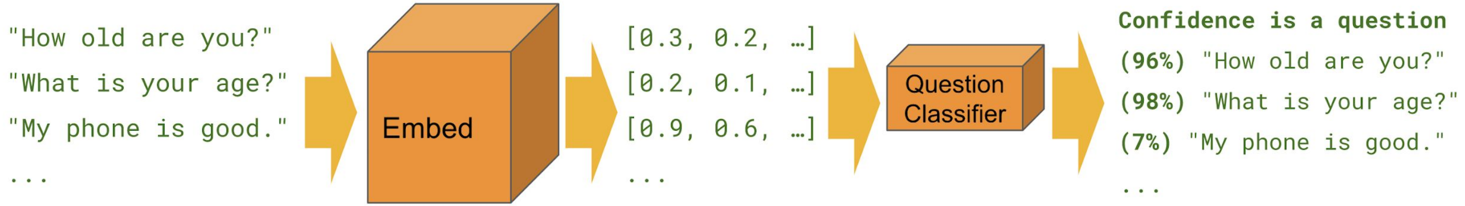
Embeddings (not just for words)

Quick discussion: what could we embed?

Embeddings (not just for words)

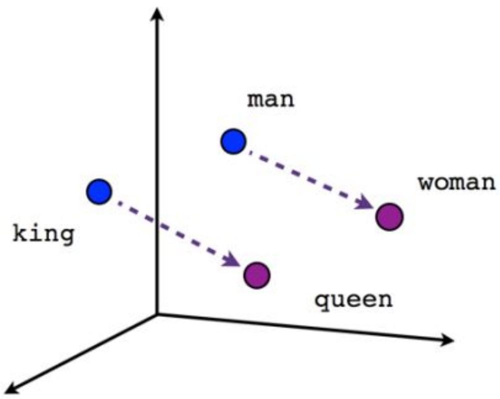
Quick discussion: what could we embed?

- Images
- Songs
- Behaviors (based on web activity)
- Conditions (based on sensor data)
- Sentences
- Etc.

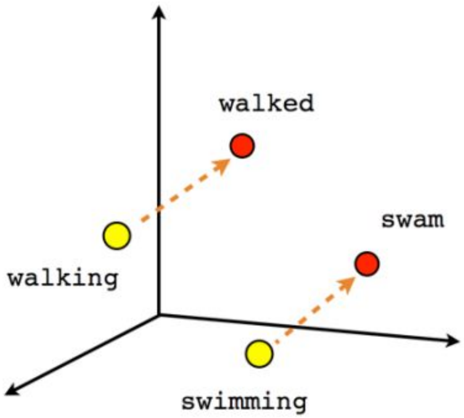


Vector arithmetic

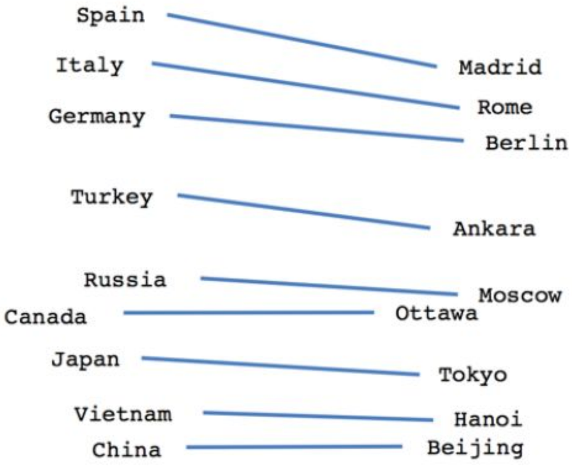
Notes: YMMV. Embeddings trained on small datasets are not likely to be as interpretable.



Male-Female



Verb tense



Country-Capital

https://www.aclweb.org/anthology/N13-1090

Transfer learning

- **Quick discussion**: how might this work with word embeddings?
- How well is it likely to work?

Transfer learning

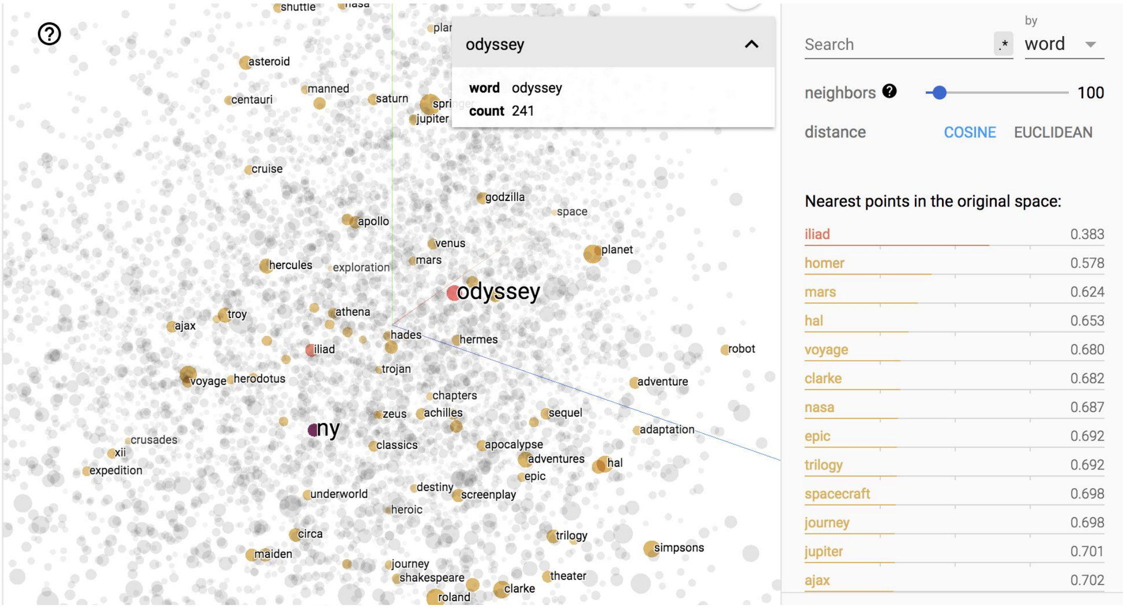
- **Quick discussion**: how might this work with word embeddings?
- How well is it likely to work?

Notes:

- **Surprisingly, I haven't had great results with it personally (similar experience to this experiment). Though this may change when using larger embeddings.**
- **Helpful example code in Deep Learning with Python.**
- **You can check http://tensorflow.org/hub for the state of the art (although skip the TF v1 code, wait a few weeks for a hopeful upgrade to TF2).**

Try the Embedding Projector (10 mins)

- Use this tutorial



When to use & when not

- You need to run a bunch of experiments and interactively explore results
- Matplotlib is easier to small stuff

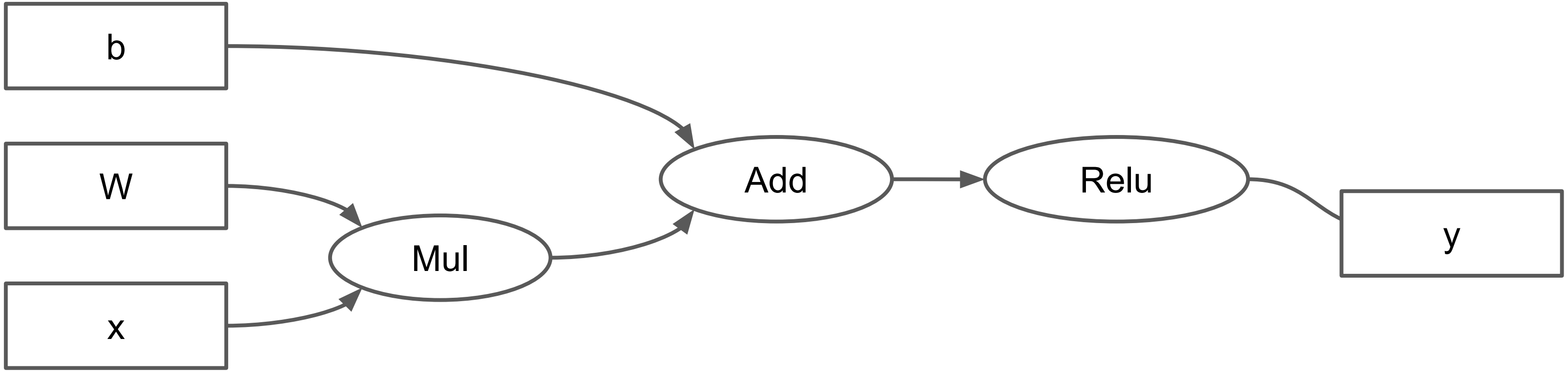
TensorFlow 1.0

Just for reference (don't spend time on this!)

- There's value in understanding this code (in case you'd like to reuse some of many papers written with it).

- TensorFlow 2.0 **much more closely matches our mental model**, and that's where you should invest energy.

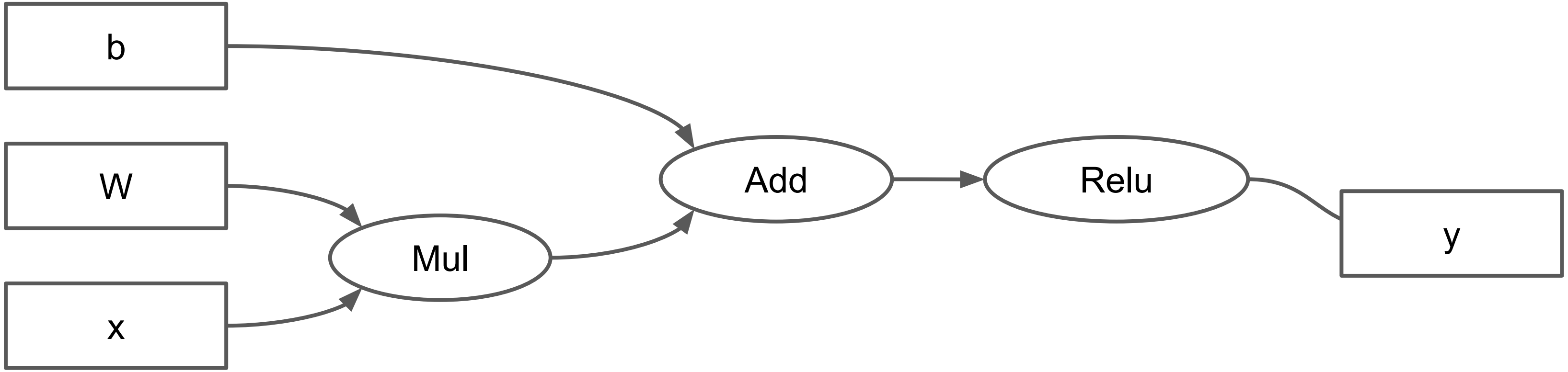
Why graphs



Why graphs

Notes

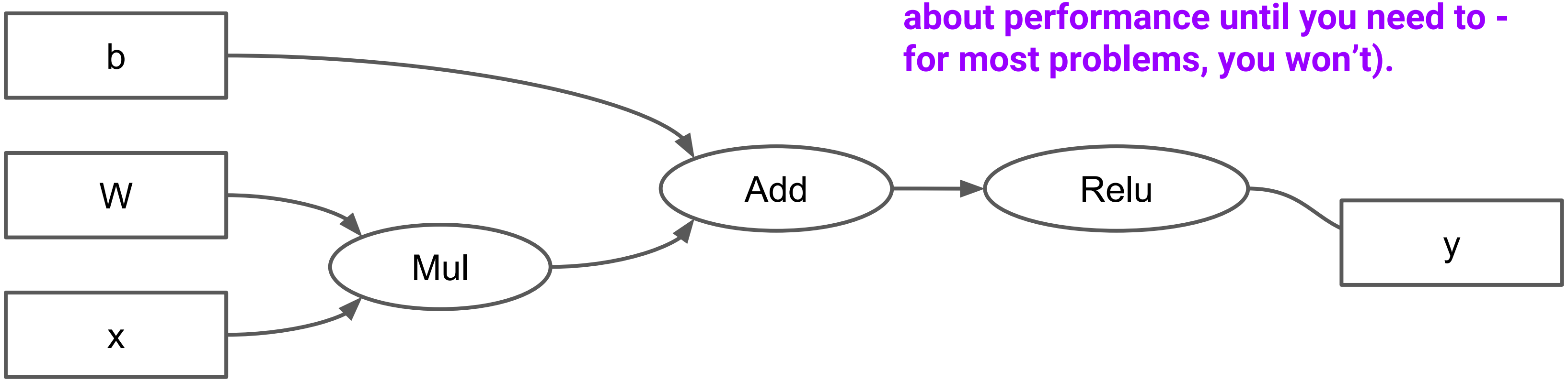
- **Portability: develop in Python, but deploy to devices (like iOS, or Android) that don't support it.**



Why graphs

Notes

- **Portability: develop in Python, but deploy to devices (like iOS, or Android) that don't support it.**
- **Performance: graphs can be optimized and distributed (as always, don't worry about performance until you need to - for most problems, you won't).**



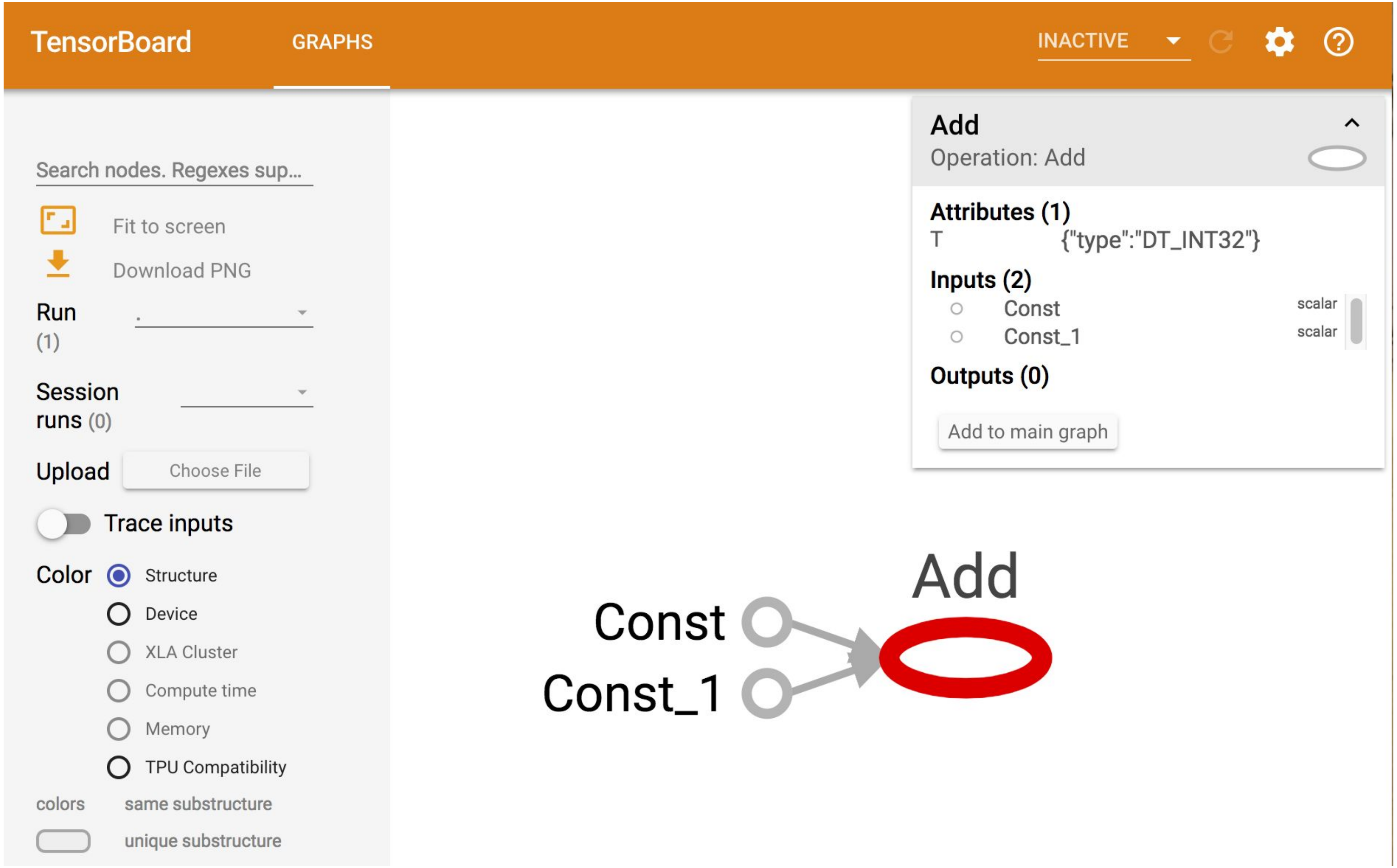
What's a graph?

1.0 code

```
import tensorflow as tf
a = tf.constant(2)
b = tf.constant(3)
x = tf.add(a, b)

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

```
# $ tensorboard --logdir=graphs/
# then browse to http://localhost:6006
```


TensorBoard
GRAPHS
INACTIVE
Search nodes. Regexes sup...
Fit to screen
Download PNG
Run (1)
Session runs (0)
Upload
Choose File
Trace inputs
Color
Structure
Device
XLA Cluster
Compute time
Memory
TPU Compatibility
colors
same substructure
unique substructure
Add
Operation: Add
Attributes (1)
T {"type":"DT_INT32"}
Inputs (2)
Const scalar
Const_1 scalar
Outputs (0)
Add to main graph
Add
Const
Const_1

1.0 code

Tensors in graph mode are symbolic

1.0 code

```
import tensorflow as tf
w = tf.Variable(10)
print(w)
# <tf.Variable shape=() dtype=int32_ref>
```

Note: printing w does not print 10!

1.0 code

Sessions

```
import tensorflow as tf
w = tf.Variable(10)
with tf.Session() as sess:
    print(sess.run(w))
```

FailedPreconditionError: Attempting to use uninitialized value

Variables must be initialized before use

1.0 code

```
import tensorflow as tf
w = tf.Variable(10)
with tf.Session() as sess:
    sess.run(w.initializer)
    print(sess.run(w))      # 10
```

1.0 code

Why doesn't this work?

```
import tensorflow as tf
w = tf.Variable(10)
with tf.Session() as sess:
    sess.run(w.initializer)
    w.assign(100)
    print(sess.run(w)) # 10
```

The op needs to be executed first

1.0 code

```
import tensorflow as tf
w = tf.Variable(10)
with tf.Session() as sess:
    sess.run(w.initializer)
    assign_op = w.assign(100)
    sess.run(assign_op)
    print(sess.run(w)) # 100
```

Not deterministic

1.0 code

```
X, y = tf.Variable(1.0), tf.Variable(1.0)
add_op = x.assign(x + y)
div_op = y.assign(y / 2)
init = tf.global_variables_initializer()
with tf.Session() as sess:
 init.run()
 for iteration in range(50):
        sess.run([add_op, div_op])
 print(sess.run(w)) # run 1: 2.0, run 2: 2.75
```

No dependency seen between operations, run in parallel. Would need to be explicitly called out.

1.0 code

Feeding and fetching

```
# create a placeholder of type float 32-bit, shape is a vector of 3 elements
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
add_op = tf.add(a,b)

with tf.Session() as sess:
    print(sess.run(add_op, {a: [1, 2, 3]})) # [6, 7, 8]
```

If you have a bunch of data to feed, you can do so in a loop - causes performance penalties (from Python -> TF -> Python -> TF)

XLA

```
tmp = tf.add(x, y)
result = tf.multiply(tmp, z)
```

```
for (i = 0; i < element_count; i++) {
  tmp[i] = x[i] + y[i];
}

for (i = 0; i < element_count; i++) {
  result = tmp[i] * z[i];
}
```

```
for (i = 0; i < element_count; i++) {
  result = (x[i] + y[i]) * z[i];
}
```

XLA compresses these two loops into one.

It was decided graph-level code was the wrong abstraction

1.0 code

Eager (aka normal) execution

1.0 code

- Feels like regular Python programming
- Easier to develop and debug

RFC for TF 2.0 based on this.

Notes: Writing graph-level code is metaprogramming (define a graph, then run it) -- making it difficult to debug. When eager mode is enabled, you could develop TensorFlow code just like you would write NumPy.

1.0 code

Eager

```
import tensorflow as tf


# must be called once at program startup
tf.enable_eager_execution()
```

Multiplying a matrix by itself

```
a = tf.constant([[1.0, 2.0],
                 [3.0, 4.0]])
print(tf.matmul(a, a))


tf.Tensor(
[[ 7. 10.]
 [15. 22.]], shape=(2, 2), dtype=float32)
```

No sessions, no graphs, no placeholders!

NumPy compatibility

```
a = tf.constant([[1.0, 2.0],
                 [3.0, 4.0]])

foo = tf.matmul(a, a)

print(foo.numpy())

array([[ 7., 10.],
       [15., 22.]], dtype=float32)
```

Works both ways

```
a = tf.constant([[1.0, 2.0],
                 [3.0, 4.0]])

foo = tf.matmul(a, a)
bar = foo.numpy()
tf.reduce_sum(bar)
```

TensorFlow operations work on NumPy data

```
<tf.Tensor: id=58, shape=(), dtype=float32, numpy=54.0>
```

NumPy operations accept tf.Tensor arguments

```
a = tf.constant([[1.0, 2.0],
                 [3.0, 4.0]])

print(type(a)) # <type 'EagerTensor'>
bar = np.dot(a,a)
print(type(bar)) # <type 'numpy.ndarray'>
```

Differences

- Tensors can be backed by accelerator memory (GPUs)

Acceleration

Note: automatic device placement is supported for most ops (so you do need to specify whether to use a GPU in practice).

```
n = 1000
def time_matmul(x):
  %timeit tf.matmul(x, x)

# Force execution on CPU
with tf.device("CPU:0"):
  x = tf.random_uniform([n, n])
  assert x.device.endswith("CPU:0")
  time_matmul(x)
```

18 ms per loop

```
# Force execution on GPU #0
if tf.test.is_gpu_available():
  with tf.device("GPU:0"): # Or GPU:N
    x = tf.random_uniform([n, n])
    assert x.device.endswith("GPU:0")
    time_matmul(x)
```

1.1 ms per loop

Gradients

Every operation (like tf.multiply, and more complex ones, like tf.nn.relu) have an associated gradient function.

To take the gradient of a user defined function:

- Operations recorded on a tape.
- Tape is played back in reverse.
- Grad functions used to compute the gradient.

Example

```
@ops.RegisterGradient("Square")
def _SquareGrad(op, grad):
  x = op.inputs[0]
  y = constant_op.constant(2.0, dtype=x.dtype)
  return math_ops.multiply(grad, math_ops.multiply(x, y))
```

Some details omitted

https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/math_grad.py

Derivative of a function

1.0 code

```
import tensorflow as tf
tf.enable_eager_execution()
tfe = tf.contrib.eager

def f(x):
  return tf.square(x)

grads = tfe.gradients_function(f)
grads(3.0) # 6.0
```

Gradient tapes

1.0 code

```
c = tfe.Variable([[2.0]])
d = tfe.Variable([[3.0]])
with tf.GradientTape() as tape:
  loss = c * d


grad = tape.gradient(loss, d)
print(grad) # 2.0
```

Traininable variables (created by tf.contrib.eager.Variable) are automatically watched.

Can also manually 'watch' other tensors with tape.watch()

You could switch between eager and graph mode

1.0 code

```
import tensorflow as tf
tf.enable_eager_execution()


print(tf.executing_eagerly()) # True


graph = tf.Graph()
with graph.as_default():
    print(tf.executing_eagerly()) # False
```

But don't. Just stay in eager unless you have a special reason not to.

1.0 code

You could also compile functions

```
import tensorflow as tf
tf.enable_eager_execution()


@tf.contrib.eager.defun
def square_sum(x, y):
    return tf.square(x+y)


result = square_sum(2., 3.)
print(result.numpy()) # 25
```

TensorFlow 2.0

Goodbye

- session.run
- placeholders
- tf.control_dependencies
- tf.global_variables_initializer
- tf.cond, tf.while_loop

Hello

- tf.function, AutoGraph

This code runs “eagerly” -- or as you would expect in regular Python. This is the default in TensorFlow 2.0 unless you explicitly specify otherwise.

```
def add(a, b):
  return a + b

add(tf.ones([2, 2]), tf.ones([2, 2])) # [[2., 2.], [2., 2.]]
```

```python
# A function is like an op

@tf.function
def add(a, b):
  return a + b

add(tf.ones([2, 2]), tf.ones([2, 2])) # [[2., 2.], [2., 2.]]
```

Adding this annotation causes code in the function be “compiled” and run in graph mode. For certain functions, this can accelerate it significantly, and makes it possible to deploy to devices without a Python interpreter.

```python
# Let's make this faster
lstm_cell = tf.keras.layers.LSTMCell(10)


def fn(input, state):
  return lstm_cell(input, state)


input = tf.zeros([10, 10]); state = [tf.zeros([10, 10])] * 2


# warm up
lstm_cell(input, state); fn(input, state)


# "benchmark"
timeit.timeit(lambda: lstm_cell(input, state), number=10) # 0.03
```

Always take benchmarks with a grain of salt (they depend on the code, hardware, network, etc). This is a simple example just to show the idea.

```python
# Let's make this faster
lstm_cell = tf.keras.layers.LSTMCell(10)


@tf.function
def fn(input, state):
  return lstm_cell(input, state)


input = tf.zeros([10, 10]); state = [tf.zeros([10, 10])] * 2


# warm up
lstm_cell(input, state); fn(input, state)


# "benchmark"
timeit.timeit(lambda: lstm_cell(input, state), number=10) # 0.03
timeit.timeit(lambda: fn(input, state), number=10) # 0.004
```

Always take benchmarks with a grain of salt (they depend on the code, hardware, network, etc). This is a simple example just to show the idea.

+/- 10x improvement (don't take that to imply this will always be the case!)

```python
# tf.function is polymorphic

@tf.function
def add(a, b):
  return a + b

add(tf.ones([2, 2]), tf.ones([2, 2])) # [[2., 2.], [2., 2.]]
```

This code will work with ints, floats, etc.

```python
# tf.function handles complex control flow


@tf.function
def f(x):
  while tf.reduce_sum(x) > 1:
    x = tf.tanh(x)
  return x


f(tf.random.uniform([10]))  # Works!
```

Autograph (or, how to never write assembly-like code again)

```python
print(tf.autograph.to_code(f)) # Autograph in action
```

AutoGraph is a Python-Python compiler, which produces code the TF-backend can compile / optimize / and accelerate in C++.

This is not a complex system at all...

```python
print(tf.autograph.to_code(f)) # Autograph in action
...
def tf__f(x):
  def loop_test(x_1):
    with ag__.function_scope('loop_test'):
      return ag__.gt(tf.reduce_sum(x_1), 1)
  def loop_body(x_1):
    with ag__.function_scope('loop_body'):
      with ag__.utils.control_dependency_on_returns(tf.print(x_1)):
        tf_1, x = ag__.utils.alias_tensors(tf, x_1)
        x = tf_1.tanh(x)
        return x,
  x = ag__.while_stmt(loop_test, loop_body, (x,), (tf,))
  return x
```

AutoGraph is a Python-Python compiler, which produces code the TF-backend can compile / optimize / and accelerate in C++.

This is not a complex system at all...

```python
# Controlling autograph

@tf.function
def f(x):
  for i in range(10): # Static python loop, not converted
    do_stuff()
  for i in tf.range(10): # Depends on a tensor, converted
    do_stuff()
```

Python statements like “if” and “while” are converted automatically. For other things, use tf.* when in doubt.

High-level APIs

- Keras [Sequential / Functional / Subclassing] (with built-in or custom training loops)

- Estimators - cool, but still not as easy to use as sklearn (prefer Keras)

Low-level APIs

- Regular Python, much like NumPy (use tf.* instead of np.*)
- GradientTape
- tf.function and AutoGraph

Best resources to start with

TF 2.0 tutorials and guides

For next time

Reading

- Deep Learning: 7.12 (Dropout), 8.3 (Basic Algorithms); 8.4 (Parameter Initialization Strategies); 8.5 (Algorithms with Adaptive Learning Rates)

- Dropout: A Simple Way to Prevent Neural Networks from Overfitting
- Understanding the difficulty of training deep feedforward neural networks

Assignments

- A3 released
- Project released