TensorFlow: Introduction.

Andrey Skripnikov

Department of Mathematics
University of Houston

August 16, 2018
"TensorFlow™ is an open source software library for numerical computation using data flow graphs."
"TensorFlow\textsuperscript{TM} is an open source software library for numerical computation using data flow graphs."

Step-by-step:

- **Open source** - no comment.
“TensorFlow™ is an open source software library for numerical computation using data flow graphs.”

Step-by-step:

- **Open source** - no comment.
- **Numerical computation** entails anything from basic operations (sum, power, compositions of functions) to hard-core modeling and algorithms (linear regression, neural networks, random forests, etc).
"TensorFlow™ is an open source software library for numerical computation using data flow graphs."

Step-by-step:

- **Open source** - no comment.
- **Numerical computation** entails anything from basic operations (sum, power, compositions of functions) to hard-core modeling and algorithms (linear regression, neural networks, random forests, etc).
- **Data flow graphs** - great way to represent numerical computation process:
Just some of the reasons *TensorFlow* might be your tool of choice for computationally demanding tasks:

- Developed and maintained by **Google**.
- Very large and active community + nice documentation.
- Python API.
- Multi-GPU support.
- *TensorBoard* - powerful visualization tool.
- Faster model compilation than most other options.
Companies Using TensorFlow

- airbnb
- NVIDIA
- UBER
- SAP
- kakao
- DeepMind
- Dropbox
- eBay
- Google
- Snapchat
- Intel
- Coca-Cola
- Xiaomi
- ZTE
- Qualcomm
- Twitter

DATA

8 tensors found
Mnist with images 10K

Color by label
- 0: 980
- 1: 1135
- 2: 1032
- 3: 1010
- 4: 982
- 5: 892
- 6: 958
- 7: 1029
- 8: 974

T-SNE PCA CUSTOM

Dimension: 2D 3D
Perplexity: 25
Learning rate: 10
Re-run Stop
Iteration: 438

How to use t-SNE effectively.
Image Style Transfer with TensorFlow

Image Style Transfer Using Convolutional Neural Networks (Gatys et. al. 2016)
What is a tensor though?

It is a multi-dimensional array of components (typically numbers).

**Examples:**

- **0-d tensor:** scalar (any single number)
What is a tensor though?

It is a multi-dimensional array of components (typically numbers).

**Examples:**

- 0-d tensor: *scalar* (any single number)
- 1-d tensor: *vector*, e.g. \((0, 5, 3)_3, (1, 1, 6, 7, 2, 6, 2)_7\)
What is a tensor though?

It is a multi-dimensional array of components (typically numbers).

**Examples:**

- **0-d tensor:** scalar (any single number)
- **1-d tensor:** vector, e.g. \((0, 5, 3)_3\), \((1, 1, 6, 7, 2, 6, 2)_7\)
- **2-d tensor:** matrix, e.g. \[
\begin{pmatrix}
1 & 3 \\
4 & 2
\end{pmatrix}_{2 \times 2},
\begin{pmatrix}
1.2 & 3.6 & 7 \\
4.2 & 2.8 & 4.9 \\
3 & 6.3 & 1.2 \\
9.1 & 0 & 1.7
\end{pmatrix}_{4 \times 3}\]
What is a tensor though?

It is a **multi-dimensional array** of components (typically numbers).

**Examples:**

- 0-\(d\) tensor: **scalar** (any single number)
- 1-\(d\) tensor: **vector**, e.g. \((0, 5, 3)_3\), \((1, 1, 6, 7, 2, 6, 2)_7\)
- 2-\(d\) tensor: **matrix**, e.g. \(\begin{pmatrix} 1 & 3 \\ 4 & 2 \end{pmatrix} \)\(2\times2\), \(\begin{pmatrix} 1.2 & 3.6 & 7 \\ 4.2 & 2.8 & 4.9 \\ 3 & 6.3 & 1.2 \\ 9.1 & 0 & 1.7 \end{pmatrix} \)\(4\times3\)
- 5-\(d\) tensor??
What is a tensor though?

It is a **multi-dimensional array** of components (typically numbers).

**Examples:**

- **0-d** tensor: scalar (any single number)
- **1-d** tensor: vector, e.g. \((0, 5, 3)_3\), \((1, 1, 6, 7, 2, 6, 2)_7\)
- **2-d** tensor: matrix, e.g. \(\begin{pmatrix} 1 & 3 \\ 4 & 2 \end{pmatrix}_{2\times2}\), \(\begin{pmatrix} 1.2 & 3.6 & 7 \\ 4.2 & 2.8 & 4.9 \\ 3 & 6.3 & 1.2 \\ 9.1 & 0 & 1.7 \end{pmatrix}_{4\times3}\)
- **5-d** tensor?? Where you need **5 indices** to access a single element.
TensorFlow: What’s a Tensor?

What is a tensor though?

It is a multi-dimensional array of components (typically numbers).

**Examples:**

- 0-\(d\) tensor: scalar (any single number)
- 1-\(d\) tensor: vector, e.g. \((0, 5, 3)_3, (1, 1, 6, 7, 2, 6, 2)_7\)
- 2-\(d\) tensor: matrix, e.g. \(\begin{pmatrix} 1 & 3 \\ 4 & 2 \end{pmatrix}_{2\times2}, \begin{pmatrix} 1.2 & 3.6 & 7 \\ 4.2 & 2.8 & 4.9 \\ 3 & 6.3 & 1.2 \\ 9.1 & 0 & 1.7 \end{pmatrix}_{4\times3}\)
- 5-\(d\) tensor?? Where you need 5 indices to access a single element.

From programmers point of view:

> \texttt{a[2,3]} \quad \# \text{To get a single element of a 2-d tensor 'a'}
> \texttt{5}

> \texttt{b[3,5,1,4,4]} \quad \# \text{To get a single element of a 5-d tensor 'b'}
> \texttt{3}
*TensorFlow* operates by using data flow graphs.
TensorFlow: Flow via Computational Graph.

TensorFlow operates by using data flow graphs.

From TensorFlow docs:

"TensorFlow programs are usually structured into

1. a construction phase, that assembles a graph, and
2. an execution phase that uses a session to execute ops in the graph."
"TensorFlow: Flow via Computational Graph."

**TensorFlow** operates by using data flow graphs.

From **TensorFlow** docs:

"TensorFlow programs are usually structured into

1. a **construction** phase, that **assembles a graph**,
2. an **execution** phase that **uses a session** to execute ops in the graph."

That leads us to the concept of "**Computational Graph**" approach:

1. Build the **GRAPH** which **represents the data flow** of the computation,
2. Run the **SESSION** which **executes** the operations graph is describing.
Graph and Session

Graph

- Nodes = Operations

\[ f(x, y) = x^2y + y + 2 \]
Graph and Session

Graph
- Nodes = Operations
- Edges = Tensors

\[ f(x,y) = x^2y + y + 2 \]
Graph and Session

Graph

- **Nodes** = Operations
- **Edges** = Tensors

Session

- **Tensor** = data
- **Tensor + flow** = data + flow

\[ f(x,y) = x^2y + y + 2 \]
Example 1:

```python
import tensorflow as tf
a = 2
b = 3
c = tf.add(a, b, name='Add')
print(c)
```

```
Tensor("Add:0", shape=(), dtype=int32)
```
Graph and Session

Example 2:

```
import tensorflow as tf
x = 2
y = 3
add_op = tf.add(x, y, name='Add')
mul_op = tf.multiply(x, y, name='Multiply')
pow_op = tf.pow(add_op, mul_op, name='Power')

with tf.Session() as sess:
    pow_out = sess.run(pow_op)
```

```python
```
```
**Graph and Session**

**Example 3:**

```python
import tensorflow as tf
x = 2
y = 3
add_op = tf.add(x, y, name='Add')
mul_op = tf.multiply(x, y, name='Multiply')
pow_op = tf.pow(add_op, mul_op, name='Power')
useless_op = tf.multiply(x, add_op, name='Useless')

with tf.Session() as sess:
    [pow_out, useless_out] = sess.run([pow_op, useless_op])
```

**Variables**

- `x`: Int 2
- `y`: Int 3
- `add_op`: Tensor("Add", shape=[], dtype=int32)
- `mul_op`: Tensor("Multiply", shape=[], dtype=int32)
- `pow_op`: Tensor("Power", shape=[], dtype=int32)
- `useless_op`: Tensor("Useless", shape=[], dtype=int32)
- `pow_out`: Int32 15625
- `useless_out`: Int32 10
In order to fully erase all the graph definitions, one uses `tf.reset_default_graph()`.

```python
In [31]:
a = 2
b = 3
c = tf.add(a, b, name="Add")

with tf.Session() as sess:
    print(sess.run(c))

5

In [32]:
tf.reset_default_graph()

with tf.Session() as sess:
    print(sess.run(c))

-----------------------------------
RuntimeError: Traceback (most recent call last)
<ipython-input-32-87c6272989e3> in <module>()
    2
    3 with tf.Session() as sess:
----> 4    print(sess.run(c))

RuntimeError: The Session graph is empty. Add operations to the graph before calling run().
```
Data types

1. **Constants** are used to create constant values.

Before:

```python
import tensorflow as tf
da = 2
b = 3
c = tf.add(a, b, name='Add')

with tf.Session() as sess:
    print(sess.run(c))
```

Result:

```
5
```
Data types

1. **Constants** are used to create constant values.

Now:

```python
import tensorflow as tf
da = tf.constant(2, name='A')
bbb = tf.constant(3, name='B')
c = tf.add(a, b, name='Add')

with tf.Session() as sess:
    print(sess.run(c))
```

```
Variables

- a = {Tensor} Tensor("A:0", shape=(), dtype=int32)
- b = {Tensor} Tensor("B:0", shape=(), dtype=int32)
- c = {Tensor} Tensor("Add:0", shape=(), dtype=int32)
```
Data types

2. Variables

```python
import tensorflow as tf

# create graph
a = tf.get_variable(name="A", initializer=tf.constant([[0, 1], [2, 3]]))
b = tf.get_variable(name="B", initializer=tf.constant([[4, 5], [6, 7]]))
c = tf.add(a, b, name="Add")

# Add an Op to initialize variables
init_op = tf.global_variables_initializer()

# launch the graph in a session
with tf.Session() as sess:
    # run the variable initializer
    sess.run(init_op)

    # now we can run the desired operation
    print(sess.run(c))
```

Variables

| a = (Variable) <tf.Variable 'A:0' shape=(2, 2) dtype=int32_ref>
| b = (Variable) <tf.Variable 'B:0' shape=(2, 2) dtype=int32_ref>
| c = (Tensor) Tensor("Add:0", shape=(2, 2), dtype=int32) |
3. Placeholder

```python
import tensorflow as tf
a = tf.constant([5, 5, 5], tf.float32, name='A')
b = tf.placeholder(tf.float32, shape=[3], name='B')
c = tf.add(a, b, name="Add")

with tf.Session() as sess:
    d = {b: [1, 2, 3]}
    print(sess.run(c, feed_dict=d))

[6. 7. 8.]
```
Variables vs Placeholders.

Stackoverflow Answer #1:

In short, you use `tf.Variable` for trainable variables, such as regression coefficients ($\beta$), weights ($w$) and biases ($b$) for your model.

`tf.placeholder` is used to feed actual training examples at execution time.

Stackoverflow Answer #2:

The difference is that with `tf.Variable` you have to provide an initial value when you declare it. With `tf.placeholder` you don’t have to provide an initial value and you can specify it at run time with the `feed_dict` argument inside `Session.run`.
Variables vs Placeholders: Neural Networks.

Let’s look at an example of variable and placeholder definitions for **Neural Networks** - which is the main application for the **TensorFlow** framework.

In the code below, we define

- The network’s **learnable parameters** - weight and bias - as **TF** variables. Those are of fixed determined sizes (or shapes), and will be changing as the model trains.
Variables vs Placeholders: Neural Networks.

Let’s look at an example of variable and placeholder definitions for Neural Networks - which is the main application for the TensorFlow framework.

In the code below, we define

- The network’s learnable parameters - weight and bias - as TF variables. Those are of fixed determined sizes (or shapes), and will be changing as the model trains.
- Input data \( X \) and response \( Y \) are defined as TF placeholders. These
  - have un-defined first dimension size (\( \equiv \text{None} \)), which corresponds to # of sample provided for training. The amount of sample provided to Neural Network may vary, and that is reflected.
  - are fed during model training.

```
W = tf.get_variable('weight', shape=(784, 10),
                   initializer=tf.truncated_normal_initializer(mean=0.0, stddev=0.01))
Bias = tf.get_variable('bias', shape=10,
                      initializer = tf.truncated_normal_initializer(mean=0.0, stddev=0.01))
X = tf.placeholder(tf.float32, shape=[None, 784], name='input')
Y = tf.placeholder(tf.float32, shape=[None, 10], name='label')
```
We provide an example of using TensorFlow to execute simple linear regression on a birth data example, where

- the sole predictor $X$ is birth rate,
- the response variable $Y$ is life expectancy.

See the notebook for details.
**TensorFlow: Building a Model.**

Most critical aspects to remember when building and training any model with **TensorFlow** is to specify:

1. **TF Placeholders** corresponding to input data and output values. E.g. for simple linear model $Y = b + wX$:
   
   ```python
   X = tf.placeholder(tf.float32, name='X')
   Y = tf.placeholder(tf.float32, name='Y')
   ```

2. **TF Variables** for the learnable parameters.
   
   ```python
   w = tf.get_variable('weights', initializer=tf.constant(0.0))
   b = tf.get_variable('bias', initializer=tf.constant(0.0))
   ```

3. The **loss function** for the training algorithm to optimize:
   
   ```python
   Y_predicted = w * X + b
   loss = tf.square(Y - Y_predicted, name='loss')
   ```

4. The **training algorithm** to optimize the loss function via estimating the learnable parameters:
   
   ```python
   optimizer = tf.train.GradientDescentOptimizer(...)
   ```

5. To train the model, run the optimizer from Step 4 and keep track of the loss function (defined in Step 3) value.
Most critical aspects to remember when building and training any model with TensorFlow is to specify:

1. **TF Placeholders** corresponding to input data and output values. E.g. for simple linear model $Y = b + wX$:
   
   ```python
   X = tf.placeholder(tf.float32, name='X')
   Y = tf.placeholder(tf.float32, name='Y')
   ```

2. **Variables** for the learnable parameters.
   
   ```python
   w = tf.get_variable('weights', initializer=tf.constant(0.0))
   b = tf.get_variable('bias', initializer=tf.constant(0.0))
   ```

3. The loss function for the training algorithm to optimize:
   
   ```python
   Y_predicted = w * X + b
   loss = tf.square(Y - Y_predicted, name='loss')
   ```

4. The training algorithm to optimize the loss function via estimating the learnable parameters
   
   ```python
   optimizer = tf.train.GradientDescentOptimizer(...)
   ```

5. To train the model, run the optimizer from Step 4 and keep track of loss function (defined in Step 3) value.

Andrey Skripnikov (University of Houston)
Most critical aspects to remember when building and training any model with TensorFlow is to specify:

1. **TF Placeholders** corresponding to input data and output values. E.g. for simple linear model $Y = b + wX$:
   
   $X = \text{tf.placeholder(tf.float32, name=’X’)}$
   
   $Y = \text{tf.placeholder(tf.float32, name=’Y’)}$

2. **TF Variables** for the learnable parameters.
   
   $w = \text{tf.get_variable(’weights’, initializer=tf.constant(0.0))}$
   
   $b = \text{tf.get_variable(’bias’, initializer=tf.constant(0.0))}$
Most critical aspects to remember when building and training any model with *TensorFlow* is to specify:

1. **TF Placeholders** corresponding to input data and output values. E.g. for simple linear model $Y = b + wX$:
   
   ```
   X = tf.placeholder(tf.float32, name='X')
   Y = tf.placeholder(tf.float32, name='Y')
   ```

2. **TF Variables** for the learnable parameters.
   
   ```
   w = tf.get_variable('weights', initializer=tf.constant(0.0))
   b = tf.get_variable('bias', initializer=tf.constant(0.0))
   ```

3. The **loss function** for the training algorithm to optimize:
   
   ```
   Y_predicted = w * X + b
   loss = tf.square(Y - Y_predicted, name='loss')
   ```
Most critical aspects to remember when building and training any model with TensorFlow is to specify:

1. **TF Placeholders** corresponding to input data and output values. E.g. for simple linear model \( Y = b + wX \):
   
   \[
   X = \text{tf.placeholder}(\text{tf.float32}, \text{name}='X')
   \]
   
   \[
   Y = \text{tf.placeholder}(\text{tf.float32}, \text{name}='Y')
   \]

2. **TF Variables** for the learnable parameters.
   
   \[
   w = \text{tf.get_variable}('weights', \text{initializer}=\text{tf.constant}(0.0))
   \]
   
   \[
   b = \text{tf.get_variable}('bias', \text{initializer}=\text{tf.constant}(0.0))
   \]

3. The **loss function** for the training algorithm to optimize:
   
   \[
   \text{Y\_predicted} = w \times X + b
   \]
   
   \[
   \text{loss} = \text{tf.square}(Y - \text{Y\_predicted}, \text{name}='loss')
   \]

4. The **training algorithm** to optimize the loss function via estimating the learnable parameters
   
   \[
   \text{optimizer} = \text{tf.train.GradientDescentOptimizer}(...)
   \]
Most critical aspects to remember when building and training any model with TensorFlow is to specify:

1. **TF Placeholders** corresponding to input data and output values. E.g. for simple linear model $Y = b + wX$:
   
   ```python
   X = tf.placeholder(tf.float32, name='X')
   Y = tf.placeholder(tf.float32, name='Y')
   ```

2. **TF Variables** for the learnable parameters.
   
   ```python
   w = tf.get_variable('weights', initializer=tf.constant(0.0))
   b = tf.get_variable('bias', initializer=tf.constant(0.0))
   ```

3. The **loss function** for the training algorithm to optimize:
   
   ```python
   Y_predicted = w * X + b
   loss = tf.square(Y - Y_predicted, name='loss')
   ```

4. The **training algorithm** to optimize the loss function via estimating the learnable parameters
   
   ```python
   optimizer = tf.train.GradientDescentOptimizer(...)
   ```

5. To train the model, **run the optimizer from Step 4** and keep track of loss function (defined in Step 3) value.