Zurich University
of Applied Sciences

# TensorFlow for Deep Learning

Oliver Dürr

Datalab-Lunch Seminar Series
Winterthur, 23 Nov, 2016

Code: github.com/oduerr/dl_tutorial/

# Leftovers

- Notes and things I forgot
  - The Mandelbrot example now also includes loop (see Control_Flow/Madelbrot)
  - Extremely nice tool (DevDocs includes TF and many other useful libs)

- Today it's about using Tensorflow for deep learning. No theory of Deep Learning!

# Outline

- Short Recap

- How to build networks
  - Scoping
- Using existing models
  - Accessing ops and tensors in existing networks
  - Fine-tuning adopt existing networks to a new task
- Debugging
  - Tensorboard
  - tf.Print()

# Most important thing from last time: compute graph

- Edges are arrays with n indices (tensors of order n)
- Nodes are operations (ops)
- These tensors flow hence the name
- To steps process (allows e.g. for symbolic differentiation)
  - Build graph in a abstract fashion
  - Put values in and out with feeds and fetches

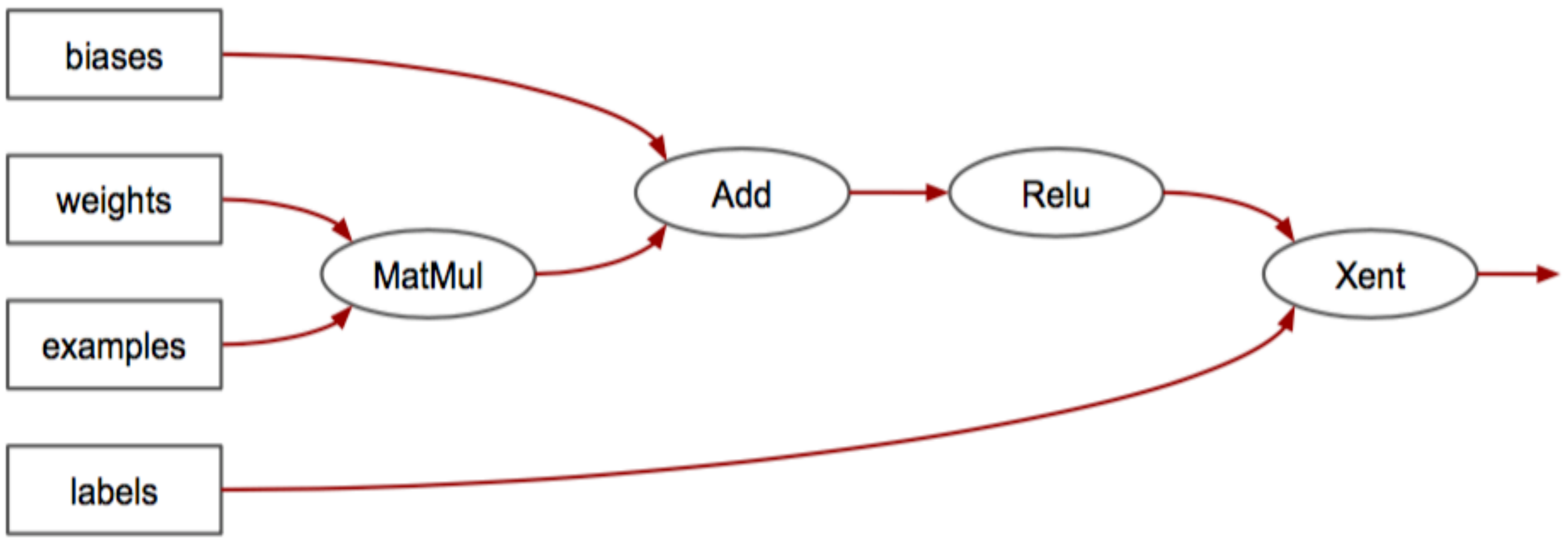



Photo credit TensorFlow documentation

# The Tensors Flowing

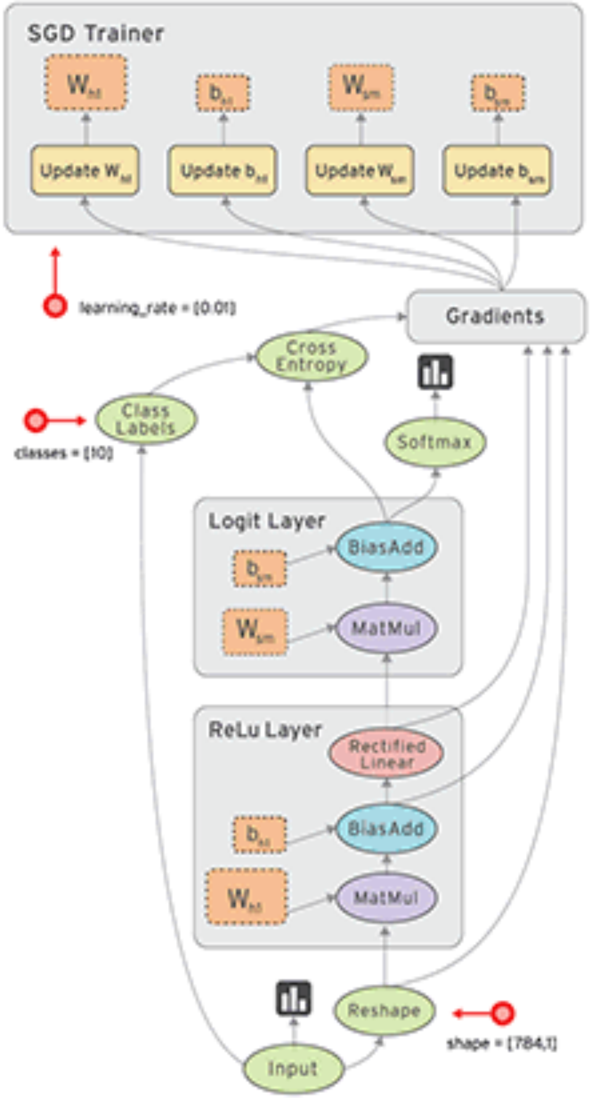
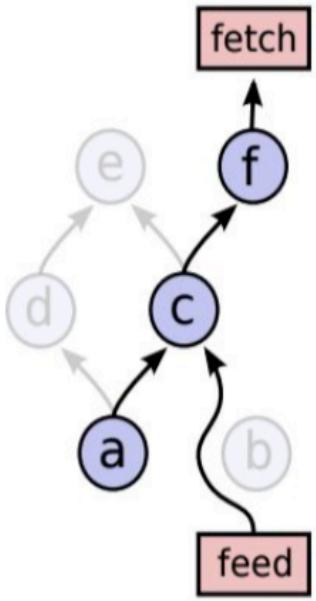



Photo credit TensorFlow documentation

# Most important thing from last time: feeds and fetches




```
res = sess.run(f, feed_dict={b:data[:,0]})
```

fetch
(the numeric value)

Fetch
f (symbolic)

symbolic

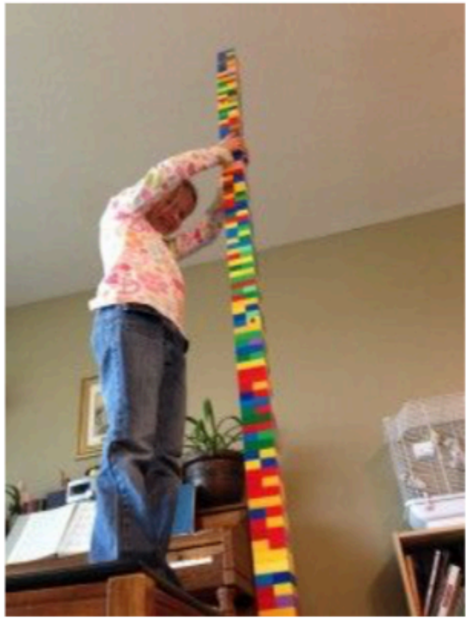
values

Photo credit TensorFlow documentation

# Libraries on top of TensorFlow

- There are lots of libraries on top of TensorFlow. Some of them are in the *tensorflow.contrib* package and are thus installed with TensorFlow
  - TF-Slim
    - nice to build networks
    - contains many pre-trained networks
  - skflow
    - scikit learn like interface (not used so far)
  - TF Learn (inside contrib)
    - I did not use it so far
- Notable exception is the TFLearn (http://tflearn.org/) library (outside TF)
    - Easy training
    - Can handle hdf5 files
    - Includes data augmentation

Since they are all build around TF they can be combined. Other libraries using TF as engine (e.g. Keras no experience so far)




# Building Blocks for Networks

Photo credit CS231n

# Example Network VGG16

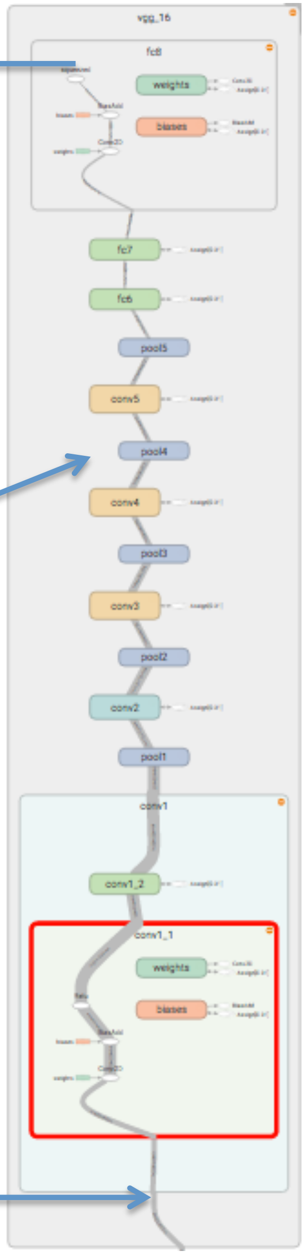
(#BS, 1, 1, 1000)

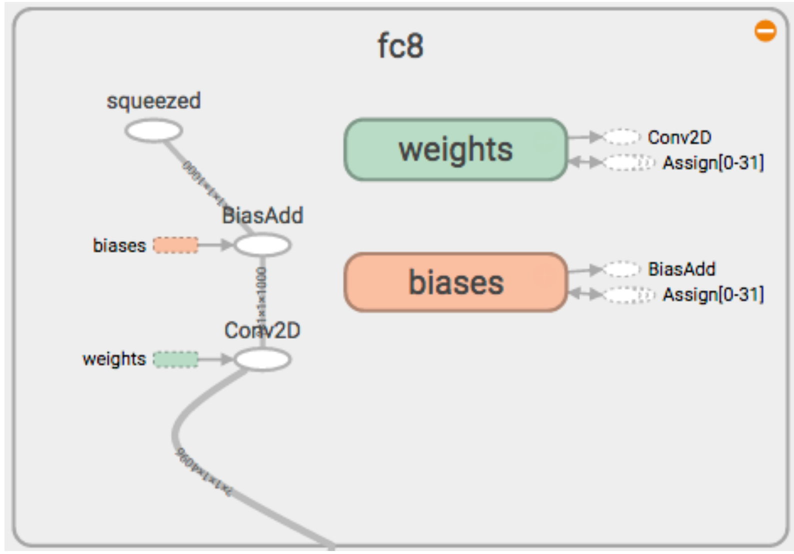
Only conv no fc layers

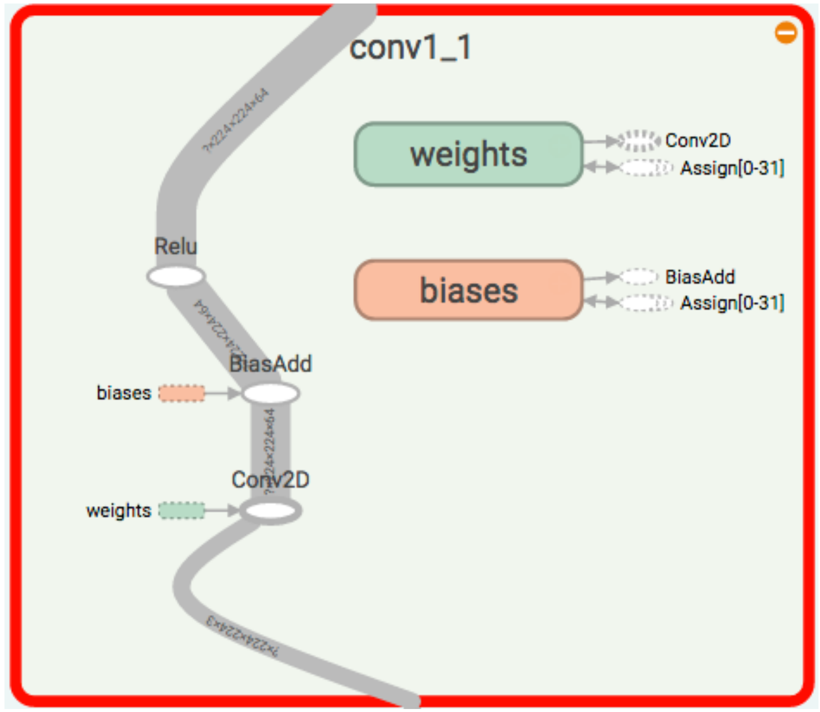
Lot's of repeated units

Only two type of stones needed! Convolutions and Maxpooling

(#BS, 224, 224, 3)

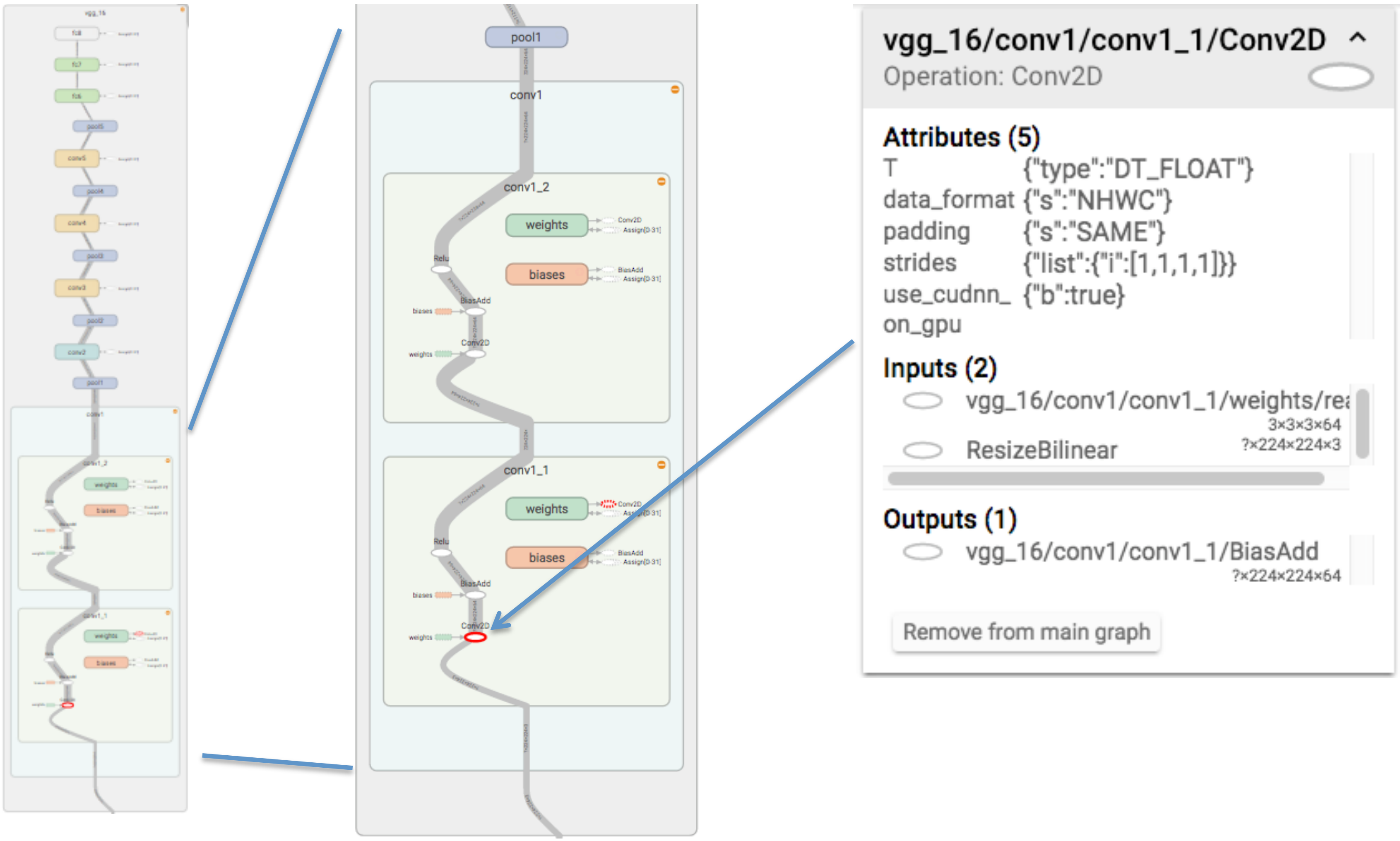


# Name scopes

- Name like vgg16/conv1/conv1_1 allow to group complex networks



# Creation of name scopes

You could simply name ops like '**conv1/Conv2d**' . However, there is a nice mechanism to do so:

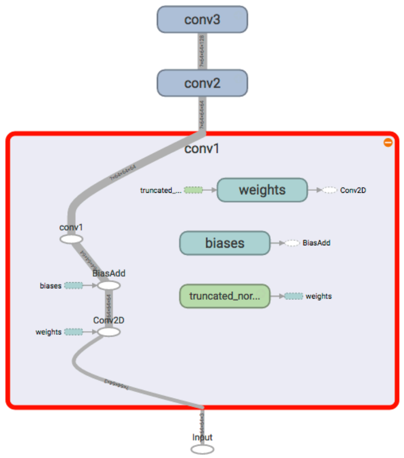
```
In [3]: tf.reset_default_graph()
        with tf.variable_scope('conv1/'):
            net = tf.placeholder(dtype='float32',
                                 shape=(None, 64, 64, 3), name='Input')
            W = tf.get_variable('W', shape=(3,3,3,10))
            var = tf.Variable(tf.truncated_normal((3,3,3,10)),
                              name='Kernels')
            ops = tf.nn.conv2d(net, W, (1,1,1,1), padding='SAME')
        var.name, ops.name

Out[3]: (u'conv1/Kernels:0', u'conv1/Conv2D:0')
```

With this mechanism it's also easy possible to create a network out of simple building blocks…

https://github.com/oduerr/dl_tutorial/blob/master/tensorflow/Building_Nice_Networks/Scoping.ipynb

# Variable scopes as building blocks




```
tf.reset_default_graph()
def conv_layer(net, shape, scope):
    with tf.variable_scope(scope) as v_scope:
        kernel = tf.Variable(tf.truncated_normal …
        conv = tf.nn.conv2d(net, kernel, [1, 1, 1, 1], padding='SAME')
        biases = tf.Variable(tf.constant(0.0, shape=[shape[3]]…
        out = tf.nn.bias_add(conv, biases)
        return tf.nn.relu(out, name=scope)

net = tf.placeholder(dtype='float32', shape=(None, 64, 64, 3),
name='Input')
net = conv_layer(net, [3, 3, 3, 64], 'conv1')
net = conv_layer(net, [3, 3, 64, 128], 'conv2')
net = conv_layer(net, [3, 3, 128, 128], 'conv3')
writer = tf.train.SummaryWriter("/tmp/dumm/scoping",
tf.get_default_graph(), 'graph.pbtxt')
```

https://github.com/oduerr/dl_tutorial/blob/master/tensorflow/Building_Nice_Networks/Scoping.ipynb

# Variable scopes to share variables

- Variable scoping is a mechanism to share the variables of (possible large) parts of a network, without the need to pass references.

- These shared variables are needed for example in Siamese Networks.

- Two function with go together:
  - `tf.variable_scope()` created the name-space or better context manager
  - `tf.get_variable()` gets or newly creates variables in the name scope
  - Here we do not use tf.Variable()

- See also
  - https://www.tensorflow.org/versions/master/how_tos/variable_scope/index.html
  - http://stackoverflow.com/questions/35919020/whats-the-difference-of-name-scope-and-a-variable-scope-in-tensorflow

# Variable scoping (new variables)

```
tf.reset_default_graph()
with tf.variable_scope('var'):
    a1 = tf.get_variable('a', shape=(1))
    #This variable is used and thus this would result in an error
    #a1_1 = tf.get_variable('a', shape=(1))
    a2 = tf.get_variable('a2', shape=(1))
a1.name,a2.name
```

```
(u'var/a:0', u'var/a2:0')
```

# Variable scoping (shared variables)

```
tf.reset_default_graph()

with tf.variable_scope('var', reuse=False):
     a1 = tf.get_variable('a', shape=(1))

with tf.variable_scope('var', reuse=True):
    a1_1 = tf.get_variable('a', shape=(1)) #This variable is reused
    #This would give an error, since that variable has not been used before
    #a2 = tf.get_variable('a2', shape=(1))
a1.name, a1_1.name
```

```
(u'var/a:0', u'var/a:0')
```

https://github.com/oduerr/dl_tutorial/blob/master/tensorflow/Building_Nice_Networks/Scoping.ipynb

## Notebook and sharing weights

Let’s have a look at the notebook. In this notebook it is also explained how to **share variables** (e.g. for Siamese Networks) with variable scopes and `tf.get_variable('v', shape=(1,10))`.

https://github.com/oduerr/dl_tutorial/blob/master/tensorflow/Building_Nice_Networks/Scoping.ipynb

# Using pre-trained networks

## Checkpointing (saving)

```
... #← Definition of the network
epochs = 1000
saver = tf.train.Saver()
with tf.Session() as sess:
    sess.run(init_op)
    for e in range(5):
        sess.run(train_op, feed_dict={x:x_data, y:y_data})
    res = sess.run([loss, a, b], feed_dict={x:x_data, y:y_data})
    print(res)
    save_path = saver.save(sess, "checkpoints/model.ckpt")
    print("Model saved in file: %s" % save_path)
print('Finished all')
```

https://github.com/oduerr/dl_tutorial/blob/master/tensorflow/linear_regression/03_checkpointing.ipynb

# Checkpointing (restoring)

```
... #← The network needs to be defined. It is not stored.
saver = tf.train.Saver()
with tf.Session() as sess:
    saver.restore(sess, "checkpoints/model.ckpt")
    res = sess.run([loss, a, b], feed_dict={x:x_data, y:y_data})
    print(res)
```

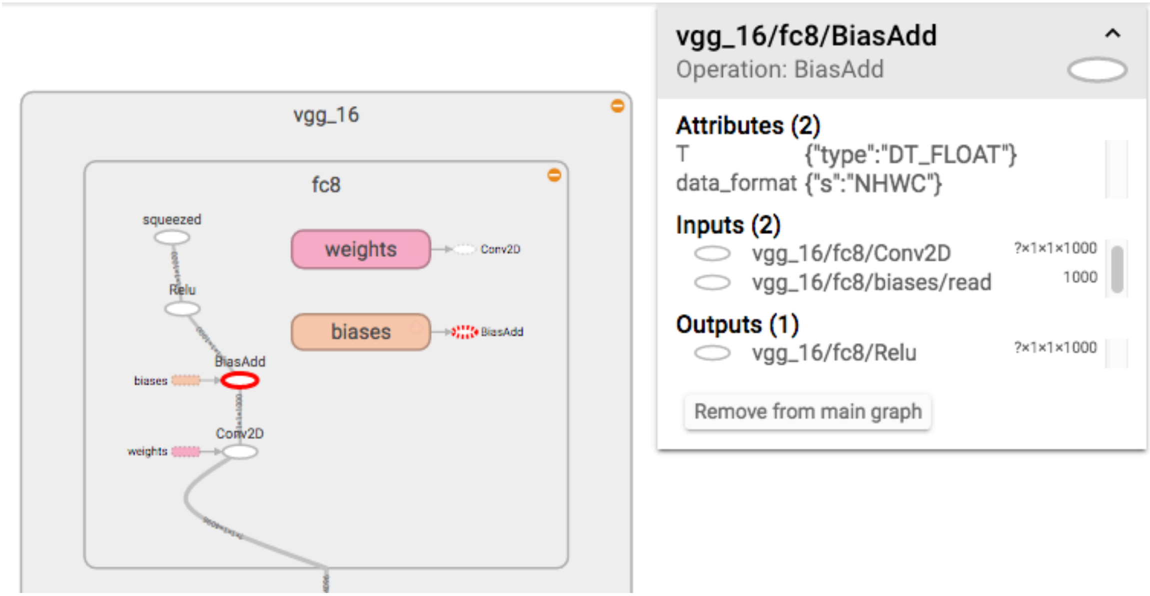
**Weights and definition of the graph**

Checkpointing just stores the weights. The definition of the network has to be defined or stored separately. If you what to do it all in one, you have to transform the weights in constants and save the network. This is referred to *freezing* the graph.

https://github.com/oduerr/dl_tutorial/blob/master/tensorflow/LinearRegression/03_checkpointing.ipynb

# Using existing models [show]


- The notebook: [Using_Trained_Nets]* shows how to access trained networks



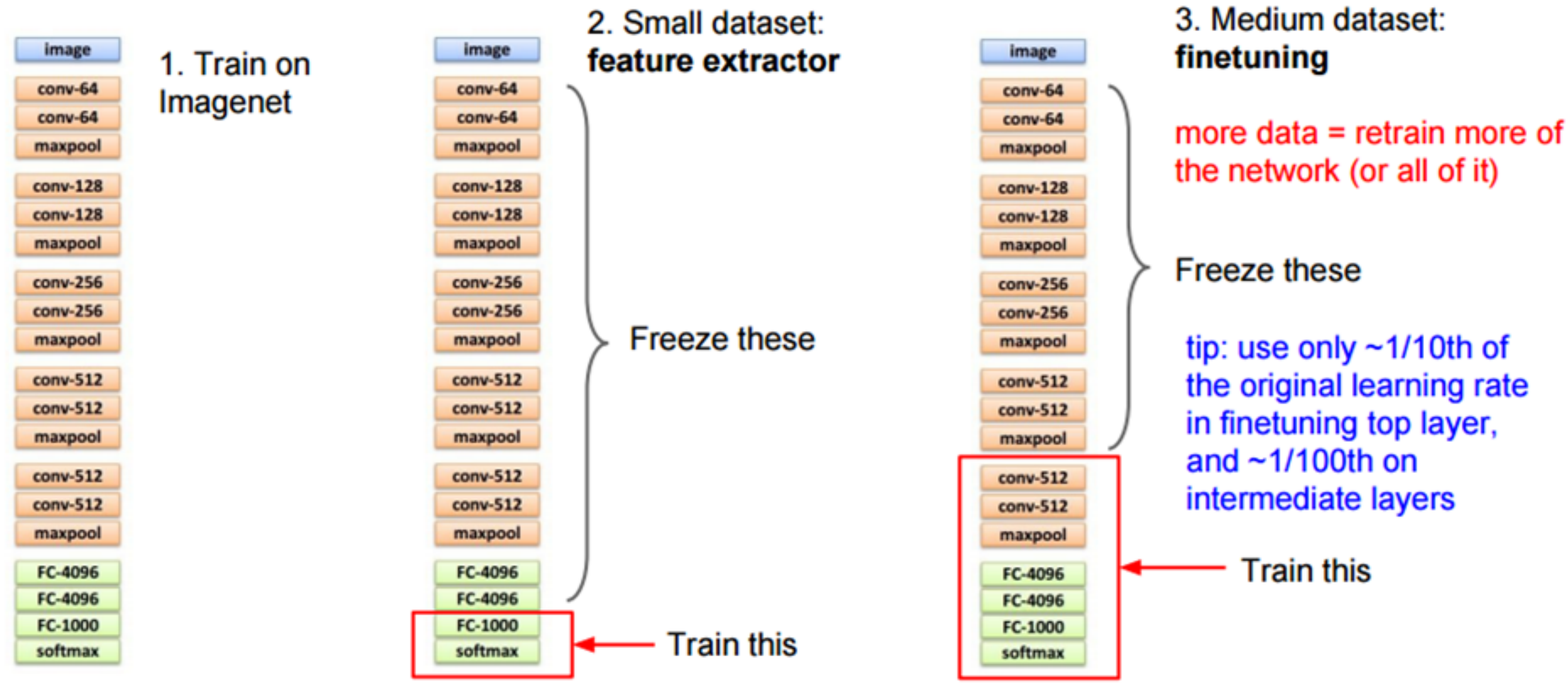

```
feed = tf.Graph.get_tensor_by_name(tf.get_default_graph(), 'Placeholder:0')
fetch = tf.Graph.get_tensor_by_name(tf.get_default_graph(), 'vgg_16/fc8/BiasAdd:0')
res = sess.run(fetch, feed_dict={feed:feed_vals})
```

**This feeding and fetching is extremely useful for debugging, see later)**

*https://github.com/oduerr/dl_tutorial/tree/master/tensorflow/stored_models

# Transfer Learning [show]

## Transfer Learning with CNNs




[*https://github.com/oduerr/dl_tutorial/tree/master/tensorflow/stored_models](https://github.com/oduerr/dl_tutorial/tree/master/tensorflow/stored_models)

# Debugging

# Debugging: run part of the graph feed and fetch

Tensorflow allows us to run parts of graph in isolation, i.e. only the relavant part of graph is executed (rather than executing *everything*)

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
bias = tf.Variable(1.0)

y_pred = x ** 2 + bias     # x -> x^2 + bias
loss = (y - y_pred)**2     # l2 loss?

# Error: to compute loss, y is required as a dependency
print('Loss(x,y) = %.3f' % session.run(loss, {x: 3.0}))

# OK, print 1.000 = (3**2 + 1 - 9)**2
print('Loss(x,y) = %.3f' % session.run(loss, {x: 3.0, y: 9.0}))

# OK, print 10.000; for evaluating y_pred only, input to y is not required
print('pred_y(x) = %.3f' % session.run(y_pred, {x: 3.0}))

# OK, print 1.000 bias evaluates to 1.0
print('bias      = %.3f' % session.run(bias))
```

Slide taken from: https://wookayin.github.io/TensorflowKR-2016-talk-debugging/#27

# Designing graphs for debugging

- You want to access the graph at different entry points

- You can get every Tensor in the graph (to feed or fetch)
  - E.g. `tf.Graph.get_tensor_by_name(tf.get_default_graph(), 'Placeholder:0')`

- However, it is much nicer to give the user handles to the tensors

- There are good and bad ways of doing so:

# Designing graphs for debugging: bad

```
def alexnet(x):
    assert x.get_shape().as_list() == [224, 224, 3]
    conv1 = conv_2d(x, 96, 11, strides=4, activation='relu')
    pool1 = max_pool_2d(conv1, 3, strides=2)
    conv2 = conv_2d(pool1, 256, 5, activation='relu')
    pool2 = max_pool_2d(conv2, 3, strides=2)
    conv3 = conv_2d(pool2, 384, 3, activation='relu')
    conv4 = conv_2d(conv3, 384, 3, activation='relu')
    conv5 = conv_2d(conv4, 256, 3, activation='relu')
    pool5 = max_pool_2d(conv5, 3, strides=2)
    fc6 = fully_connected(pool5, 4096, activation='relu')
    fc7 = fully_connected(fc6, 4096, activation='relu')
    output = fully_connected(fc7, 1000, activation='softmax')
    return conv1, pool1, conv2, pool2, conv3, conv4, conv5, pool5, fc6, fc7


# At construction time
conv1, conv2, conv3, conv4, conv5, fc6, fc7, output = alexnet(images)  # ?!

# During the training loop
_, loss_, conv1_, conv2_, conv3_, conv4_, conv5_, fc6_, fc7_ = session.run(
        [train_op, loss, conv1, conv2, conv3, conv4, conv5, fc6, fc7],
        feed_dict = {...})
```

Quite messy code!

Slide taken from: https://wookayin.github.io/TensorflowKR-2016-talk-debugging/#27

# Designing graphs for debugging: good with dictionary

```
def alexnet(x, net={}):
    assert x.get_shape().as_list() == [224, 224, 3]
    net['conv1'] = conv_2d(x, 96, 11, strides=4, activation='relu')
    net['pool1'] = max_pool_2d(net['conv1'], 3, strides=2)
    net['conv2'] = conv_2d(net['pool1'], 256, 5, activation='relu')
    net['pool2'] = max_pool_2d(net['conv2'], 3, strides=2)
    net['conv3'] = conv_2d(net['pool2'], 384, 3, activation='relu')
    net['conv4'] = conv_2d(net['conv3'], 384, 3, activation='relu')
    net['conv5'] = conv_2d(net['conv4'], 256, 3, activation='relu')
    net['pool5'] = max_pool_2d(net['conv5'], 3, strides=2)
    net['fc6'] = fully_connected(net['pool5'], 4096, activation='relu')
    net['fc7'] = fully_connected(net['fc6'], 4096, activation='relu')
    net['output'] = fully_connected(net['fc7'], 1000, activation='softmax')
    return net['output']


net = {}
output = alexnet(images, net)
# access intermediate layers like net['conv5'], net['fc7'], etc.
```

Better

Slide taken from: https://wookayin.github.io/TensorflowKR-2016-talk-debugging/#27

# Designing graphs for debugging: good with class

```
class AlexNetModel():
    # ...
    def build_model(self, x):
        assert x.get_shape().as_list() == [224, 224, 3]
        self.conv1 = conv_2d(x, 96, 11, strides=4, activation='relu')
        self.pool1 = max_pool_2d(self.conv1, 3, strides=2)
        self.conv2 = conv_2d(self.pool1, 256, 5, activation='relu')
        self.pool2 = max_pool_2d(self.conv2, 3, strides=2)
        self.conv3 = conv_2d(self.pool2, 384, 3, activation='relu')
        self.conv4 = conv_2d(self.conv3, 384, 3, activation='relu')
        self.conv5 = conv_2d(self.conv4, 256, 3, activation='relu')
        self.pool5 = max_pool_2d(self.conv5, 3, strides=2)
        self.fc6 = fully_connected(self.pool5, 4096, activation='relu')
        self.fc7 = fully_connected(self.fc6, 4096, activation='relu')
        self.output = fully_connected(self.fc7, 1000, activation='softmax')
        return self.output

model = AlexNetModel()
output = model.build_model(images)
# access intermediate layers like self.conv5, self.fc7, etc.
```

Better

Slide taken from: https://wookayin.github.io/TensorflowKR-2016-talk-debugging/#27

## Summaries

Taken from: [LinearRegression/02_Inspecting_the_graph.ipynb]()

```
resi = a*x + b - y
loss = tf.reduce_sum(tf.square(resi), name='loss')
...
#Definition of ops to be stored
loss_summary = tf.scalar_summary("loss_summary", loss) #<-- creates op!
resi_summart = tf.histogram_summary("resi_summart", resi)
merged_summary_op = tf.merge_all_summaries()#<-----  Combine all ops to be stored

sess.run(init_op)
#Where to store
writer = tf.train.SummaryWriter("/tmp/dumm/run1", tf.get_default_graph(),
'graph.pbtxt')
for e in range(epochs): #Fitting the data for 10 epochs
    ...
    #Running the graph to produce output
    sum_str = sess.run(merged_summary_op, feed_dict={x:x_vals, y:y_vals})
    writer.add_summary(sum_str, e) #<--- writing out the output
```

# Print- a bit non-trivial at the first sight

The op, which is watched. Creates identity op, with side effect printing.

This, is the op with
is used from now

Tensors with are printed as side effect

```
loss = (y - y_pred)**2
loss = tf.Print(loss, [y, y_pred, loss],
                message='Debug y, y_pred ', name='Debug_Print', first_n=5)
# The loss function is now decorated with the print function
with tf.Session() as sess:
    …
    print('Loss : {}'.format(sess.run(loss, feed_dict={x:10, y:10})))

# Not working is
# tf.Print(loss, [y, y_pred, (y-y_pred)**2], name='Name', first_n=5)
# loss = tf.Print(loss, y, name='Name', first_n=5)

I tensorflow/core/kernels/logging_ops.cc:79] Debug y, y_pred [10][21][121]
Loss : 121.0
```

# More Debugging

- `tf.Assert()`
  - Creates runtime assertions

- Possibility to use python code as tensorflow op.
  - [`tf.py_func()`]()

```
def my_func(x):
  # x will be a numpy array with the contents of the placeholder below
  return np.sinh(x)
inp = tf.placeholder(tf.float32, [...])
y = py_func(my_func, [inp], [tf.float32])
```

The above snippet constructs a tf graph which invokes a numpy sinh(x) as an op in the graph.

For a detailed explanation of the above concepts see:
https://wookayin.github.io/TensorflowKR-2016-talk-debugging/

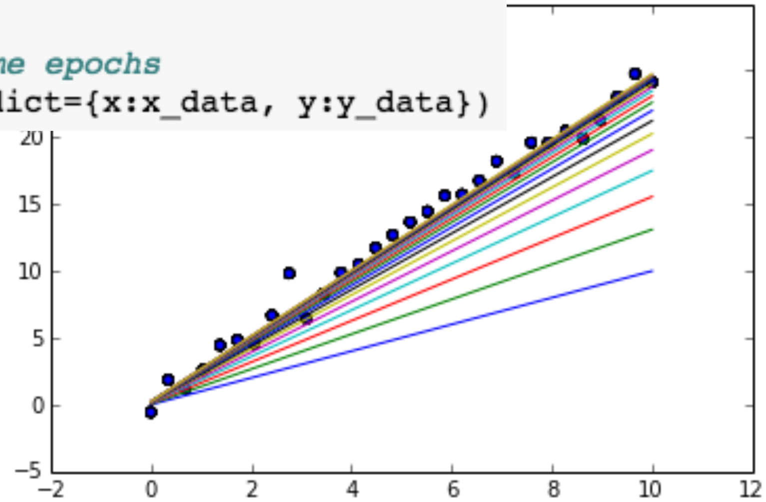
# Debugging with embedded python code

```
def _debug_plot(a_val, b_val, x_val, y_val):
    plt.scatter(x_val,y_val)
    ablineValues = [a_val * x_ + b_val for i, x_ in enumerate(x_val)]
    plt.plot(x_val, ablineValues)
    return False
```

Decorating the loss function

```
debug_op = tf.py_func(_debug_plot, [a, b, x, y], [tf.bool])
with tf.control_dependencies(debug_op):
    loss = tf.identity(loss, name='out')
```

```
epochs = 20
with tf.Session() as sess:
    sess.run(init_op) #Running the initialization
    for e in range(epochs): #Fitting the data for some epochs
        res = sess.run([loss, train_op, a, b], feed_dict={x:x_data, y:y_data})
```




See debugging/debug_with_python.ipynb