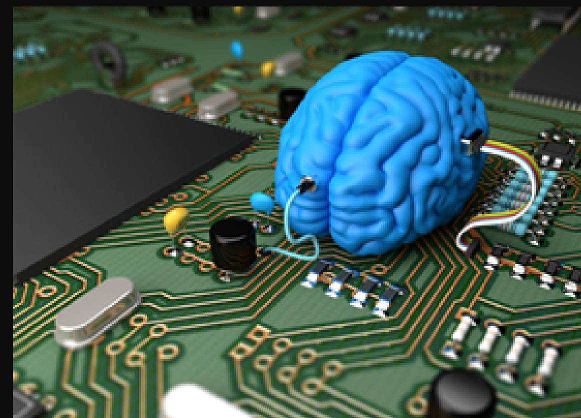
Deep Learning
Hands-on

Elisa Ricci

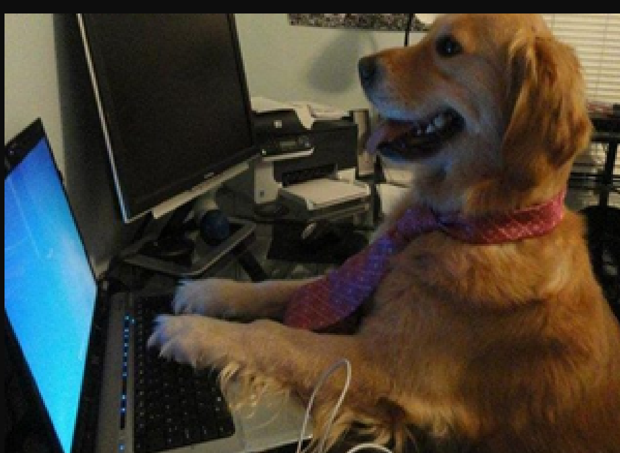
Deep Learning

What society thinks I do



What my friends think I do

What other computer scientists think I do



What mathematicians think I do

What I think I do

```
from theano import *
```

What I actually do

Outline

- Deep Learning Frameworks
- Introduction to TensorFlow
  - Examples (linear regression, MNIST)
- Introduction to Keras
  - Examples (MNIST MLP & CNN)

Deep Learning Frameworks

Deep Learning Frameworks

- Many different frameworks over the past few years...

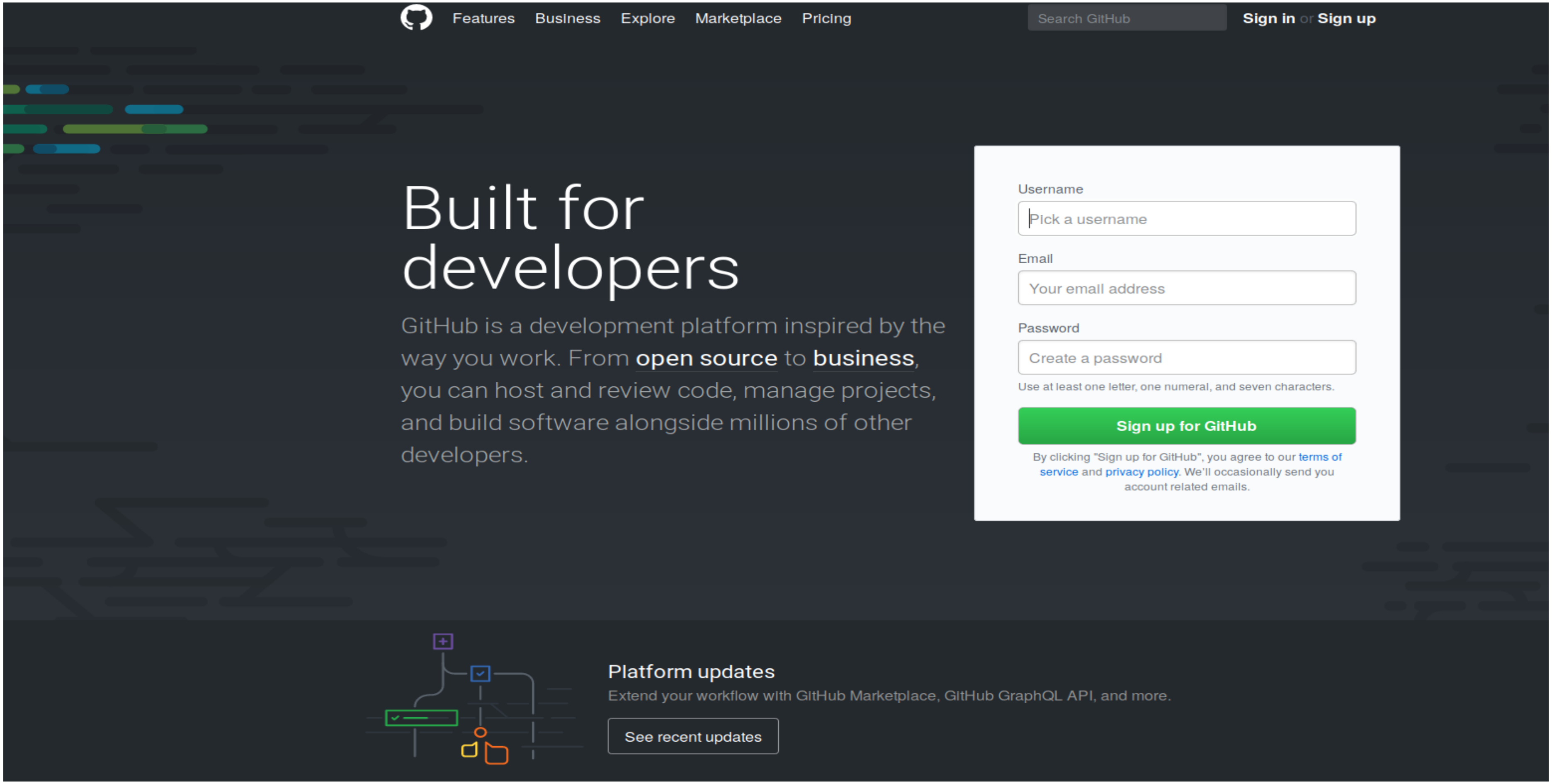
Caffe
Chainer
DL4J Deeplearning4j
KERAS
Microsoft CNTK
MatConvNet
MINERVA
mxnet
Purine
TensorFlow
theano
torch

Deep Learning Frameworks

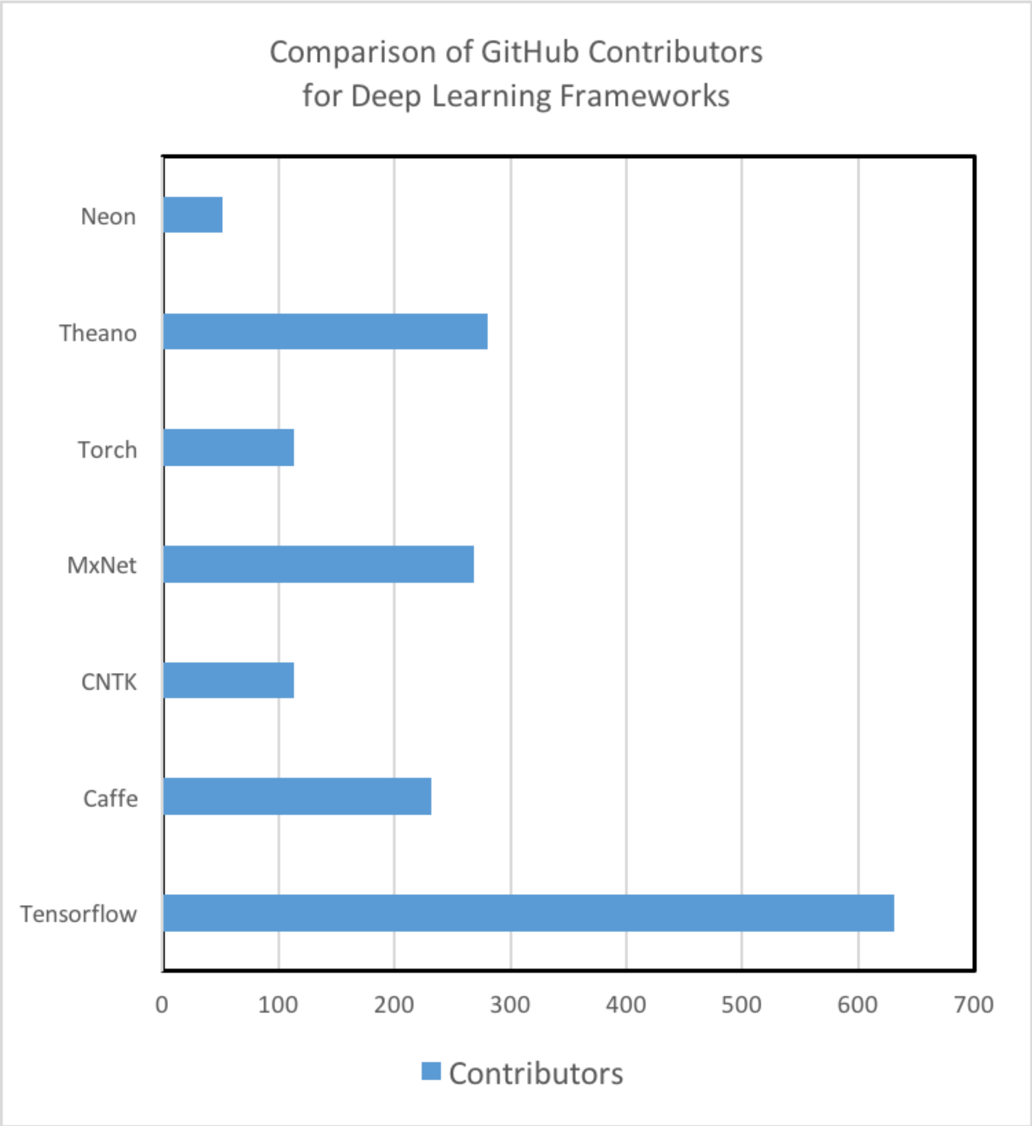
TensorFlow	Google Brain, 2015 (rewritten DistBelief)
Theano	University of Montréal, 2009
Keras	François Chollet, 2015 (now at Google)
Torch	Facebook AI Research, Twitter, Google DeepMind
Caffe	Berkeley Vision and Learning Center (BVLC), 2013

Deep Learning Frameworks

- Which framework to choose? Look at GitHub...

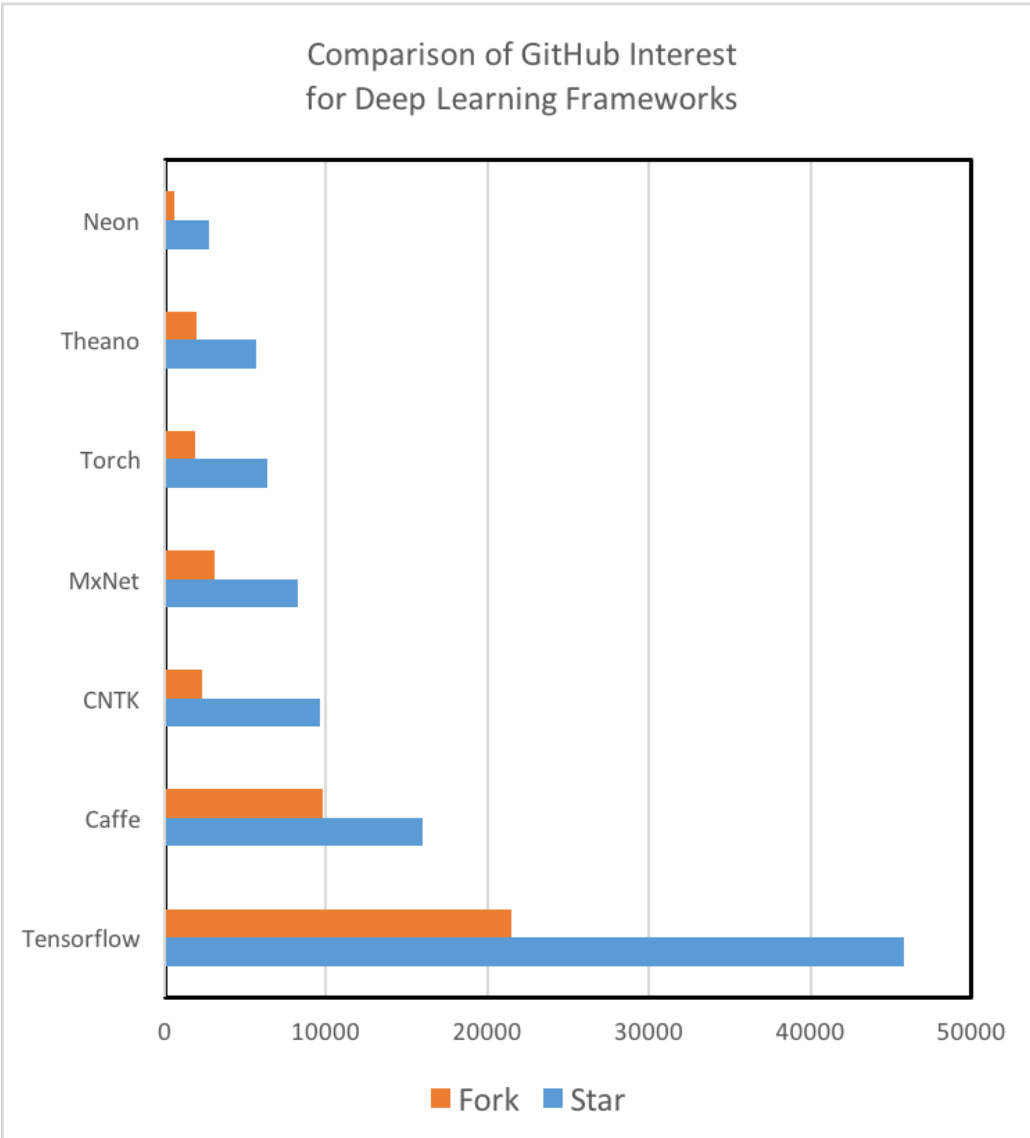


Deep Learning Frameworks



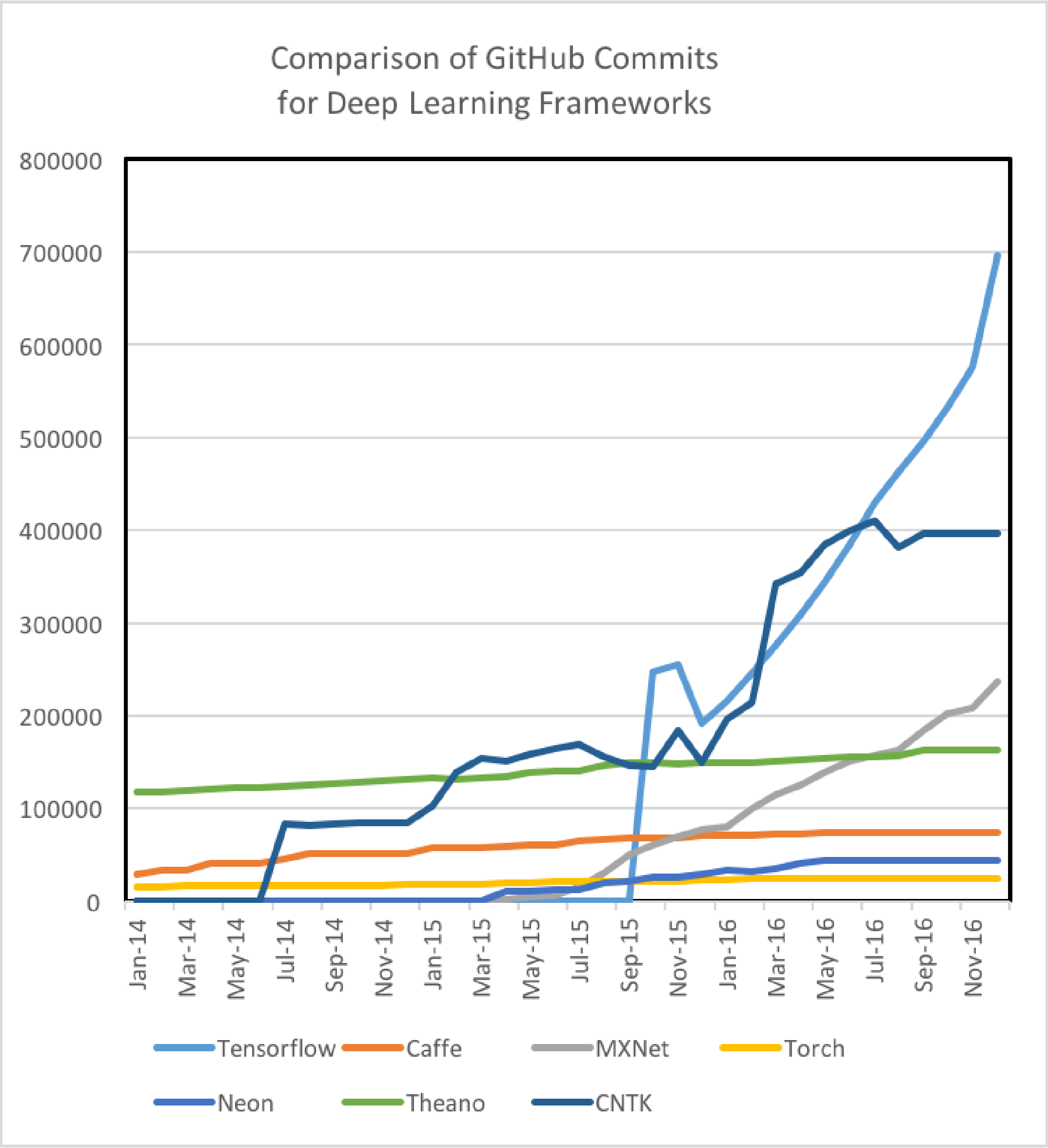
[Rubashkin]

Deep Learning Frameworks



[Rubashkin]

Deep Learning Frameworks



[Rubashkin]

Community and Resources

- (Github, groups, discussions…)
  - For CNNs Caffe has the largest community
  - TensorFlow’ s is already large and growing
  - Keras’ community is growing
  - Theano’ s and Lasagne’ s community are declining

Theano

- Maintained by Montréal University group
- Pioneered the use of a computational graph
- General machine learning tool
- Symbolic differentiation
- Use of Lasagne and Keras
- Very popular in the research community, but not much elsewhere. Falling behind

Torch

- Mixed language:
  - C/CUDA backend built on common backend libraries
  - Lua frontend
- Flexibility: existing building blocks from the community can be easily integrated
- Automatic differentiation
- Modularity
- Speed
- *(People hate Lua) → very recently PyTorch*

Caffe

- Pros:
  - Especially good for CNN and Computer Vision
  - Extremely easy to code
  - Easy to use pretrained models
  - Matlab and Python interface
  - Easy to include different libraries
  - Layer as building block and many layers already implemented online

Caffe

- Cons:
  - No auto-differentiation
  - Need to write C++/CUDA for new GPU layers
  - Not good for RNN
  - Cumbersome for big networks (ResNet)

Caffe

- Main steps:
  - creation of the training network for learning and test network(s) for evaluation
  - iterative optimization by calling forward/backward and parameter updating
  - (periodical) evaluation of the test networks
  - snapshotting of the model and solver state throughout the optimization

Caffe

Caffe

- Models:

```
layer {
  name: "pool1"
  type: "Pooling"
  pooling_param {
    kernel_size: 2
    stride: 2
    pool: MAX
  }
  bottom: "conv1"
  top: "pool1"
}
```

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 0
    decay_mult: 0
  }
  convolution_param {
    num_output: 64
    kernel_size: 3
    pad: 1
   }
}
```

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "fc8"
  bottom: "label"
  top: "loss"
}
```

Caffe

Caffe

- Solver:

```
base_lr: 0.01           # begin training at a learning rate of 0.01 = 1e-2

lr_policy: "step"       # learning rate policy: drop the learning rate in "steps"
                        # by a factor of gamma every stepsize iterations

gamma: 0.1              # drop the learning rate by a factor of 10
                        # (i.e., multiply it by a factor of gamma = 0.1)

stepsize: 100000        # drop the learning rate every 100K iterations

max_iter: 350000        # train for 350K iterations total
```

Which framework to chose

	Languages	Tutorials and training materials	CNN modeling capability	RNN modeling capability	Architecture: easy-to-use and modular front end	Speed	Multiple GPU support	Keras compatible
Theano	Python, C++	++	++	++	+	++	+	+
Tensor-Flow	Python	+++	+++	++	+++	++	++	+
Torch	Lua, Python (new)	+	+++	++	++	+++	++	
Caffe	C++	+	++		+	+	+	

[Rubashkin]

Which framework to chose

- You work in industry:
  - TensorFlow, Caffe

- You want to work "seriously" on new models (research-oriented):
  - TensorFlow, Theano, (Torch)

- You don't have time and you are just curious about deep learning:
  - Keras, Caffe

- You want to use deep learning for educational purposes:
  - Keras, Caffe

TensorFlow

TensorFlow

- An open-source software library for Machine Intelligence
- Especially useful for Deep Learning
- For research & industry

TensorFlow

TensorFlow™ | Install | Develop | API r1.1 | Deploy | Extend | Resources | Versions | TFRC | Search | GITHUB

TensorFlow 1.2rc0 has arrived!

We're excited to announce the release of TensorFlow 1.2rc0! Check out the release notes for all the latest.

UPGRADE NOW

Introducing TensorFlow Research Cloud

We're making 1,000 Cloud TPUs available for free to accelerate open machine learning research.

LEARN MORE

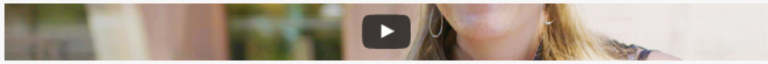
The 2017 TensorFlow Dev Summit

Thousands of people from the TensorFlow community participated in the first flagship event. Watch the keynote and talks.

WATCH VIDEOS

About TensorFlow

TensorFlow™ is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.



News

Announcing TensorFlow 1.0

In just its first year, TensorFlow has helped researchers, engineers, artists, students, and many others make progress with everything from language translation to early detection of skin cancer and preventing blindness in diabetics. We're excited to see people using TensorFlow in over 6000 open-source repositories online.

Celebrating TensorFlow's First Year

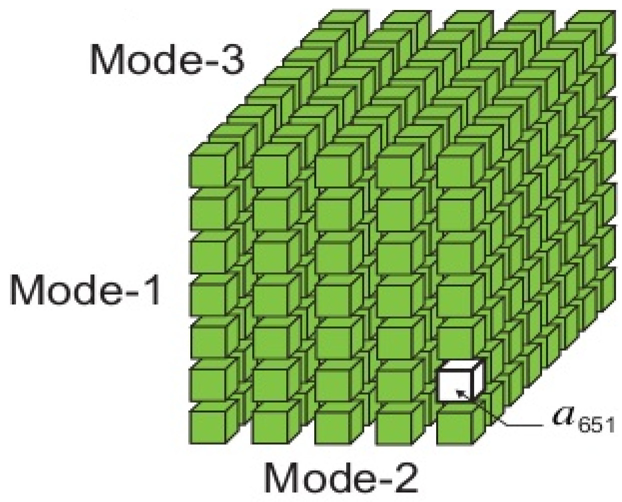
It has been an eventful year since the Google Brain Team open-sourced TensorFlow to accelerate machine learning research and make technology work better for everyone. There has been an amazing amount of activity around the project: more than 480 people have contributed directly to TensorFlow.

A Neural Network for Machine Translation, at Production Scale

Ten years ago, we announced the launch of Google Translate, together with the use of Phrase-Based Machine Translation as the key algorithm behind this service. Since then, rapid advances in machine intelligence have improved our speech recognition and image recognition capabilities,

TensorFlow

Tensors: multidimensional arrays



TensorFlow

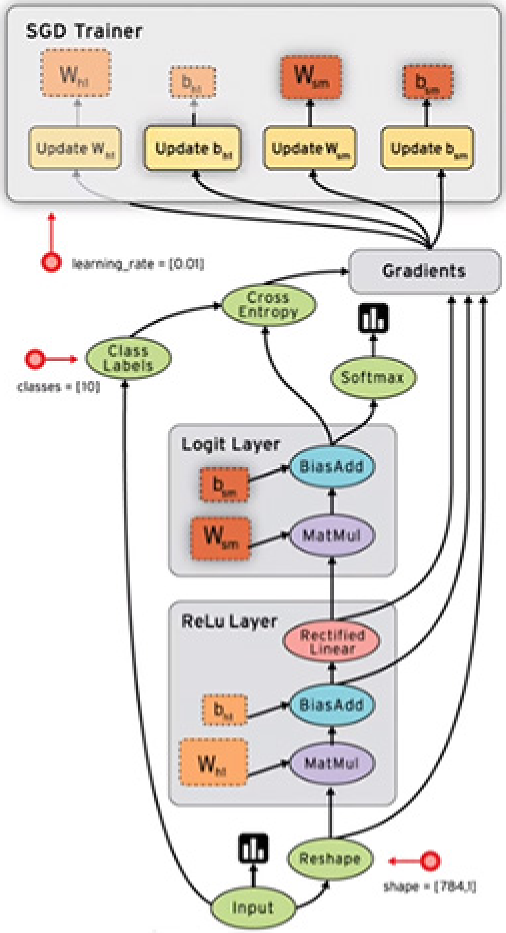
Tensors: multidimensional arrays

The central unit of data in TensorFlow is the **tensor**. A tensor consists of a set of primitive values shaped into an array of any number of dimensions. A tensor's **rank** is its number of dimensions. Here are some examples of tensors:

```
3 # a rank 0 tensor; this is a scalar with shape []
[1. ,2., 3.] # a rank 1 tensor; this is a vector with shape [3]
[[1., 2., 3.], [4., 5., 6.]] # a rank 2 tensor; a matrix with shape [2, 3]
[[[1., 2., 3.]], [[7., 8., 9.]]] # a rank 3 tensor with shape [2, 1, 3]
```

TensorFlow

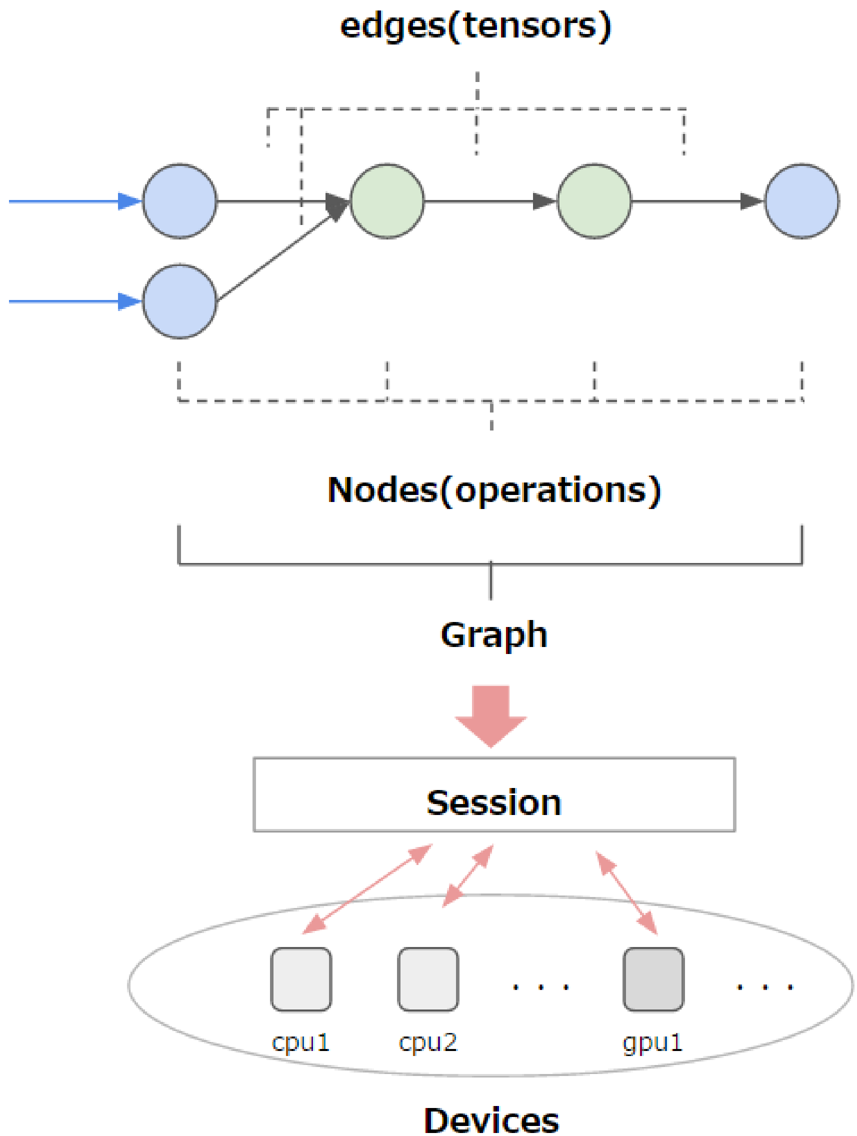
Flow: Graph describing operations



DataFlow Graph

- Computation is defined as a directed acyclic graph (DAG) to optimize an objective function
- Graph is defined in high-level language (Python, C++)
- Graph is compiled and optimized
- Graph is executed (in parts or fully) on available low level devices (CPU, GPU, Android)
- Data (tensors) flow through the graph

TensorFlow Idea



Automatic differentiation

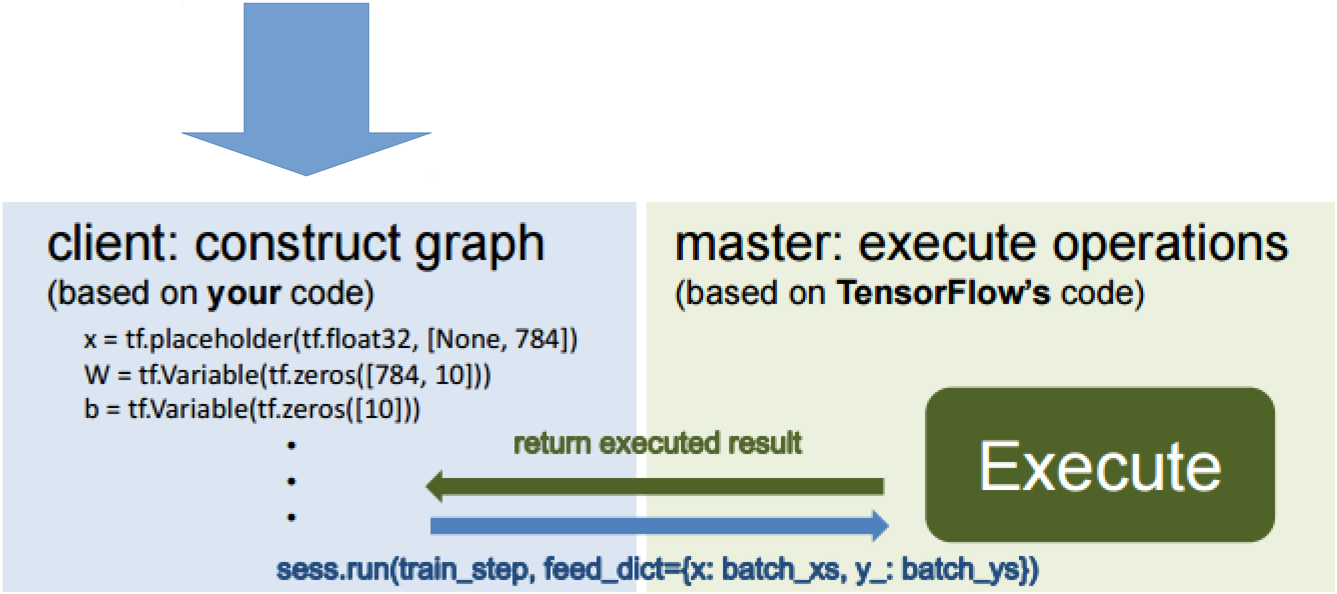
- TensorFlow can compute gradients automatically
  - Reverse automatic differentiation
  - In a nutshell:
    - When you define an operator (op), you also define together how its derivatives are computed (of course most of the common ops are already provided).
    - After you write a function by stacking a series of ops, the program can figure out by itself how should the corresponding derivatives be computed (usually by keeping some computation graphs and using the chain rule).
    - The benefit is obvious as it saves us from working out the math, writing the code, verifying the derivatives numerically…

Main Components

- The main components of Tensorflow:
  - **Variables:** Retain values between sessions, use for weights/bias
  - **Nodes:** The operations
  - **Tensors:** Signals that pass from/to nodes
  - **Placeholders:** used to send data between your program and the tensorflow graph
  - **Session:** Place when graph is executed.

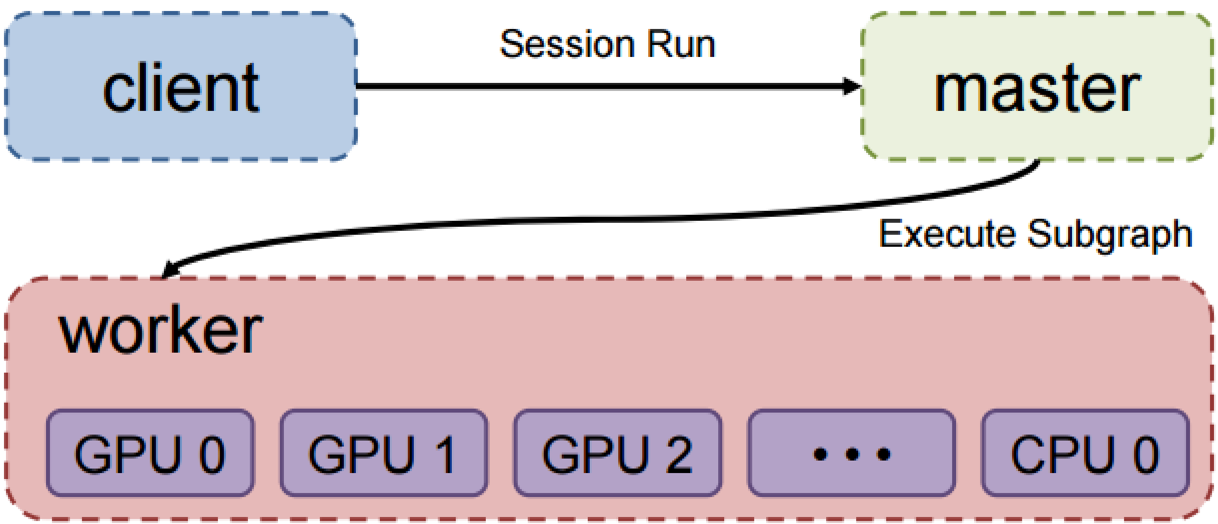
What we do

- Create a graph using code C++ or Python and ask TensorFlow to execute this graph.



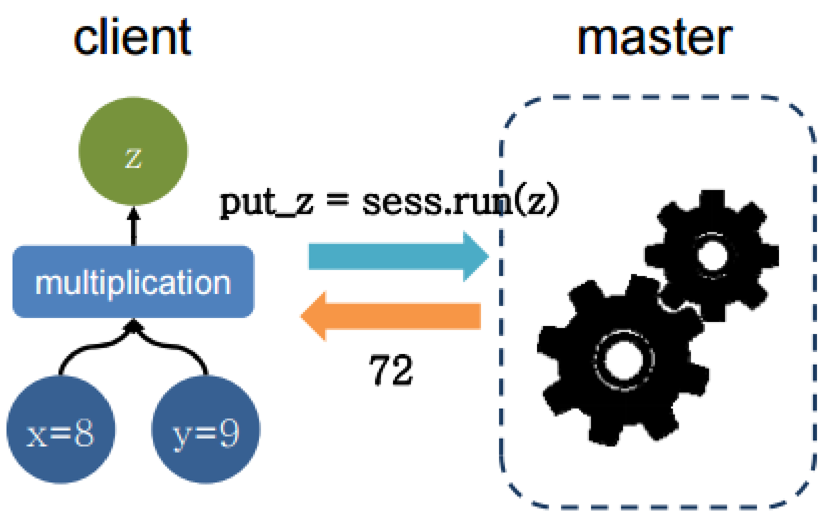
What we do

- Execution



Hello world

- Multiply two numbers
- Main phases:
  - Import TensorFlow library
  - Build the graph
  - Create a session
  - Run the session



Hello world

- Multiply two numbers

mult.py

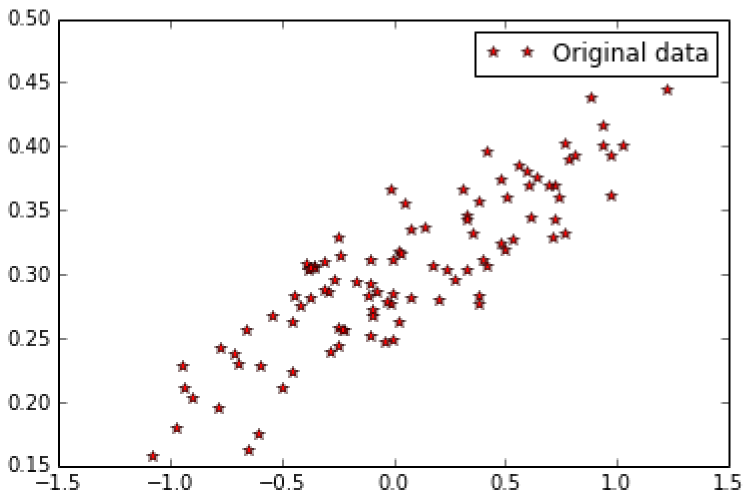
```python
import tensorflow as tf

#build graph
x = tf.constant(8)
y = tf.constant(9)
z = tf.multiply(x,y)

#create_session
sess =tf.Session()

#run_session
out_z = sess.run(z)

print("out_z: %d"%out_z)

```

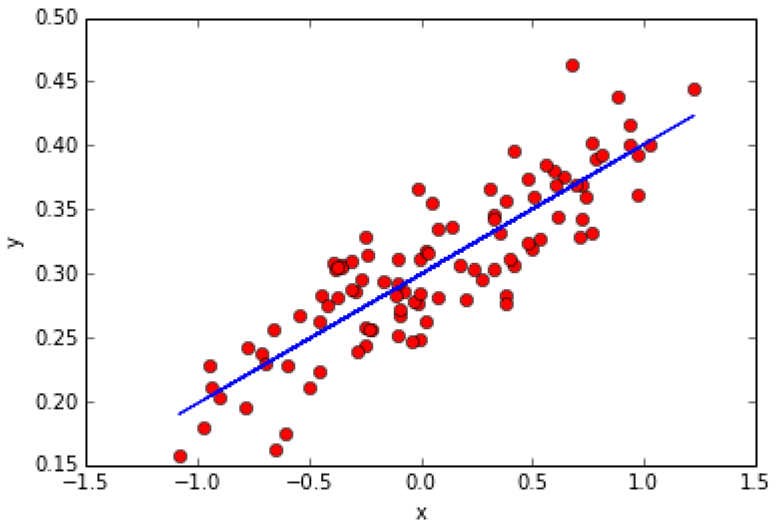
Placeholders

- Allow exchanging data with your graph variables through "placeholders".
- They can be assigned when we ask the session to run

```python
# Import tensorflow
import tensorflow as tf

# Build graph
a = tf.placeholder('float')
b = tf.placeholder('float')

# Graph
y = tf.multiply(a,b)

# Create session passing the graph
session = tf.Session()
# Put the values 3,4 on the placeholders a,b
print session.run(y,feed_dict={a: 3, b:4})
```

Linear Regression





Linear Regression

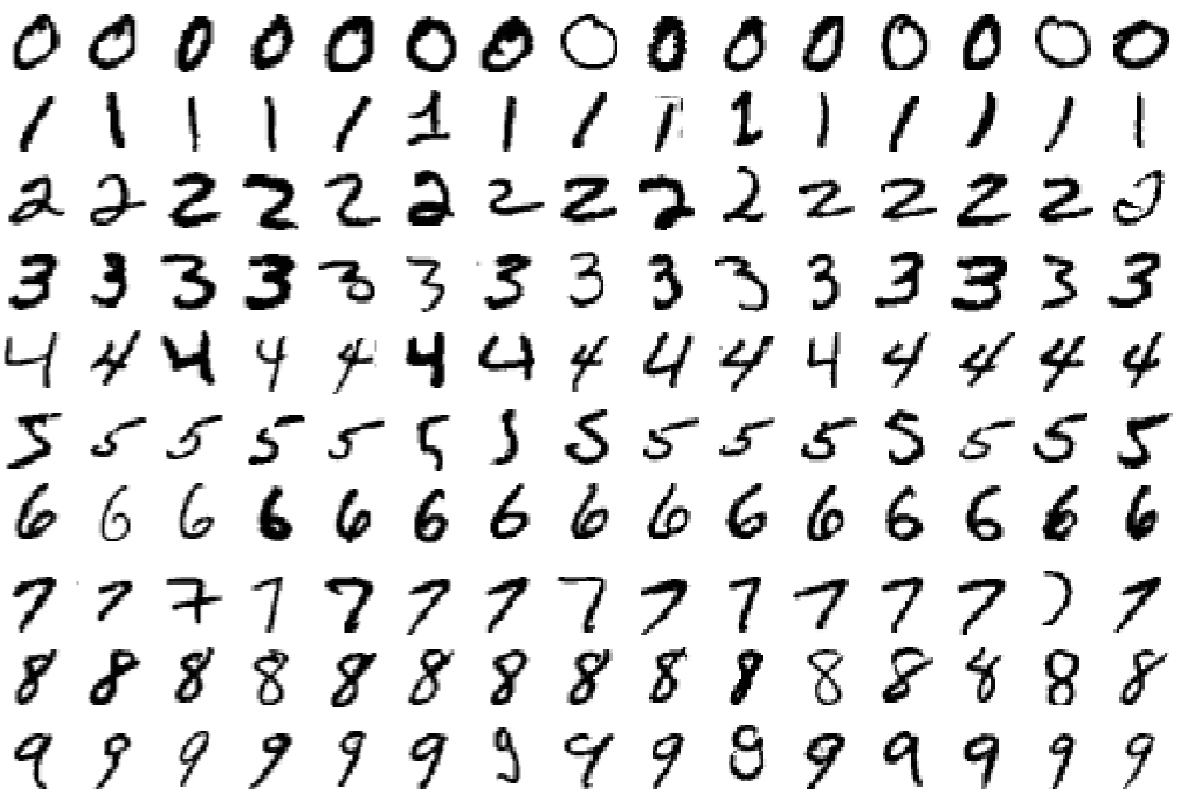
```python
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)

# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)

# loss
loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
```

Linear Regression

```
# optimizer
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)

# training data
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]

# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init) # reset values to wrong
for i in range(1000):
    sess.run(train, {x:x_train, y:y_train})
    # evaluate training accuracy
    curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
    print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

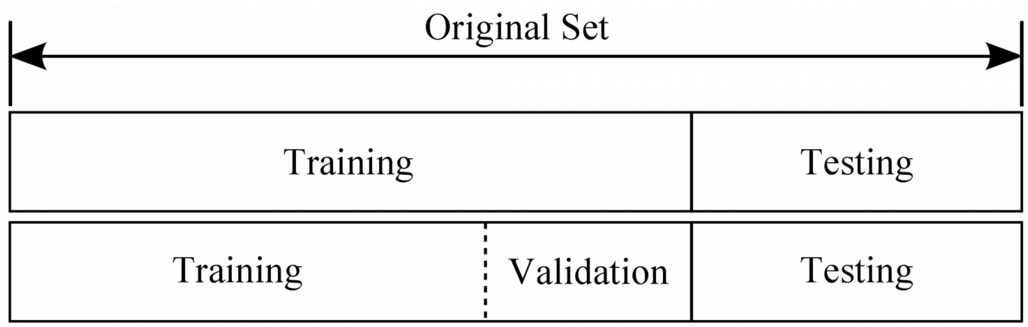
MNIST

- Classification of hand-written digits (0-9) from 28x28 pixel greyscale images (MNIST data set).
- Full data set of 70k examples: http://yann.lecun.com/exdb/mnist



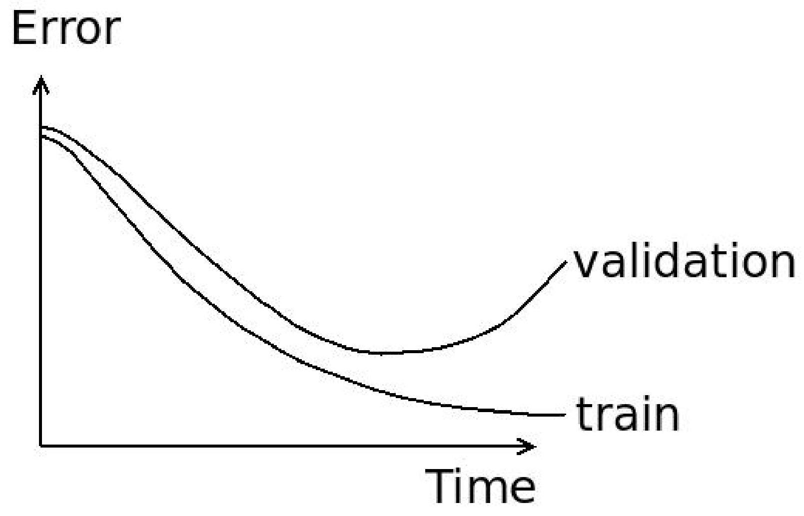
MNIST

- As common in machine learning, the MNIST data is split into three parts:
  - Training: 55,000 images
  - Test: 10,000 images
  - Validation: 5,000 images.
  - Dataset contains pair of images and labels.
  - Useful to test hyper parameters and generalization performance



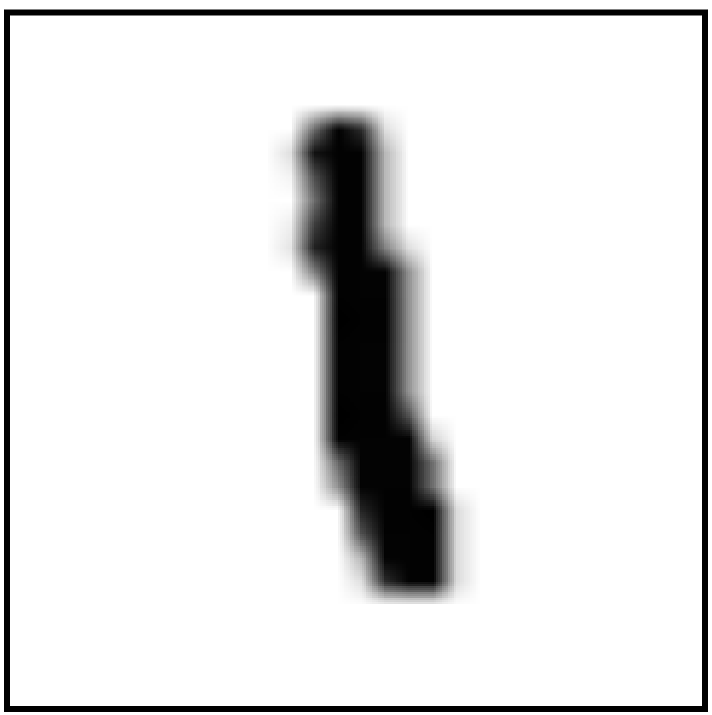
MNIST

- As common in machine learning, the MNIST data is split into three parts:
  - Training: 55,000 images
  - Test: 10,000 images
  - Validation: 5,000 images.
  - Dataset contains pair of images and labels.
  - Useful to test hyper parameters and generalization performance



MNIST

- Each image is 28 pixels by 28 pixels.
  - We can flatten this array into a vector of 28x28 = 784 numbers.
  - Vector representation but loosing structure.



$$\simeq \begin{bmatrix}
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&.6&.8&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.5&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.7&0&0&0&0&0\\
0&0&0&0&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&.9&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&.3&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}$$

Import data

- Download and read the data automatically:

MNIST_softmax.py ×

No Python interpreter configured for the project

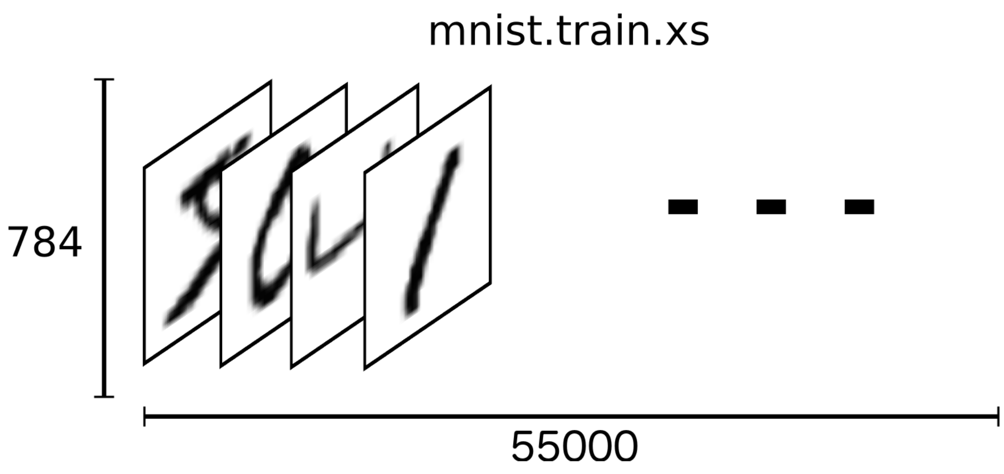
```python
# limitations under the License.
# ==============================================================================

"""A very simple MNIST classifier.

See extensive documentation at
http://tensorflow.org/tutorials/mnist/beginners/index.md
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import sys

from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf

FLAGS = None


def main(_):
  # Import data
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

  # Create the model
  x = tf.placeholder(tf.float32, [None, 784])
  W = tf.Variable(tf.zeros([784, 10]))
  b = tf.Variable(tf.zeros([10]))
  y = tf.matmul(x, W) + b
```

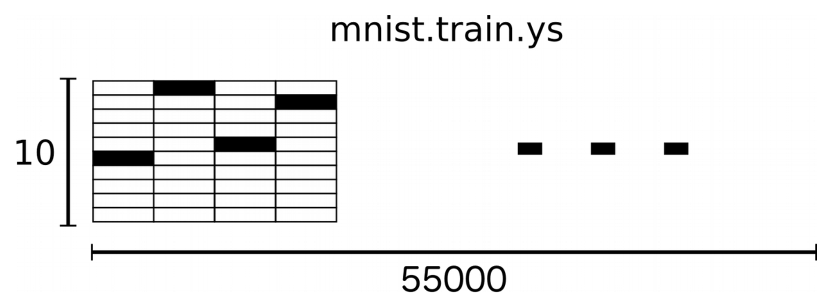
Import data

- We get:

  `mnist.train.images`: tensor with a shape of [55000, 784]

  `mnist.train.labels:` a [55000, 10] array of floats – vector notation for class labels.





NN training

- Several things to decide (data, hyperparameters):
  - Training data
    - Representation (vectors, images, text).
    - Normalization
  - Architecture
    - Layers: type, shape, number.
    - Activation functions
    - Output type (according to task, e.g. classification/regression) and loss function.
  - Learning algorithm
    - Initialization.
    - Update scheme.
    - Learning rate.
    - Momentum.
    - Regularization (weight decay, dropout).
    - Batch normalization
    - Stopping criteria

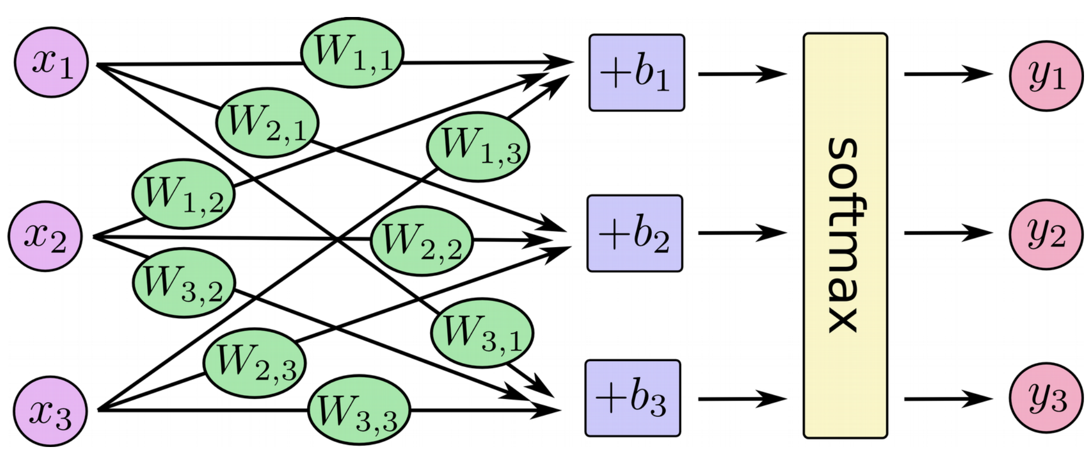
Softmax regression

- Recap: softmax regression to output probabilities
- Two steps: add up the evidence of our input being in certain classes and then convert evidences into probabilities.

$$\text{evidence}_i = \sum_j W_{i,\,j} x_j + b_i$$

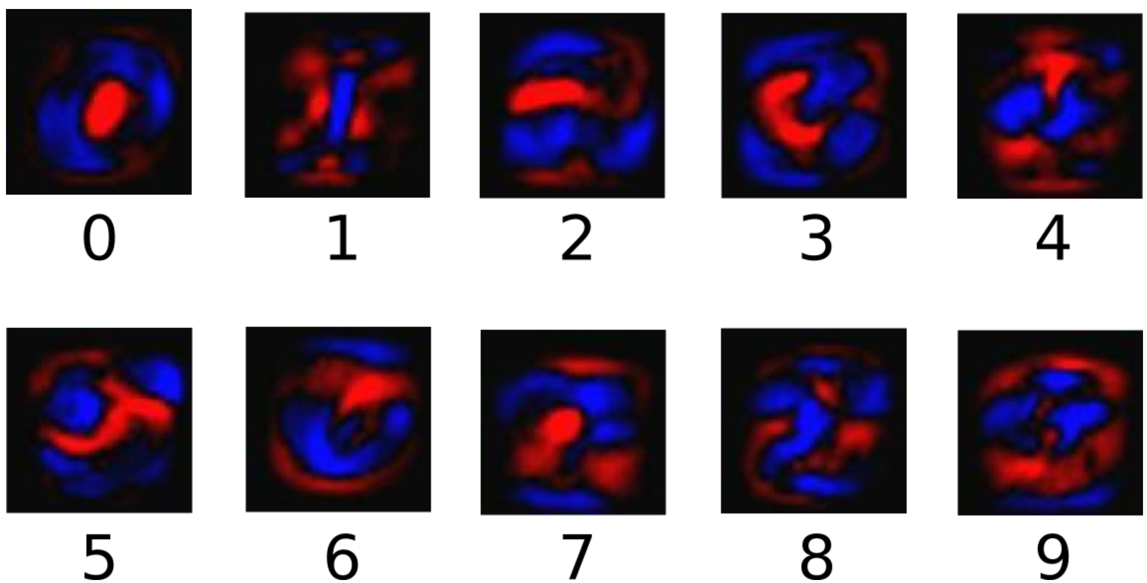
$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

$$y = \text{softmax}(\text{evidence})$$



Softmax regression

- Output: As we do a weighted sum of the pixel intensities we can inspect them.
- Red: negative weights.
- Blue: positive weights.



Softmax regression

- Matrix Notation

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

MNIST

- We use variables and placeholders to create the model:
  - Look at the dimensionality
  - What is missing?

```
"""
import ...

FLAGS = None


def main(_):
  # Import data
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

  # Create the model
  x = tf.placeholder(tf.float32, [None, 784])
  W = tf.Variable(tf.zeros([784, 10]))
  b = tf.Variable(tf.zeros([10]))
  y = tf.matmul(x, W) + b
```

MNIST

- Model training:
  - Use cross-entropy $H_{y'}(y) = -\sum_i y'_i \log(y_i)$
  - Optimize with gradient descent with a learning rate 0.5.
  - Many other optimizers (link)

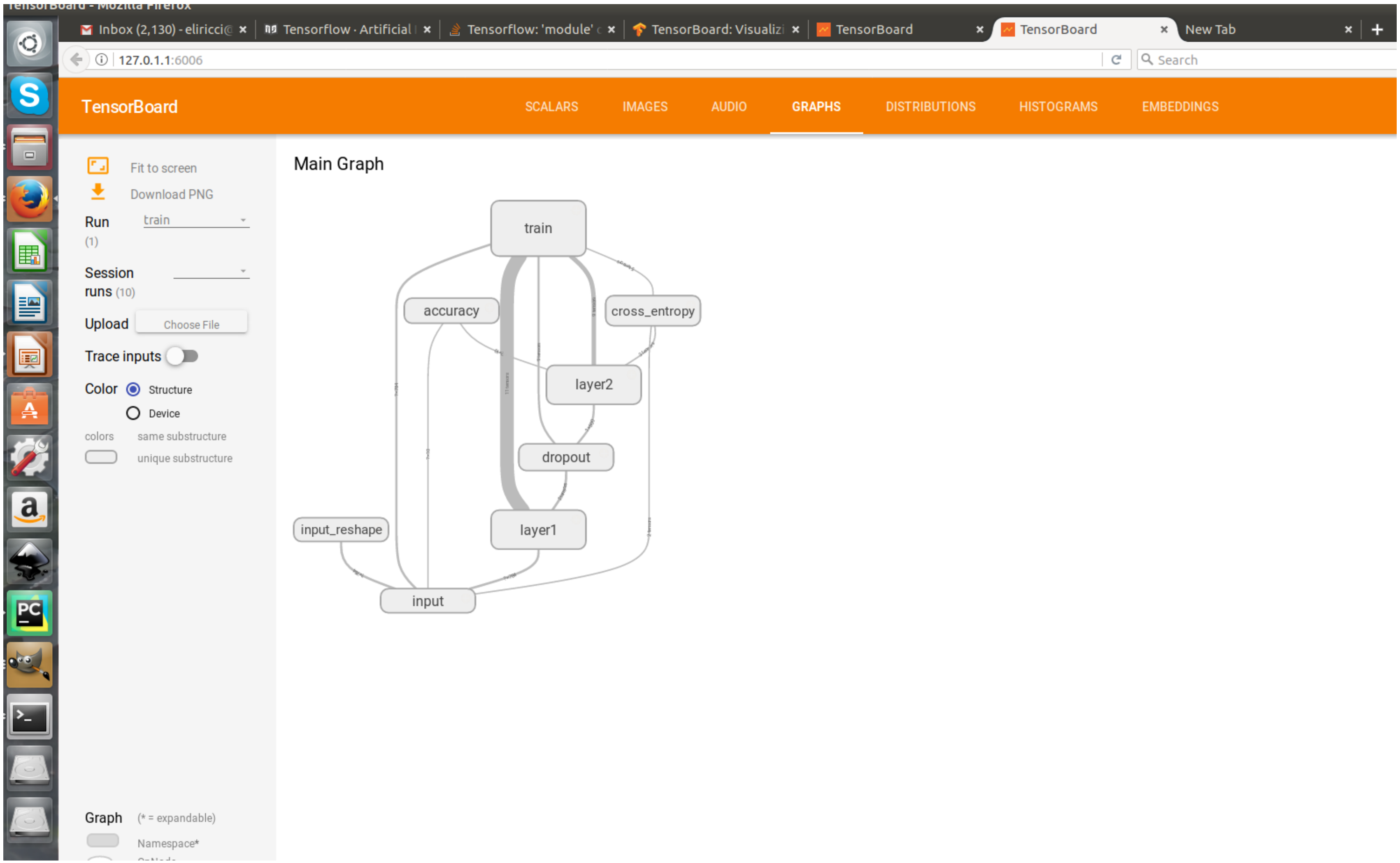
```
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])

# The raw formulation of cross-entropy,
#
#   tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(tf.nn.softmax(y)),
#                                 reduction_indices=[1]))
#
# can be numerically unstable.
#
# So here we use tf.nn.softmax_cross_entropy_with_logits on the raw
# outputs of 'y', and then average across the batch.
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

MNIST

- Run the session
  - Training considering mini-batches
  - Evaluate performance (are they good?)

```
  sess = tf.InteractiveSession()
  tf.global_variables_initializer().run()
  # Train
  for _ in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

  # Test trained model
  correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
  print(sess.run(accuracy, feed_dict={x: mnist.test.images,
                                      y_: mnist.test.labels}))

if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  parser.add_argument('--data_dir', type=str, default='/tmp/tensorflow/mnist/input_data',
                      help='Directory for storing input data')
  FLAGS, unparsed = parser.parse_known_args()
  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

TensorBoard



TensorBoard

- Training a massive deep neural network can be complex and confusing.
- TensorBoard: visualization tools to facilitate models understanding and debug.
- Visualize graph, plot quantitative metrics about the execution of the graph, show additional data like images used, visualize statistics.

TensorBoard

- Modify code to generate summary data.

  (1) Create graph and decide which nodes you would like to collect summary data.

  Example MNIST:

  - Monitor learning rate and loss.
  - Use `tf.summary.scalar` for to the nodes that output the learning rate and loss respectively.

TensorBoard

- Modify code to generate summary data.

  (1) Create graph and decide which nodes you would like to collect summary data.

  Example MNIST:

  - Visualize the distributions of activations coming off a particular layer, or the distribution of gradients or weights.
  - Use *tf.summary.histogram*.

TensorBoard

- Modify code to generate summary data.

  (1) Create graph and decide which nodes you would like to collect summary data.

  The summary nodes are peripheral nodes added to the graph: none of the ops we are currently running depend on them.

TensorBoard

- Modify code to generate summary data.

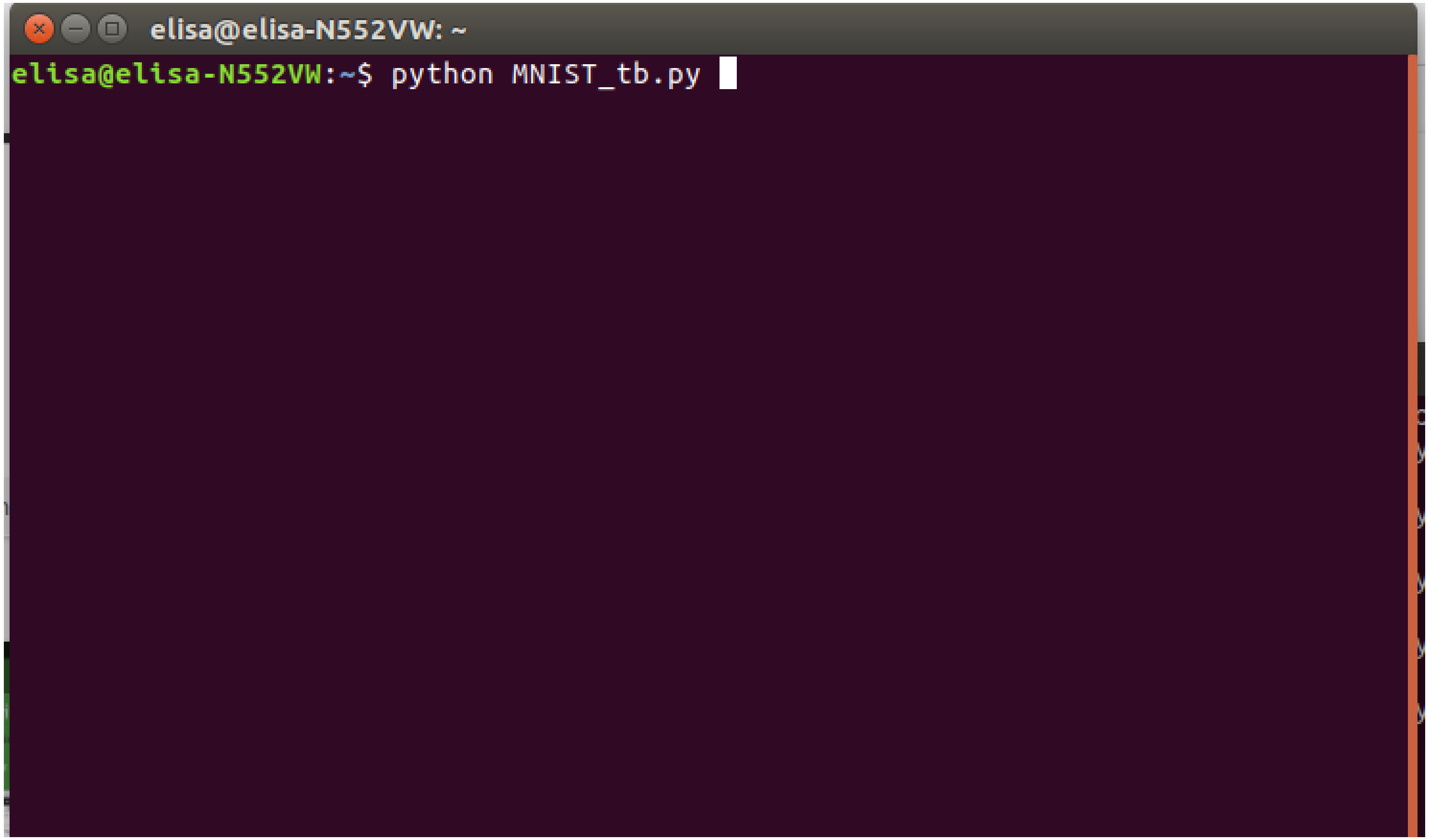
  (2) To generate summaries, run all of the summary nodes.

  (2a) Use `tf.summary.merge_all` to combine them.

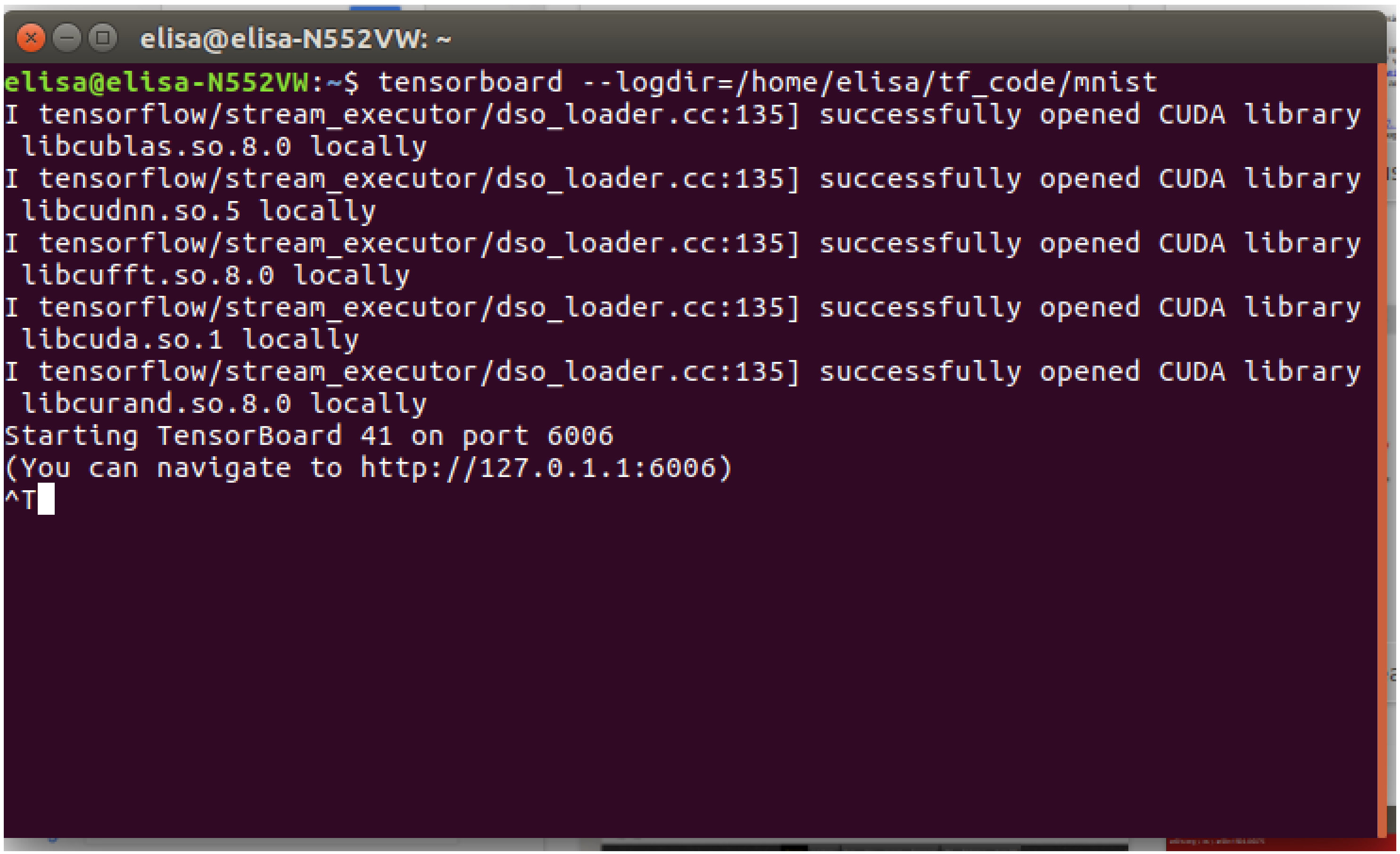
  (2b) Run the merged summary op, which will generate a serialized Summary protobuf object with all of your summary data at a given step.

  (5) Write summary data to disk, pass the summary protobuf to a `tf.summary.FileWriter`.

TensorBoard

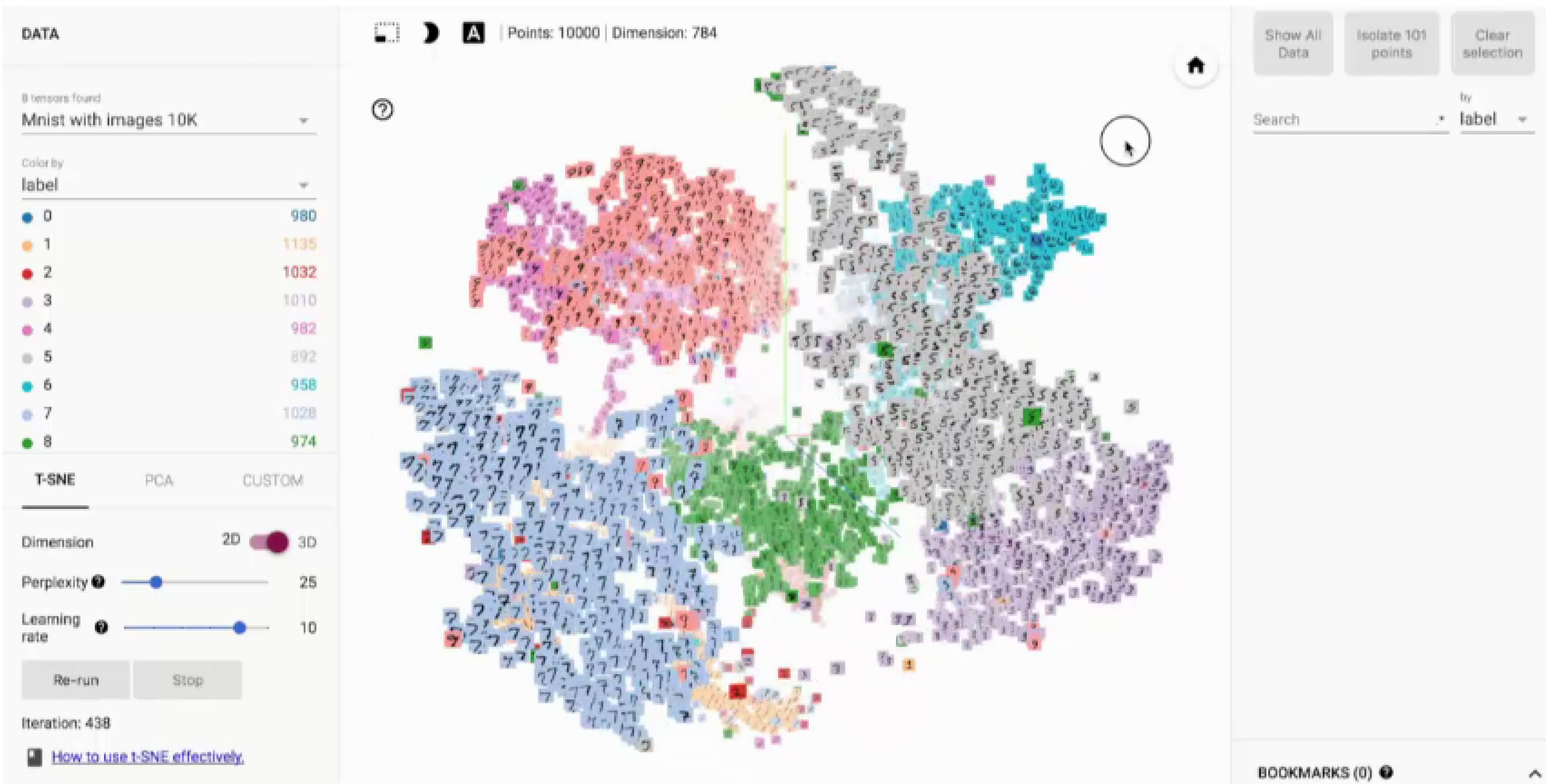


TensorBoard



TensorBoard

- Other features: Embedding visualization.



Keras

Keras

- Keras (κέρας) means *horn* in Greek.
- In the Odyssey it is mentioned that dream spirits are divided between:
  - those who deceive men with false visions, who arrive to Earth through a gate of ivory
  - those who announce a future that will come to pass, who arrive through a gate of horn.

Keras

- Easy-to-use Python library
- Why Python? Easy to learn, powerful libraries (scikit-learn, matplotlib...)
- It wraps Theano and TensorFlow (it benefits from the advantages of both)
- Guiding principles: modularity, minimalism, extensibility.

Keras

- Use both GPU and CPUs
- Easy to use both convolutional networks and recurrent networks and combinations of the two.
- Supports arbitrary connectivity schemes (including multi-input and multi-output training)
- Many easy-to-use tools: real-time data augmentation, callbacks (Tensorboard visualization)

Keras

- Keras gained official Google support

fast.ai

Making neural nets uncool again

Home
About
Our MOOC

© fast.ai 2017. All rights reserved.

Big deep learning news: Google Tensorflow chooses Keras

03 Jan 2017 *Rachel Thomas*

Buried in a Reddit comment, Francois Chollet, author of Keras and AI researcher at Google, made an exciting announcement: Keras will be the first high-level library added to core TensorFlow at Google, which will effectively make it TensorFlow's default API. This is excellent news for a number of reasons!

As background, Keras is a high-level Python neural networks library that runs on top of either TensorFlow or Theano. There are other high level Python neural networks libraries that can be used on top of TensorFlow, such as TF-Slim, although these are less developed and not part of core TensorFlow.

Using TensorFlow makes me feel like I'm not smart enough to use TensorFlow; whereas using Keras makes me feel like neural networks are easier than I realized. This is because TensorFlow's API is verbose and confusing, and because Keras has the most thoughtfully designed, expressive API I've ever experienced. I was too embarrassed to publicly criticize TensorFlow after my first few frustrating interactions with it. It felt so clunky and unnatural, but surely this was my failing. However, Keras and Theano confirm my suspicions that tensors and neural networks don't have to be so painful. (In addition, in part 2 of our deep learning course Jeremy will be showing some tricks to make it easier to write custom code in Tensorflow.)

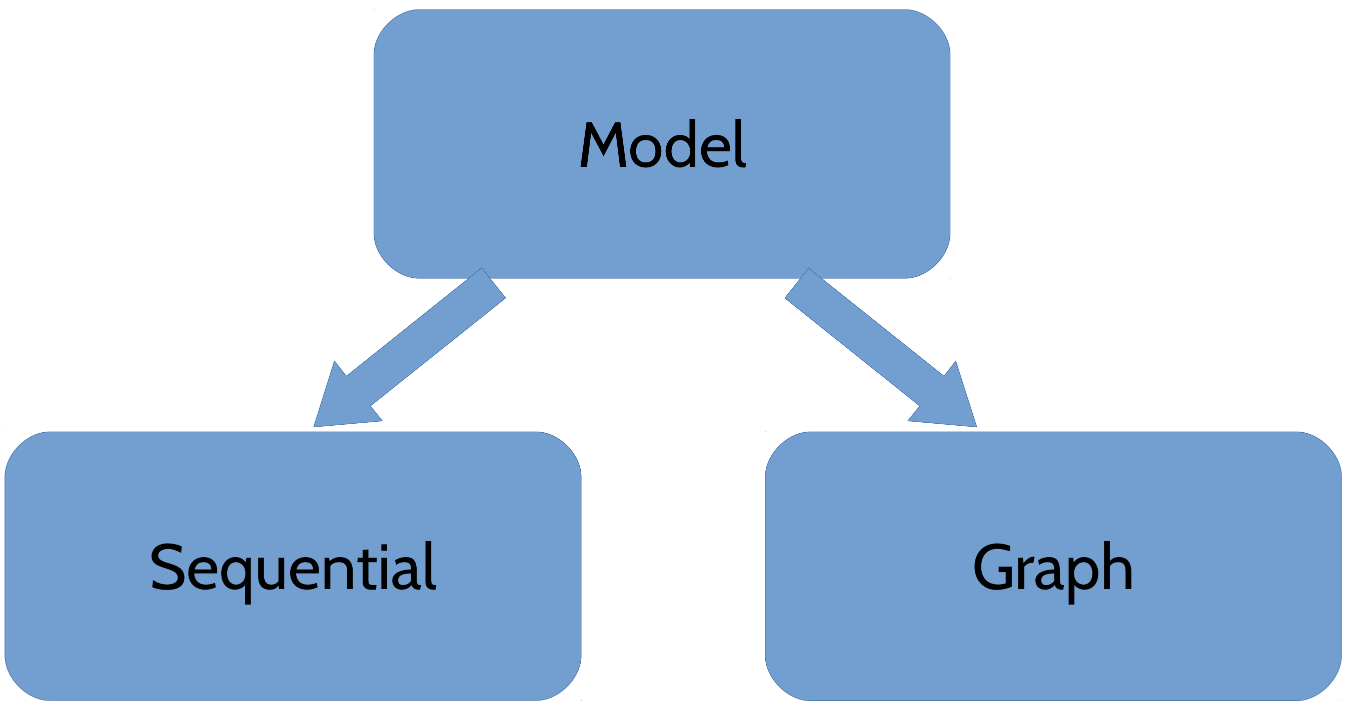
For a college assignment, I once used a hardware description language to code division by adding and shifting bits in the CPU's registers. It was an interesting exercise, but I certainly wouldn't want to code a neural network this way. There are a number of advantages to using a higher level language: quicker coding, fewer bugs, and less pain. The benefits of Keras go beyond this: it is so well-suited to the concepts of neural networks, that Keras has improved how Jeremy and I think about neural networks and facilitated new discoveries. Keras makes me better at neural networks, because the language abstractions match up so well with neural network concepts.

Keras

- Weaknesses:
  - Less flexible
  - Some stuff not there yet (no RBM for example)
  - Less projects available online (e.g. with respect to Caffe)

Model

- A model is a ***sequence*** or a ***graph*** of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible.



Modularity

- A model is a ***sequence*** or a ***graph*** of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible.
- Modules:
  - neural layers
  - cost functions
  - optimizers
  - initialization schemes
  - activation functions
  - regularization schemes
  - ***your own module***

Keras

- Extensibility: modules are easy to add.
- Simplicity: modules should be made extremely simple.

TensorFlow:

```
kernel = tf.Variable(tf.truncated_normal([3,3,64,64],type=tf.float32,stddev=1e-1), name='weights')
conv = tf.nn.conv2d(self.conv1_1, kernel, [1, 1, 1, 1], padding='SAME')
biases = tf.Variable(tf.constant(0.0, shape=[64], dtype=tf.float32), trainable=True, name='biases')
out = tf.nn.bias_add(conv, biases)
self.conv1_2 = tf.nn.relu(out, name='block1_conv2')
```

Keras:

```
x = Convolution2D(64, 3, 3, activation='relu', border_mode='same', name='block1_conv2')(x)
```

Install Keras

- Extremely easy:

```
>> source tensorflow/bin/activate
>> python
>> pip install keras
>> import keras as k
```

Sequential model

- Sequential models are linear stack of layers
- Treat each layer as object that feeds the next layer

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
```

Graph model

- Useful to create two or more independent networks to diverge or merge
- Useful to create multiple separate inputs or outputs
- Different merging layers (sum or concatenate)

```
model =  Graph()
# Load the input
model.add_input(name='input1', ndim=4)
# Convolution Neural Network architecture (5 convolution layers, 3 pooling layers)

model.add_node(Convolution2D(nb_filters[0], image_dimensions, nb_conv[0], nb_conv[0], activation='relu', border_mode='full'), name='conv2', input='input1')
model.add_node(Convolution2D(nb_filters[0], nb_filters[0], nb_conv[0], nb_conv[0], activation='relu', border_mode='full'), name='conv3', input='conv2')
model.add_node(MaxPooling2D(poolsize=(nb_pool[0], nb_pool[0])), name='pool1', input='conv3')

model.add_node(Convolution2D(nb_filters[1], nb_filters[0], nb_conv[0], nb_conv[0], activation='relu', border_mode='full'), name='conv4', input='pool1')
model.add_node(Convolution2D(nb_filters[1], nb_filters[1], nb_conv[1], nb_conv[1], activation='relu', border_mode='full'), name='conv5', input='conv4')
model.add_node(MaxPooling2D(poolsize=(nb_pool[1], nb_pool[1])), name='pool2', input='conv5')

model.add_node(Flatten(), name='flatten', input='pool2')

model.add_node(Dense(nb_filters[-1] * (shapex / nb_pool[0] / nb_pool[1]) * (shapey / nb_pool[0] / nb_pool[1]), 512, activation='relu', init='uniform'),  name
model.add_node(Dense(512, nb_classes, activation='softmax', init='uniform'), name='dense2', input='dense1')

model.add_output(name='output1', input='dense2', merge_mode='sum')
model.compile('sgd', {'output1':'categorical_crossentropy'})
model.get_config(verbose=1)

model.fit({'input1':X_train, 'output1':Y_train},batch_size=batch_size, nb_epoch=nb_epoch)

#model.predict({'input1':X_test})

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
```

Let's run MNIST again

- Homepage

  https://keras.io/

  https://keras.io/getting-started/sequential-model-guide/#getting-started-with-the-keras-sequential-model

- Examples:

  https://github.com/fchollet/keras/tree/master/examples

- Let's compare a MLP and a CNN…

Questions?