API for creating, optimizing and evaluating mathematical expressions, particularly those involving multi-dimensional arrays (a.k.a. tensors).

Designed for Machine-Learning applications by Google.

Particularly good with multi-dimensional arrays, the basis of most neural-net representations.

Automatic derivation of gradients (derivatives), a central aspect of many core neural-net computations.

Seamless exploitation of GPU’s for major performance improvements.

NOT a drag-and-drop construction kit for neural nets, but a programming paradigm (that takes a little getting used to).

The source of info: http://tensorflow.org

Runs in Python (most stable), C++, Java, and Go

Code for this tutorial: tflowtools.py, tutor1.py, tutor2.py, tutor3.py
Function graph = scaffolding for computation; variables (but no values) and operations.

Inside a session run, the graph receives and produces actual data (dark shapes), which the user can access.
TensorFlow variables have the same Python class type, though they may house everything from scalars to multi-dimensional arrays.

TensorFlow variables have (optional) names that serve no important purpose in TensorFlow but are useful for the user when displaying the function graph. These names do NOT need to match the names of the Python variables whose values are the TensorFlow variables.

When declared, variables must be given an initial value, denoting the value that the variable will receive when run in a TensorFlow session.

```python
>>> import tensorflow as tf
>>> import numpy as np
>>> import tensorflow tools as TFT

>>> x = tf.Variable(37, name='x')
>>> y = tf.Variable(np.array([[1,2],[3,4]]), name='2x2-array')

>>> type(x)
<class 'tensorflow.python.ops.variables.Variable'>

>>> type(y)
<class 'tensorflow.python.ops.variables.Variable'>
```
def tfex1(a,b):
    x = tf.constant(a)  # Scalar constant variables
    y = tf.constant(b)
    z = x * y  # z becomes a multiplication operator
    sess = tf.Session()  # open a new session
    result = sess.run(z)  # Run the operator z
    sess.close()  # Close session to release memory
    return result

Operators are also tensors, since they produce a tensor variable.
As the product of tf variables, z is automatically declared as a tf multiplication operator.
TensorFlow compiles the entire expression in a function graph that connects inputs (x,y) to outputs (z).

>>> from tutor1 import *
>>> tfex1(5,7)
35
def tfex2(a, b):
    x = tf.Variable(a, name='x')
    y = tf.Variable(b, name='y')
    z = x * y  # Create operator z
    sess = tf.Session()
    sess.run(x.initializer)  # Each var has an initializer
    sess.run(y.initializer)
    result = sess.run(z)
    sess.close()
    return result

- x and y are now variables (not constants) with initial values (a,b).
- All variables at the leaves of the function graph must be initialized internally (by calling the variable’s initializer). Alternatively, some leaves can be placeholders, which are fed values from outside via the init_dict argument to Session.run(), as described later.
def tfex2b(a, b):
    x = tf.Variable(a, name='x')
    y = tf.Variable(b, name='y')
    z = x*y  # Create operator z
    sess = tf.Session()
    sess.run(tf.global_variables_initializer())
    result = sess.run(z)
    sess.close()
    return result

- The call `tf.global_variables_initializer()` initializes all variables in the function graph.
- This is the standard practice for initializing all variables at the start of a session.
def quickrun(operators):
    sess = tf.Session()
    sess.run(tf.global_variables_initializer())
    result = sess.run(operators)
    sess.close()
    TFT.showvars(result)

def showvars(vals, names=None, msg=" "):
    print("\n"+msg, end="\n")
    for i, v in enumerate(vals):
        if names:
            print("\n"+names[i] + "=", end="\n")
        print(v, end="\n\n")

The result of sess.run is a list of output values, one per operator.

showvars (from file tflowtools.py) just pretty-prints the values of variables, which, in this case, all represent the outputs of operators.
Matrix Operations for Neural Networks

Neural network simulations often boil down to many matrix and vector (i.e., tensor) operations.

Keith L. Downing
Tensor Flow: A Brief Introduction and Tutorial
The bias $\approx$ activation threshold; *it can be learned just like a weight.*

Keith L. Downing    Tensor Flow: A Brief Introduction and Tutorial
def tfex3():
    x = tf.Variable(np.random.uniform(0, 1, size=(3, 5)), name='x')
    y = tf.Variable(np.random.uniform(10, 20, size=(5, 1)), name='y')
    z = tf.matmul(x, y)
    return quickrun(z)

- x is a 3x5 matrix, while y is a 5x1 column vector.
- z is a matrix multiplication, producing a 3x1 column vector.
- Below, tfex3b sends 2 operations to quickrun (z1 and z2) while also sending w, so the run will also return the value of w.

def tfex3b():
    w = tf.Variable(np.random.uniform(0, 1, size=(3, 5)), name='w')
    x = tf.Variable(np.random.uniform(10, 20, size=(5, 1)), name='x')
    y = tf.Variable(np.random.uniform(100, 110, size=(3, 1)), name='y')
    z1 = tf.matmul(w, x)
    z2 = z1 + y
    return quickrun([w, z1, z2])
```python
def tfex4():
    x = tf.Variable(np.random.uniform(1, 2, size=(5, 1)), name='x')
    x2 = tf.Variable(x.initialized_value())  # Only doing x2 = x would NOT do the copying.
    x2 = x2.assign(x + np.random.uniform(100, 200, size=(5, 1)))
    return quickrun([x, x2])
```

- x2's initial value is declared to be a COPY of x's initial value. Be sure to use x.initialized_value() in these situations, not just x.
- x2.assign is an operation that will give x2 a new value DURING the session run.

```python
>>> tfex4()
[[ 1.78439086], [ 1.60791754], [ 1.83206691],
 [ 1.62277005], [ 1.03705036]]

[[ 195.89901858], [ 184.40446403], [ 131.03902211],
 [ 152.79362131], [ 155.98459017]]
```
# Partial contents of file tflowtools.py

```python
import os

def viewprep(session, dir='probeview', flush=120, queue=10):
    return tf.summary.FileWriter(dir, session.graph
                                 ,flush_secs=flush, max_queue=queue)

def fireup_tensorboard(logdir):
    os.system('tensorboard --logdir=' + logdir)
```

- FileWriter stores information about the function graph and the values of any user-specified variables.
- It is the basis for a Tensorboard, which can be activated from a command line or from Python via the os module.
- After the call to fireup_tensorboard, open a browser (Chrome works best) and go to localhost:6006.
- The tensorboard reads data from the directory "logdir" specified in the call. This is the same directory written to by the FileWriter.
def quickrun2(operators, grabbed_vars = None, dir='probeview'):
    sess = tf.Session()
    probe_stream = TFT.viewprep(sess, dir=dir)
    sess.run(tf.global_variables_initializer())
    results = sess.run([operators, grabbed_vars])
    sess.close()
    TFT.show_results(results[1], grabbed_vars, dir)
    return results

def show_results(grabbed_vals, grabbed_vars=None, dir='probeview'):
    showvars(grabbed_vals, names = [x.name for x in grabbed_vars],
             msg="The Grabbed Variables:")

In quickrun2, we can specify variables to "grab" during the run, as well as the
directory in which to write summary information (for later viewing in the
tensorboard).

show_results is defined in tflowtools.py. It provides variable names (in addition to values) to showvars.
```python
def tfex5():
    w = tf.Variable(np.random.uniform(0,1,size=(3,5)),name='w')
    x = tf.Variable(np.random.uniform(10,20,size=(5,1)),name='x')
    y = tf.Variable(np.random.uniform(100,110,size=(3,1)),name='y')
    z = tf.matmul(w, x) + y
    return quickrun2([z],[w,x,y])
```

- Multiply a column vector by a matrix and then add another column vector to the product.
- This is a common operation for neural networks.
Choose the "Graphs" option from the top menu of TensorBoard.

This is highly interactive, enabling zooming and other exploratory operations.
def tfex5b ():
    w = tf.Variable(np.random.uniform(0,1,size=(3,5)), name='w')
    x = tf.Variable(np.random.uniform(10,20,size=(5,1)), name='x')
    y = tf.Variable(np.random.uniform(100,110,size=(3,1)), name='y')
    z = tf.add(tf.matmul(w,x), y, name="mult-add")
    return quickrun2([z],[w,x,y])

By using tf.add, we can name the operation for ease of viewing in TensorBoard.

The name has many restrictions and cannot be just any string. It cannot contain blanks, for example.
def tfex6():
    x = tf.placeholder(tf.float64, shape=(5,1),name='x')
    y = tf.placeholder(tf.float64, shape=(3,1),name='y')
    w = tf.Variable(np.random.uniform(0, 1, size=(3, 5)), name='w')
    z = tf.matmul(w, x) + y
    feeder = {x: np.random.uniform(10,20,size=(5,1)),
              y: np.random.uniform(100,110,size=(3,1))}
    return quickrun3([z],[w,x,y],feed_dict=feeder)

- Placeholders are variables whose values will be filled in via the feed_dict during Session.run.
- If given a shape of None, they can accept any tensor.
- In this example, only w gets initialized by its initializer; x and y get values from the feeder.
def quickrun3(operators, grabbed_vars = None,
               dir='probeview', feed_dict=None):
    sess = tf.Session()
    probe_stream = TFT.viewprep(sess, dir=dir)
    sess.run(tf.global_variables_initializer())
    results = sess.run([operators, grabbed_vars], feed_dict=feed_dict)
    sess.close()
    TFT.show_results(results[1], grabbed_vars, dir)
    return results

- The feed_dict must include values for all placeholders.
- These values are fed into the placeholders at the start of the run.
- Initialization = providing values at startup, normally the beginning of a session.
- Feeding = done during each run, of which there may be many per session.
Opening and Closing a Session

```python
def gen_initialized_session(dir='probeview'):
    sess = tf.Session()
    sess.probe_stream = viewprep(sess, dir=dir)
    sess.viewdir = dir
    sess.run(tf.global_variables_initializer())
    return sess

def close_session(sess, view=True):
    sess.close()
    if view: fireup_tensorboard(sess.viewdir)
```

- This attaches the probe stream and the view directory as new slots on the session object.
- Opening and closing are moved to separate functions so that the new quickrun (see below) can accept pre-opened sessions and simply do additional runs in them.
- Closing of a session can also invoke the TensorBoard, merely as a convenience for this tutorial.
- Both are defined in tflowtools.py
def quickrun4(operators, grabbed_vars = None, dir='probeview', session=None, feed_dict=None, step=1, show_interval=1):
    sess = session if session
    else TFT.gen_initialized_session(dir=dir)

    results = sess.run([[operators, grabbed_vars], feed_dict=feed_dict)
    if show_interval and (step % show_interval) == 0:
        TFT.show_results(results[1], grabbed_vars, dir)
    return results[0], results[1], sess

Quickrun4 accepts previously-opened sessions.

It only displays the grabbed-variables periodically (via show_interval).
**Placeholders vs Variables in a Run Loop**

```python
def tfex7(n=5):
    w = tf.Variable(np.random.uniform(-1, 1, size=(5,5)), name='w')
    x = tf.Variable(np.zeros((1,5)), name='x')
    y = tf.placeholder(tf.float64, shape=(1,5), name='y')
    feeder = {y: np.random.uniform(-1,1, size=(1,5))}
    update_x = x.assign(tf.matmul(x,w) + y)
    _,_ , sess = quickrun4([update_x],[w,x,y], feed_dict=feeder)
    for step in range(n):
        quickrun4([update_x],[x], session=sess, feed_dict=feeder)
    TFT.close_session(sess)
```

- Use `quickrun4` to call `session.run` several times in a loop, each time updating var `x`, while var `y`, a placeholder, is set externally (via `feed_dict`) but to the same value on each call to `session.run`.
- This is a standard activity for a neural network, except that the placeholder (`y`) often gets a new value each time through the loop.
Now we have the tools to do gradient-descent optimization, the core of supervised learning in neural networks.

```python
def tfex8(size=5, steps=50, tvect=None, learning_rate=0.5, showint=10):
target = tvect if tvect else np.ones((1, size))
w = tf.Variable(np.random.uniform(-.1,.1, size=(size, size)), name='weights')
b = tf.Variable(np.zeros((1, size)), name='bias')
x = tf.placeholder(tf.float64, shape=(1, size), name='input')
y = tf.sigmoid(tf.matmul(x, w) + b, name='out-sigmoid')
error = tf.reduce_mean(tf.square(target - y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
training_operator = optimizer.minimize(error)
feeder = {x: np.random.uniform(-1,1, size=(1, size))}
sess = TFT.gen_initialized_session()
for step in range(steps):
    quickrun4([training_operator], [w, b, y], session=sess, feed_dict=feeder, step=step, show_interval=showint)
TFT.close_session(sess)
```
An Autoencoder Neural Network

Output ≈ Input, but it must pass through the hidden layer.

Hidden activation pattern = compression.
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as PLT
import tfflowtools as TFT

class autoencoder():
    def __init__(self, nh=3, lr=.1):
        self.cases = TFT.gen_all_one_hot_cases(2**nh)
        self.learning_rate = lr
        self.num_hiddens = nh
        self.build_neural_network(nh)

This autoencoder is in file tutor2.py

nh = # hidden nodes
lr = learning rate
One_hot cases are those with all zeros except for one 1.
def build_neural_network(self, nh):
    ios = 2**nh  # ios = input–and output–layer size
    self.w1 = tf.Variable(np.random.uniform(-1,.1,size=(ios,nh)),
                          name='Weights−1')
    self.w2 = tf.Variable(np.random.uniform(-1,.1,size=(nh,ios)),
                          name='Weights−2')
    self.b1 = tf.Variable(np.random.uniform(-1,.1,size=nh),
                          name='Bias−1')
    self.b2 = tf.Variable(np.random.uniform(-1,.1,size=ios),
                          name='Bias−2')

    self.input = tf.placeholder(tf.float64,shape=(1,ios),
                                 name='Input')
    self.target = tf.placeholder(tf.float64,shape=(1,ios),
                                 name='Target')
    self.hidden = tf.sigmoid(tf.matmul(self.input,self.w1) + self.b1,
                              name="Hiddens")
    self.output = tf.sigmoid(tf.matmul(self.hidden,self.w2) + self.b2,
                             name = "Outputs")

    ....
def build_neural_network(self, nb, nh):
    ......

    self.error = tf.reduce_mean(tf.square(self.target - self.output),
                                 name='MSE')

    self.predictor = self.output
    # Defining the training operator
    optimizer = tf.train.GradientDescentOptimizer(self.learning_rate)
    self.trainer = optimizer.minimize(self.error, name='Backprop')

    - The trainer is the main operation for training/learning
    - The predictor is the main operation for testing
    - Note that TensorFlow versions of both squaring(tf.square) and averaging (tf.reduce_mean) combine to form the error function, used by the optimizer, which derives gradients from the function graph.
    - The tf.contrib.losses module contains many built-in loss/error functions.
def do_training(self, epochs=100, test_interval=10):
    self.current_session = sess = TFT.gen_initialized_session()
    for i in range(epochs):
        error = 0
        grabvars = [self.error]
        for c in self.cases:
            feeder = {self.input: [c[0]], self.target: [c[1]]}
            grabvals, = self.run_one_step([self.trainer], grabvars, 
                                           session=sess, feed_dict=feeder)
            error += grabvals[0]
```python
def run_one_step(self, operators, grabbed_vars=None, dir='probeview', session=None, feed_dict=None, step=1, show_interval=1):
    sess = session if session else TFT.gen_initialized_session(dir=dir)
    results = sess.run([operators, grabbed_vars], feed_dict=feed_dict)
    if show_interval and (step % show_interval == 0):
        TFT.show_results(results[1], grabbed_vars, dir)
    return results[0], results[1], sess
```

This is the same as Quickrun, but now it is a method instead of a function.
For this example, the main purpose of testing is to find the hidden-node activation patterns for each input case. Ideally, they should be well separated in 2-d space.

def do_testing(self, sess, scatter=True):
    hidden_activations = []
    grabvars = [self.hidden]
    for c in self.cases:
        feeder = {self.input: [c[0]]}
        _, grabvals, _ = quickrun4([self.predictor], grabvars, session=sess, feed_dict=feeder)
        hidden_activations.append(grabvals[0][0])
General Artificial Neural Net (GANN)

**Goal:**
Easily build ANNs of different sizes for different data sets.

See file tutor3.py for the Gann class.

```python
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as PLT
import tensorflow_tools as TFT

class Gann:
    def __init__(self, dims, cman, lrate=.1, showint=None, mbs=10, vint=None):
        self.learning_rate = lrate
        self.caseman = cman  # Case Manager
        self.layer_sizes = dims  # Sizes of each layer of neurons
        self.show_interval = showint
        self.global_training_step = 0
        self.grabvars = []  # Variables monitored by gann code
        self.grabvar_figures = []  # A matplotlib figure per grabvar
        self.minibatch_size = mbs  # Common in Deep Learning
        self.validation_interval = vint; self.validation_history = []
        self.modules = []  # Combo of neurons + weights + biases
        self.build()
```

Keith L. Downing  Tensor Flow: A Brief Introduction and Tutorial
Probed variables are managed by a TensorFlow FileWriter and displayed on the TensorBoard.

Grabbed variables are managed by our own Gann code, so we have a lot of flexibility in visualizing them. Each grabvar has an associated matplotlib figure in which to display it.

A data case = (input vector, target vector)

A Gann Module = a layer of neurons + incoming weights and biases.
The GANN consists of an input placeholder + modules, each containing one layer of neurons along with their incoming weights and biases.

- \text{shape} = (\text{None, num}) \rightarrow 2D \text{ with the first dimension unspecified (and hence flexible). This enables the use of minibatches of cases, rather than only single cases.}

- Output of the last module = output of the whole network.
The optimizer knows to gather up all trainable variables in the function graph and compute derivatives of the error function with respect to each such variable, e.g. each weight and bias tensor of each module.
```python
def do_training(self, sess, cases, epochs=100, continued=False):
    if not(continued): self.error_history = []
    for i in range(epochs):
        error = 0; step = self.global_training_step + i
        gvars = [self.error] + self.grabvars; nmb = math.ceil(ncases / mbs)
        mbs = self.minibatch_size; ncases = len(cases)
        for cstart in range(0, ncases, mbs):  # Loop through minibatches
            cend = min(ncases, cstart + mbs)
            minibatch = cases[cstart:cend]
            inputs = [c[0] for c in minibatch]
            targets = [c[1] for c in minibatch]
            feeder = {self.input: inputs, self.target: targets}
            _, grabvals, = self.run_one_step([self.trainer], gvars,
                                             self.probes, session=sess, feed_dict=feeder,
                                             step=step, show_interval=self.show_interval)
            error += grabvals[0]
        self.error_history.append((step, error/nmb))
        self.consider_validation_testing(step, sess)
        self.global_training_step += epochs
        TFT.plot_training_history(self.error_history, self.validation_history)
```

- Cycle through minibatches of cases, supplying input and target values.
- Run the trainer op in current session to produce a new error value.
```python
def run_one_step(self, operators, grabbed_vars=None, probed_vars=None, dir='probeview', session=None, feed_dict=None, step=1, show_interval=1):
    sess = session if session else TFT.gen_initialized_session(dir=dir)
    if probed_vars is not None:
        results = sess.run([operators, grabbed_vars, probed_vars], feed_dict=feed_dict)
        sess.probe_stream.add_summary(results[2], global_step=step)
    else:
        results = sess.run([operators, grabbed_vars], feed_dict=feed_dict)
    if show_interval and (step % show_interval == 0):
        self.display_grabvars(results[1], grabbed_vars, step=step)
    return results[0], results[1], sess
```

- Similar to the earlier autoencoder class, but this handles probed and grabbed variables.
- Note use of probe_stream.add_summary to add the values of probed variables to the TensorBoard.
def training_session(self, epochs, sess=None, dir="probeview", continued=False):
    self.roundup_probes()
    session = sess if sess else TFT.gen_initialized_session(dir=dir)
    self.current_session = session
    self.do_training(session, self.caseman.get_training_cases(),
                     epochs, continued=continued)

**Use of parameter** continued **supports training runs that are extended**
**with additional training epochs after their original completion** (using the
**method** runmore).
In testing, we run cases through the network and ONLY record error. No learning occurs. Hence, the main Tensorflow operator is self.error, not self.trainer.

Minibatch = entire set of test cases, since no learning occurs.

Two common forms of testing are: a) final - when training is finished, and b) validation - intermixed with the training epochs. Final testing usually involves a separate test set, not yet seen by the network.
def testing_session(self, sess):
    cases = self.caseman.get_testing_cases()
    if len(cases) > 0:
        self.do_testing(sess, cases, msg='Final Testing')

def consider_validation_testing(self, epoch, sess):
    if self.validation_interval and (epoch % self.validation_interval == 0):
        error = self.do_testing(sess, self.caseman.get_validation_cases(),
                                 msg='Validation Testing')
        self.validation_history.append((epoch, error))

def test_on_trains(self, sess):
    self.do_testing(sess, self.caseman.get_training_cases(),
                    msg='Total Training')

Each uses the self.do_testing method, but with different case sets.
def run(self, epochs=100, sess=None, continued=False):
    PLT.ion()
    self.training_session(epochs, sess=sess, continued=continued)
    self.test_on_trains(sess=self.current_session)
    self.testing_session(sess=self.current_session)
    self.close_current_session()
    PLT.ioff()

def runmore(self, epochs=100):
    self.reopen_current_session()
    self.run(epochs, sess=self.current_session, continued=True)

When training finishes, run through the entire training set ONCE, without learning, and record the average error.

After a run finishes, runmore enables additional training, picking up where we left off after the last call to run (or to runmore). Use of the continued parameter along with self.global_training_step allows easy updating of the error graph to account for additional runs.
This supports the use of *runmore* and situations in which you first train the network and then test it using different probed or grabbed variables. The test runs will normally employ the weight and bias values learned during the training phase.
class Gannmodule():

def __init__(self, ann, index, invariable, insize, outsize):
    self.ann = ann
    self.insize = insize
    self.outsize = outsize  # Number of neurons in this module
    self.input = invariable
    self.index = index
    self.name = "Module−" + str(self.index)
    self.build()

- Insize = number of neurons in the immediate upstream layer of neurons.
- Input = tensorflow variable: either the GANN’s main input or the output of the module immediately upstream.
- Outsize = number of neurons in this module.
def build(self):
    basename = "Module-"+str(self.index)
    n = self.outsize
    self.weights = tf.Variable(np.random.uniform(-.1,.1,
        size=(self.insize,n)),
        name=basename+ '-wgt',
        trainable=True)
    self.biases = tf.Variable(np.random.uniform(-.1,.1, size=n),
        name=basename+ '-bias',
        trainable=True)
    self.output = tf.nn.relu(tf.matmul(self.input, self.weights) +
        self.biases,name=basename+ '-out')
    self.ann.add_module(self)

def getvar(self,type):  # type = (in, out, wgt, bias)
    return {'in': self.input, 'out': self.output,
        'wgt': self.weights, 'bias': self.biases}[type]

- Multiply weights times inputs, then add to biases, and then pass the
  sum through the (ReLU) activation function.
- The resulting tensor is the output of the module.
- Note: nothing extra was needed to handle minibatches!
def gen_probe(self, type, spec):
    var = self.getvar(type)
    base = self.name + '_' + type
    with tf.name_scope('probe_ '):
        if ('avg' in spec) or ('stdev' in spec):
            avg = tf.reduce_mean(var)
        if 'avg' in spec:
            tf.summary.scalar(base + '/avg/', avg)
        if 'max' in spec:
            tf.summary.scalar(base + '/max/', tf.reduce_max(var))
        if 'min' in spec:
            tf.summary.scalar(base + '/min/', tf.reduce_min(var))
        if 'hist' in spec:
            tf.summary.histogram(base + '/hist/', var)

spec = list with one or more of: avg, min, max and histogram.

All summaries are aggregated in Gann.roundup.probes().

In run_one_step, the values of probed variables are written to the TensorBoard.
def autoex(epochs=300, nbits=4, lrate=0.03, showint=100, mbs=None, vfrac=0.1, tfrac=0.1, vint=100):
    size = 2**nbits
    mbs = mbs if mbs else size
    case_generator = (lambda : TFT.gen_all_one_hot_cases(2**nbits))
    cman = Caseman(cfunc=case_generator, vfrac=vfrac, tfrac=tfrac)
    ann = Gann(dims=[size, nbits, size], cman=cman, lrate=lrate, showint=showint, mbs=mbs, vint=vint)
    ann.gen_probe(0, 'wgt', ('hist', 'avg'))
    ann.gen_probe(1, 'out', ('avg', 'max'))
    ann.add_grabvar(0, 'wgt')
    ann.run(epochs)
    ann.runmore(epochs * 2)
    return ann

- Caseman used to manage the training, testing, and validation cases. See tutor2.py for Caseman code.
- Creates 2 probes, each with 2 statistics: (hist, avg) and (avg,max)
- Probed values and function graph are available in the Tensorboard.
- One grabvar is created and displayed during the run.
These slides do NOT contain every detail of the code. See the files `tflowtools.py`, `tutor1.py`, `tutor2.py` and `tutor3.py` for everything needed to run these neural networks.