# TensorFlow

AN OPEN-SOURCE SOFTWARE LIBRARY FOR MACHINE INTELLIGENCE

# What is tensorflow?

- Open-source library by Google for **numerical computation** (not just deep learning!)
- Build a **computational graph** with the mathematical operations you want to perform
- Core **backend** developed in **C++** for efficiency, supports multiple CPUs, multiple GPUs with use of **CUDA** for GPU acceleration
- Interface with the backend using APIs for **Python**, Java, Go, …
- Multiplatform: Linux, MacOS, Windows, Android, iOS
- Not the only exisiting one: Torch (Facebook), Theano (U. of Montreal), …


# What is a tensor?

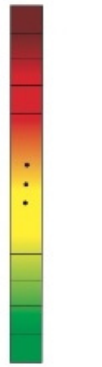
**Tensor = multidimensional array**

- 1D tensor: vector
- 2D tensor: matrix
- 3D tensor : RGB image

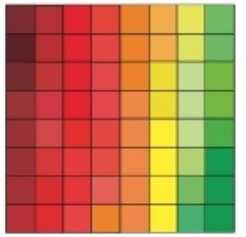
Example: a sequence of images as a 4D tensor

( 100, 256, 256, 3 )

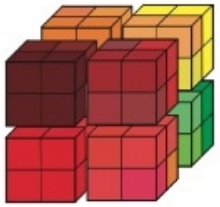
- Number of images
- Number of rows
- Number of columns
- Number of color channels



$\mathbf{v} \in \mathbb{R}^{64}$



$X \in \mathbb{R}^{8x8}$



$\mathcal{X} \in \mathbb{R}^{4x4x4}$

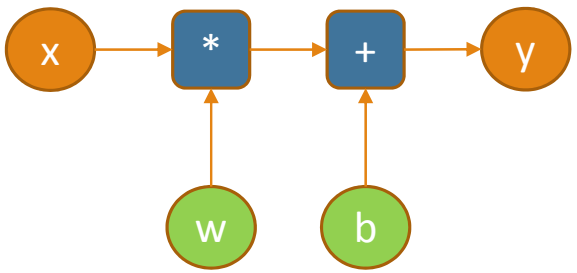
# What is a computational graph?

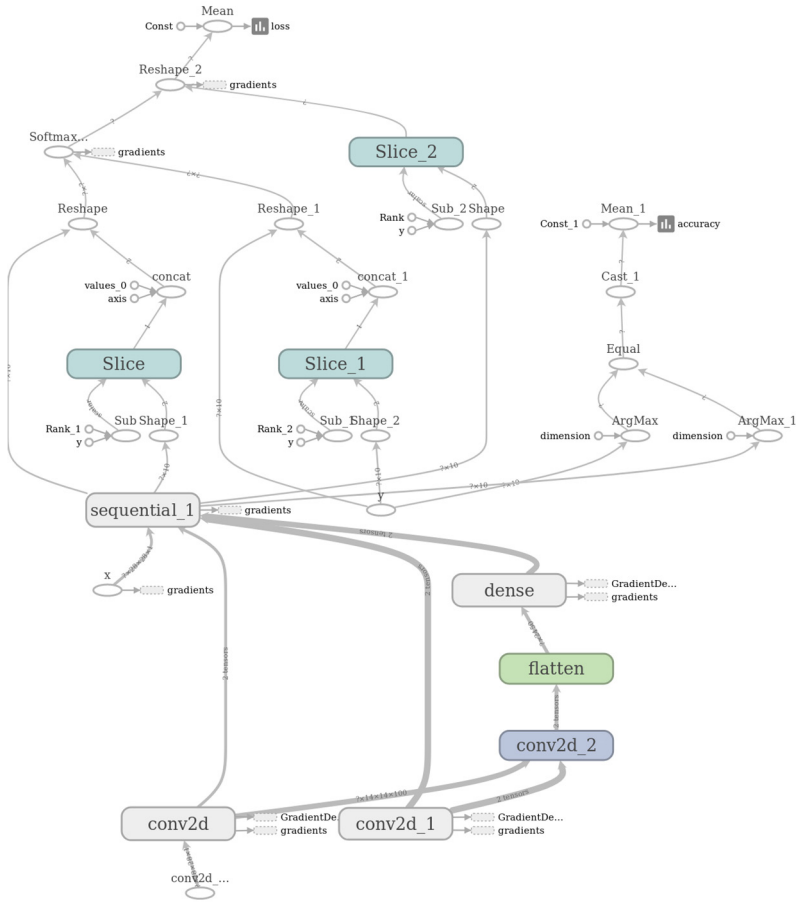
**Computational graph**

**=**

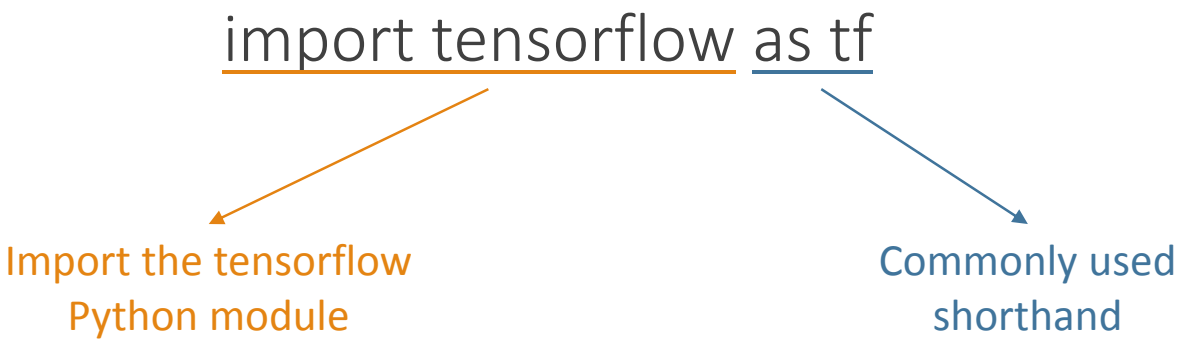
**sequence of mathematical operations connected to each other as a graph of nodes**

Example: $y = w * x + b$



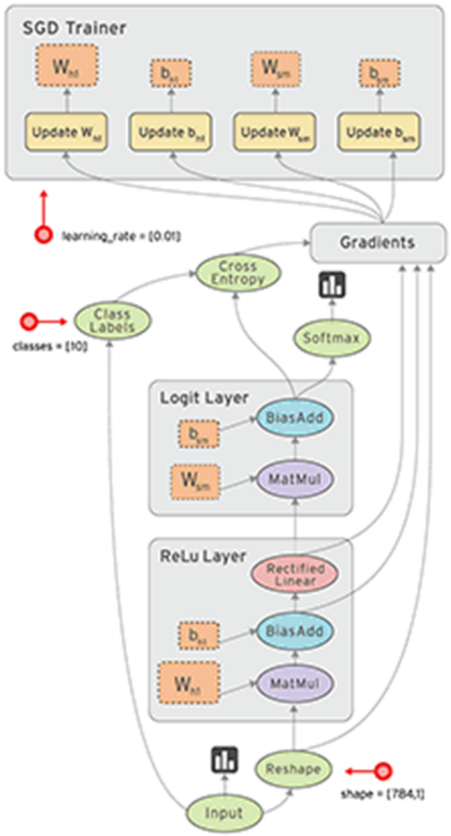



# Guide to the Python API



# Guide to the Python API

1. Define a computational **graph**
2. Create a **Session** to run parts of the graph with some input data



# Types of nodes

- **Constant**
  - A tensor with constant value

```
c = tf.constant(4.0, dtype=tf.float32)
```

value

type:

tf.float32 (default)
tf.int8
tf.uint8
tf.int16
tf.int32
tf.bool
tf.string
…

*TF documentation: class tf.DType*

# Types of nodes

- **Placeholder**
  - Placeholders are used to *provide inputs*, i.e. the placeholder will be filled in with some value provided at runtime.
    Example: define a placeholder that at runtime will hold a batch of images to be processed

```
x = tf.placeholder(tf.float32, shape=(None, 256, 256, 3), name='input_images')
```

Type

Tensor size:
*(batch_size, num_rows, num_columns, num_channels)*
**None**: this dimension can be of any length

Useful reminder of what this placeholder should contain

**Tip**: defining the dimension of the placeholder corresponding to the number of samples in the batch as *None* allows you to run the same code for any batch size without modifications

# Types of nodes

- **Variable**
  - Variables are used to define the *trainable tensors* of the graph, e.g. the layer weights in a neural net.

```
W = tf.Variable(tf.truncated_normal(shape=(100,10), sdtdev=0.1), name='trainable_weights')
```

Initial value

tf.truncated_normal: random Gaussian (without extreme values) tensor

tf.constant: constant tensor

tf.zeros: constant tensor filled with 0s

tf.ones: constant tensor filled with 1s

…

*TF Documentation:*
*https://www.tensorflow.org/api_guides/python/constant_op*

Useful reminder of what this variable is

# Mathematical operations

- Tensorflow provides functions to compute most mathematical operations. These functions call efficient implementations in the backend
- For these functions Tensorflow knows both the
  - **forward** behavior: from inputs and internal parameters compute output
  - **backward** behavior: compute derivative of loss with respect to input and internal parameters

  This allows optimizers to apply the **chain rule** to compute derivatives across all the operations in the graph

```
h = tf.matmul(x, W) + b # tensor product and sum x*W+b
h = x*W + b # same as before (*,+) are recognized by tf
```

# A few useful operations

```
out = tf.nn.relu(input) # ReLU activation

out = tf.nn.sigmoid(input) #  sigmoid activation

out = tf.nn.conv2d(x, W, strides=(1,1,1,1),padding='SAME') # 2D convolution

out = tf.nn.max_pool(input, ksize=(1,2,2,1), strides=(1,2,2,1), padding='SAME') # Max pooling

out = tf.nn.dropout(input, keep_prob) # Dropout, keep_prob is a placeholder

out = tf.equal(a, b) # Returns the boolean truth value of (x == y) element-wise

out = tf.cast(input, tf.float32) # cast input tensor to specified type

out = tf.argmax(input, axis=1) # Returns the index with the largest value across the specified dimension (axis) of a tensor

out = tf.reduce_mean(input, axis=0) # Computes the mean of elements across the specified dimension of a tensor. e.g.: if
input has shape (100,10), axis=0, output has shape (1,10)
```

https://www.tensorflow.org/api_guides/python/ for many more

# Note on multinomial logistic regression

- Multinomial logistic regression = logistic regression for **multiclass classification** problems
- Last layer of network must have
  - **Softmax** activation $\mathrm{s_i} = softmax(h_i) = \frac{e^{h_i}}{\sum_j e^{h_j}}$
  - **Cross-entropy** loss $D(\boldsymbol{s}, \boldsymbol{y}) = -\sum_i y_i \log(s_i)$

Tensorflow allows you to do this but THIS IS NOT NUMERICALLY STABLE and will work poorly

```
s = tf.nn.softmax(h) # softmax
loss = - tf.reduce_sum(y * tf.log(s), 1) # cross entropy
```

**DO NOT DO THIS!**

Instead, the softmax and cross-entropy are combined in a single **numerically-stable** operation:

```
loss = tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=h)  # numerically stable
```

# Optimization

- Now that you defined all the computations and the loss function, you must specify if you want to **optimize some variable** and **how**
- The **train API** provides optimizers to minimize loss functions with respect to Variables

```
optimizer = tf.train.GradientDescentOptimizer(0.5) # 0.5 is the learning rate
train_step = optimizer.minimize(loss)
```

*train_step* is a new node in the graph, executing it runs **one iteration** of gradient descent

Optionally:
*optimizer.minimize(loss, var_list=…)*
Provide a list of variable instead of optimizing over all of them

- tf.train has several functions such as more advanced optimizers (tf.train.AdamOptimizer, …) and options such as learning rate decay

# Sessions – Creating them

- You need a **Session** to run a graph or parts of it

Connection to
the C++ backend

```
sess = tf.InteractiveSession()
```

- *tf.InteractiveSession()* is the same as *tf.Session()* except it is less annoying because it automatically sets itself as default session

# Sessions – Initialize variables

- Before training anything make sure your **variables are initialized** by calling

```
tf.global_variables_initializer().run()
```

- This operation will initialize all the variables using the **initialization functions or tensors** you specified when you declared them with tf.Variable

# Sessions – Run

- Once you have a Session (and have initialized the Variables if you want to train them), you can **run parts of the graph**

```
sess.run(train_step, feed_dict={x: batch_xs, y: batch_ys})
```

feed_dict is a **Python dictionary** (syntax is *key: value*) that takes tensors that will be put inside **placeholders**.
The key is the placeholder name.
The value is a tensor (this is an input, not part of the graph, it has numbers in it)

This is the **node of the graph** we want to run: in this case is a training operation of the optimizer that will **run one iteration** of SGD

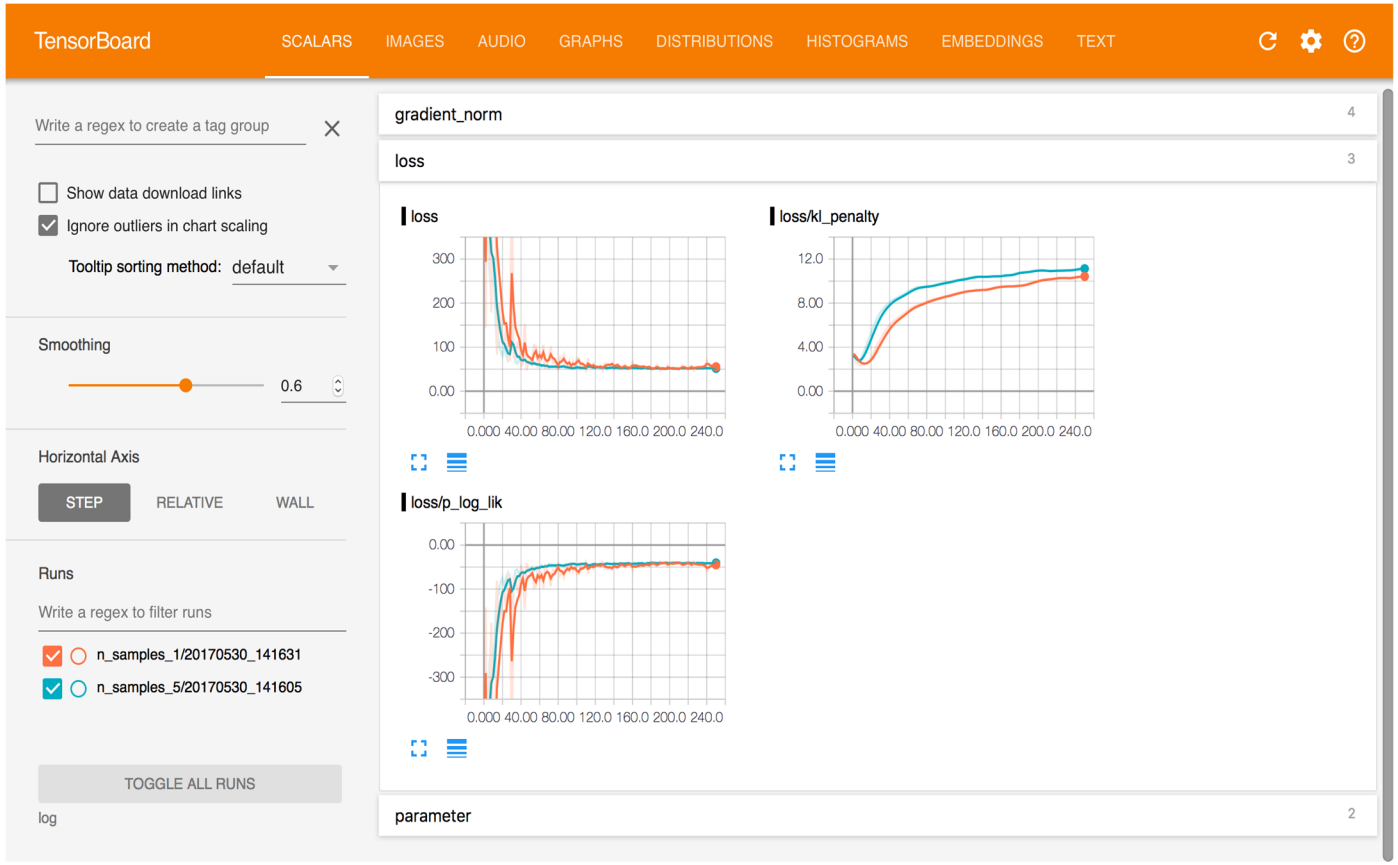
# Sessions – Run

- You can **run any node** in the graph to get its **value** (it will perform a forward pass from the input to node you specified)
- Suppose you want to compute an accuracy value of your method

```
accuracy = ... # define the operation to compute accuracy as node in the graph
accuracy_value = sess.run(accuracy, feed_dict={x: batch_xs, y: batch_ys}) # run the operation to get the value
```

# Tensorboard

- Tensorboard is a **graphical interface** to visualize tensor values, statistics, the computational graph , …
- **Very useful for debugging**: keep track of the loss function during a long training, visualize a histogram of weight values
- You must **write instructions** in your program to **save** information to an **Event File** that is read by Tensorboard for visualization



# Tensorboard - Summaries

- When you have a **tensor** you want to keep track of, you attach a **tf.summary** to it
- There are different types of summaries depending on the tensor and visualization:

```
tf.summary.scalar('loss', loss) # a scalar, e.g. loss function or accuracy
tf.summary.histogram('weights_layer1', W1) # a histogram of all the values in the tensor, e.g. check that your weights are not all 0s!
tf.summary.image('input_images', x) # visualize a batch of images (batch_size, Nrows, Ncols, Nchannels)
```

- They are all nodes in the graph; once you defined all the **summary nodes** you want, you need to **run them** to get the actual values
- **Merge all the summaries** into one object for simplicity:

```
merged = tf.summary.merge_all()
```

# Tensorboard - Event File

- All the data in summaries must be written into a file that is read by the Tensorboard program
- To do that we use **FileWriter** that takes the **merged summaries** and **serializes them to a file**

1. Instantiate the FileWriter object

```
train_writer = tf.summary.FileWriter(FLAGS.log_dir + '/train', sess.graph)
```

Logging directory where to write the file

Notice you need a session

2. Run the merged summaries to get the values

```
summary_train, _ = sess.run([merged, train_step], feed_dict={x: batch_xs, y: batch_ys}) # do one step of training and compute summaries
```

3. Write summaries

```
train_writer.add_summary(summary_train, i) # i is the iteration counter
```

# Keras

- **Keras** is a high-level API written in Python to code neural networks more easily
- It runs **on top of Tensorflow** (or Theano)
- Keras can be used independently (without ever writing Tensorflow code) or can be integrated into the native Tensorflow flow

**Independent Keras**
Its own model compiling, optimization, …
Agnostic to backend (can run on Theano without code modifications)

**tf.contrib.keras**
Use tensorflow optimizers, Tensorboard, …
WE WILL USE THIS

Note: from tensorflow 1.4, Keras is no longer a contributor package but officially supported as **tf.keras**

# Keras - Models

- Everything works as in pure Tensorflow (placeholders, sessions, tensorboard)
- Except there are high-level definitons for many common neural networks layers so that you don’t have to worry about variables
- A **Keras model is a sequence of layers**
  - **Sequential model:** linear stack of layers
  - **Functional API:** allows to defined arbirarly complex arrangements of layers

# Keras – Sequential model

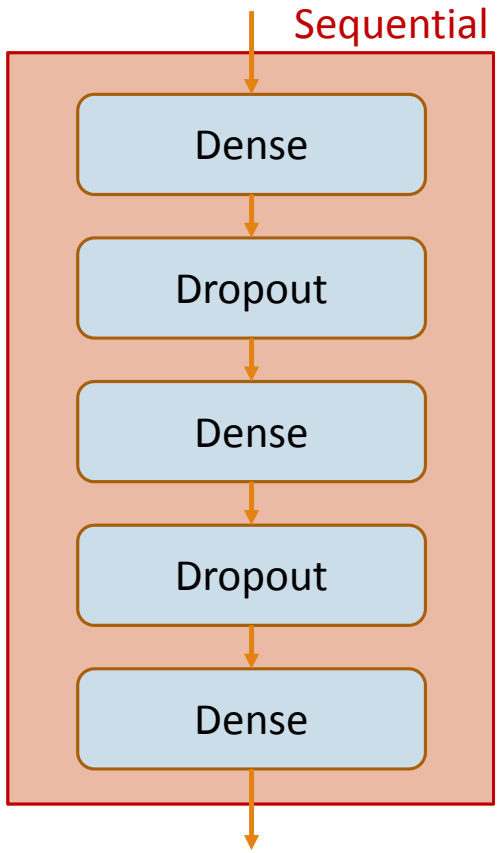
- **Linear stack of layers**: the output of a layer is the input of the next layer

```
def my_model():

  model = models.Sequential()
  model.add(layers.Dense(64, activation='relu', input_dim=20))
  model.add(layers.Dropout(0.5))
  model.add(layers.Dense(64, activation='relu'))
  model.add(layers.Dropout(0.5))
  model.add(layers.Dense(10))

  return model
```

Create a Sequential model

Add layers with .add()

First layer only: must specify input dimension (without batch size)



# Keras – Layers, Initializers, Regularizers

- Visit https://keras.io/layers/about-keras-layers/ for a comprehensive list of layers. Examples:

```
layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

layers.Flatten()
```

- For the initializer parameters you can use the Keras initializer functions:

```
initializers.Zeros()
initializers.Ones()
initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None)
initializers.glorot_normal(seed=None)
```

- For the regularizers you can use:

```
regularizers.l1(0.01)
regularizers.l2(0.01)
regularizers.l1_l2(0.01)
```

# Keras – Using a model

- Once you create your model you can use it as a function on any tensor in the graph

```
my_net = my_model()
h = my_net(x) # x is a placeholder
```

- Keras models have extra methods and attributes. For example:

```
my_net.summary()
```

will print a summary of the model with
all the tensor sizes and
number of trainable parameters

| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense_1 (Dense) | (None, 64) | 1344 |
| dropout_1 (Dropout) | (None, 64) | 0 |
| dense_2 (Dense) | (None, 64) | 4160 |
| dropout_2 (Dropout) | (None, 64) | 0 |
| dense_3 (Dense) | (None, 10) | 650 |

Total params: 6,154
Trainable params: 6,154
Non-trainable params: 0