# MML minor #9

Введение в TensorFlow

# TensorFlow DL framework

- We will use it in Jupyter Notebook with Python 3 kernel

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

- We will overview Python API for TensorFlow 1.2+

- APIs in other languages exist: Java, C++, Go
  - Python API is at present the most complete and the easiest to use
  - https://www.tensorflow.org/api_docs/

# What is TensorFlow?

1. A tool to describe computational graphs
   - The foundation of computation in TensorFlow is the **Graph** object. This holds a network of nodes, each representing one **operation**, connected to each other as inputs and outputs.

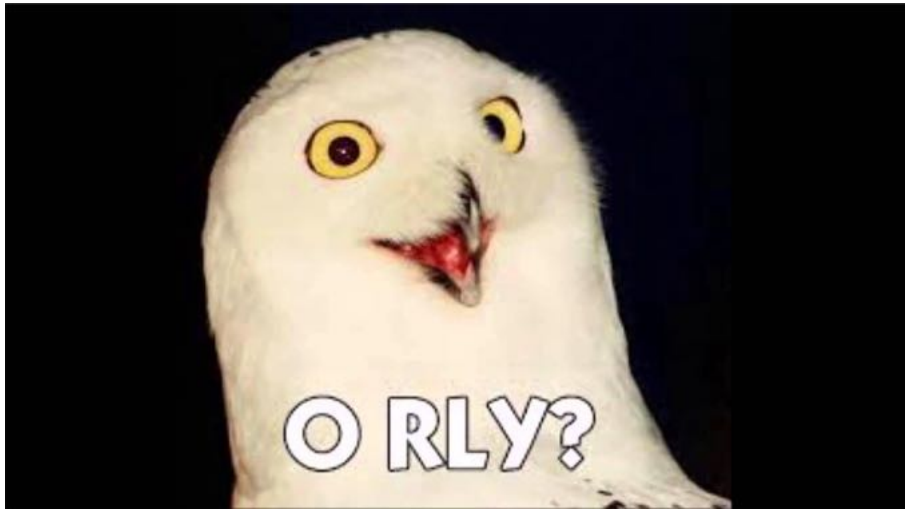
2. A runtime for execution of these graphs
   - On CPU, GPU, TPU, …
   - On one node or in distributed mode

# Why this name

- **Input** to any operation will be a collection of **tensors** (multi-dimensional arrays)

- **Output** will be a collection of tensors as well.

- We will have a graph of operations, each of which transforms tensors into another tensors, so it’s a kind of a flow of tensors



# How the input looks like

- **Placeholder**
  - This is placeholder for a tensor, which will be fed during graph execution (e.g. input features)
  - x = tf.placeholder(tf.float32, (None, 10))
- **Variable**
  - This is a tensor with some value that is updated during execution (e.g. weights matrix in MLP)
  - w = tf.get_variable("w", shape=(10, 20), dtype=tf.float32)
  - w = tf.Variable(tf.random_uniform((10, 20)), name="w")
- **Constant**
  - This is a tensor with constant value, that cannot be changed
  - c = tf.constant(np.ones((4, 4)))

# Operation example

- Matrix product:

```
x = tf.placeholder(tf.float32, (None, 10))
w = tf.Variable(tf.random_uniform((10, 20)), name="w")
z = x @ w
# z = tf.matmul(x, w)
print(z)
```

- Output:

```
Tensor("matmul:0", shape=(?, 20), dtype=float32)
```

- We don’t do any computations here, we just **define** the graph!

# Computational graph

- TensorFlow creates a **default graph** after importing
  - All the operations will go there by default
  - You can get it with tf.get_default_graph(), which returns an instance of tf.Graph.

- You can **create your own** graph variable and define operations there:

```
g = tf.Graph()
with g.as_default():
    pass
```

- You can **clear** the default graph like this:

```
tf.reset_default_graph()
```

# Jupyter Notebook cells

- If you run this cell **3 times**:

```
x = tf.placeholder(tf.float32, (None, 10))
```

- This is what you get in your **default graph**:
  - Using tf.get_default_graph().get_operations()

```
[<tf.Operation 'Placeholder' type=Placeholder>,
 <tf.Operation 'Placeholder_1' type=Placeholder>,
 <tf.Operation 'Placeholder_2' type=Placeholder>]
```

- **Your graph is cluttered**!
  - Clear your graph with tf.reset_default_graph()

# Operations and tensors

- Every **node** in our graph is an **operation**:

```
x = tf.placeholder(tf.float32, (None, 10), name="x")
```

- Listing nodes with tf.get_default_graph().get_operations():

[<tf.Operation **'x'** type=Placeholder>]

- How to get **output tensors** of operation:
  - tf.get_default_graph().get_operations()[0].outputs
  - Output: [<tf.Tensor **'x:0'** shape=(?, 10) dtype=float32>]

# Running a graph

- A **tf.Session** object encapsulates the environment in which tf.Operation objects are executed, and tf.Tensor objects are evaluated.

- Create a session:

```
s = tf.InteractiveSession()
```

- Defining a graph:

```
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
```

- Running a graph:

```
print(c)  # here just looking at the type
print(s.run(c))  # that's how you run the graph
```

- Output:

```
Tensor("mul:0", shape=(), dtype=float32)
30.0
```

# Running a graph

- Operations are written in C++ and executed on CPU or GPU.

- tf.Session owns necessary resources to execute your graph, such as tf.Variable, that occupy RAM.

- It is important to release these resources when they are no longer required with tf.Session.close()

# Initialization of variables

- A variable has an initial value:
  - Tensor: tf.Variable(tf.random_uniform((10, 20)), name="w")
  - Initializer: tf.get_variable("w", shape=(10, 20), dtype=tf.float32)
- You need to run some code to **compute that initial value** in graph execution environment
- This is done with a call in your session s:

  s.run(tf.global_variables_initializer())

- Without it you will get “Attempting to use uninitialized value” errors

# Example

- Definition:

```
tf.reset_default_graph()
a = tf.constant(np.ones((2, 2), dtype=np.float32))
b = tf.Variable(tf.ones((2, 2)))
c = a @ b
```

- Running attempt:

```
s = tf.InteractiveSession()
s.run(c)
```

Output: **“Attempting to use uninitialized value”** error

- Running properly:

```
s.run(tf.global_variables_initializer())
s.run(c)
```

Output: array([[ 2.,2.],[ 2.,2.]], dtype=float32)

# Feeding placeholder values

- Definition:

```
tf.reset_default_graph()
a = tf.placeholder(np.float32, (2, 2))
b = tf.Variable(tf.ones((2, 2)))
c = a @ b
```

- Running attempt:

```
s = tf.InteractiveSession()
s.run(tf.global_variables_initializer())
s.run(c)
```

Output: “**You must feed a value for placeholder tensor**” error

- Running properly:

```
s.run(tf.global_variables_initializer())
s.run(c, feed_dict={a: np.ones((2, 2))})
```

Output: array([[ 2.,2.],[ 2.,2.]], dtype=float32)

# Summary

- TensorFlow: defining and running computational graphs
- Nodes of a graph are operations, that convert a collection of tensors into another collection of tensors
- In Python API you **define** the graph, you **don’t execute** it along the way
  - In 1.5+ the latter mode is supported: eager execution
- You create a **session** to execute your graph (fast C++ code on CPU or GPU)
- Session **owns** all the resources (tensors eat RAM)

# Optimizers in TensorFlow

- Let's define **f** as a square of variable **x**:

```
import numpy as np
import tensorflow as tf

tf.reset_default_graph()
x = tf.get_variable("x", shape=(), dtype=tf.float32)
f = x ** 2
```

- Let's say we want to *minimize* the value of **f** w.r.t **x**:

```
optimizer = tf.train.GradientDescentOptimizer(0.1)
step = optimizer.minimize(f, var_list=[x])
```

# Trainable variables

- You don’t have to specify all the optimized variables:

```
step = optimizer.minimize(f, var_list=[x])
step = optimizer.minimize(f)
```

- Because all variables are **trainable** by default**:**

```
x = tf.get_variable("x", shape=(), dtype=tf.float32)
x = tf.get_variable("x", shape=(), dtype=tf.float32, trainable=True)
```

- You can get all of them:

```
tf.trainable_variables()
```

- Output:

```
[<tf.Variable 'x:0' shape=() dtype=float32_ref>]
```

# Making gradient descent steps

- Now we need to create a **session** and **initialize** variables:

```
s = tf.InteractiveSession()
s.run(tf.global_variables_initializer())
```

- We are ready to make 10 **gradient descent** steps:

```
for i in range(10):
    _, curr_x, curr_f = s.run([step, x, f])
    print(curr_x, curr_f)
```

- Output:

```
0.448929 0.314901
0.359143 0.201537
…
0.0753177 0.00886368
0.0602542 0.00567276
```

*GD step is already applied to x*

# Logging with tf.Print

- We can **evaluate** tensors and print them like this:

```
for i in range(10):
    _, curr_x, curr_f = s.run([step, x, f])
    print(curr_x, curr_f)
```

- Or we can pass our tensor of interest through **tf.Print**:

```
...
f = x ** 2
f = tf.Print(f, [x, f], "x, f:")
...
for i in range(10):
    s.run([step, f])
```

```
x, f:[1.5879565][2.521606]
x, f:[1.2703652][1.6138278]
...
x, f:[0.26641488][0.070976891]
x, f:[0.2131319][0.04542521]
```

*This is Jupyter Server stdout.*
*Not visible in the Notebook!*

# Logging with TensorBoard

- We can add so-called **summaries**:

```
tf.summary.scalar('curr_x', x)
tf.summary.scalar('curr_f', f)
summaries = tf.summary.merge_all()
```

- This is how we log these summaries:

*Run number*

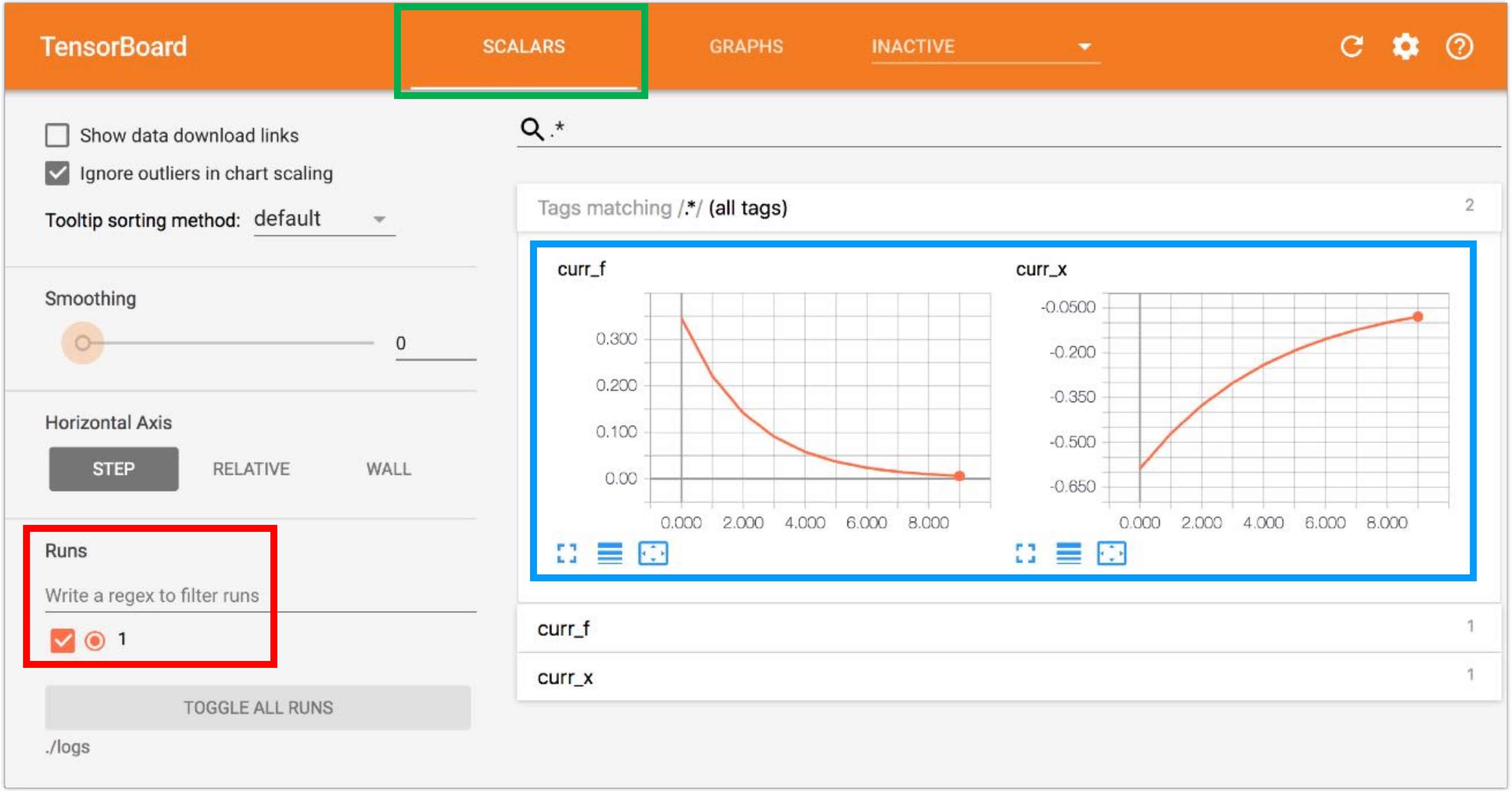
```
s = tf.InteractiveSession()
summary_writer = tf.summary.FileWriter("logs/1", s.graph)
s.run(tf.global_variables_initializer())
for i in range(10):
    _, curr_summaries = s.run([step, summaries])
    summary_writer.add_summary(curr_summaries, i)
    summary_writer.flush()
```

# Launching TensorBoard

- Now you can launch TensorBoard via bash:

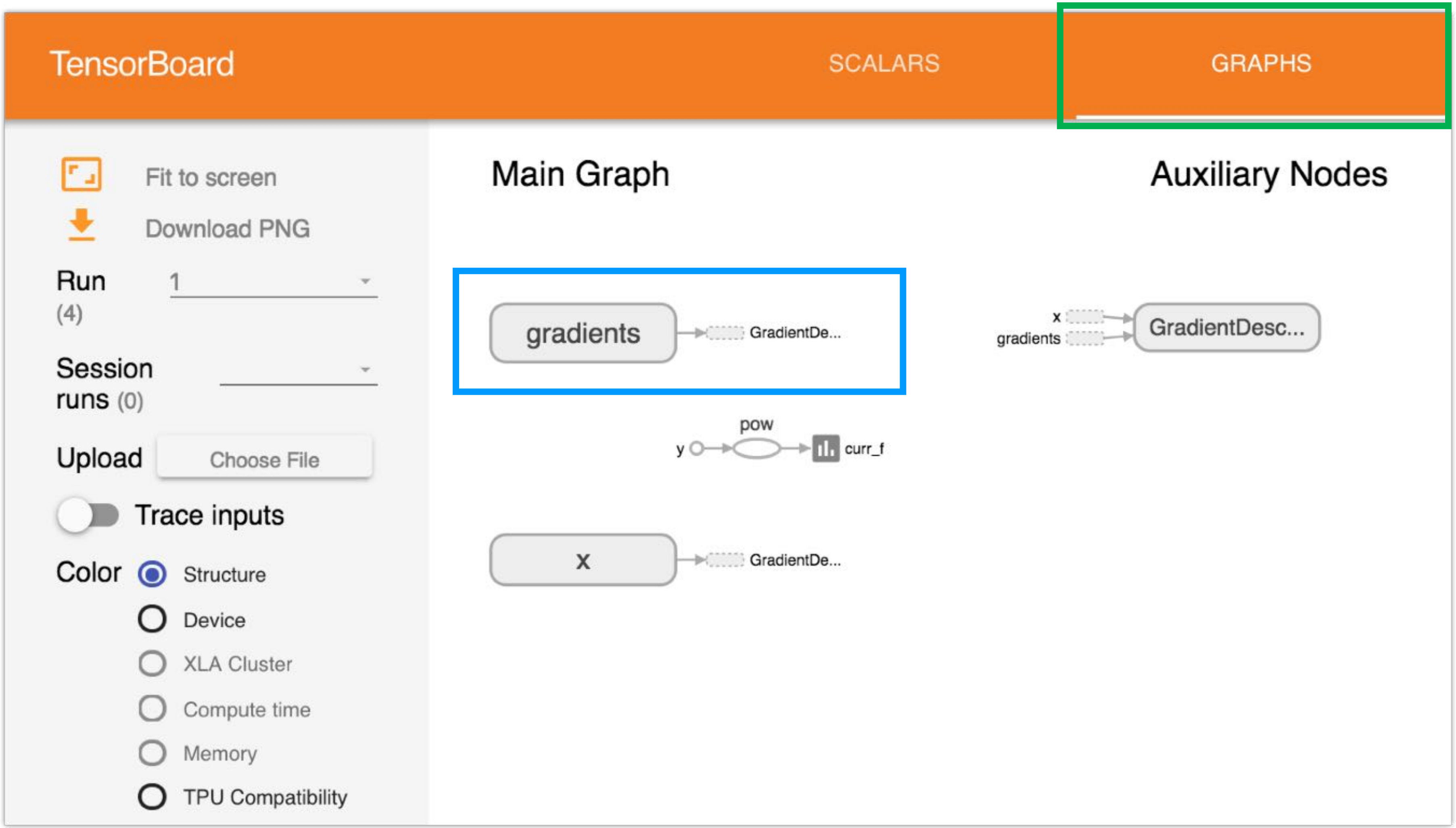
tensorboard --logdir=./logs

- And open **http://127.0.0.1:6006** in your browser.



# Visualizing graph in TensorBoard

- You can see that **gradients computation** is a part of our graph:



# Solving a linear regression

- Let's generate a model dataset:

```
N = 1000
D = 3
x = np.random.random((N, D))
w = np.random.random((D, 1))
y = x @ w + np.random.randn(N, 1) * 0.20
```

# Solving a linear regression

- We will need **placeholders** for input data:

```
tf.reset_default_graph()
features = tf.placeholder(tf.float32, shape=(None, D))
target = tf.placeholder(tf.float32, shape=(None, 1))
```

- This is how we make **predictions**:

```
weights = tf.get_variable("w", shape=(D, 1), dtype=tf.float32)
predictions = features @ weights
```

- And define our **loss**:

```
loss = tf.reduce_mean((target - predictions) ** 2)
```

- And **optimizer**:

```
optimizer = tf.train.GradientDescentOptimizer(0.1)
step = optimizer.minimize(loss)
```

# Solving a linear regression

- Gradient descent:

```
s = tf.InteractiveSession()
s.run(tf.global_variables_initializer())
for i in range(300):
    _, curr_loss, curr_weights = s.run(
        [step, loss, weights], feed_dict={features: x, target: y})
    if i % 50 == 0:
        print(curr_loss)
```

*Filling placeholders*

- Ground truth weights:

[ 0.11649134,0.82753164,0.46924019]

- Found weights:

[ 0.13715988,0.79555332,0.47024861]

# Model checkpoints

- We can save variables' state with **tf.train.Saver**:

```
s = tf.InteractiveSession()
saver = tf.train.Saver(tf.trainable_variables())
s.run(tf.global_variables_initializer())
for i in range(300):
    _, curr_loss, curr_weights = s.run(
        [step, loss, weights], feed_dict={features: x, target: y})
    if i % 50 == 0:
        saver.save(s, "logs/2/model.ckpt", global_step=i)
        print(curr_loss)
```

# Model checkpoints

- We can **list** last checkpoints:

```
saver.last_checkpoints
```

```
['logs/2/model.ckpt-50', 'logs/2/model.ckpt-100',
'logs/2/model.ckpt-150', 'logs/2/model.ckpt-200',
'logs/2/model.ckpt-250']
```

- We can **restore** a previous checkpoint like this:

```
saver.restore(s, "logs/2/model.ckpt-50")
```

- Only variables' values are restored, which means that you need to define a graph in *the same* way **before restoring** a checkpoint.

# Summary

- TensorFlow has **built-in optimizers** that do back-propagation automatically.

- TensorBoard provides **tools for visualizing** your training progress.

- TensorFlow allows you to **checkpoint** your graph to restore its state later *(you need to define it in exactly the same way though)*

# Ссылки

- https://www.oreilly.com/learning/hello-tensorflow
- https://www.tensorflow.org/guide/low_level_intro