Introduction to TensorFlow

Fundamentals and Practical Tips

The Big Idea

TensorFlow is a framework composed of:

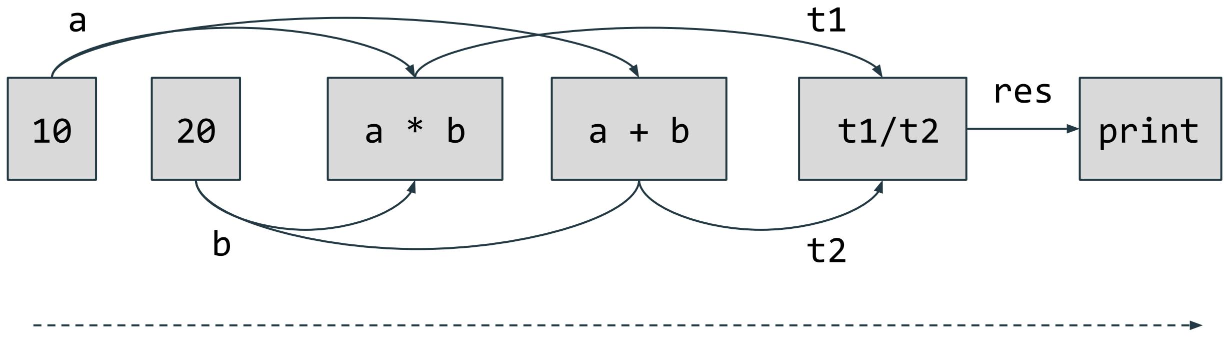
- A **library** for defining **computational graphs**,
- A **runtime** for executing such graphs on a variety of different hardware.

Computational graphs are an abstract way of describing computations as directed graph:

- The **edges** correspond to multidimensional arrays (**Tensors**).
- The **nodes** create or manipulate these Tensors according to specific rules (**Ops**).

Programs as Computational Graphs

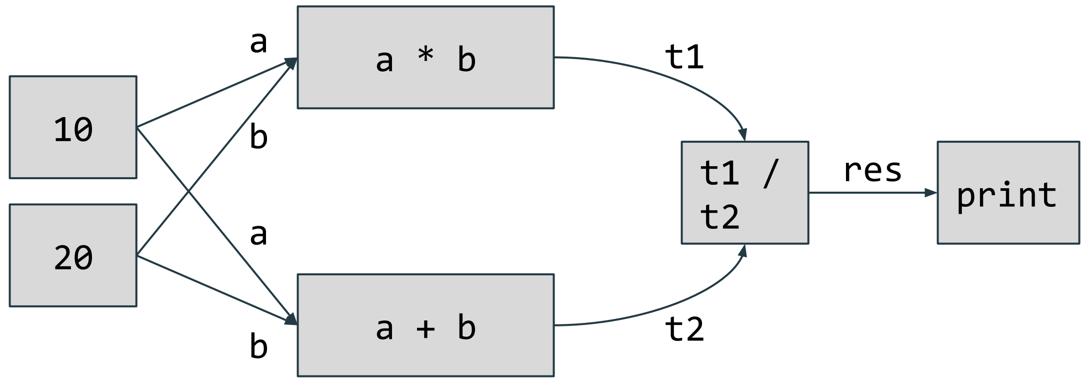
```
a = 10
b = 20
tmp1 = a * b
tmp2 = a + b
res = tmp1 / tmp2
print(res)
```



the program specifies the order of execution

Dependency Driven Scheduling

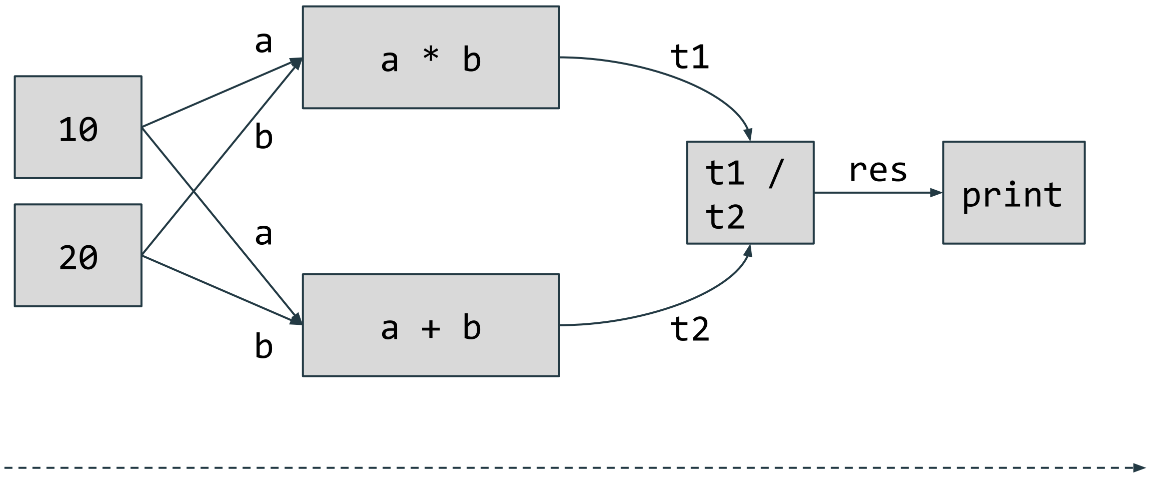
```
a = 10
b = 20
tmp1 = a * b
tmp2 = a + b
res = tmp1 / tmp2
print(res)
```



the data dependencies specify the order of execution,
operations that do not depend on each other can schedule in parallel

TensorFlow Computational Graphs

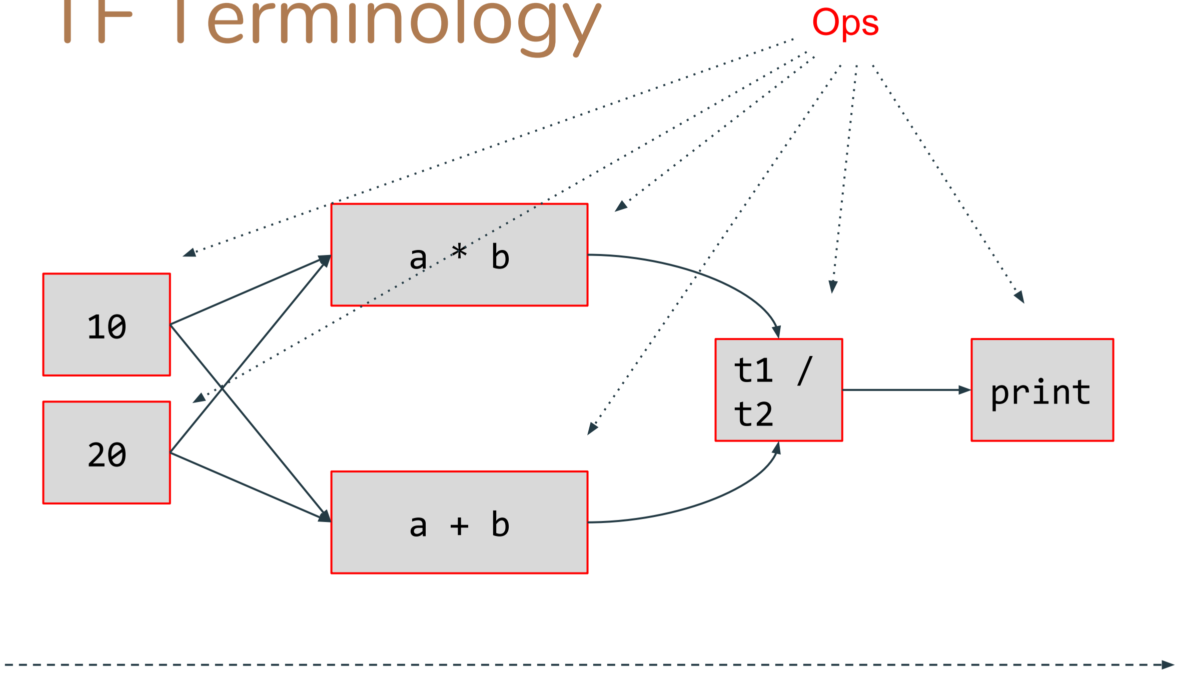
```
a=tf.constant(10)
b=tf.constant(20)

t1=tf.multiply(a,b)
t2=tf.add(a,b)
res=tf.divide(t1,t2)

tf.Print(res)
```



the data dependencies specify the order of execution,
operations that do not depend on each other can schedule in parallel

TF Terminology

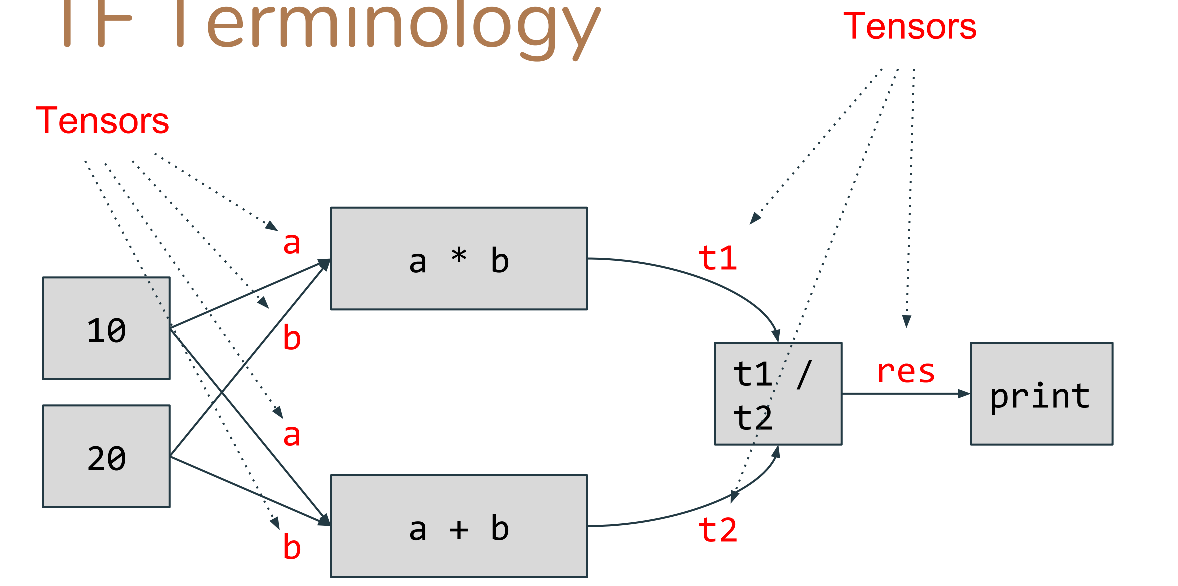
```
a=tf.constant(10)
b=tf.constant(20)

t1=tf.multiply(a,b)
t2=tf.add(a,b)
res=tf.divide(t1,t2)

tf.Print(res)
```



the data dependencies specify the order of execution,
operations that do not depend on each other can schedule in parallel

TF Terminology

```
a=tf.constant(10)
b=tf.constant(20)

t1=tf.multiply(a,b)
t2=tf.add(a,b)
res=tf.divide(t1,t2)

tf.Print(res)
```



the data dependencies specify the order of execution,
operations that do not depend on each other can schedule in parallel

TF Computational Graphs

TensorFlow graphs supports very general forms of computation, including **stateful**, **conditional**, **iterative** and **asynchronous** computation.
All these different forms of computation are supported thanks to a variety of in-built Ops:

- **Variables ops**, to read and update writable values that persist across executions.
- **Conditional ops**, to make part of the computation conditional on other parts.
- **Loop ops**, to allow to efficiently specify computational graphs with cycles.
- **Control ops**, to enforce an order in the execution of pairs of ops.
- **Queue ops**, to describe asynchronous computation.

Definition vs Execution

Any TensorFlow program is composed of two separate parts:

- **definition** of the computational graph.
- **execution** of the computational graph (or some subset of it).

When we define the graph we make use of a set of TensorFlow **library** functions to specify some computations as a `tf.Graph`, at execution time we use the TensorFlow **runtime** to execute (a subset of) those computations through a `tf.Session`.

tf.Graph

- Describes computations through Ops and Tensors.
- A tensor is a description of a multidimensional array.
- Tensors may have a shape and a data type, but don’t have actual values.

```
a = tf.zeros((int(1e12), int(1e12)))  # perfectly defined quantity
a = np.zeros((int(1e12), int(1e12)))  # out of memory error
```

- Tensor shapes can usually be derived from the graph (**shape inference**).

```
a = tf.zeros((10, 10))  # a.shape → (10, 10)
b = tf.concat([a, a], axis=0)  # b.shape → (20, 10)
```

tf.Session

```python
# define the graph
a = tf.constant(1.0)
b = tf.constant(1.0)
c = tf.constant(4.0)
d = tf.div(tf.add(a, b), c)

# execute the graph
with tf.Session() as session:
    print(session.run(d))  # 0.5
```

Session (`tf.Session`):

- Carries out the actual computations.

- `session.run(tensor)` executes the graph and returns the value of `tensor`.

- `session.run([t1, t2, ...])` computes the value of all `tensors` in the list.

Execution Model

When `session.run()` is called, TensorFlow identifies and executes the **smallest set of nodes** that **needs** to be evaluated in order to compute the requested tensors.

- TensorFlow is aware of all functional dependencies, and additional dependencies can be added through special control ops.
- TensorFlow can schedule the execution of ops which do not depend on each other in parallel across available **cores**, **devices** and **machines**.

Programming languages

Tensorflow comes with **bindings** for multiple programming languages:

- `Python` is the main supported language (and the one we will use in this course),
- C++ implements most of the back end and can also be interfaced directly,
- Go, `Java` have experimental support by the TF team,
- `Haskell` and `Rust` are supported by the open-source community.

Whichever language you choose to interact with TensorFlow, most of the computation actually happens in the highly optimized TensorFlow C++ backend.

Variables - Definition

Variables enable **learning**, by preserving state across execution of the graph:

- All trainable parameters of machine learning models are `tf.Variables`.

A variable is defined by its **name**, **type**, **shape**, and **initialization** procedure.
e.g. a 2x2 matrix of floats, initialized from a normal distribution with σ = 0.5 is defined by:

```
v = tf.get_variable(
    "name", dtype=tf.float32, shape=[2, 2],
    initializer=tf.random_normal_initializer(stddev=0.5))
```

Variables - Usage

Variables can be **read** and used as any other Tensor in the computational graph.

```
y = tf.matmul(v, tf.constant([[1, 2], [3, 4]]))
```

Variables can be **assigned** new values, and will maintain them across graph executions until the next update. Assignments are also ops, exposed by the same python object:

```
increment_op = v.assign(v + 1)  # syntax 1
increment_op = tf.assign(v, v + 1)  # syntax 2
```

Variables - Initialization

- Defining a variable adds the corresponding ops to the computational graph.
- But a variable instance **exists** and holds values in the context of a specific **session**.
- Any variable we define must be explicitly **initialized** before its first use in a session.
- You can initialize all variables in the graph at once as follows:

```
create graph()  # including variables
init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    ...
```

Variables - Initialization

- Instead of **initializing** the variables manually at the beginning of the session you can use `tf.train.MonitoredSession`, that takes care of this for you.
- Note that `MonitoredSession` also **finalizes** the graph: any attempt to add ops to the graph after the instantiation of the monitored session will raise an error.

```
create_graph()  # including variables
with tf.train.MonitoredSession() as session:
    ...  # can execute graph directly without manually initializing vars.
```

Working with Data

- Very often we need a way to **feed data into the graph**.
- If the dataset is very small we can, in principle, embed it in the graph definition itself, loading the data in a numpy array and defining a constant tensor with its values.

```
d = tf.constant(<some numpy array>)
```

This is **NOT recommended**: you’ll be out of memory if the dataset is of even moderate size; also, the whole dataset will be serialized whenever the graph is serialized.

Placeholders and Feeds

Use **placeholders** and **feed dictionaries** to inject data in the graph at execution time. Placeholders are used in the graph definition as tensors, but at each execution of the graph they'll take the value specified in the **feed dictionary** provided to `session.run`.

```
a = tf.placeholder(tf.float32, [])
b = tf.constant(1.0)
c = a + b
with tf.Session() as session:
  print(session.run(c, feed_dict={a: 3.0}))  # result: 4
  print(session.run(c, feed_dict={a: 4.0}))  # result: 5
```

Placeholders Pros & Cons

Using placeholders puts the onus on you to manage the data:

- TF expects a dictionary of numpy arrays for each call to `session.run()`.
- The rest is up to you: load, pre-process, batch, queue, etc., all in Python code.
- Very flexible, but can be somewhat labour intensive.
- TF offers additional built-in functionalities for working with data (e.g. tf.data).

Simple Linear Regression - Problem

In **univariate regression** the aim is to learn a function $f(x): \mathbf{R} \rightarrow \mathbf{R}$ from data

In **linear regression** we assume that such function is linear $y = wx + b$

$$\widehat{w}, \widehat{b} = argmin_{w,b} \left\{ \frac{\sum(y_i - wx_i + b)^2}{N} \right\}$$

Typically, the relation will not be exactly linear, but for any dataset of pairs (x_i, y_i) we can find values w and b that minimize the **mean squared error** (**MSE**) on the dataset.

Simple Linear Regression - Solution

We can find the solution to the linear regression problem directly in **closed form**:

$$\widehat{w} = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y}))}{\sum_i (x_i - \overline{x})^2} \qquad \widehat{b} = \overline{y} - w\overline{x} \quad / \quad \overline{x} = \frac{\sum_i x_i}{N} \qquad \overline{y} = \frac{\sum_i y_i}{N}$$

Or we can use **gradient descent** to iteratively improve our estimates of w and b:

$$w_{t+1} = w_t - \alpha \frac{\partial MSE}{\partial w_t} \qquad b_{t+1} = b_t - \alpha \frac{\partial MSE}{\partial b_t}$$

Gradient descent generalizes to more complex problems where closed form solution **do not exist** or are too **computationally expensive** to solve.

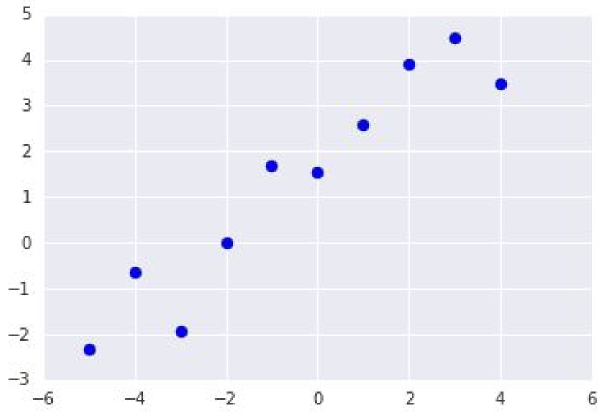
Linear Regression in TensorFlow - Data

Let’s generate a small dataset for a 1D Linear Regression problem.

We generate the data according to the following rule: $y = w x + b + \eta$

Let η be a random noise with gaussian distribution (zero mean and unit variance):

```
num_samples, w, b = 20, 0.5, 2.
xs = numpy.asarray(range(num_samples))
ys = numpy.asarray([
    x * w + b + numpy.random.normal()
    for x in range(num_samples)])
matplotlib.pyplot.plot(xs, ys)
```



Linear Regression in TensorFlow - Model

Next we can create a simple linear regression model.

The linear model is parametrized by two variables only: the *slope* (w) and the *offset* (b).

```
class Linear(object):

  def __init__(self):
    self.w = tf.get_variable(
        "w",dtype=tf.float32,shape=[],initializer=tf.zeros_initializer())
    self.b = tf.get_variable(
        "b",dtype=tf.float32,shape=[],initializer=tf.zeros_initializer())

  def __call__(self, x):
    return self.w * x + self.b
```

Linear Regression in TensorFlow - Solver

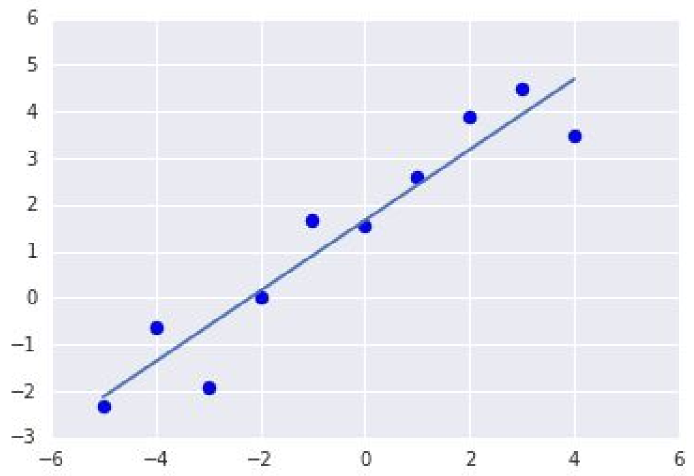
We can define the solver for the linear regression problem as part of the graph itself.

```
xtf = tf.placeholder(tf.float32, [num_samples], "xs")
ytf = tf.placeholder(tf.float32, [num_samples], "ys")
model = Linear()
model_output = model(xtf)

cov = tf.reduce_sum((xtf-tf.reduce_mean(xtf))*(ytf-tf.reduce_mean(ytf)))
var = tf.reduce_sum(tf.square(xtf-tf.reduce_mean(xtf)))
w_hat = cov / var
b_hat = tf.reduce_mean(ytf)-w_hat*tf.reduce_mean(xtf)

solve_w = model.w.assign(w_hat)
solve_b = model.b.assign(tf.reduce_mean(ytf)-w_hat*tf.reduce_mean(xtf))
```

Linear Regression in TensorFlow - Execution

We can now fit a model to any specific input dataset, and then use it for prediction.

```
with tf.train.MonitoredSession() as sess:
  sess.run(
      [solve_w, solve_b],
      feed_dict={xtf: xs, ytf: ys})
  preds = sess.run(
      model_output,
      feed_dict={xtf: xs, ytf: ys})

matplotlib.pyplot.plot(xs, ys)
matplotlib.pyplot.plot(xs, preds)
```



Automatic Differentiation

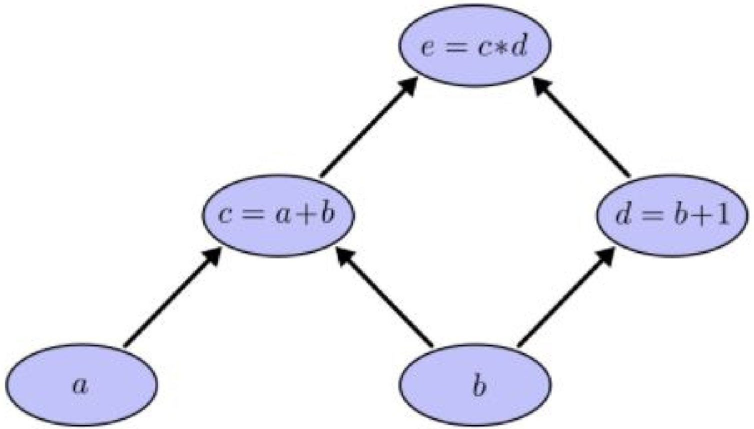
Describing our computations as graphs allows to easily compute gradients.

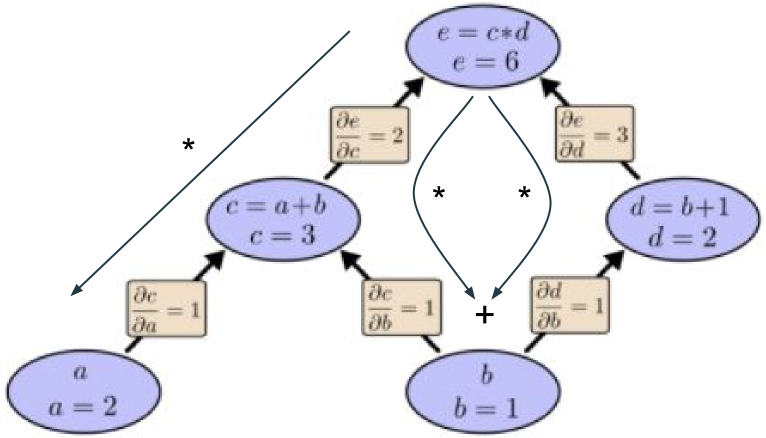
If we know the gradients of the output of each op with respect to its direct inputs, the **chain rule** provides us with gradients for any Tensor with respect to any other.

The process is known as **reverse mode auto-differentiation**, and enables the computation of the gradient of a node in the graph with respect to all others in a single sweep.

Note that autodiff is **different** from numerical methods such as finite differences in that it provides you with **exact gradients** not approximations.

Automatic Differentiation





$e = c*d$
$e = 6$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$c = a+b$
$c = 3$
$d = b+1$
$d = 2$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
a
$a = 2$
b
$b = 1$
*
*
*
+

Autodiff simply starts from the output node and iterates backwards **multiplying** the gradients of individual ops along paths, and **summing** them when paths join.

Gradients

TensorFlow's `tf.gradients` function constructs a subgraph implementing reverse mode auto differentiation. It does so in a fully transparent way:

```
grads = tf.gradients(my_tensor, var_list)
```

`grads` is a list of Tensors of length `len(var_list)` holding, in order, the gradients of `my_tensor` with respect to each of the variables in `var_list`. If `my_tensor` does not depend on some of the variables, the corresponding entry in `grads` is None.

Note: if `my_tensor` is multidimensional, it sums the gradients over its dimensions.

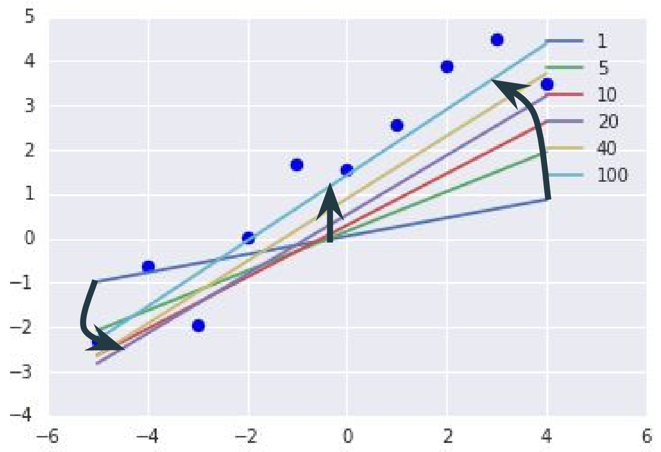
Linear Regression - Gradient Descent

We can also define the **gradient descent solver** as part of the graph.
In this case we’ll need to **execute** the update op **repeatedly**, until convergence.

```
loss = tf.losses.mean_squared_error(ytf, model_output)
grads = tf.gradients(loss, [model.w, model.b])

update_w = tf.assign(model.w, model.w - 0.001 * grads[0])
update_b = tf.assign(model.b, model.b - 0.001 * grads[1])
update = tf.group(update_w, update_b)
```

Linear Regression - Gradient Descent

```
matplotlib.pyplot.plot(xs, ys)
feed_dict = {xs_tf: xs, ys_tf: ys}

with tf.train.MonitoredSession() as sess:
  for i in xrange(500):
    sess.run(update, feed_dict=feed_dict)
    if i in [1, 5, 25, 125, 499]:
      preds = sess.run(
            model_output, feed_dict=feed_dict)
       matplotlib.pyplot.plot(
            xs, preds, label=str(i))

matplotlib.pyplot.legend()
```



TensorFlow Optimizers

Instead of manually computing the gradients and applying the updates, we can also Delegate to **higher level components**, such as a `tf.train.Optimizer`.

```
loss = tf.losses.mean_squared_error(ytf, model_output)
update = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(loss)
```

`Optimizers` may implement more sophisticated update rule than plain SGD.
e.g. `AdamOptimizer` or `RMSPropOptimizer` are good choices to get good results with more limited amount of tuning (compared to `GradientDescentOptimizer`).

Control Logic

TensorFlow’s computational graphs support **conditional** and **iterative** computation, as well as the specification of ad-hoc **control dependencies** between Ops.
This is especially important to write high performance RL agents in TensorFlow.

The main Ops used in order to specify the control logic in the graph are:

- `tf.control_dependencies`: add dependencies between nodes.
- `tf.cond`: true/false conditional.
- `tf.case`: multi-case conditional.
- `tf.while_loop`: while loop.

Understanding how to use these Ops can be complicated initially, but very important.

Control Dependencies - I

What is the output of this code?

```
x = tf.get_variable("x", shape=(), initializer=tf.zeros_initializer())
assign_x = tf.assign(x, 10.0)
z = x + 1.0

with tf.train.MonitoredSession() as session:
  print(session.run(z))
```

The answer is `1.0`: the op `assign_x` is not a dependency of x or z, and never evaluated.

Control Dependencies - II

What is the output of this code?

```
x = tf.get_variable("x", shape=(), initializer=tf.zeros_initializer())
assign_x = tf.assign(x, 10.0)
z = x + 1.0

with tf.train.MonitoredSession() as session:
  print(session.run([assign_x, z]))
```

The output could be (`10.0, 1.0`) or (`10.0, 11.0`). The ops `assign_x` and `z` are racing!

Control Dependencies - III

What is the output of this code?

```
x = tf.get_variable("x", shape=(), initializer=tf.zeros_initializer())
assign_x = tf.assign(x, 10.0)
with tf.control_dependencies([assign_x]):
  z = x + 1.0

with tf.train.MonitoredSession() as session:
  print(session.run(z))
```

The answer is now `11.0`.

Conditional Evaluation - I

The `tf.cond` op enables to make the execution of some portion of the computational graph conditional upon the result of the execution of some other portion of the graph.

```
v1 = tf.get_variable("v1", shape=(), initializer=tf.zeros_initializer())
v2 = tf.get_variable("v2", shape=(), initializer=tf.zeros_initializer())
switch = tf.placeholder(tf.bool)
cond = tf.cond(switch,
               lambda: tf.assign(v1, 1.0),
               lambda: tf.assign(v2, 2.0))
with tf.train.MonitoredSession() as session:
  session.run(cond, feed_dict={switch: False})
  print(session.run([v1, v2]))                       # Output: (0.0, 2.0)
```

Conditional Evaluation - II

The graph that must be executed conditionally has to be created within the `tf.cond`.
Tensors used but created outside will be dependencies of `tf.cond` and always executed.

```python
v1 = tf.get_variable("v1", shape=(), initializer=tf.zeros_initializer())
v2 = tf.get_variable("v2", shape=(), initializer=tf.zeros_initializer())
switch = tf.placeholder(tf.bool)
assign_v1 = tf.assign(v1, 1.0)
assign_v2 = tf.assign(v2, 1.0)
cond = tf.cond(switch,
               lambda: assign_v1,
               lambda: assign_v2)
with tf.train.MonitoredSession() as session:
  session.run(cond, feed_dict={switch: False})
  print(session.run([v1, v2]))                          # Output: (1.0, 2.0)
```

While Loops

You can have cycles in the graph, resulting in **iterative computation** of variable length.

```
k = tf.constant(2)
matrix = tf.ones([2, 2])
condition = lambda i, _: i < k  # i is a tensor here, and < is thus tf.less
body = lambda i, m: (i+1, tf.matmul(m, matrix))
final_i, power = tf.while_loop(
    cond=condition,
    body=body,
    loop_vars=(0, tf.diag([1., 1.])))
```

`power` will be the k-th power of `matrix`.

Dynamic Unrolling

In Deep Learning and Reinforcement Learning it is common to have to apply the same transformation (**core**) recursively to some **state**, in order to accumulate some sequence of **outputs** along the way. This is important for:

- Time series prediction
- Sequence to sequence models
- Take decisions in partially observable domains

Tensorflow provides ad-hoc utility functions to do this in graph.

Dynamic Unrolling - I

The **fibonacci** sequence is initialized with `(0, 1)` and then constructed by **recursively** summing the latest two elements. At each step we only need to remember the last two generated elements (we shall therefore refer to these as our **state**). This is sufficient to compute the next element (which we'll refer as **output**) and to update the state.

```
0, 1 || 1, 2, 3, 5, 8, 13, 21, …
```

In order to generate the sequence in-graph we only need to specify the **core**:

output, next_state = fibonacci_core(old_state)

where: old_state = $(\text{elem}_{i-1}, \text{elem}_i)$

output = $\text{elem}_{i-1} + \text{elem}_i$

next_state = $(\text{elem}_i, \text{output})$

Dynamic Unrolling - II

```
class fibonacci_core(object):

  def __init__(self):
    self.output_size = 1
    self.state_size = tf.TensorShape([1,1])

  def __call__(self, input, state):
    return state[0]+state[1], (state[1], state[0]+state[1])

  def zero_state(self, batch_size, dtype):
    return (tf.zeros((batch_size, 1), dtype=dtype),
            tf.ones((batch_size, 1), dtype=dtype))

  def initial_state(self, batch_size, dtype):
    return zero_state(self, batch_size, dtype)
```

A core must specify how to generate the next output and state in the `_call_`.

And expose `zero_state()`, `initial_state()`, `output_size`, `state_size`.

Dynamic Unrolling - III

```
inputs = tf.reshape(tf.range(10), [10, 1, 1])

fib_seq = tf.nn.dynamic_rnn(
    cell=fibonacci_core(),
    inputs=inputs,
    dtype=tf.float32,
    time_major=True)

with tf.train.MonitoredSession() as sess:
  print(sess.run(fib_seq))
```

Output: (1, 2, 3, 5, 8, 13, 21, 34, 55, 89)

Advanced Features

Specifying programs in terms of computational graphs and then delegating the execution to TensorFlow’s runtime library has a powerful advantage:

- Trasparent multithreaded execution on **multi-core CPUs**
- Transparent execution on specialized hardware such as **GPUs**
- Transparent distributed execution across multiple machines in a **cluster**

All these advanced features just require to **annotate** the graph appropriately, and then the TF backend takes care of all the actual work.

GPU Example - I

If the machine that you are running your program on has a GPU that supports CUDA, You can specify what should execute on CPU and which on GPU as follows:

```
with tf.device("/cpu:0"):
 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')

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

with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
 print(sess.run(c))
```

GPU Example - II

Note that for ops that you do not explicitly assign to a specific device, tensorflow will decide the placement automatically. And if a GPU is available TF will assign most ops to it.

```
with tf.device("/cpu:0"):
 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)  # if a GPU is available will be assigned to /gpu:0

with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
 print(sess.run(c))
```

Neural Network Libraries

There are many higher level different neural network libraries that have been built on top of TensorFlow in order to simplify constructing large neural network models:

- `Sonnet`
- `Keras`

In this course you’ll also be required to build neural networks with raw TensorFlow; understanding how to do this is essential to be able to use confidently the above libraries.

Sonnet

Sonnet is a TensorFlow library with two main aims:

- Provide implementations of Deep Learning modules and architectures
- Enable easy sharing of weights between modules

Each sonnet module is a class and follows a **configure-then-connect** paradigm:

1. First you **instantiate** the class,
2. Then you **call** the resulting object to create the corresponding graph
   a. You provide the input tensors to the module as arguments to the call.
   b. If you call the same module multiple times with different input tensors the various subgraphs that are generated all share variables.

Sonnet

Sonnet provides many standard deep learning modules out of the box,
It’s also easy to write your own sonnet modules and get the sharing capabilities for free:

```
class Linear(sonnet.AbstractModule):

  def __init__(self):
     super(Linear, self).__init__(name=name)

  def _build(self, x):
    w = tf.get_variable(
         "w",dtype=tf.float32,shape=[],initializer=tf.zeros_initializer())
    b = tf.get_variable(
         "b",dtype=tf.float32,shape=[],initializer=tf.zeros_initializer())
    return w * x + b
```

All variables created in `_build` with `tf.get_variable` are shared across different calls to `build`

Alternative Frameworks

There are many other Deep Learning frameworks available.

- Torch / PyTorch: with interfaces in Lua and Python respectively,
- Chainer: also in Python, with similar features to Torch,
- Caffe: C++ and Python,
- Theano: similar in some ways to TF (but not supported anymore).

The End

Questions?