Introduction to Tensorflow

Feb 7th, 2019

Outline

- Tensor
- Placeholder and Feed Dictionary
- Variable
- Optimizer
- Computation Graph
  - Build Graph
  - Run Graph
- Automatic Differentiation
- Use GPU
- Parallel and Distributed Training
- Tensorboard
- Example

Tensorflow

- Tensorflow is an open source software library for high performance numerical computation.
- Define functions on tensors and automatically compute gradients.
- Design, build and train deep learning models.
- Can be used for many kinds of applications, including, computer vision, natural language processing, reinforcement learning, etc.
- Multiple devices, Parallel and Distributed training

Tensor

- Scalars: single number
  Vectors: an array of number
  Matrices: 2-D array of numbers

- Tensor is generalization of scalar, vector and matrix,   multidimensional array
  Example, computer vision, 4-D tensors are used,
  dimensions with batch size, image width, image height, and color channels.
  my_image = tf.zeros([10, 224, 224, 3])

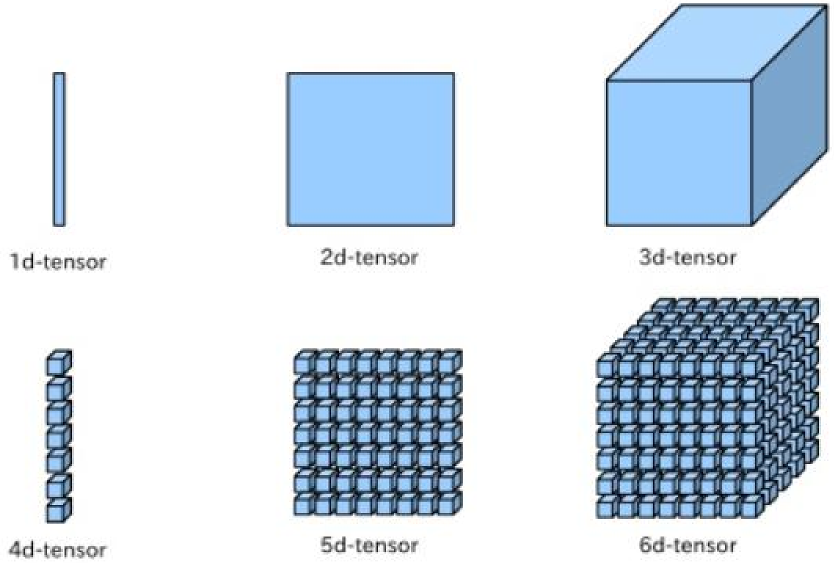


Image credit to knoldus

Placeholder and Feed Dictionary

- A placeholder is a variable that we will assign data to at a later time.
- Create operations and build computation graph, without needing the data.
- Tensors that depend on placeholders can't be evaluated without providing a value for the placeholder.

  Example

```
#build graph
p = tf.placeholder(tf.float32)
t = p + 1.0
#run graph
t.eval()  wrong! t depends on p (placeholder)
t.eval(feed_dict={p:2.0})  correct!
```

Variable

- A tf.Variable represents a tensor whose value can be changed by running ops on it.

- Create a variable is to call the tf.get_variable function
  Example,
  cons = tf.get_variable("scalar", initializer=tf.constant(3))
  matrix = tf.get_variable("matrix", initializer=tf.constant([[1, 2], [3, 4]]))

- Initializing variables
  By default, tf.Variable gets placed in the tf.GraphKeys.GLOBAL_VARIABLES
  we can initialize these variables by calling by simply calling
  session.run(tf.global_variables_initializer())

Variable

```
def conv_relu(input, kernel_shape, bias_shape):

    weights = tf.get_variable("weights", kernel_shape,
        initializer=tf.random_normal_initializer())

    biases = tf.get_variable("biases", bias_shape,
        initializer=tf.constant_initializer(0.0))

    conv = tf.nn.conv2d(input, weights,
        strides=[1, 1, 1, 1], padding='SAME')
    return tf.nn.relu(conv + biases)


input1 = tf.random_normal([1,10,10,32])
input2 = tf.random_normal([1,20,20,32])


x = conv_relu(input1, kernel_shape=[5, 5, 32, 32], bias_shape=[32])
x = conv_relu(x, kernel_shape=[5, 5, 32, 32], bias_shape = [32])
```

wrong! tensorflow does not know to create new variable or use existing ones!

Variable

- Create new variables

```
def my_image_filter(input_images):
    with tf.variable_scope("conv1"):
        # Variables created here will be named "conv1/weights", "conv1/biases".
        relu1 = conv_relu(input_images, [5, 5, 32, 32], [32])
    with tf.variable_scope("conv2"):
        # Variables created here will be named "conv2/weights", "conv2/biases".
        return conv_relu(relu1, [5, 5, 32, 32], [32])
```

- Sharing variables

```
with tf.variable_scope("model"):
    output1 = my_image_filter(input1)
with tf.variable_scope("model", reuse=True):
    output2 = my_image_filter(input2)
```

way 1

```
with tf.variable_scope("model") as scope:
  output1 = my_image_filter(input1)
  scope.reuse_variables()
  output2 = my_image_filter(input2)
```

way 2

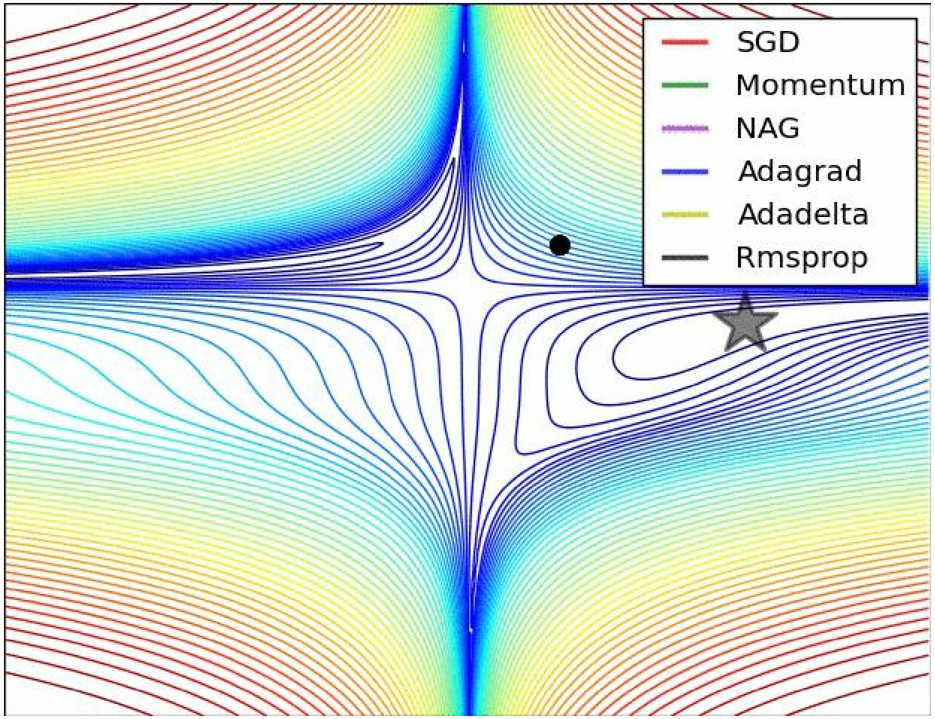
Optimizer

- Training neural network involves minimizing loss function $J(\theta)$

  The simplest approach would be stochastic gradient descent

  $$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

- Other approaches, e.g. Momentum, Adagrad, etc, use momentum or history gradient information



Optimizer

- add training operation to the computation graph, based on variants of gradient descent algorithms

- Direct usage:

```
opt = GradientDescentOptimizer(learning_rate=0.1)
opt_op = opt.minimize(cost, var_list=<list of variables>)
opt_op.run()
```

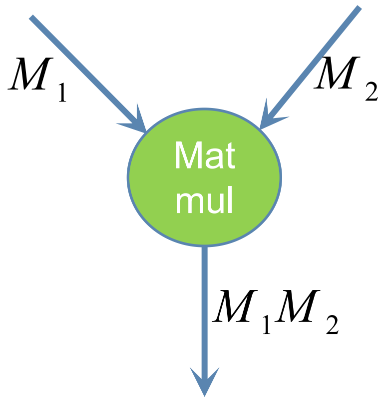
- Processing gradients before applying them:

```
opt = GradientDescentOptimizer(learning_rate=0.1)
grads_and_vars = opt.compute_gradients(loss, <list of variables>)
capped_grads_vars = [[tf.clip_by_value(g, -max_norm,  max_norm), v]  for g, v in grads_and_vars]
opt.apply_gradients(capped_grads_and_vars)
```

- You could use other optimizers, e.g.,
  AdagradDAOptimizer,
  AdamOptimizer.

Computation Graph

- A directed, acyclic graph
- Tensorflow graph
  nodes represent computation operation (a simple function of one or more variable)
  edges represent data needed for the operation or the results of the computation
- Example, tf.matmul operation would correspond to
  a single node (matrix multiplication operation)
  two incoming edges (the matrices to be multiplied)
  one outgoing edge (the result of the multiplication).



Why Computation Graph

- Advantage of using computation graph
  - Parallel computation
    using edges to represent dependencies between operations,
    can be used for identifying operations that can run in parallel.

  - Distributed Computation
    using edges to represent the data that flow between operations
    TensorFlow can partition program across multiple devices (CPUs, GPUs, and TPUs)

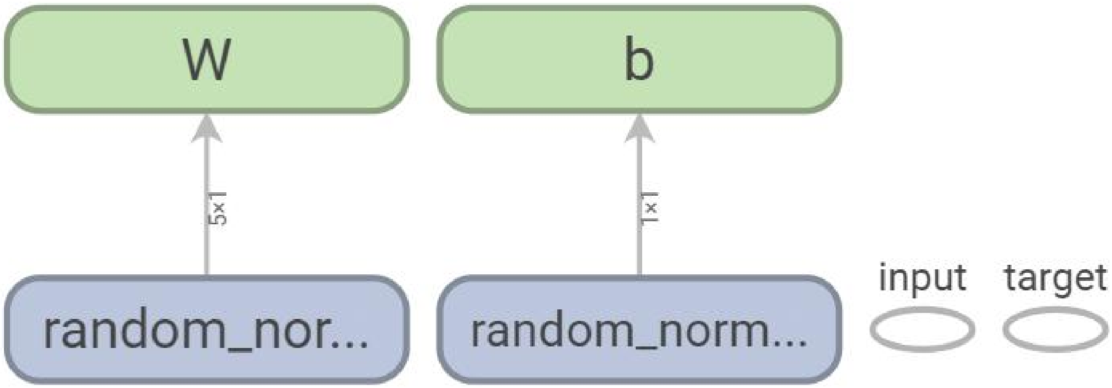
  - Auto-differentiation
    Partition the computation into small, differential component to facilitate calculating gradient

- Tensorflow program consist of two steps:
  step1: build computation graph, tf.Graph
  step2: run the computation graph, tf.Session

Build Graph

- Example, build the computation graph of training logistic regression

```
W = tf.Variable(tf.random_normal(shape=[5, 1]), name='W')
b = tf.Variable(tf.random_normal(shape=[1, 1]), name='b')
input = tf.placeholder(dtype=tf.float32, shape=[None, 5], name='input')
target = tf.placeholder(dtype=tf.float32, shape=[None, 1], name='target')
```

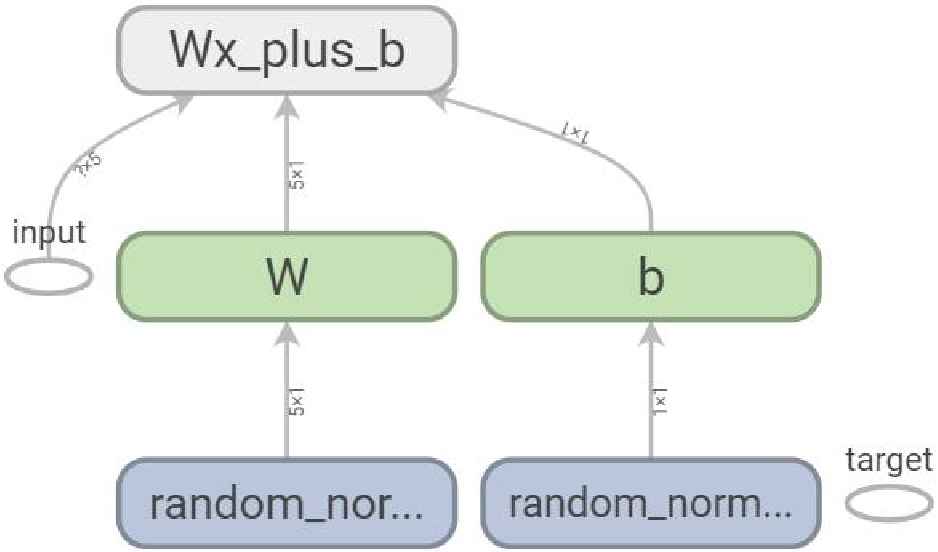


visualization by tensorboard

Build Graph

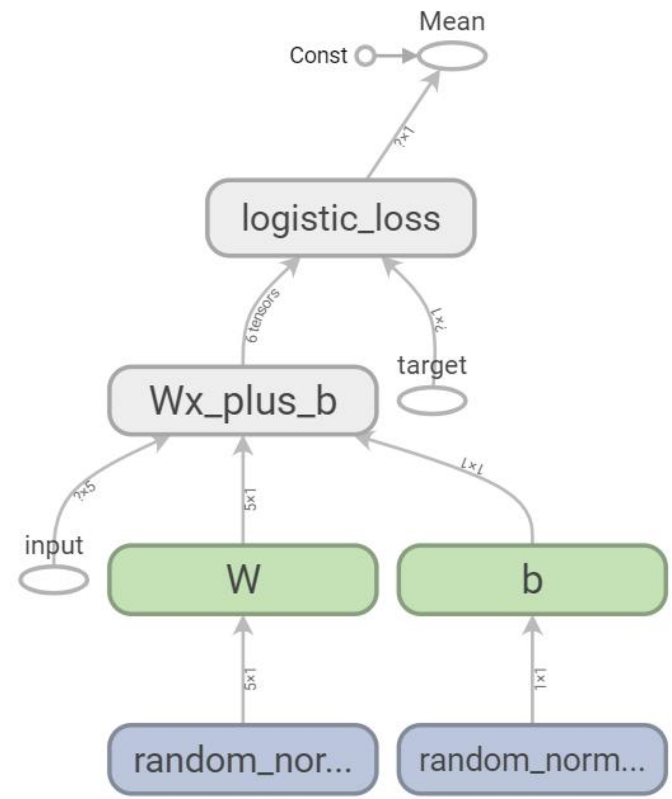
```
W = tf.Variable(tf.random_normal(shape=[5, 1]), name='W')
b = tf.Variable(tf.random_normal(shape=[1, 1]), name='b')
input = tf.placeholder(dtype=tf.float32, shape=[None, 5], name='input')
target = tf.placeholder(dtype=tf.float32, shape=[None, 1], name='target')
```

```
with tf.name_scope('Wx_plus_b'):
    pred_logits = tf.matmul(input, W) + b
```



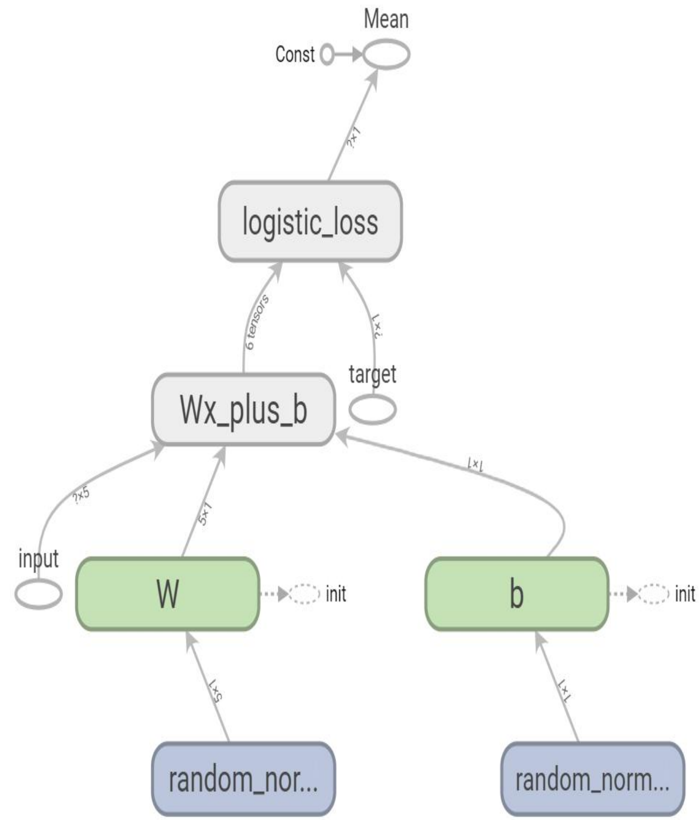
Build Graph

```
W = tf.Variable(tf.random_normal(shape=[5, 1]), name='W')
b = tf.Variable(tf.random_normal(shape=[1, 1]), name='b')
input = tf.placeholder(dtype=tf.float32, shape=[None, 5],
name='input')
target = tf.placeholder(dtype=tf.float32, shape=[None, 1],
name='target')

with tf.name_scope('Wx_plus_b'):
    pred_logits = tf.matmul(input, W) + b

loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with
_logits(logits=pred_logits, labels=target))
```



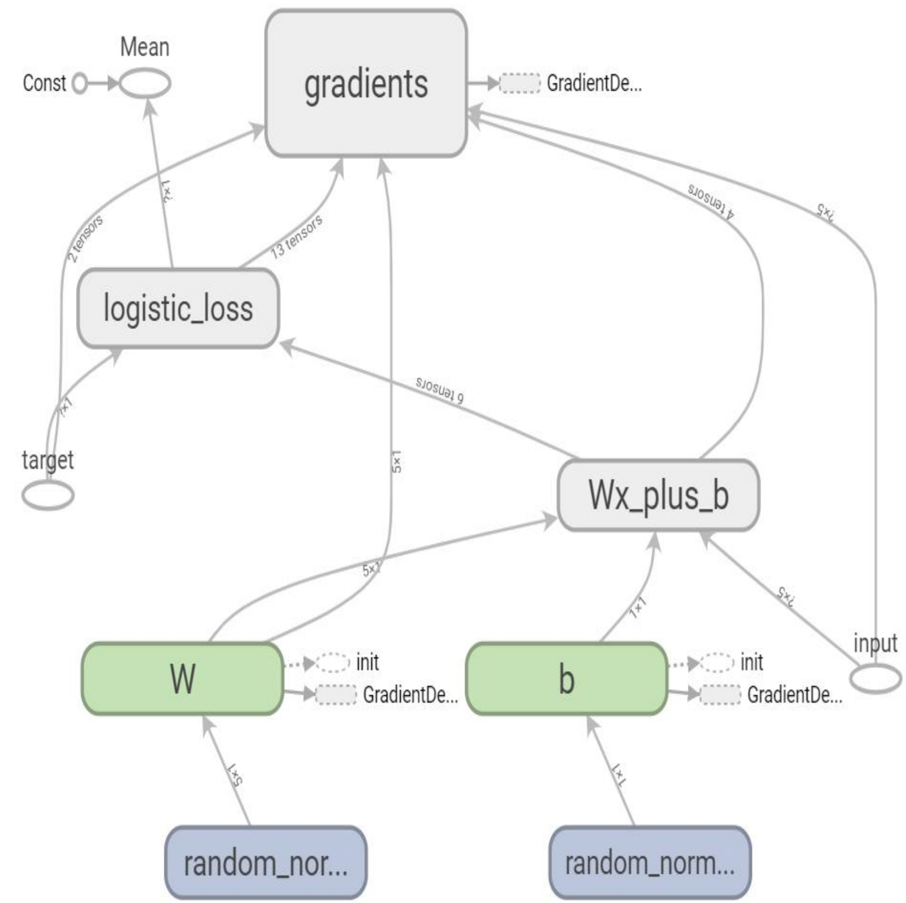
Build Graph

```
W = tf.Variable(tf.random_normal(shape=[5, 1]), name='W')
b = tf.Variable(tf.random_normal(shape=[1, 1]), name='b')
input = tf.placeholder(dtype=tf.float32, shape=[None, 5],
name='input')
target = tf.placeholder(dtype=tf.float32, shape=[None, 1],
name='target')

with tf.name_scope('Wx_plus_b'):
    pred_logits = tf.matmul(input, W) + b

loss =tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits
(logits=pred_logits, labels=target))

init = tf.global_variables_initializer()
```



Build Graph

```
W = tf.Variable(tf.random_normal(shape=[5, 1]), name='W')
b = tf.Variable(tf.random_normal(shape=[1, 1]), name='b')
input = tf.placeholder(dtype=tf.float32, shape=[None, 5],
name='input')
target = tf.placeholder(dtype=tf.float32, shape=[None, 1],
name='target')

with tf.name_scope('Wx_plus_b'):
    pred_logits = tf.matmul(input, W) + b

loss=tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with
_logits(logits=pred_logits, labels=target))
init = tf.global_variables_initializer()

learning_rate = 0.003
opt = tf.train.GradientDescentOptimizer(learning_rate)
train_op = opt.minimize(loss)
```



Run the Graph with Session

- Create a tf.Session for the current default graph

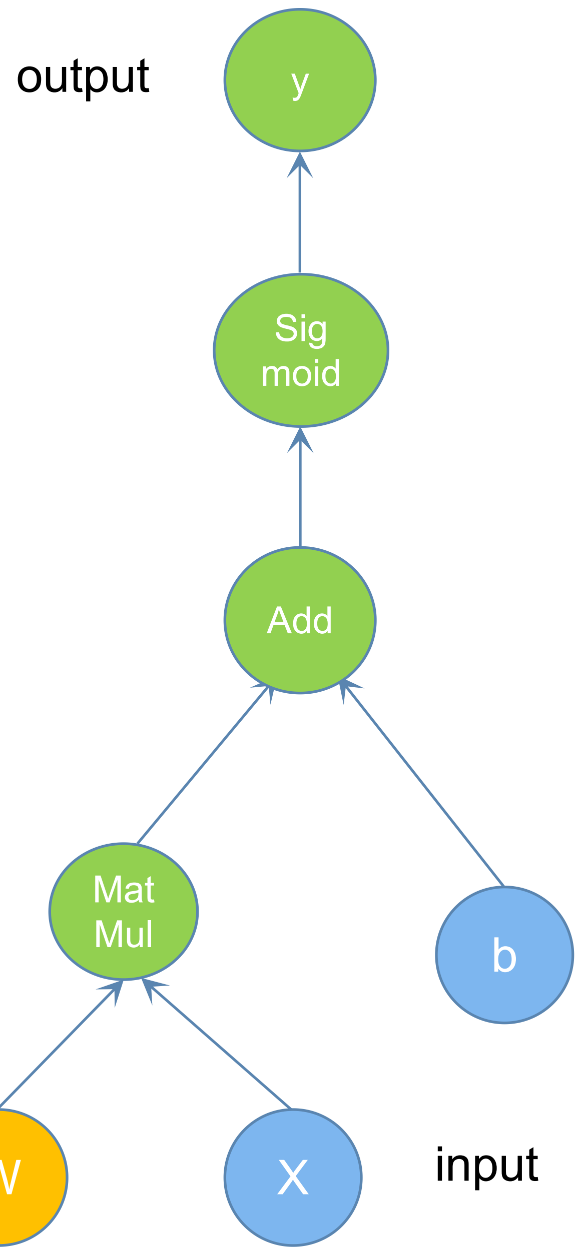
- tf.Session.run method is the main mechanism for
  running a tf.Operation or,
  evaluating a tf.Tensor

- You can pass one or more tf.Operation or tf.Tensor objects to tf.Session.run
  TensorFlow will execute the operations that are needed to compute the result.

- Run the training operation
  continue previous example

```
x_input = np.random.normal(loc =0.0, scale = 1.0, size = [3,5])
y_true = np.array([[0], [0], [1]])
sess = tf.Session()
sess.run(init)
for i in range(iter_num):
    _, pred, logss = sess.run([train_op, pred_logits, loss], feed_dict={input: x_input,
 target: y_true})
```

Automatic Differentiation



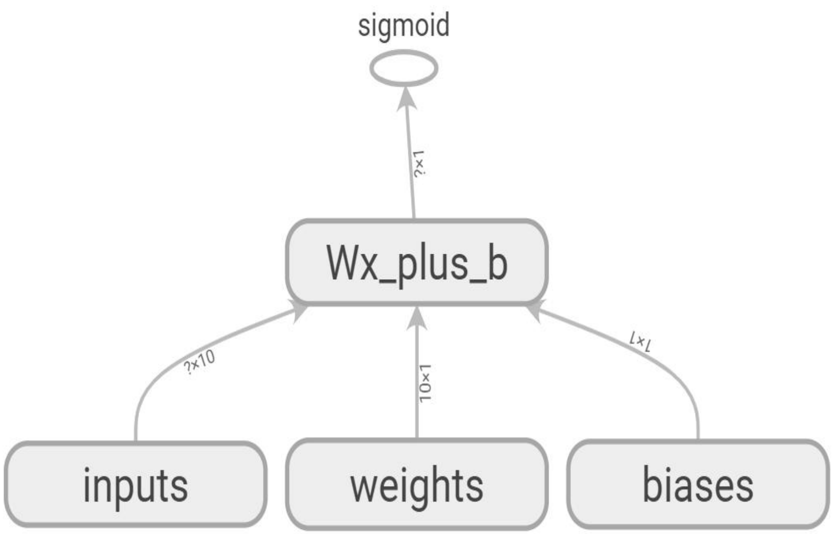
How to calculate the gradient? $\frac{\partial y}{\partial W}$ $\frac{\partial y}{\partial b}$

Simply use the function tf.gradients
grad = tf.gradients(ys, xs)
Constructs symbolic derivatives of sum of ys w.r.t. x in xs.

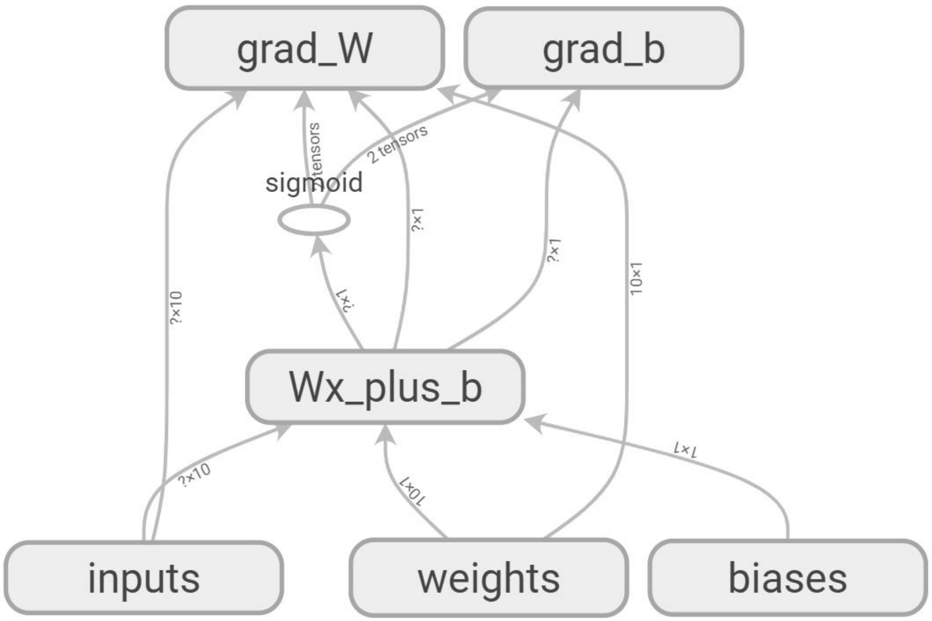
grad_W = tf.gradients(y, W)
grad_b = tf.gradients(y, b)

Automatic Differentiation

- tf.gradients() adds ops to the graph to output the derivatives of ys with respect to xs



graph before taking gradient



graph after taking gradient

visualization by tensorboard

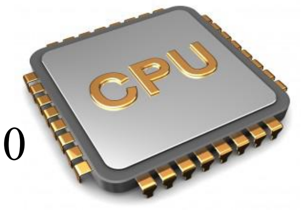
Use GPU

- using GPU to speed up computation
  supported device types are CPU and GPU
  "/cpu:0"   CPU of your machine
  "/device:GPU:0" first GPU of your machine
  "/device:GPU:1" second GPU of your machine

- Manual device placement
  # build graph.

```
with tf.device('/cpu:0'):
    a = tf.constant([1.0, 2.0, 3.0, 4.0], shape=[2, 2], name='a')
    b = tf.constant([3.0, 4.0, 5.0, 6.0], shape=[2, 2], name='b')
```

cpu:0




```
with tf.device('/gpu:0'):
    c = tf.matmul(a, b)
```

gpu:0

```
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
# run graph.
print(sess.run(c))
```

Use GPU

- Allowing GPU memory growth
  By default, TensorFlow maps nearly all of the GPU memory of all GPUs
  allocate only as much GPU memory based on runtime allocations

```
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
session = tf.Session(config=config, ...)
```

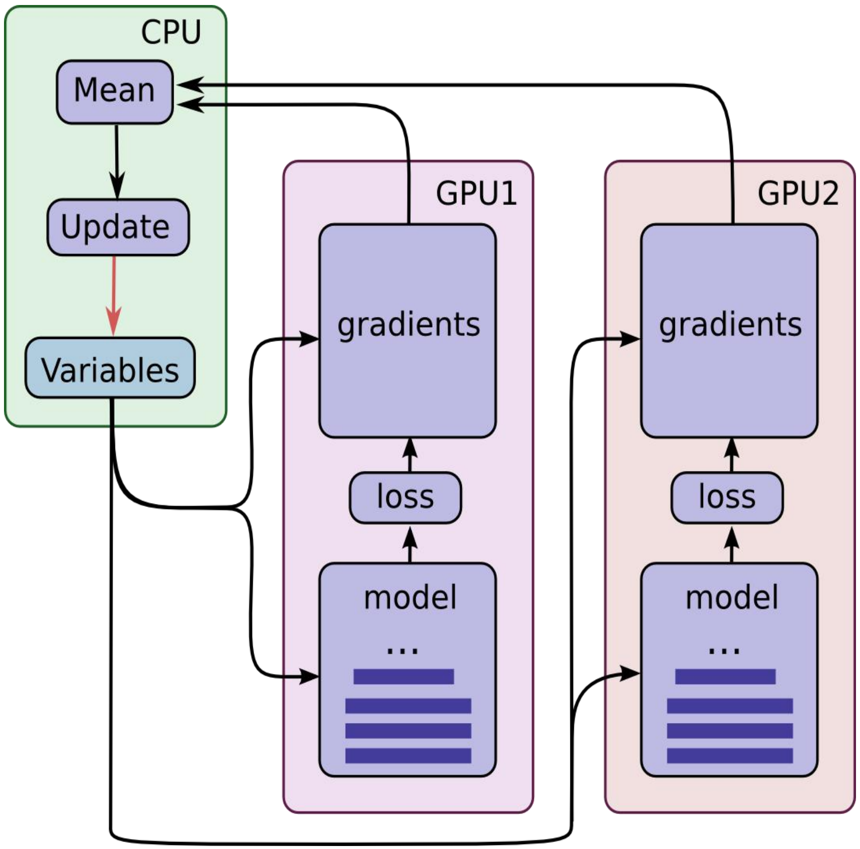
- Using a single GPU on a multi-GPU system

```
#build graph.
with tf.device('/device:GPU:2'):
        a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
        b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
        c = tf.matmul(a, b)
# Creates a session with log_device_placement set to True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
# run graph
print(sess.run(c))
```

gpu:0 gpu:1 gpu:2

Use GPU

- Use multi-GPUs on a single machine
  copy the model on each GPU
  split the batch data and send to each GPU
  wait for all GPUs to finish processing a batch of data and then update the model parameters



Parallel and Distributed training

- Data and Model Parallelism

  - Data Parallelism

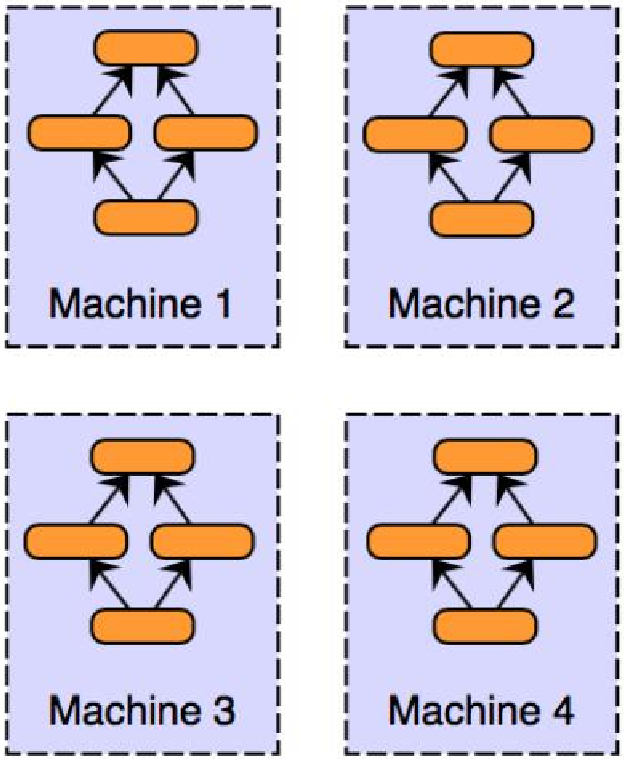
    large scale of training data, e.g., hundreds of billions of data
    data may not be fit into single GPU memory
    slow to train neural networks using single GPU
    can be solved by data parallelism
    e.g., each device uses different parts of batch data

  - Model Parallelism

    large scale models, e.g., billions of parameters
    model may not be fit into single GPU memory
    can be solved by model parallelism
    different devices run different part of the computation graph

Parallel and Distributed training

- data-parallel training
  - copy model on each machine
  - split the batch data across the multiple machines
  - synchronously/asynchronously update the model parameters



Parallel and Distributed training

- Synchronous Data Parallelism
  - all devices train their model using different parts of a (large) batch data.
  - only after all devices have successfully computed and sent their gradients to parameter devices, model parameters are updated.
  - updated model is then sent to all devices along with splitted next batch data.

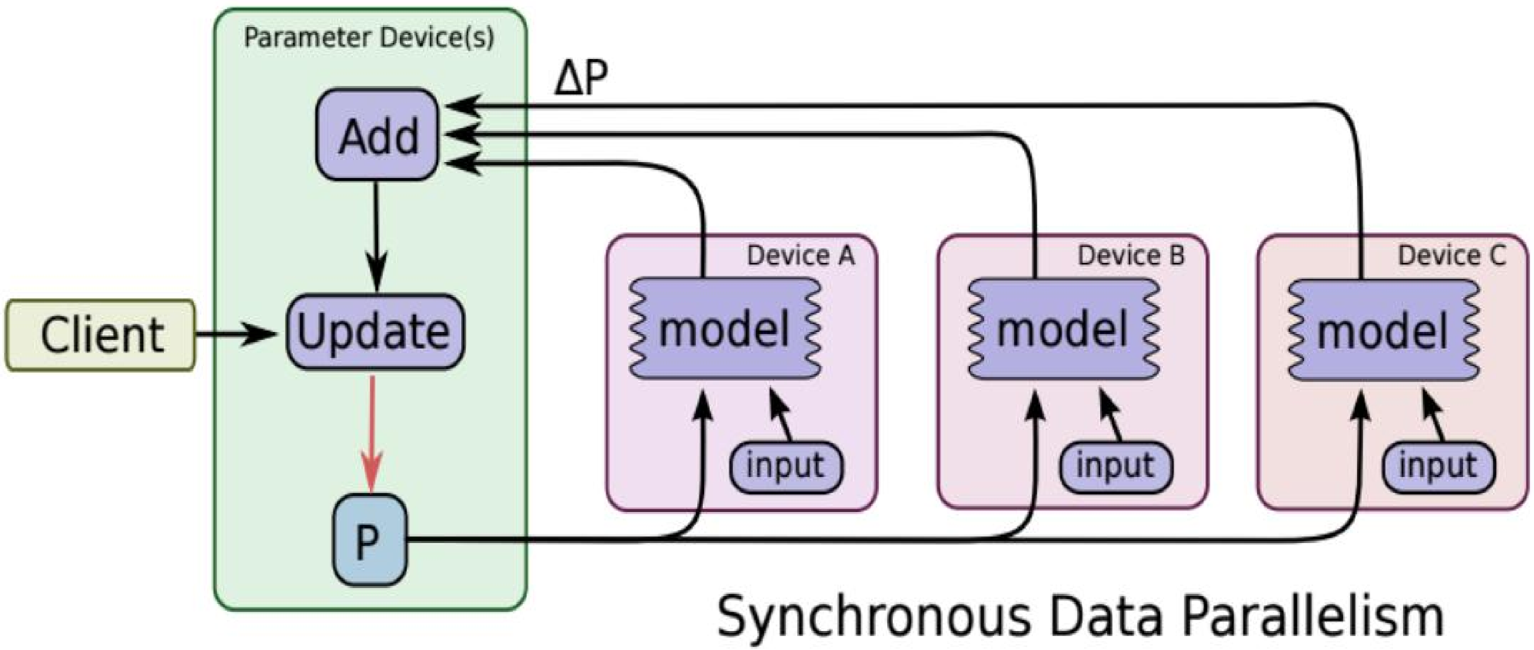
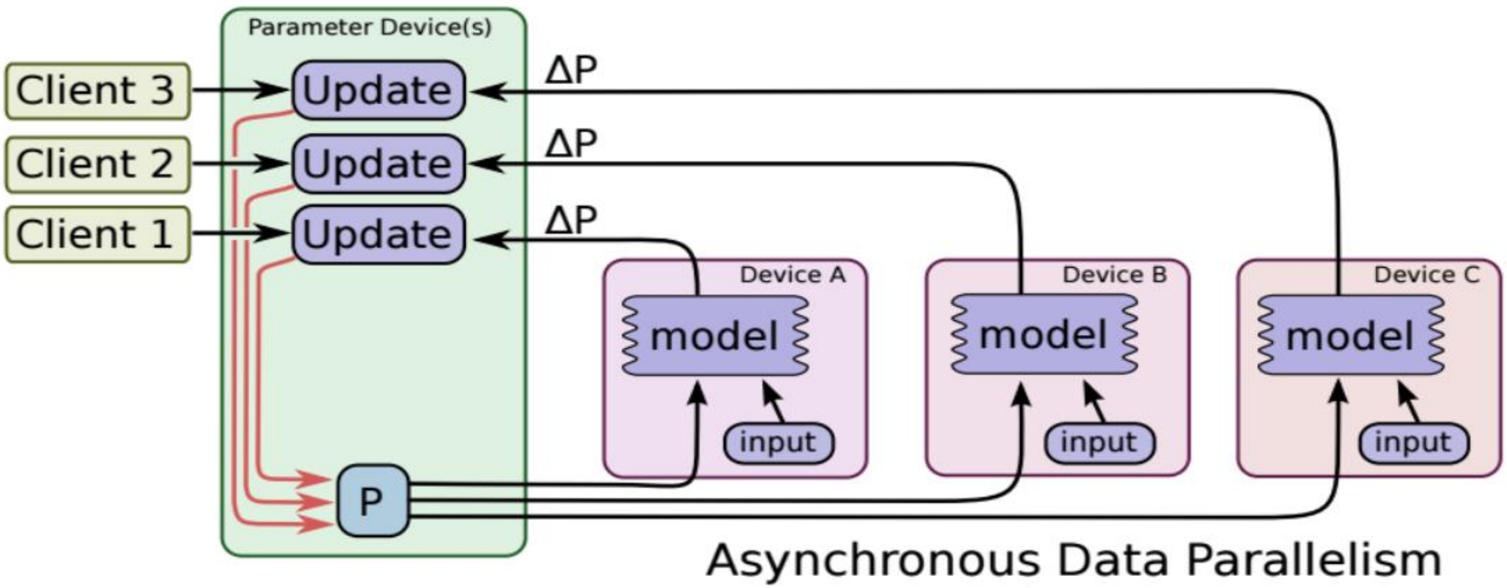


image from Large-Scale Machine Learning on Heterogeneous Distributed Systems. Tensorflow whitepaper

Parallel and Distributed training

- Asynchronous Data Parallelism
  - device does not need to wait for updates from other devices (run independently)
  - communicate through one or more central servers known as "parameter" servers
  - each device calculate and send gradients to parameter server when they are finished
  - parameter servers summarize the gradients, update model parameters and send new parameters to the device that locally calculate the gradients



Scaling Distributed Machine Learning with the Parameter Server. Mu Li, etc.

image from Large-Scale Machine Learning on Heterogeneous Distributed Systems. Tensorflow whitepaper

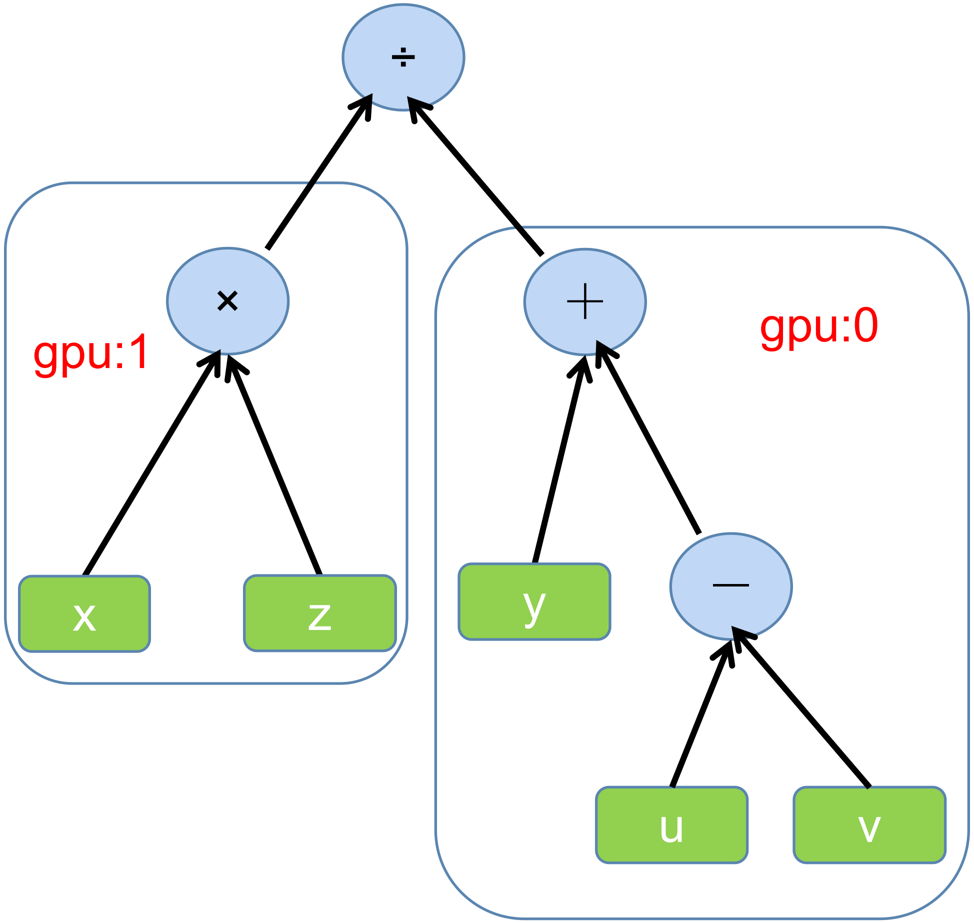
Parallel and Distributed training

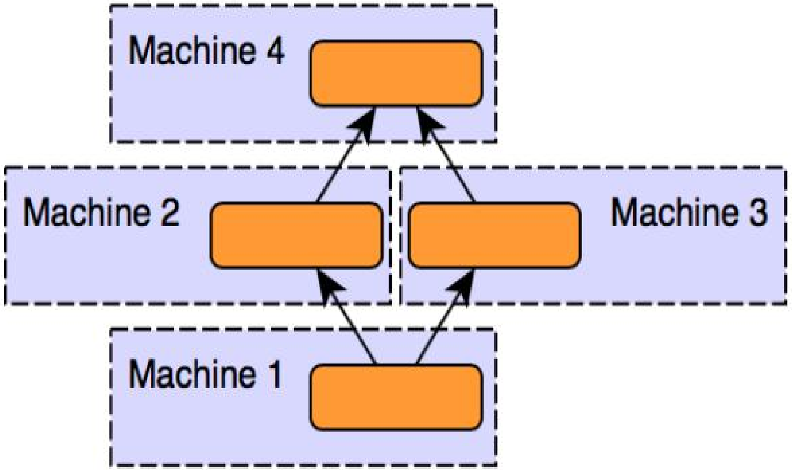
- Example, Multiple GPU training (Data Parallelism)

```
a = tf.random_uniform([1000, 100])
b = tf.random_uniform([1000, 100])
split_a = tf.split(a, 2)
split_b = tf.split(b, 2)

split_c = []
for i in range(2):
    with tf.device(tf.DeviceSpec(device_type="GPU", device_index=i)):
        split_c.append(split_a[i] + split_b[i])

c = tf.concat(split_c, axis=0)
```

Parallel and Distributed training

- Model Parallelism
  - partition the computation graphs into several subgraphs especially when our graph is too large to be stored on a single GPU
  - run them parallelly across multiple CPUs, GPU, etc





Parallel and Distributed training

- Example, Multiple GPU training (Model Parallelism)

```
with tf.device('/gpu:0'):
    w1=tf.Variable(...)
    b1=tf.Variable(...)
    fc1 = tf.add(tf.matmul(fc1,w1),b1)
    fc1=tf.nn.relu(fc1)
```

→ place the first layer

gpu:0

```
with tf.device('/gpu:1'):
    w2=tf.Variable(...)
    b2=tf.Variable(...)
    fc2 = tf.add(tf.matmul(fc1,w2),b2)
    fc2=tf.nn.relu(fc2)
```

→ place the second layer

gpu:1

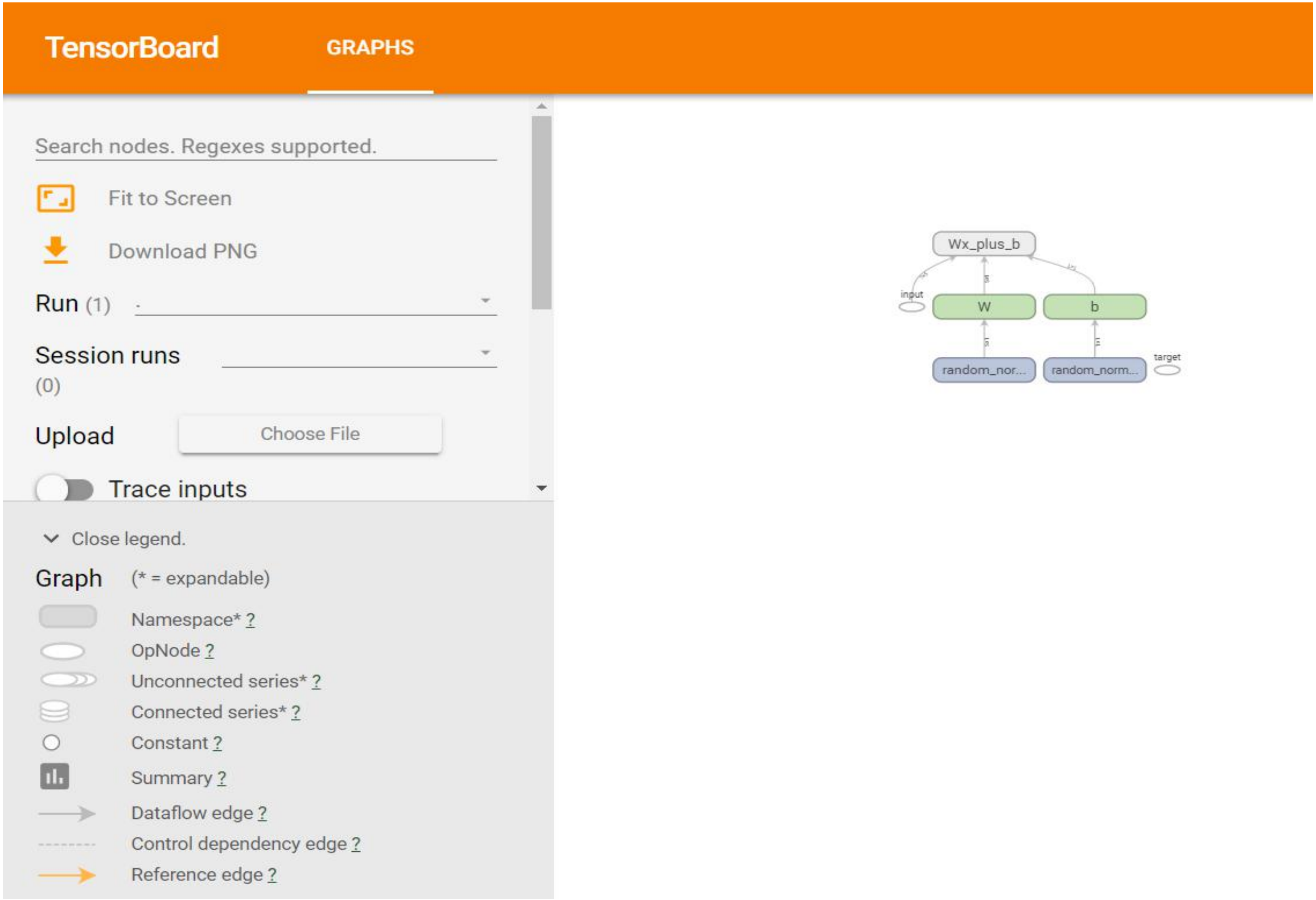
TensorBoard

- Make it easier to understand, debug, and optimize TensorFlow programs.

- Visualize your TensorFlow graph, plot quantitative metrics about the execution of your graph

- Step 1, create the TensorFlow graph to collect summary data from summary operations to annotate nodes.
- Step 2, tf.summary.merge_all to combine them into a single op that generates all the summary data.
- Step 3, run the merged summary op.
- Step 4, pass the summary protobuf to a tf.summary.FileWriter to write this summary data to disk.
- step 5, launch tensorboard, type command
  tensorboard --logdir=path/to/log-directory
  Step 3 and step 4 are optional.

TensorBoard

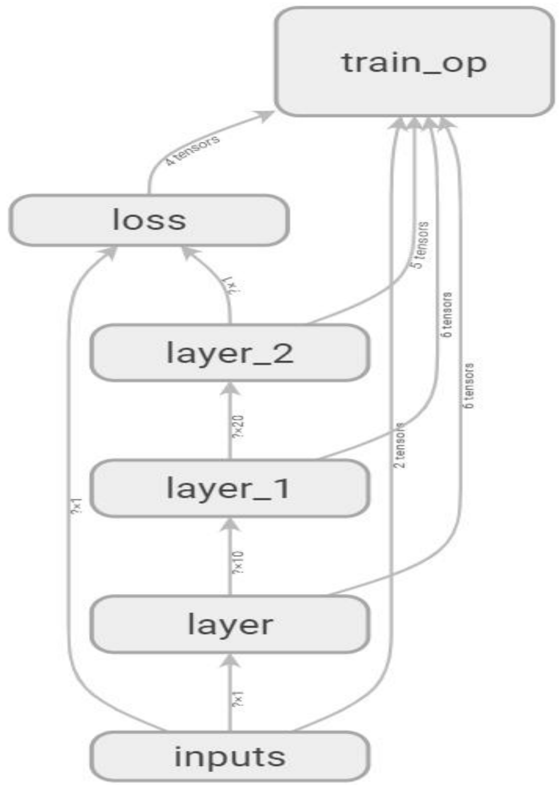


TensorBoard

- Visualization of computation graph

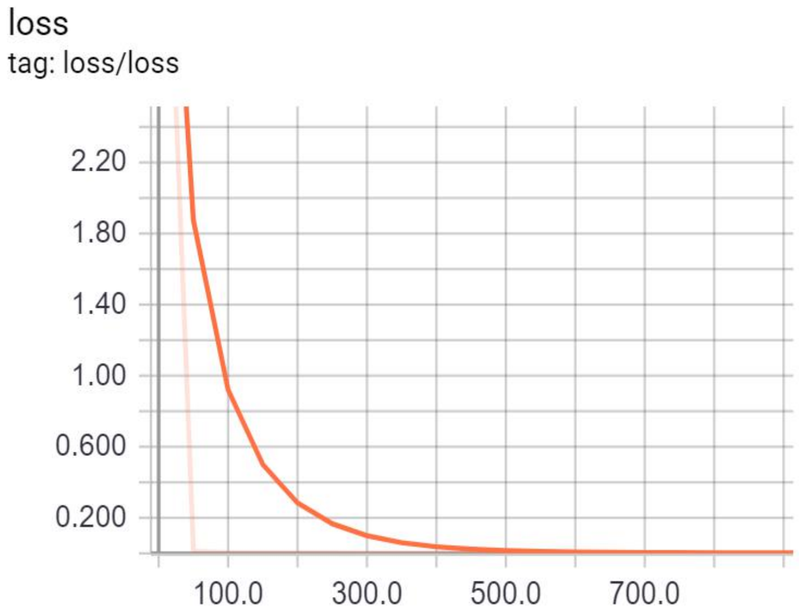
  sess = tf.Session()

  writer = tf.summary.FileWriter("logs/", sess.graph)



- Visualization of loss

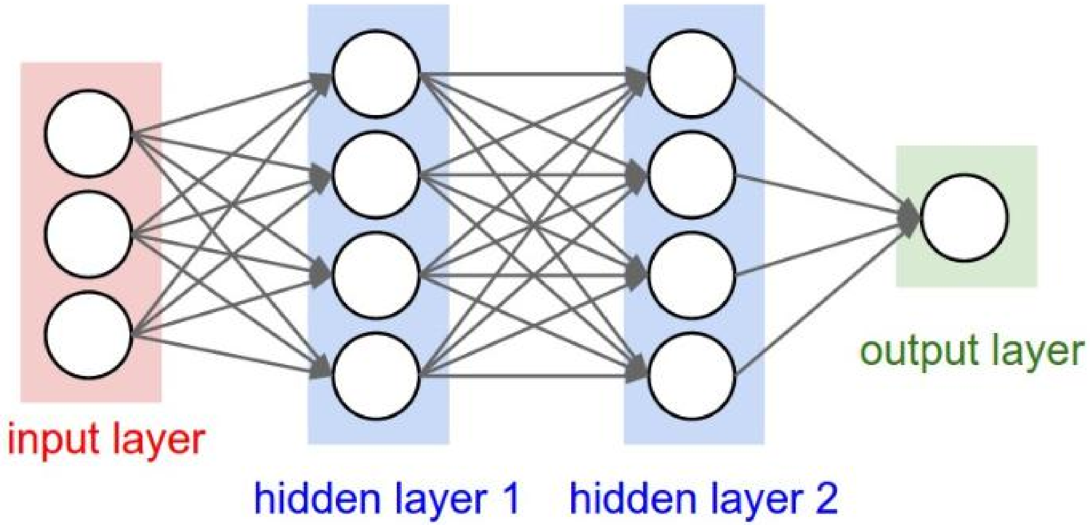
  tf.summary.scalar('loss', loss)



Example

- MNIST dataset, 10 classes (0-9) hand-written digit classification task
- train a model using the 60,000 training images
  test classification accuracy on the 10,000 test images
- fully connected networks to learn to classify 10 classes images

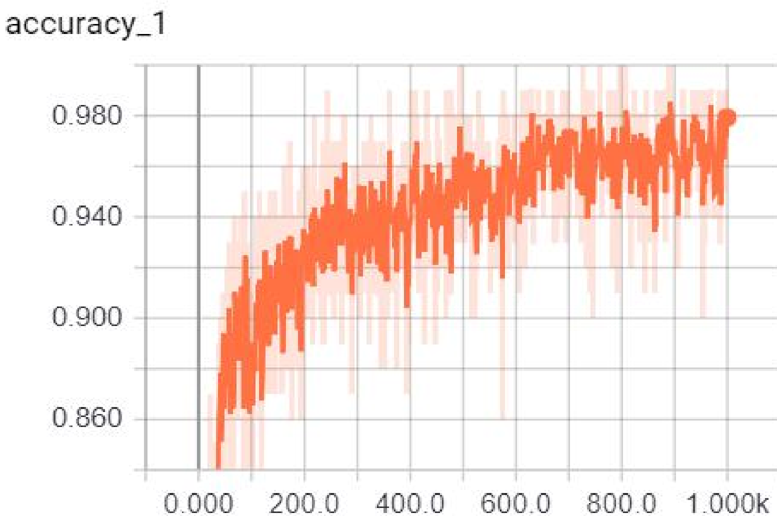
Code to be shown in class



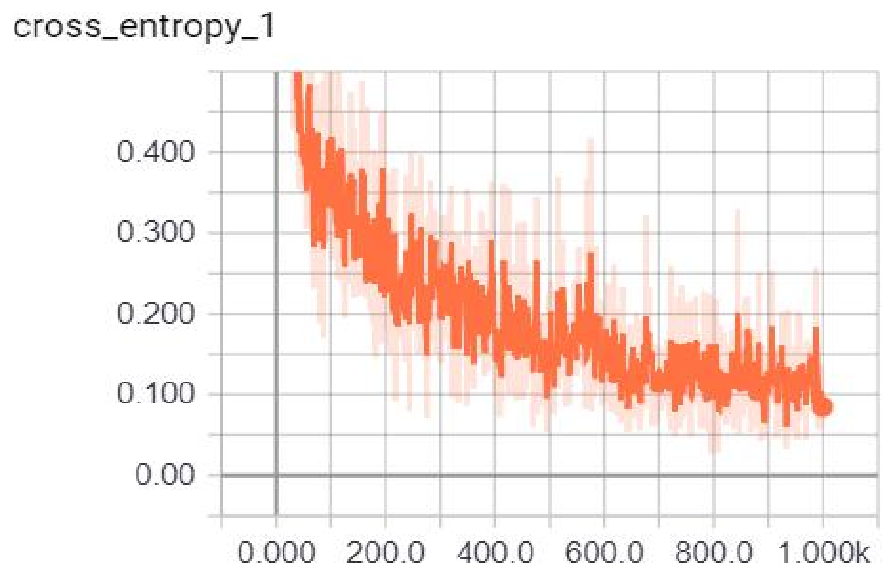
Example

- tensorboard visualization of training accuracy and loss function

training accuracy

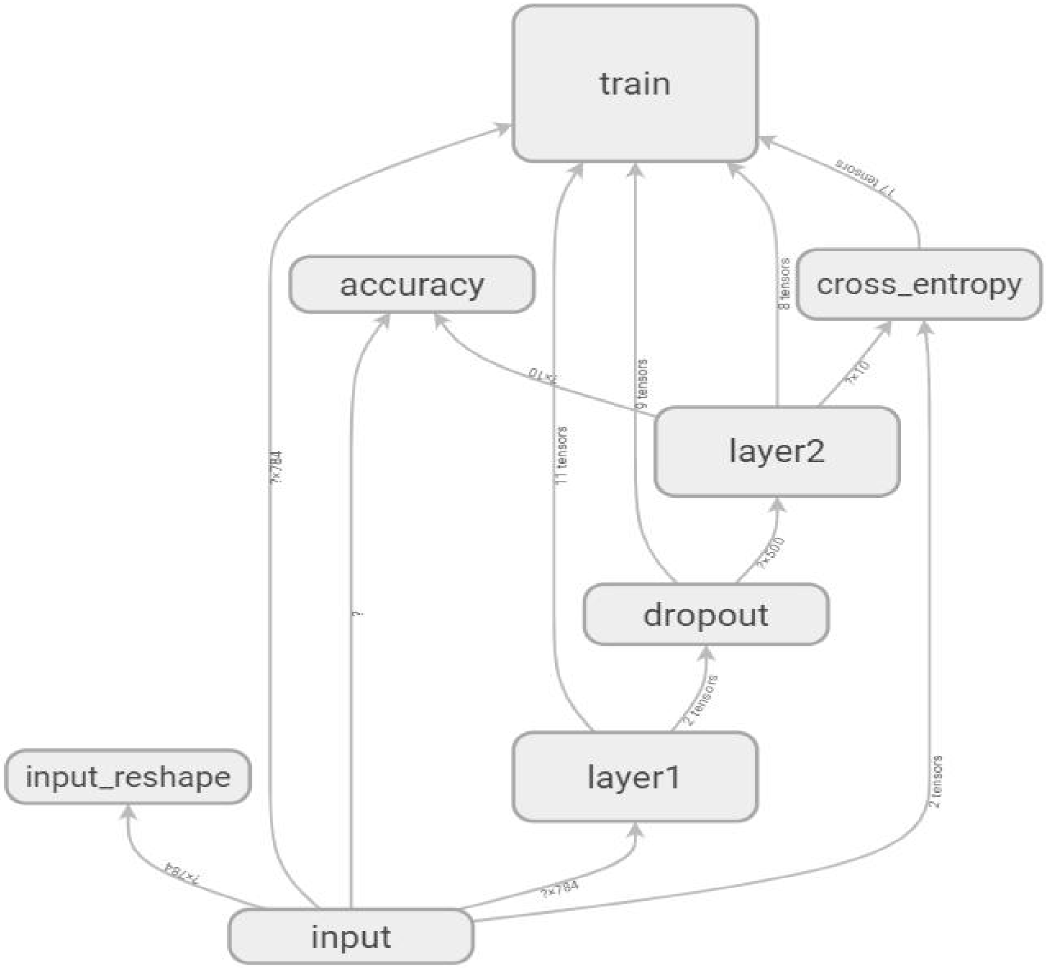


training loss



Example

- visualization of computation graph by tensorboard



Example

- visualization of layer1 weights and activation value by tensorboard