Deep Learning Software

Tensorflow/PyTorch

(Based on stanford cs231n slides)

A zoo of frameworks!

PaddlePaddle
(Baidu)

Chainer

Caffe
(UC Berkeley)

Caffe2
(Facebook)

MXNet
(Amazon)
Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

CNTK
(Microsoft)

Torch
(NYU / Facebook)

PyTorch
(Facebook)

JAX
(Google)

Theano
(U Montreal)

TensorFlow
(Google)

And others...

We'll focus on these

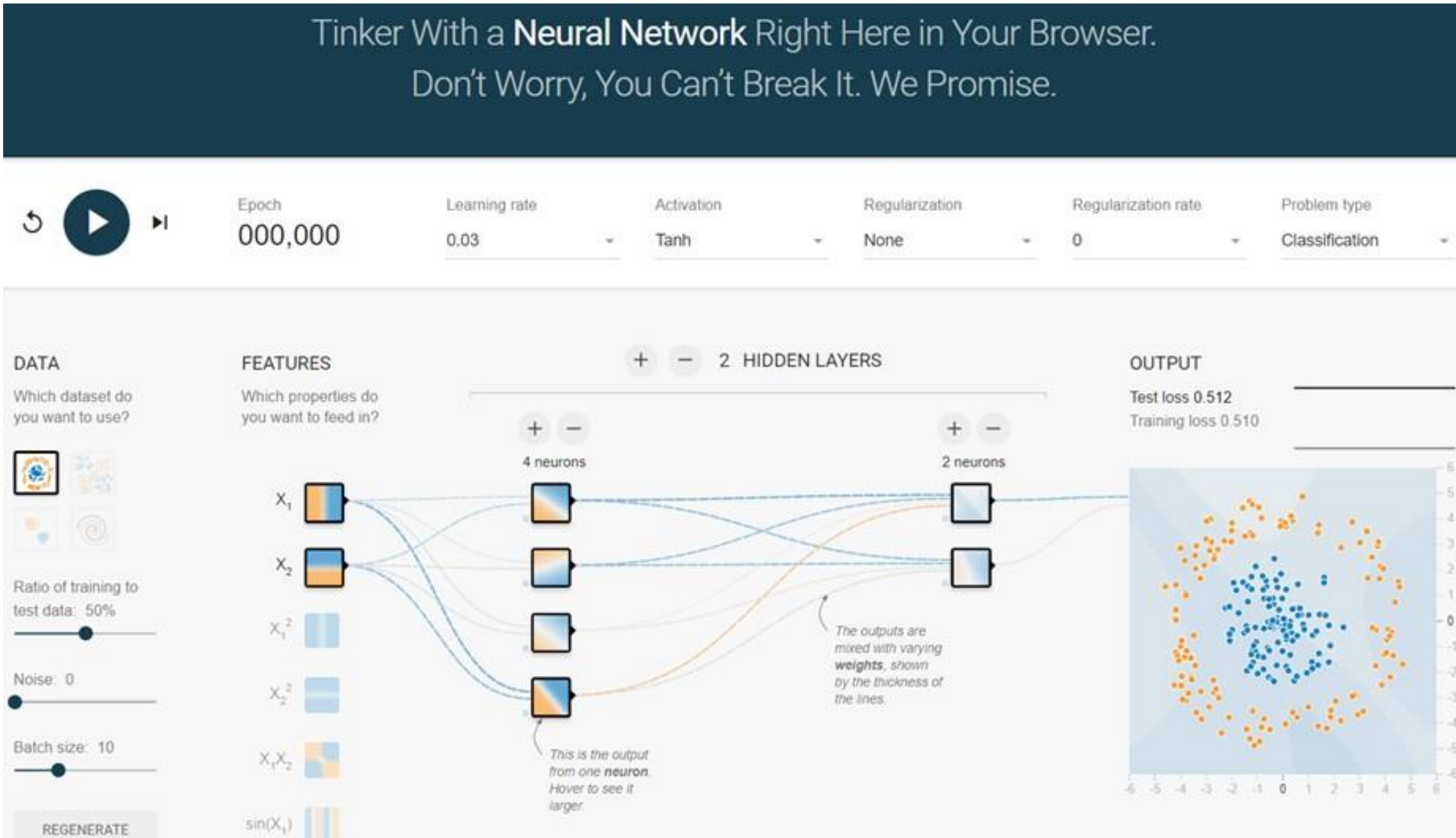
Choose a deep learning software

- Efficiency
- Memory Usage
- Coding Language Supporting
- Debugging Information
- Documentation
- Organization & Community

Tensorflow

- From Google
- All about computation graphs
- Easy visualizations (TensorBoard)
- Multi-GPU and multi-node training
- Easy deploying
- Have a try: http://playground.tensorflow.org/

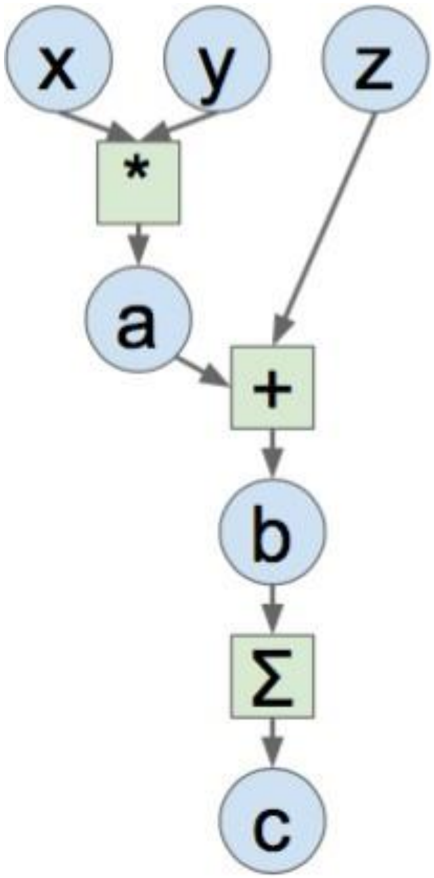
Tensorflow: Playground



Tensorflow: Version

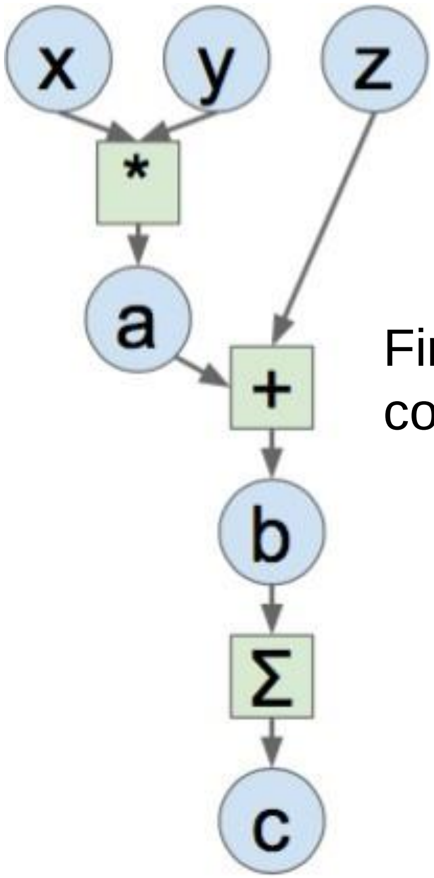
- For this course, we are using Tensorflow 1.13
- Why we don’t use Tensorflow 2.0?
  - Eager Execution
  - Unstable APIs

Tensorflow: Computational Graphs




```python
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

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

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

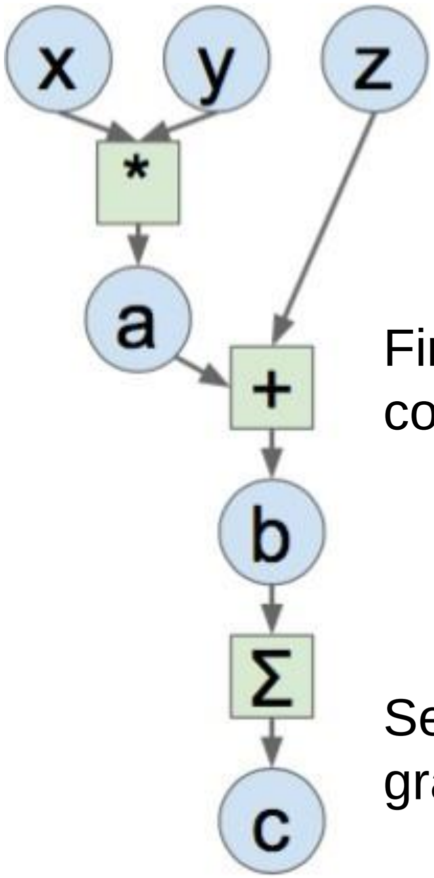
Tensorflow: Computational Graphs



First, **define** the computational graph

```python
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

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

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Tensorflow: Computational Graphs




```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4
```

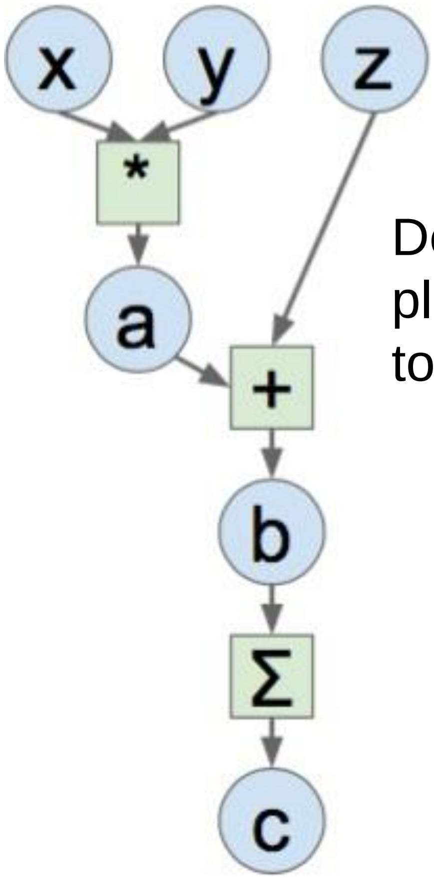
First, **define** the computational graph

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])
```

Second, **run** the graph many times

```
with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

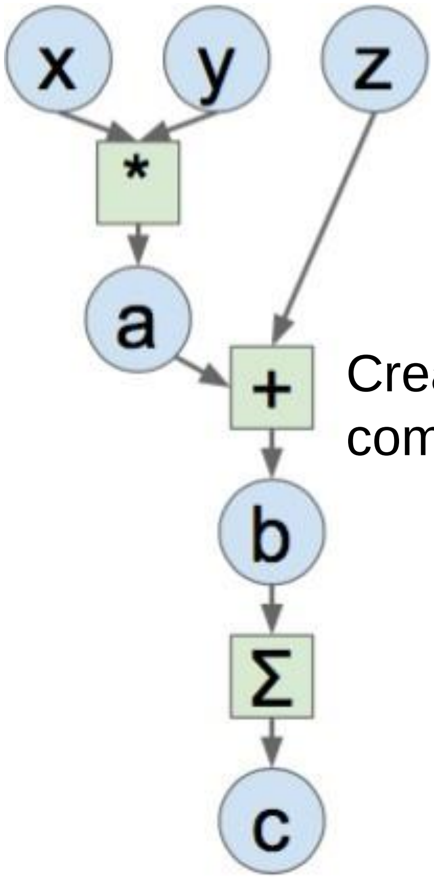
Tensorflow: Computational Graphs



Define symbolic placeholders: inputs to the graph

```python
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

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

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

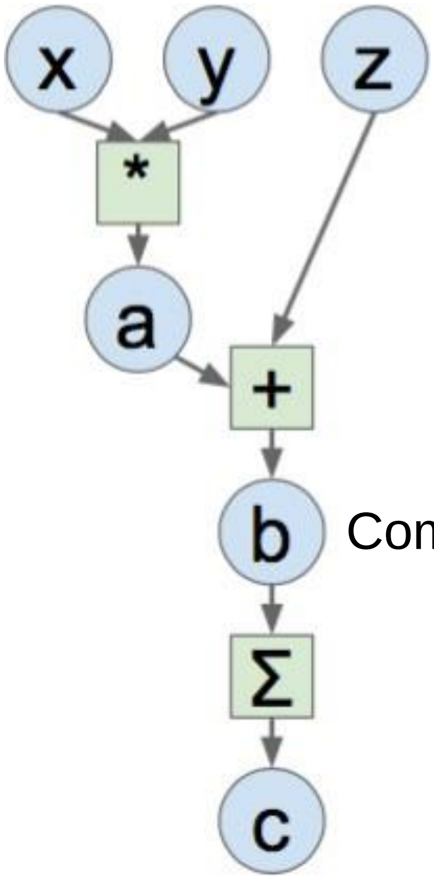
Tensorflow: Computational Graphs



Create forward computational graph

```python
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

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

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

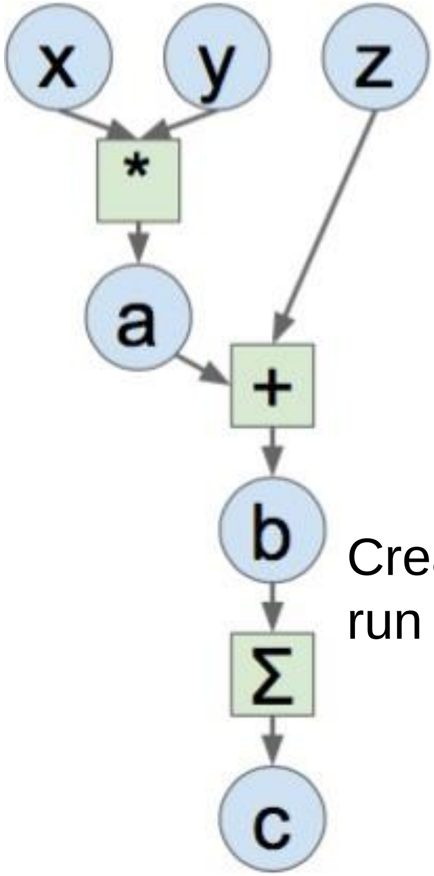
Tensorflow: Computational Graphs



Compute gradients

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

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

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

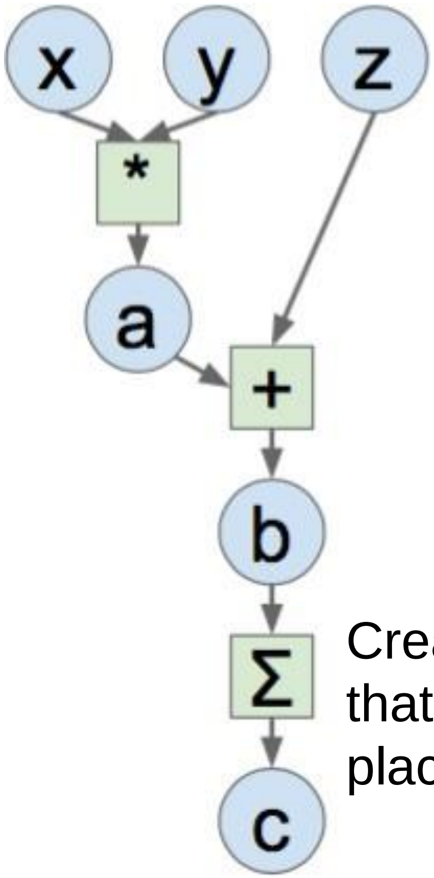
Tensorflow: Computational Graphs



Create **session** to run the graph

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

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

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

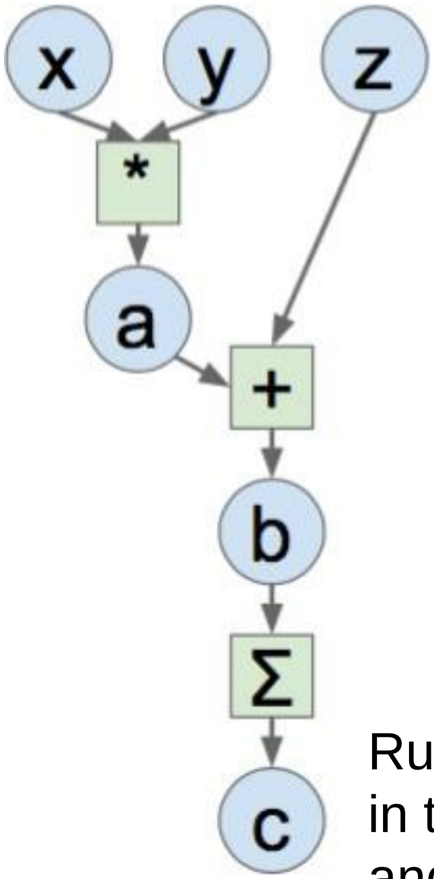
Tensorflow: Computational Graphs



Create numpy arrays that will fill in the placeholders above

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

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

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Tensorflow: Computational Graphs



Run the graph: feed in the numpy arrays and get c_val and gradients

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

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

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Tensorflow: About Tensor

- A+B Problem

```
import tensorflow as tf

a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
c = tf.add(a, b)

with tf.Session() as session:
    result = session.run(c, {a:1, b:2})
    print(result) # 3
    result = session.run(c, {a:[5,2,1], b:[3,6,5]})
    print(result) # [8., 8., 6.]
```

Tensorflow: About Tensor

- Everything is Tensor

```
import tensorflow as tf

a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
c = tf.add(a, b)

print(a) # Tensor("Placeholder:0", dtype=float32)
print(b) # Tensor("Placeholder_1:0", dtype=float32)
print(c) # Tensor("add:0", dtype=float32)
```

Tensorflow: About Tensor

- About Placeholder
  - 1.dtype, 2.shape, 3.name

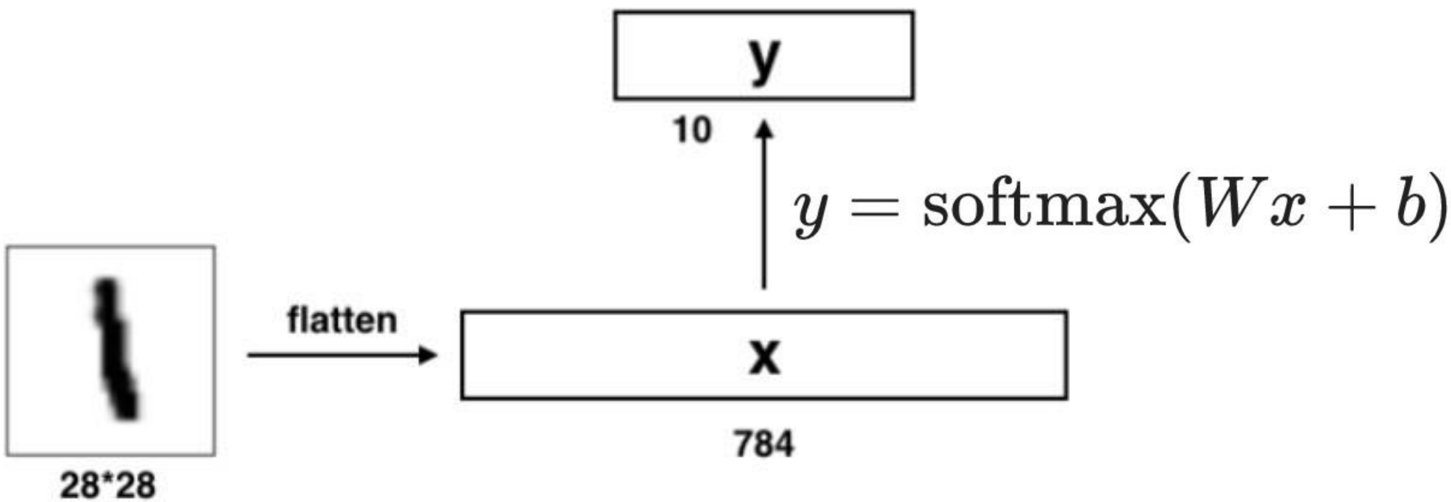
```
a = tf.placeholder(tf.float32, shape=[3,5])
print(a)
# Tensor("Placeholder:0",shape=(3,5),dtype=float32)

b = tf.placeholder(tf.int32,
    shape=[3,None,2], name='qq')
print(b)
# Tensor("qq:0",shape=(3,?,2),dtype=int32)
```

Tensorflow: Work on MNIST

- A handwritten digits classification task.

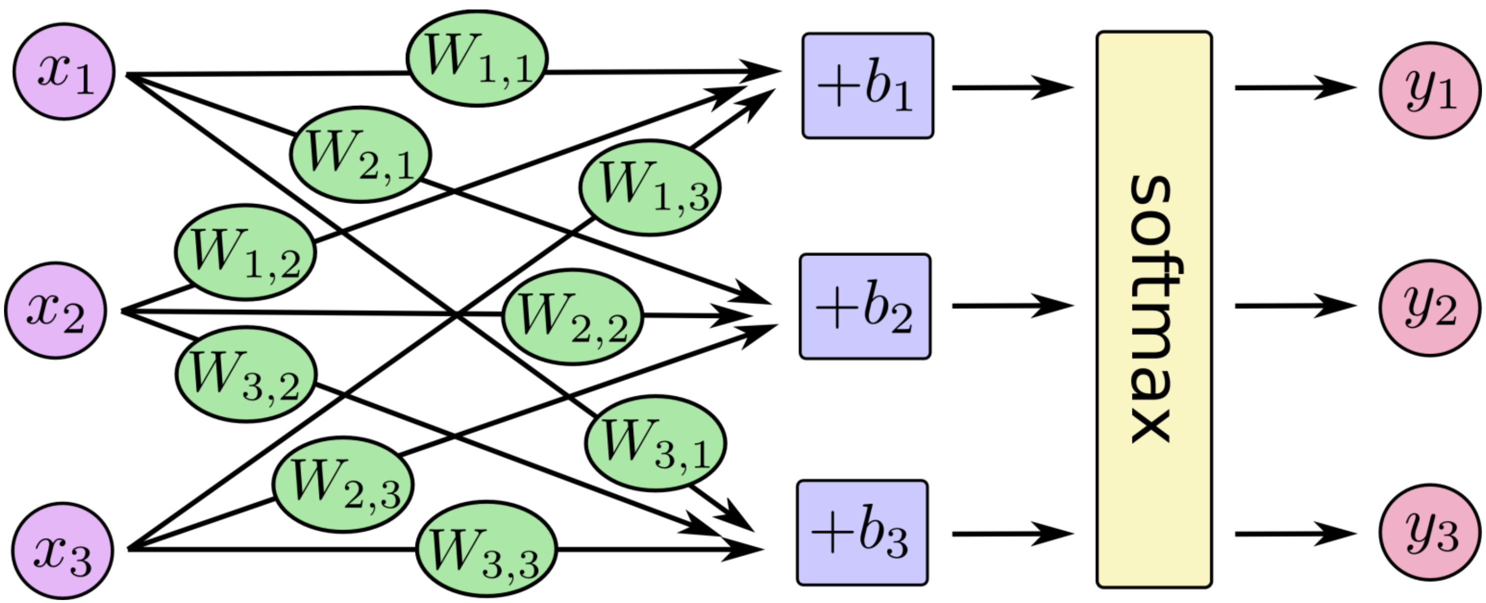
Tensorflow: Work on MNIST

- A simple model
  - inputs: x, outputs: y
  - parameters of model: W, b



Tensorflow: Work on MNIST

- A simple model
  - inputs: x, outputs: y
  - parameters of model: W, b



Tensorflow: Work on MNIST

- A simple model
  - inputs: x, outputs: y
  - parameters of model: W, b

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\begin{pmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{pmatrix}$$

Tensorflow: Work on MNIST

- A simple model
  - inputs: x, outputs: y
  - parameters of model: W, b

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

Tensorflow: Work on MNIST

- A simple model
  - inputs: x, outputs: y
  - parameters of model: W, b

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

$$y = \text{softmax}(Wx + b)$$

Tensorflow: Work on MNIST

- Build the graph

```
import tensorflow as tf

x = tf.placeholder(tf.float32, [None, 784], name='x')
# shape of x is (?,784)
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.matmul(x, W) + b
# shape of y is (?,10)
z = tf.argmax(y, 1, name='z')
```

Tensorflow: Work on MNIST

- Compute the loss & gradients

```
y_hat = tf.placeholder(tf.float32, [None, 10])
loss = tf.nn.softmax_cross_entropy_with_logits(
       labels=y_hat, logits=y)
# shape of loss is (?)
loss_avg = tf.reduce_mean(loss)
# shape of loss_avg is ()

opt = tf.train.GradientDescentOptimizer(0.5)
gradients = opt.compute_gradients(loss_avg)
train_step = opt.apply_gradients(gradients)
```

Tensorflow: Work on MNIST

- Training & Saving

```
mnist = input_data.read_data_sets(DIR, one_hot=True)
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    for _ in range(1000):
        xs, ys = mnist.train.next_batch(100)
        sess.run(train_step, {x: xs, y_hat: ys})

    saver = tf.train.Saver(tf.global_variables())
    saver.save(sess, 'train/train')
```

Tensorflow: Work on MNIST

- Restoring & Testing

```
mnist = input_data.read_data_sets(DIR, one_hot=True)
saver = tf.train.import_meta_graph('train/train.meta')
with tf.Session() as sess:
    saver.restore(sess, 'train/train')
    result = sess.run('z:0',{'x:0':mnist.test.images})
```

Tensorflow: More Details

- Variable Scope

```
with tf.variable_scope('scope1'):
    v1 = tf.get_variable('v', shape=[23])
with tf.variable_scope('scope2'):
    v2 = tf.get_variable('v', shape=[23])

for item in tf.global_variables():
    print((item.name, item.get_shape()))

# (u'scope1/v:0', TensorShape([Dimension(23)]))
# (u'scope2/v:0', TensorShape([Dimension(23)]))
```

Tensorflow: More Details

- Reuse
  - bad case

```
with tf.variable_scope('scope1'):
    v1 = tf.get_variable('v', shape=[23])
with tf.variable_scope('scope1'):
    v2 = tf.get_variable('v', shape=[23])
```

ValueError: Variable scope1/v already exists

Tensorflow: More Details

- Reuse
  - good case

```
with tf.variable_scope('scope1'):
    v1 = tf.get_variable('v', shape=[23])
with tf.variable_scope('scope1', reuse=True):
    v2 = tf.get_variable('v', shape=[23])
```

v2 and v1 share the same parameter

Tensorflow: More Details

- Trainable Variable

```
with tf.variable_scope('scope1'):
    v1 = tf.get_variable('v', shape=[23])

with tf.variable_scope('scope2'):
    v2 = tf.get_variable('v', shape=[23],
        trainable=False)

for item in tf.trainable_variables():
    print((item.name, item.get_shape()))

# (u'scope1/v:0', TensorShape([Dimension(23)]))
```

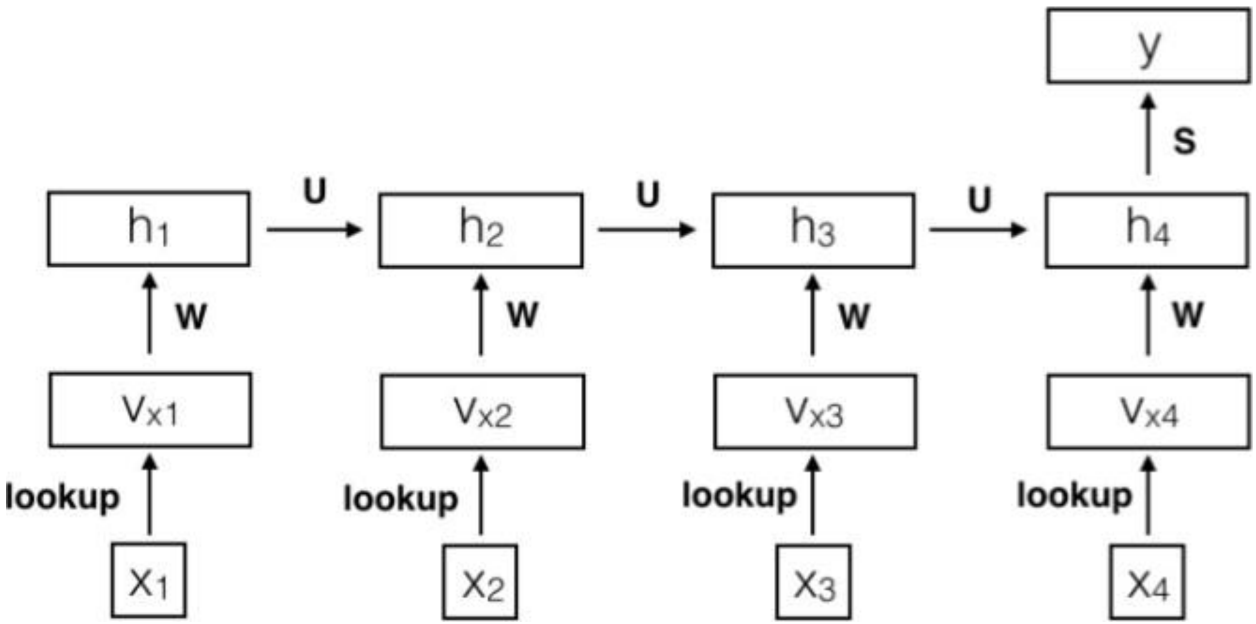
Tensorflow: More Details

- Print all trainable variables for debugging

```
embed:0: (40000, 100)
encoder/multi_rnn_cell/cell_0/gru_cell/gates/weights:0: (1124, 2048)
encoder/multi_rnn_cell/cell_0/gru_cell/gates/biases:0: (2048,)
encoder/multi_rnn_cell/cell_0/gru_cell/candidate/weights:0: (1124, 1024)
encoder/multi_rnn_cell/cell_0/gru_cell/candidate/biases:0: (1024,)
encoder/multi_rnn_cell/cell_1/gru_cell/gates/weights:0: (2048, 2048)
encoder/multi_rnn_cell/cell_1/gru_cell/gates/biases:0: (2048,)
encoder/multi_rnn_cell/cell_1/gru_cell/candidate/weights:0: (2048, 1024)
encoder/multi_rnn_cell/cell_1/gru_cell/candidate/biases:0: (1024,)
encoder/multi_rnn_cell/cell_2/gru_cell/gates/weights:0: (2048, 2048)
encoder/multi_rnn_cell/cell_2/gru_cell/gates/biases:0: (2048,)
encoder/multi_rnn_cell/cell_2/gru_cell/candidate/weights:0: (2048, 1024)
encoder/multi_rnn_cell/cell_2/gru_cell/candidate/biases:0: (1024,)
encoder/multi_rnn_cell/cell_3/gru_cell/gates/weights:0: (2048, 2048)
encoder/multi_rnn_cell/cell_3/gru_cell/gates/biases:0: (2048,)
encoder/multi_rnn_cell/cell_3/gru_cell/candidate/weights:0: (2048, 1024)
encoder/multi_rnn_cell/cell_3/gru_cell/candidate/biases:0: (1024,)
attention_keys/weights:0: (1024, 1024)
decoder/multi_rnn_cell/cell_0/gru_cell/gates/weights:0: (2148, 2048)
decoder/multi_rnn_cell/cell_0/gru_cell/gates/biases:0: (2048,)
decoder/multi_rnn_cell/cell_0/gru_cell/candidate/weights:0: (2148, 1024)
decoder/multi_rnn_cell/cell_0/gru_cell/candidate/biases:0: (1024,)
decoder/multi_rnn_cell/cell_1/gru_cell/gates/weights:0: (2048, 2048)
decoder/multi_rnn_cell/cell_1/gru_cell/gates/biases:0: (2048,)
decoder/multi_rnn_cell/cell_1/gru_cell/candidate/weights:0: (2048, 1024)
decoder/multi_rnn_cell/cell_1/gru_cell/candidate/biases:0: (1024,)
decoder/multi_rnn_cell/cell_2/gru_cell/gates/weights:0: (2048, 2048)
decoder/multi_rnn_cell/cell_2/gru_cell/gates/biases:0: (2048,)
decoder/multi_rnn_cell/cell_2/gru_cell/candidate/weights:0: (2048, 1024)
decoder/multi_rnn_cell/cell_2/gru_cell/candidate/biases:0: (1024,)
decoder/multi_rnn_cell/cell_3/gru_cell/gates/weights:0: (2048, 2048)
decoder/multi_rnn_cell/cell_3/gru_cell/gates/biases:0: (2048,)
decoder/multi_rnn_cell/cell_3/gru_cell/candidate/weights:0: (2048, 1024)
decoder/multi_rnn_cell/cell_3/gru_cell/candidate/biases:0: (1024,)
attention_construct/weights:0: (2048, 1024)
decoder/output_projection/weights:0: (1024, 40000)
decoder/output_projection/biases:0: (40000,)
```

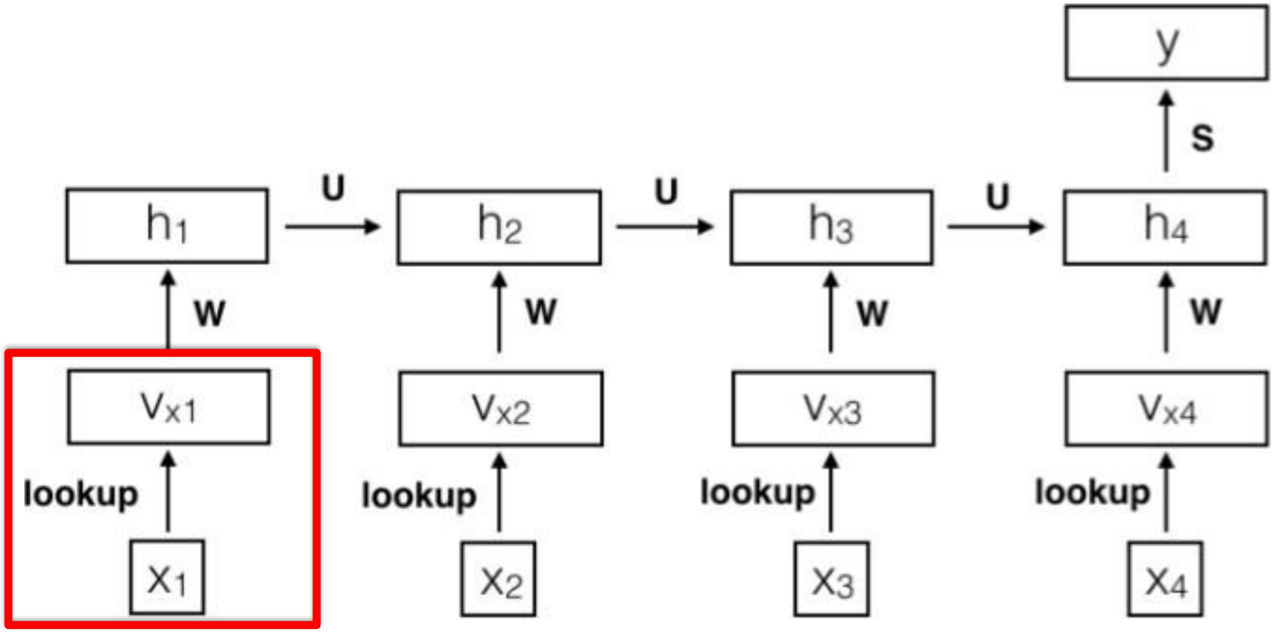
Tensorflow: About RNN

- RNN for Sentence Classification



Tensorflow: About RNN

- RNN for Sentence Classification
  - Word Embedding



Tensorflow: About RNN

- Word Embedding

```
# initialize randomly
embed = tf.get_variable('embed', [wocab_size, d],
        tf.float32)
```

```
# initialize by pre-trained word vectors
embed = tf.get_variable('embed', dtype=tf.float32,
        initializer=init_embed)
```

```
# using pre-trained word vectors without tuning
embed = tf.get_variable('embed', dtype=tf.float32,
        initializer=init_embed, trainable=False)
```

Tensorflow: About RNN

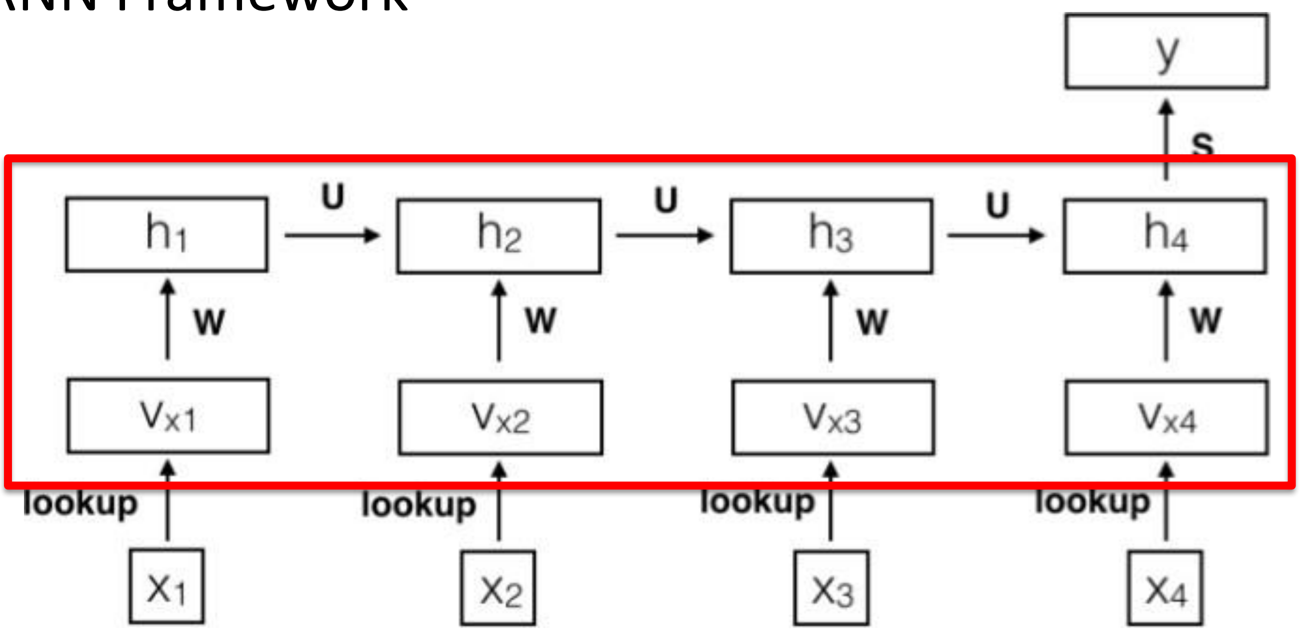
- Embedding Lookup

```
# initialize randomly
embed = tf.get_variable('embed', [wocab_size, d],
        tf.float32)
```

```
x = tf.placeholder(tf.int32, shape=[None, max_step])
v = tf.nn.embedding_lookup(embed, x)
# the shape of v is (?,max_step,d)
```

Tensorflow: About RNN

- RNN for Sentence Classification
  - RNN Framework



Tensorflow: About RNN

- A Simple RNN

```
h = tf.zeros((1, d))
for step in range(max_step):
    with tf.variable_scope('rnn', reuse=step>0):
        with tf.variable_scope('W'):
            parts_w = layers.linear(v[:,step,:], d)
        with tf.variable_scope('U'):
            parts_u = layers.linear(h, d)
        h = tf.nn.tanh(parts_w+parts_u)

with tf.variable_scope('classifier'):
    y = layers.linear(h, 5)

# 'embed:0': (10000, 300)
# 'rnn/W/fully_connected/weights:0': (300, 300)
# 'rnn/W/fully_connected/biases:0': (300)
# 'rnn/U/fully_connected/weights:0': (300, 300)
# 'rnn/U/fully_connected/biases:0': (300)
# 'classifier/fully_connected/weights:0': (300, 5)
# 'classifier/fully_connected/biases:0': (5)
```

Tensorflow: About RNN

- Using RNNCell Module

```
from tensorflow.contrib.rnn import BasicRNNCell

cell = BasicRNNCell(d)
h = cell.zero_state(batch_size, tf.float32)

for step in range(max_step):
    with tf.variable_scope('rnn', reuse=step>0):
        _, h = cell(v[:,step,:], h)

# 'rnn/basic_rnn_cell/weights:0': (600, 300)
# 'rnn/basic_rnn_cell/biases:0': (300)
```

Tensorflow: About RNN

- Using LSTMCell or GRUCell Module

```
from tensorflow.contrib.rnn import GRUCell, LSTMCell

cell = LSTMCell(d)
# 'rnn/lstm_cell/weights:0': (600, 1200)
# 'rnn/lstm_cell/biases:0': (1200)

cell = GRUCell(d)
# 'rnn/gru_cell/gates/weights:0': (600, 600)
# 'rnn/gru_cell/gates/biases:0': (600)
# 'rnn/gru_cell/candidate/weights:0': (600, 300)
# 'rnn/gru_cell/candidate/biases:0': (300)
```

Tensorflow: About RNN

- Using DynamicRNN

```
x = tf.placeholder(tf.int32, shape=[None, None])
x_len = tf.placeholder(tf.int32, shape=[None])
# an example for x and x_len
# x = [[2, 4, 6, 0], [3, 5, 6, 1], [2, 4, 0, 0]]
# x_len = [3, 4, 2]

v = tf.nn.embedding_lookup(embed, x)
# the shape of v is (?,?,d)
```

```
from tensorflow.nn import dynamic_rnn

cell = BasicRNNCell(d)
_, h = dynamic_rnn(cell, v, x_len, dtype=tf.float32)

# 'rnn/basic_rnn_cell/weights:0': (600, 300)
# 'rnn/basic_rnn_cell/biases:0': (300)
```

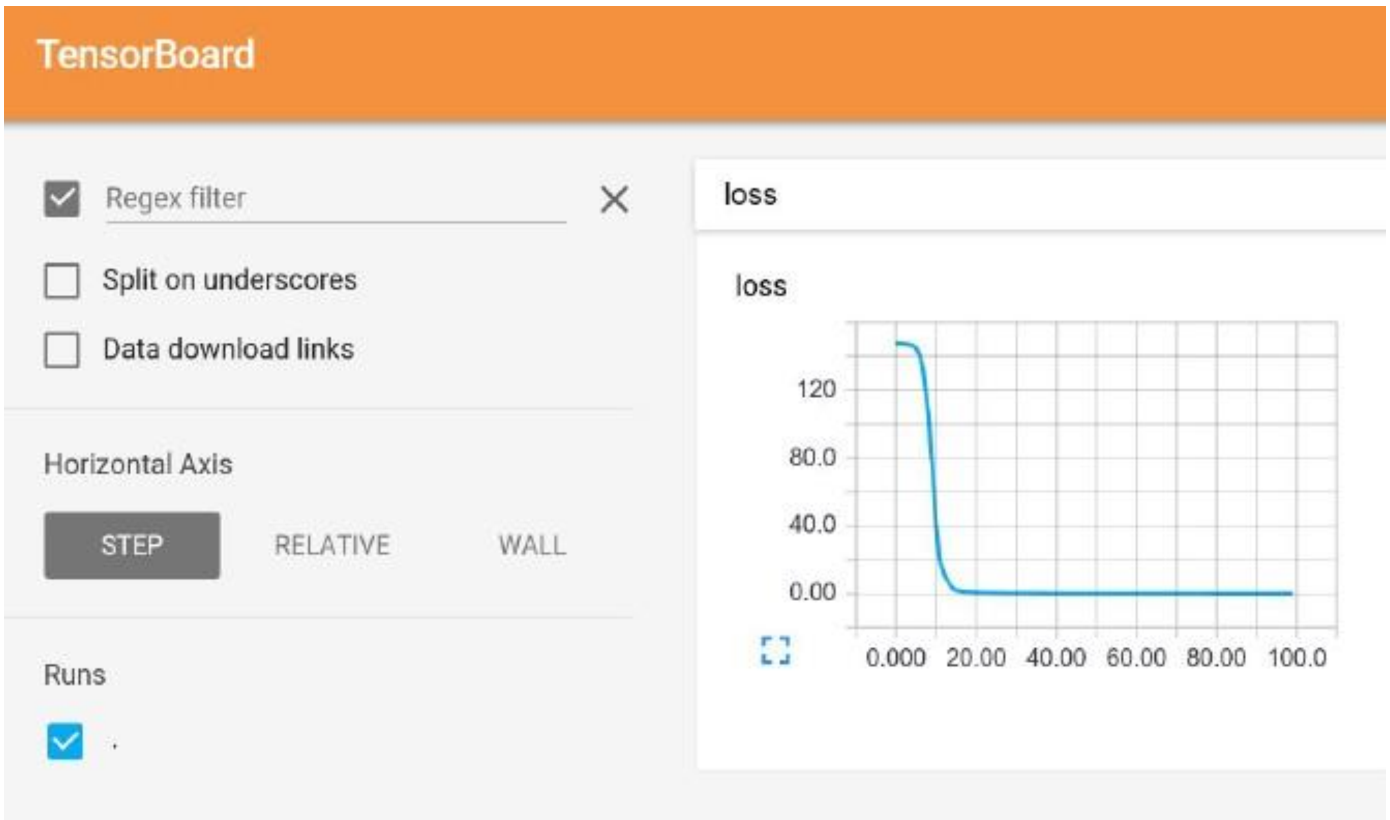
Tensorflow: How to Debug

- tf.Print

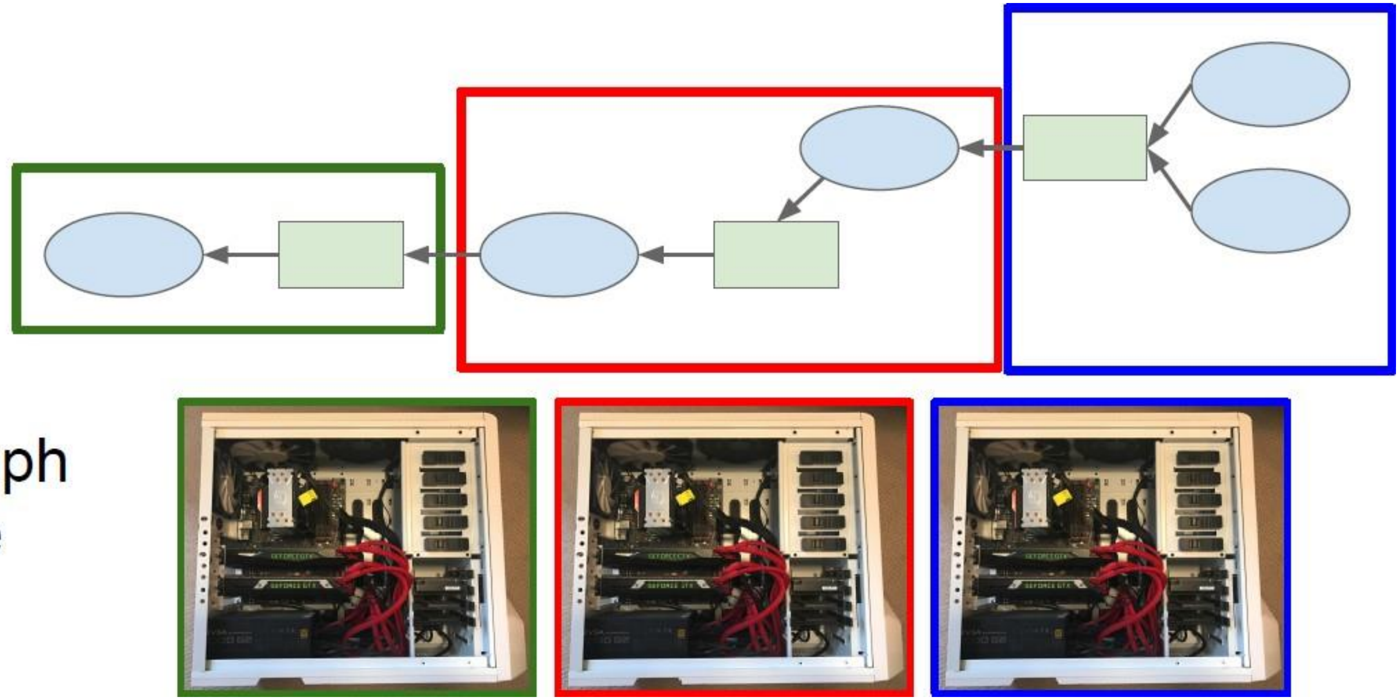
```
import tensorflow as tf

a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
c = tf.add(a, b)
c = tf.Print(c, [tf.shape(c)], summarize=10)

with tf.Session() as session:
    result = session.run(c, {a:[1, 2, 3], b:[4, 5, 6]})
    #I tensorflow/core/kernels/logging_ops.cc:79] [3]

    result = session.run(c, {a:[[1,2],[3,4]], b:[[4,5], [4,6]]})
    #I tensorflow/core/kernels/logging_ops.cc:79] [2 2]
```

Tensorflow: Tensorboard

- Add logging to code to record loss, stats, etc.



Tensorflow: Distributed Version



Split one graph over multiple machines!

https://tensorflow.google.cn/deploy/distributed

Tensorflow: Summary

- Placeholder(dtype,shape,name)
- Variable(dtype,shape,name,scope,reuse,trainable)
- Tensor(dtype,shape)
- Operation, Graph, Session
- Gradient, Optimizer
- Save, Restore, Debug (Tensorboard)
- Distributed Version

Tensorflow: High-level Wrappers

- tf.layers (https://tensorflow.google.cn/api_docs/python/tf/layers)
- tf.contrib.layers (https://tensorflow.google.cn/api_docs/python/tf/contrib/layers)
- tf.estimator (https://tensorflow.google.cn/api_docs/python/tf/estimator)
- tf.keras (https://tensorflow.google.cn/api_docs/python/tf/keras)
- Sonnet (https://github.com/deepmind/sonnet)
- TFLearn (http://tflearn.org/)
- TensorLayer (https://tensorlayer.readthedocs.io/en/latest/)
- ZhuSuan (https://zhusuan.readthedocs.io/en/latest/)

A zoo of frameworks!

PaddlePaddle
(Baidu)

Chainer

Caffe
(UC Berkeley)

Caffe2
(Facebook)

MXNet
(Amazon)
Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

CNTK
(Microsoft)

Torch
(NYU / Facebook)

PyTorch
(Facebook)

JAX
(Google)

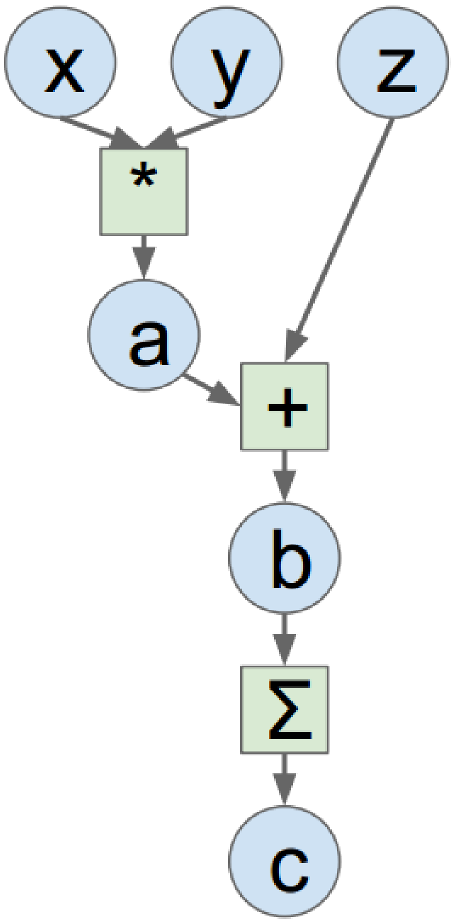
Theano
(U Montreal)

TensorFlow
(Google)

And others...

We'll focus on these

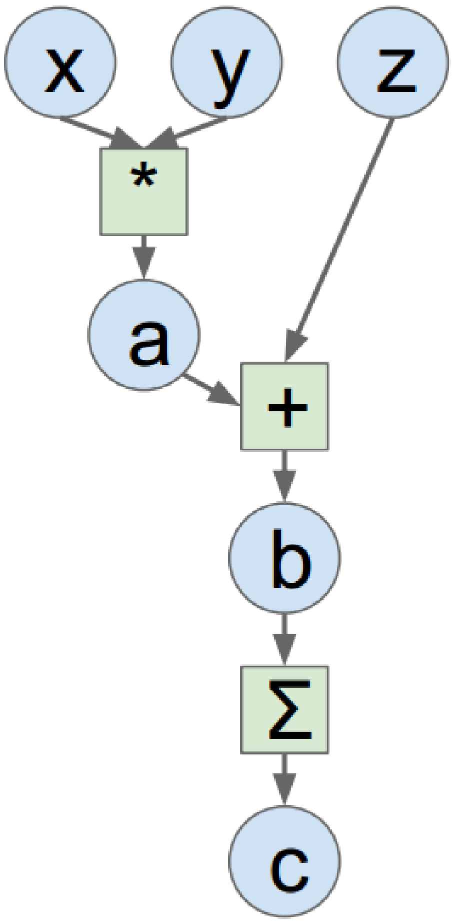
PyTorch: An example




```python
import torch

N, D = 3, 4
x = torch.randn(N, D)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)
```

Looks exactly like numpy!

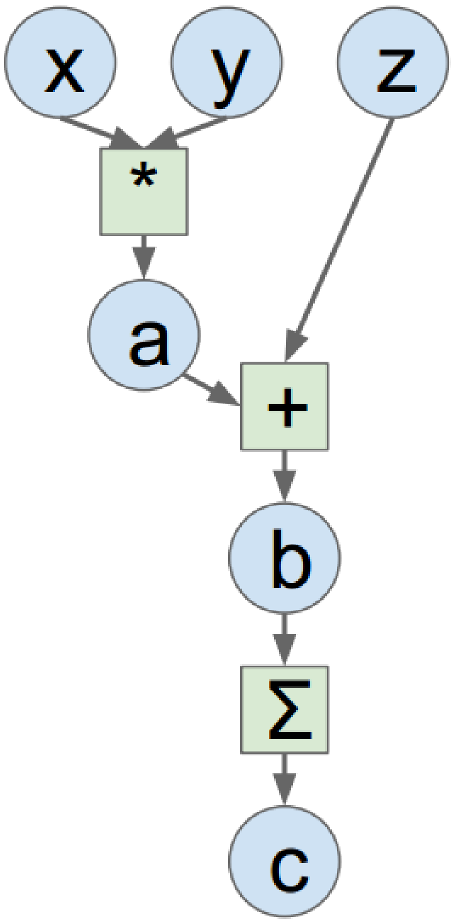
PyTorch: An example




```
import torch

N, D = 3, 4
x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

PyTorch handles gradients for us!

PyTorch: An example




```
import torch

device = 'cuda:0'
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True,
                device=device)
y = torch.randn(N, D, device=device)
z = torch.randn(N, D, device=device)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

Trivial to run on GPU
– just construct arrays on a different device!

PyTorch: Version

- For this slides, we are using PyTorch 1.0
- Be careful if you are looking at older PyTorch code!

PyTorch: Fundamental Concepts

- **Tensor**: Like a numpy array, but can run on GPU
- **Autograd**: Package for building computational graphs out of Tensors, and automatically computing gradients
- **Module**: A neural network layer; may store state or learnable weights

PyTorch: Autograd

We will not want gradients (of loss) with respect to data

Do want gradients with respect to weights

Operations on Tensors with requires_grad=True cause PyTorch to build a computational graph

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

Forward pass looks exactly the same as before, but we don’t need to track intermediate values - PyTorch keeps track of them for us in the graph

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

Compute gradient of loss with respect to w1 and w2

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

Make gradient step on weights, then zero them. Torch.no_grad means “don’t build a computational graph for this part”

PyTorch: Autograd

Clear the gradients, waiting for next loop

PyTorch methods that end in underscore modify the Tensor in-place; methods that don't return a new Tensor

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: nn

Higher-level wrapper for working with neural nets

Use this! It will make your life easier

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

PyTorch: optim

Use an **optimizer** for different update rules

After computing gradients, use optimizer to update params and zero gradient

```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

PyTorch: nn -- Define new Modules

A PyTorch **Module** is a neural net layer; it inputs and outputs Tensors

Modules can contain weights or other modules

You can define your own Modules using autograd!

```
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

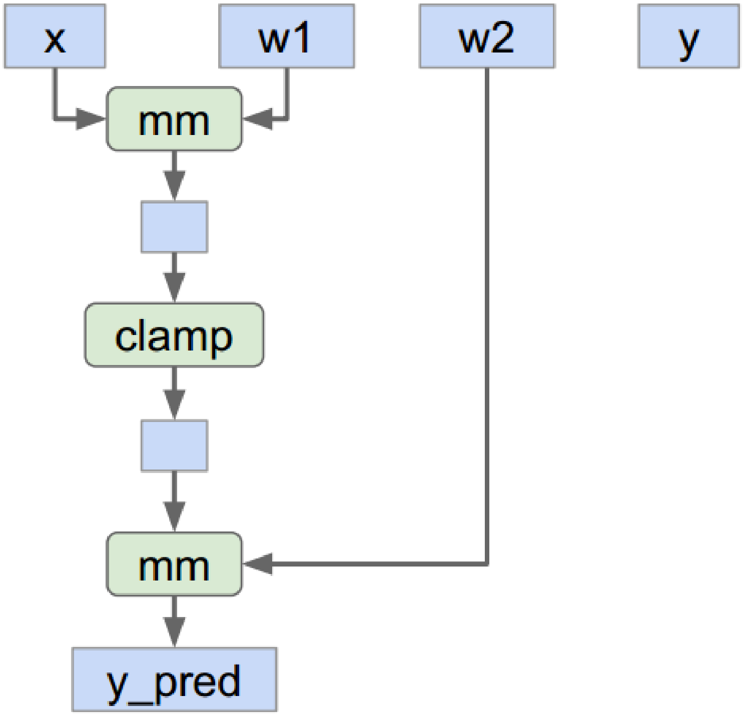
PyTorch: **Dynamic** Computation Graphs

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

PyTorch: **Dynamic** Computation Graphs

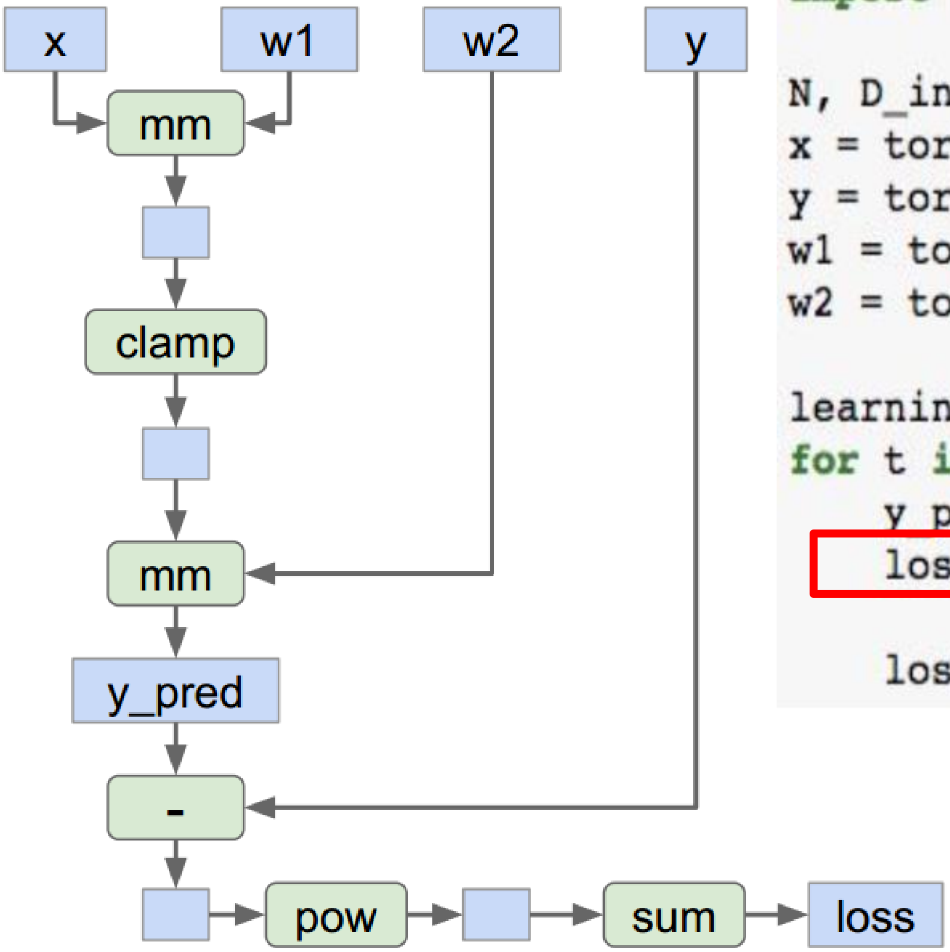

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

PyTorch: **Dynamic** Computation Graphs



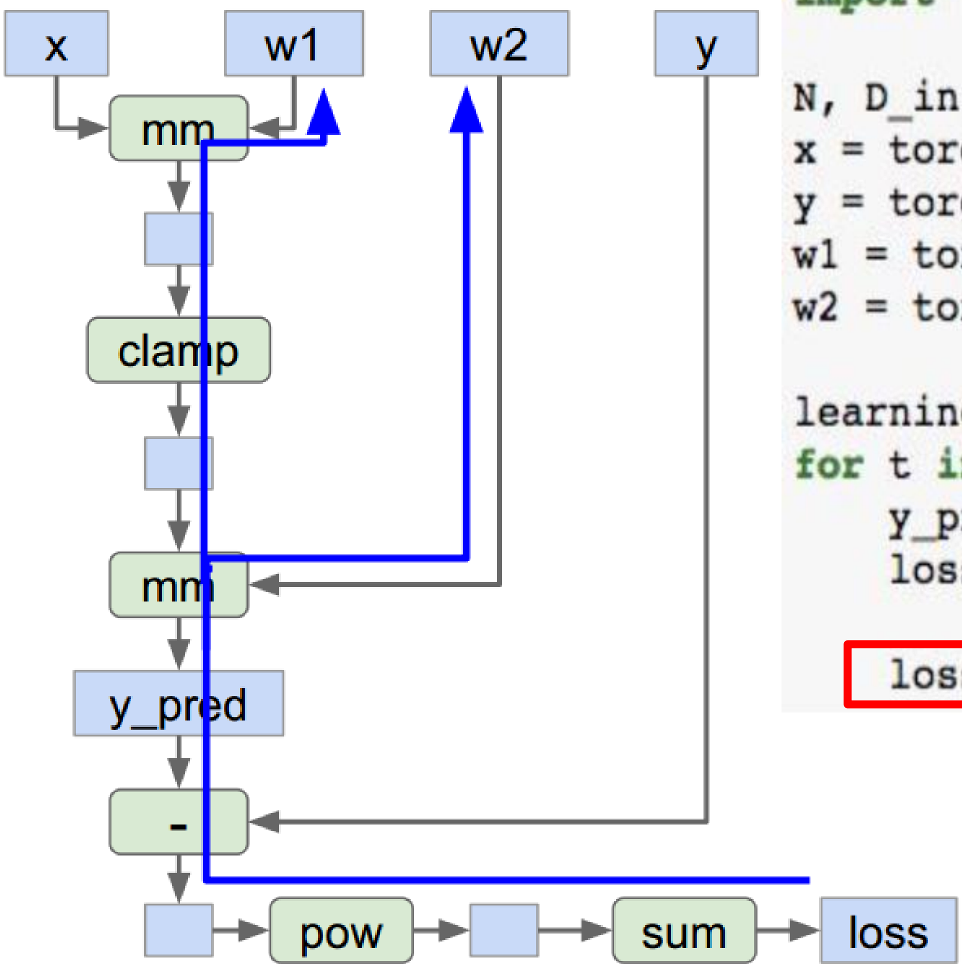

```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

PyTorch: **Dynamic** Computation Graphs



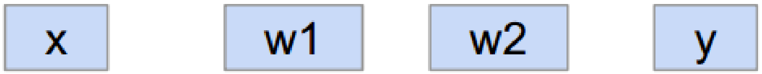

```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

PyTorch: **Dynamic** Computation Graphs




```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

PyTorch: **Dynamic** Computation Graphs




```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Throw away the graph, backprop path, and rebuild it from scratch on every iteration

PyTorch: **Dynamic** Computation Graphs

Building the graph and **computing** the graph happen at the same time.

Seems inefficient, especially if we are building the same graph over and over again...

But useful if you are tackle with varied-length data

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Pytorch: TensorboardX

- A python wrapper around Tensorflow’s web-based visualization tool.
  - pip install tensorboardx

Static vs Dynamic Graphs

TensorFlow: Build graph once, then run many times (**static**)

PyTorch: Each forward pass defines a new graph (**dynamic**)

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)
```

Build graph

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

Run each iteration

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

New graph each iteration

Dynamic TensorFlow: Eager Execution

TensorFlow 2.0 supports **eager execution** which allows dynamic graphs!

Convert input numpy arrays to TF **tensors**. Create weights as tf.Variable

```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights

with tf.GradientTape() as tape:
  h = tf.maximum(tf.matmul(x, w1), 0)
  y_pred = tf.matmul(h, w2)
  diff = y_pred - y
  loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
gradients = tape.gradient(loss, [w1, w2])
```

Use tf.GradientTape() context to build **dynamic** computation graph.

All forward-pass operations in the contexts (including function calls) gets traced for computing gradient later.

tape.gradient() uses the traced computation graph to compute gradient for the weights

Personal Advice

- **TensorFlow pre2.0:** safe bet for most projects, not perfect but huge community, wide usage. Better pair with high-level wrapper (Keras, Sonnet, etc)

- **Tensorflow 2.0:** still new

- **PyTorch**: Personal choice. Best for research. Lighter than tensorflow, easy to debug.

Links

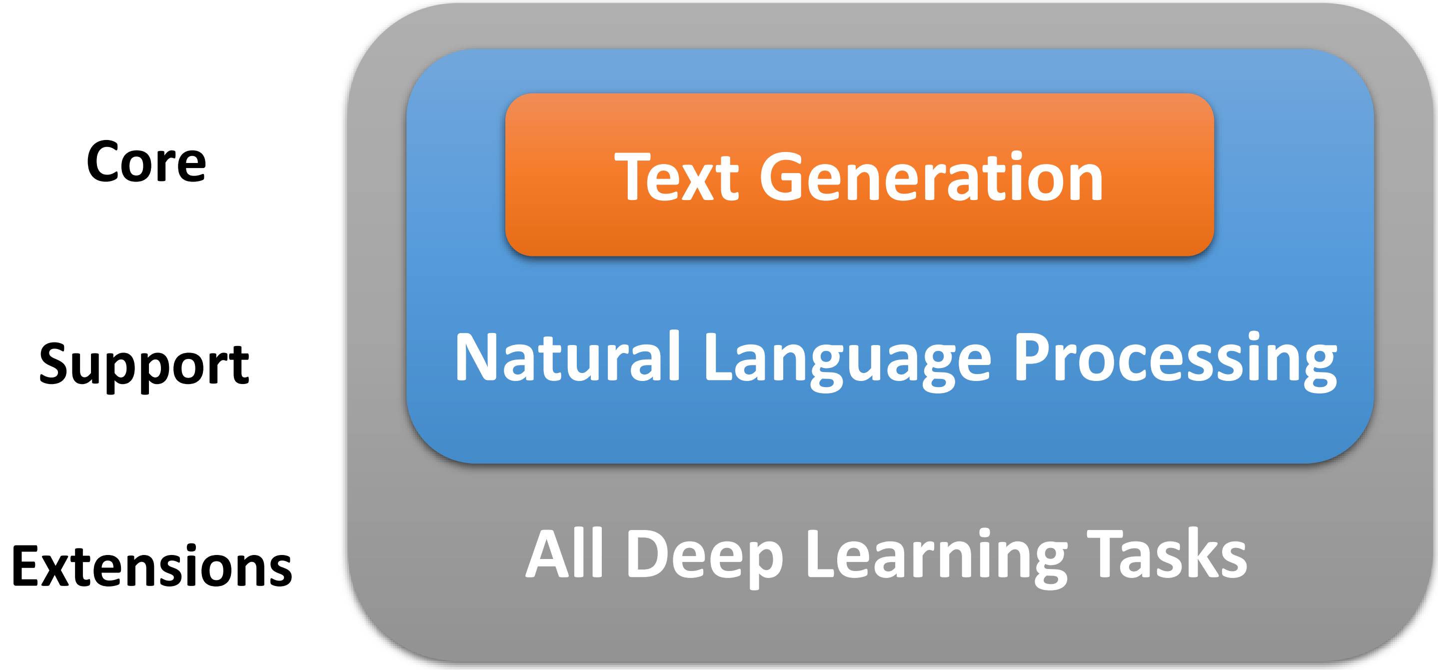
- https://tensorflow.google.cn
- https://pytorch.org
- https://keras.io

Conversational Toolkits (CoTK)

HUANG Fei

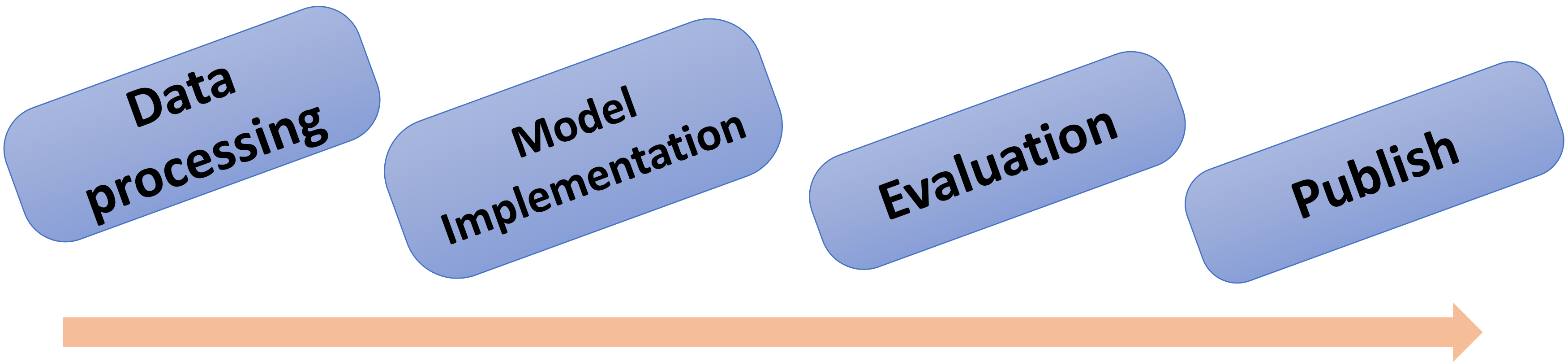
What is CoTK

- A toolkit for deep learning researchers



Design Concept

What can CoTK do



Help you throughout your workflow

Data processing	Datasets and Preprocess	torchtext
Model Implementation	Baselines	Github
Evaluation	Metrics	NLTK
Publish	Reproduce Experiments	SACRED

Advantages

- **A few lines** to start
- Light-weight and **framework independent**
- Predefined standard datasets
- Predefined baseline models
- Compare models **fairly**
- **Reproduce** your and others' experiments

Overview

- An example: Implement a GRU LM
- Quick Start
  - Installation
  - Dataloader
  - Metrics
  - Publish Experiments
  - Reproduce Experiments
  - Predefined models
- Extending CoTK

Quick Start

Installation

- cmd: pip install cotk
- Github: github.com/thu-coai/cotk
- Homepage:
  http://coai.cs.tsinghua.edu.cn/dialtk/cotk/
- Tutorials:
  https://thu-coai.github.io/cotk_docs/

Dataloader

- Automatically download online resources
  - cotk.dataloader.MSCOCO("resources://MSCOCO_small")
- Download from a url
  - cotk.dataloader.MSCOCO("http://cotk-data.s3-ap-northeast-1.amazonaws.com/mscoco_small.zip#MSCOCO")
- Import from local file
  - cotk.dataloader.MSCOCO("./MSCOCO.zip#MSCOCO")

NAME @ SOURCE # Preprocessor

Dataloader

- Inspect vocabulary list
  - dataloader.vocab_size

    Vocabulary size: 2588
  - dataloader.vocab_list[:10]

    ['<pad>', '<unk>', '<go>', '<eos>', '.', 'a', 'A', 'on', 'of', 'in’]

- Convert between ids and strings
  - dataloader.convert_tokens_to_ids(["<go>", "hello", "world", "<eos>"])
  - dataloader.convert_ids_to_tokens([2, 1379, 1897, 3])

Dataloader

- Iterative over batch
  - for data in dataloader.get_batch("train", batch_size=1):
    print(data)

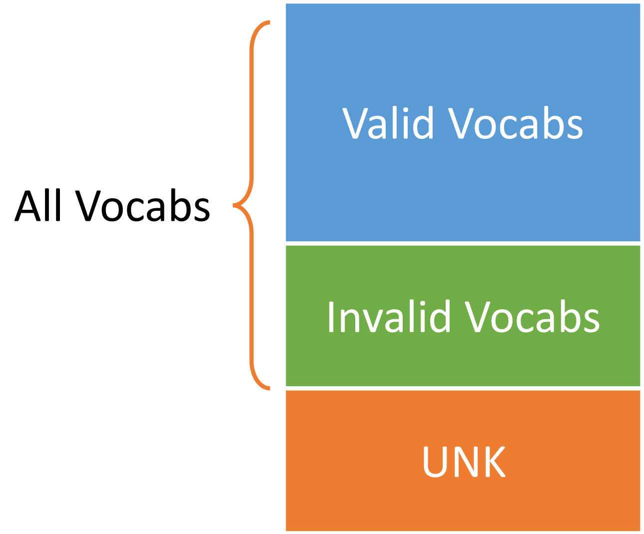
{**'sent**’:

array([[2, 181, 13, 26, 145, 177, 8, 22, 12, 5, **1,** 1099, 4, 3]]),
<go> This is an old photo of people and a <unk> wagon.
'**sent_allvocabs**’:
array([[2, 181, 13, 26, 145, 177, 8, 22, 12, 5, **3755**, 1099, 4, 3]]),
<go> This is an old photo of people and a horse-drawn wagon.
'**sent_length**': array([14])

}

Valid / Invalid / Unknown Vocabs

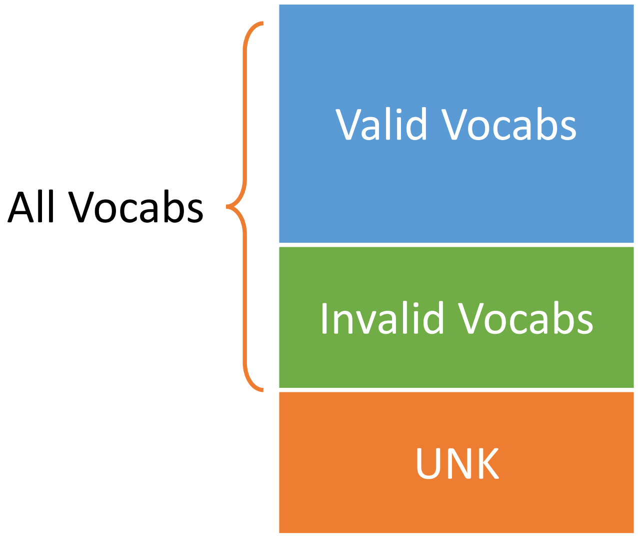
- Valid vocabs
  - From training set
  - Appear > min_vocab_times
  - Model should read and generate



- Invalid vocabs
  - From train & dev & test set
  - Appear > invalid_vocab_times, but not valid
  - Model can optionally read or generate in test stage (like copyNet)

Valid / Invalid / Unknown Vocabs

- Unknown vocabs
  - Not valid vocabs or invalid vocabs
  - Model can’t know or generate



- Why ?
  - Transfer models between different datasets
  - Compare between models with the same allvocabs
  - Compare between common generate model & copyNet

Valid / Invalid / Unknown Vocabs

- How ?
  - Most models only care about valid vocabs
    - Transferable if using the same valid vocabs

  - Metrics care about valid & invalid vocabs
    - Perplexity -> smoothing <unk> to invalid vocabs
    - Bleu -> <unk> never matched
    - Comparable if all_vocabs is the same

Dataloader

- LanguageGeneration
  - MSCOCO
- SingleTurnDialog
  - OpenSubtitles
- BERTSingleTurnDialog
  - BERTOpenSubtitles
- MultiTurnDialog
  - UbuntuCorpus
- SentenceClassification
  - SST

Metrics

- Metric Pipeline

```
metric = cotk.metric.xxMetric(dataloader)
metric.forward(…)
metric.close()
```

- MetricChain: Merge multiple metrics

```
metric = cotk.metric.MetricChain()
metric.add_metric(metricA)
metric.add_metric(metricB)
metric.forward(…)
metric.close()
```

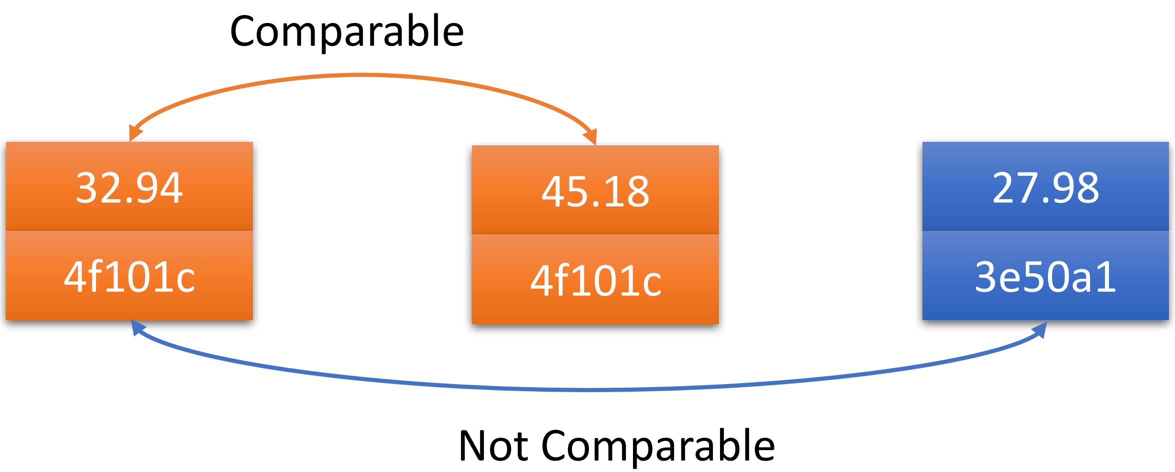
Metrics

- Predefined metrics for given tasks

```
metric = dataloader.get_inference_metric(gen_key="gen")
metric.forward({
    "gen":
        [[2, 181, 13, 26, 145, 177, 8, 22, 12, 5, 3755, 1099, 4, 3],
         [2, 46, 145, 500, 1764, 207, 11, 5, 93, 7, 31, 4, 3]]
})
print(metric.close())
{
   'self-bleu': 0.0220,
    'self-bleu hashvalue': 'c206..',
    'fw-bleu': 0.383, 'bw-bleu': 0.0259, 'fw-bw-bleu': 0.0486
    'fw-bw-bleu hashvalue': '530d...',
}
```

Metrics

- Hashvalue: Make sure we have the same
  - References
  - Settings
  - Version

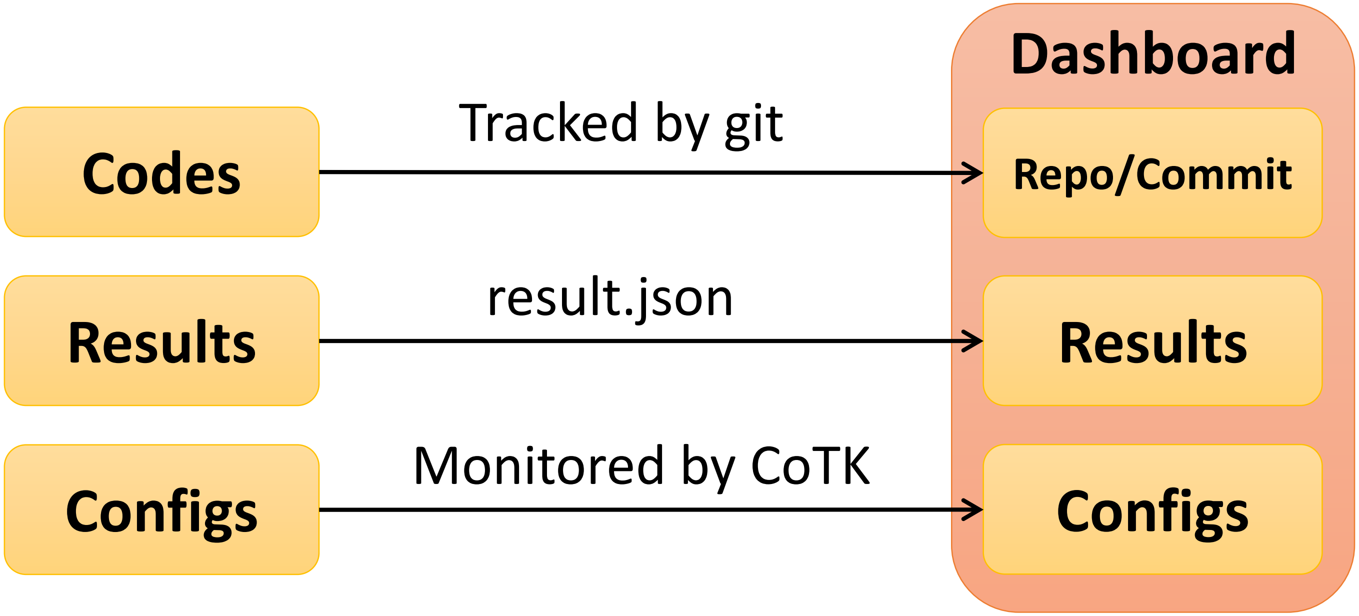


Metrics

- PerplexityMetric
- BleuCorpusMetric
- SelfBleuCorpusMetric
- FwBwBleuCorpusMetric
- BleuPrecisionRecallMetric
- EmbSimilarityPrecisionRecallMetric
- NgramFwBwPerplexityMetric

Publish Experiments

- Make a git repo
- Write your model
  - Output your result to result.json
- Use “cotk run” command



Publish Experiments

Dashboard Home Records Log in Sign up

Extra columns (Corresponding to keys in 'result', separated by commas):

perplexity,perplexity hashvalue,bleu,bleu hashvalue [Submit]

Uploaded by | Github commit | file_id | [Search]

Show 10 entries

ID	User	Github commit	Dataloader	perplexity	perplexity hashvalue
#5	hzhwcmhf	thu-coai/seq2seq-pytorch@aa1869	OpenSubtitles (resources://OpenSubtitles_small)	94.698	460f26
#4	Hikari	thu-coai/seq2seq-pytorch@eff99e	OpenSubtitles (resources://OpenSubtitles_small)	94.698	460f26
#3	Hikari	thu-coai/seq2seq-pytorch@5860c0	OpenSubtitles (resources://OpenSubtitles)	43.868	104528

Publish Experiments

- Dashboard
  - Compare with others
  - Manage your experiments
  - Hold competitions

Reproduce Experiments

- Download code from dashboard
  - cotk download ID
- Download code from github
  - cotk download USERNAME/REPO/COMMIT

- Cotk recover codes, cmdlines
  - but not weights
  - Publishers should make their codes reproducible

Predefined models

- cotk download thu-coai/seq2seq-pytorch/master

- LanguageModel
- VAE
- SeqGAN
- Seq2seq
- HRED
- CVAE

Extending CoTK

New dataset

- Use existing dataloader & Write your preprocessor
  - LanguageGeneration("./path/to/your_data#your_processor")

- Override the base dataloader
  - Class MyDataloader(LanguageGeneration)
  - Just define some fields, don’t need to build vocab list again

New metrics

- Override MetricBase
  - Implement __init__, forward, close

```python
class AverageLengthMetric(MetricBase):
    def __init__(self, dataloader, gen_key="gen"):
        super().__init__()
        self.dataloader = dataloader
        self.gen_key = gen_key
        self.token_num = 0
        self.sent_num = 0

    def forward(self, data):
        gen = data[gen_key]
        for sent in gen:
            self.token_num += len(self.dataloader.trim_index(sent))
            self.sent_num += 1

    def close(self):
        return {"len_avg": self.token_num / self.sent_num}
```

New models

- Just make your model reproducible
- And upload it to dashboard

- We’ll check whether the report is reproducible
- You can become one of our baseline

Contribution to CoTK

- We will create a package named cotk_contrib

- Define your datasets, metrics, models
- Every contribution can be made regardless of quality
- Good contribution will be merged into CoTK

THANK YOU

- cmd: pip install cotk
- Github: github.com/thu-coai/cotk