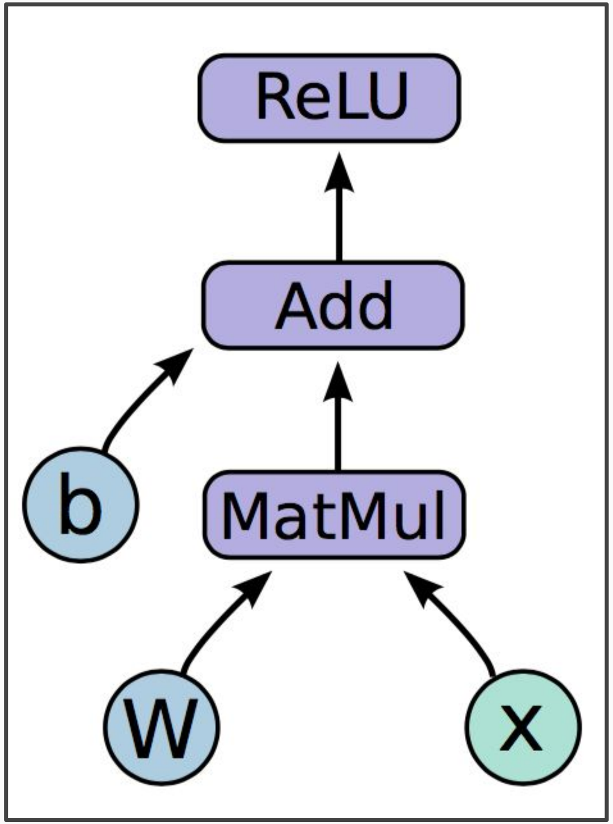
Programming model

Big idea: express a numeric computation as a **graph**.

- Graph nodes are **operations** which have any number of inputs and outputs
- Graph edges are **tensors** which flow between nodes

Programming model

$$h = ReLU(Wx + b)$$



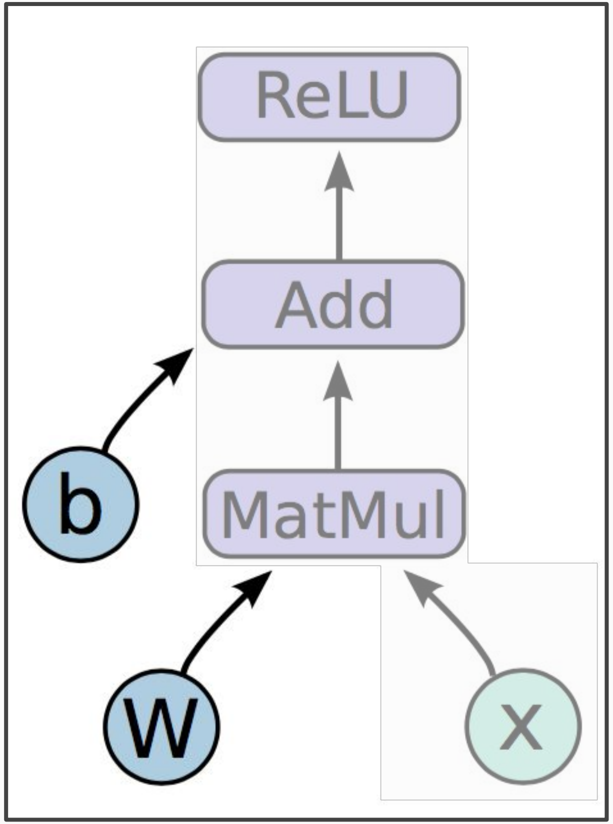
Programming model

$$h = ReLU(Wx + b)$$

Variables are stateful nodes which output their current value.

State is retained across multiple executions of a graph

(mostly parameters)

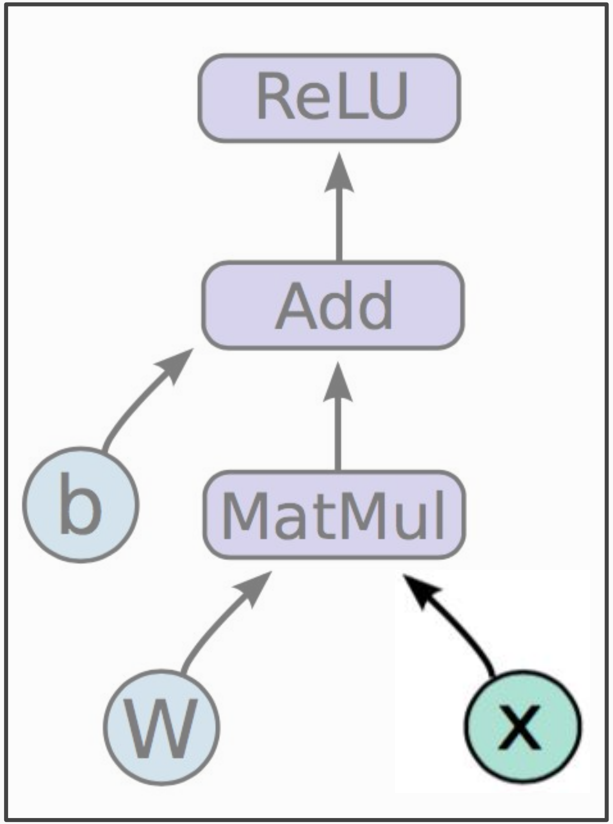


Programming model

$$h = ReLU(Wx + b)$$

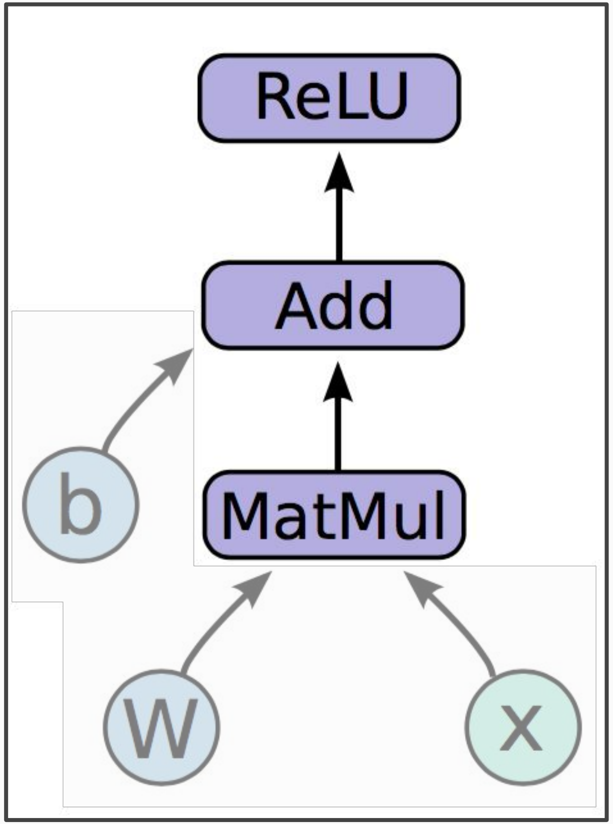
Placeholders are nodes whose value is fed in at execution time

(inputs, labels, ...)



Programming model

$$h = ReLU(Wx + b)$$



Mathematical operations:

MatMul: Multiply two matrix values.
Add: Add elementwise (with broadcasting).
ReLU: Activate with elementwise rectified linear function.

In code,

1. Create weights, including initialization

   W ~ *Uniform*(-1, 1); b = **0**

2. Create input placeholder x

   m * 784 input matrix

3. Build flow graph

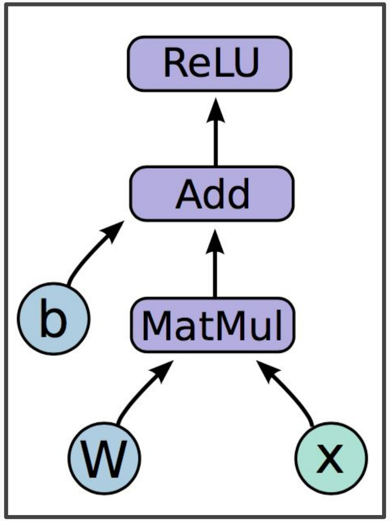
```
import tensorflow as tf

b = tf.Variable(tf.zeros((100,)))
W = tf.Variable(tf.random_uniform((784, 100), -1, 1))

x = tf.placeholder(tf.float32, (100, 784))

h = tf.nn.relu(tf.matmul(x, W) + b)
```

$$h = ReLU(Wx + b)$$



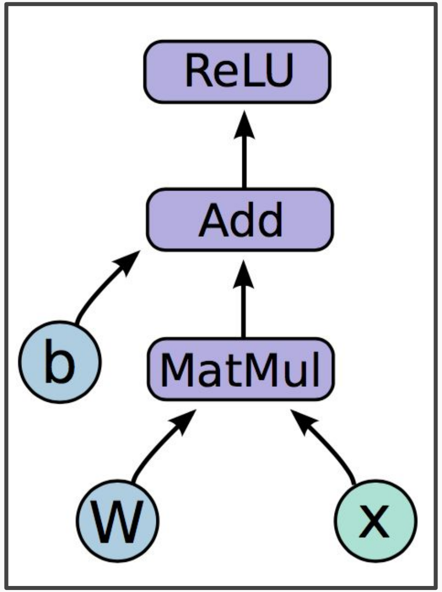
But where is the graph?

New nodes are automatically built into the underlying graph!

tf.get_default_graph().get_operations():

zeros/shape
zeros/Const
zeros
Variable
Variable/Assign
Variable/read
random_uniform/shape
random_uniform/min
random_uniform/max
random_uniform/RandomUniform
random_uniform/sub
random_uniform/mul
random_uniform
Variable_1
Variable_1/Assign
Variable_1/read
Placeholder
MatMul
add
Relu == h

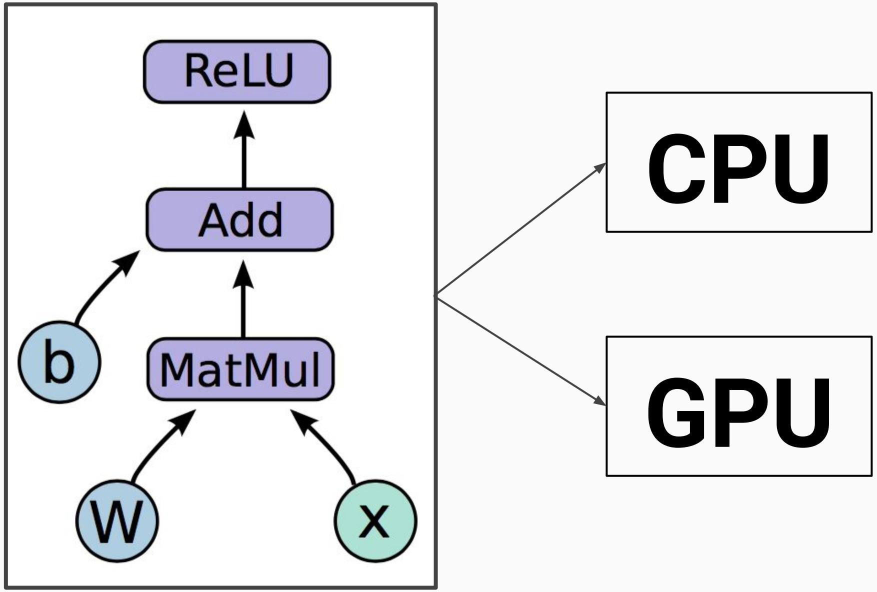
h refers to an op!



How do we run it?

So far we have defined a **graph**.

We can deploy this graph with a **session**: a binding to a particular execution context (e.g. CPU, GPU)



Getting output

```
sess.run(fetches, feeds)
```

Fetches: List of graph nodes. Return the outputs of these nodes.

Feeds: Dictionary mapping from graph nodes to concrete values. Specifies the value of each graph node given in the dictionary.

```
import numpy as np
import tensorflow as tf

b = tf.Variable(tf.zeros((100,)))
W = tf.Variable(tf.random_uniform((784, 100),
                -1, 1))

x = tf.placeholder(tf.float32, (100, 784))
h = tf.nn.relu(tf.matmul(x, W) + b)

sess = tf.Session()
sess.run(tf.initialize_all_variables())
sess.run(h, {x: np.random.random(100, 784)})
```

So what have we covered so far?

We first built a **graph** using **variables** and **placeholders**

We then deployed the graph onto a **session**, which is the **execution environment**

Next we will see how to **train** the **model**

How do we define the loss?

Use **placeholder** for **labels**

Build loss node using labels and **prediction**

```
prediction = tf.nn.softmax(...)  #Output of neural network
label = tf.placeholder(tf.float32, [100, 10])

cross_entropy = -tf.reduce_sum(label * tf.log(prediction), axis=1)
```

How do we compute Gradients?

```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

- `tf.train.GradientDescentOptimizer` is an **Optimizer** object
- `tf.train.GradientDescentOptimizer(lr).minimize(cross_entropy)` adds optimization **operation** to computation graph

TensorFlow graph **nodes** have **attached gradient operations**

Gradient with respect to **parameters** computed with **backpropagation**

...automatically

Creating the train_step op

```
prediction = tf.nn.softmax(...)
label = tf.placeholder(tf.float32, [None, 10])

cross_entropy = tf.reduce_mean(-tf.reduce_sum(label * tf.log(prediction),
reduction_indices=[1]))

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

Training the Model

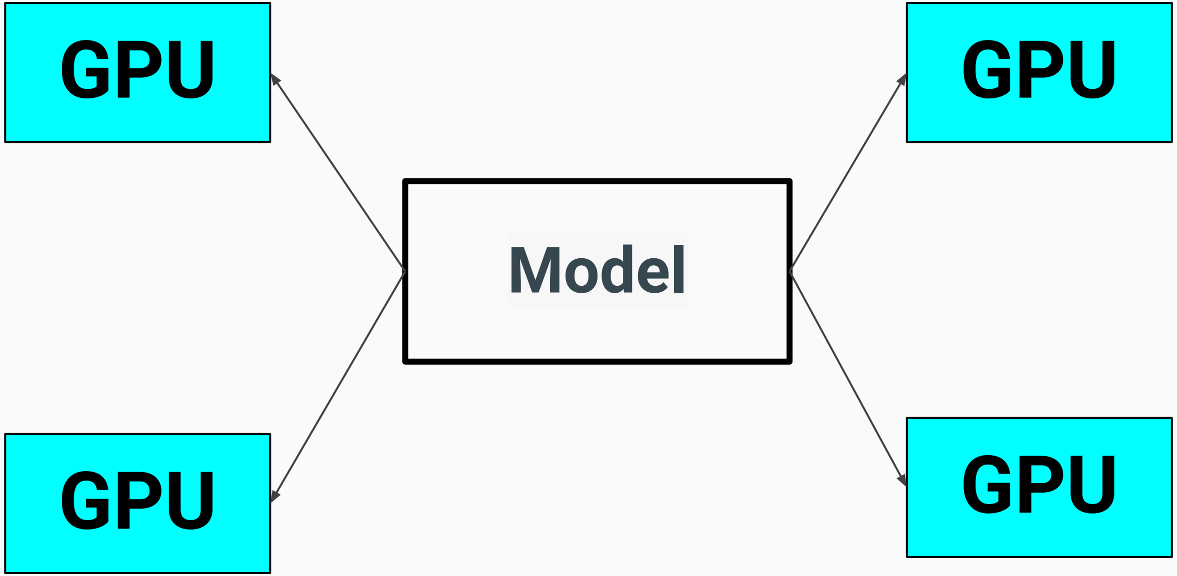
`sess.run(train_step, feeds)`

1. Create Session
2. Build training schedule
3. Run `train_step`

```
sess = tf.Session()
sess.run(tf.initialize_all_variables())

for i in range(1000):
    batch_x, batch_label = data.next_batch()
    sess.run(train_step, feed_dict={x: batch_x,
                                    label: batch_label}
```

Variable sharing



Variable sharing: naive way

```
variables_dict = {
     "weights": tf.Variable(tf.random_normal([782, 100]),
                                name="weights")
     "biases": tf.Variable(tf.zeros([100]), name="biases")
     }
```

Not good for encapsulation!

What’s in a Name?

`tf.variable_scope()` provides simple name-spacing to avoid clashes

`tf.get_variable()` creates/accesses variables from within a variable scope

```
with tf.variable_scope("foo"):
    v = tf.get_variable("v", shape=[1])  # v.name == "foo/v:0"

with tf.variable_scope("foo", reuse=True):
    v1 = tf.get_variable("v")         # Shared variable found!

with tf.variable_scope("foo", reuse=False):
    v1 = tf.get_variable("v")         # CRASH foo/v:0 already exists!
```

In Summary:

1. Build a graph
    a. Feedforward / Prediction
    b. Optimization (gradients and train_step operation)

2. Initialize a session

3. Train with `session.run(train_step, feed_dict)`