How to Use TensorBoard?

Aryan mobiny
Jun 9, 2018 · 10 min read

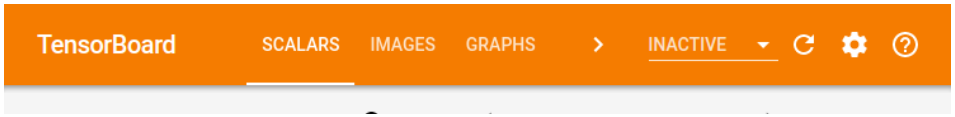
The two main advantages of TensorFlow over many other available libraries are flexibility and visualization. Imagine if you can visualize whats happening in the code (in this case code represents the computational graph that we create for a model), it would be so convenient to deeply understand and observe the inner workings of the graph. Not just that, it also helps in fixing things that are not working the way they should. TensorFlow provides a way to do just that using TensorBoard!

TensorBoard is a visualization software that comes with any standard TensorFlow installation. In Google's words:

> *"The computations you'll use TensorFlow for many things (like training a massive deep neural network) and they can be complex and confusing. To make it easier to understand, debug, and optimize TensorFlow programs, we've included a suite of visualization tools called TensorBoard."*

TensorFlow programs can range from a very simple to super complex problems (using thousands of computations), and they all have two basic components, Operations, and Tensors. As explained in the previous tutorials, the idea is that you create a model that consists of a set of operations, feed the data into the model and the tensors will flow between the operations until you get an output tensor i.e., your result. TensorBoard provides us with a suite of web applications that help us to inspect and understand the TensorFlow runs and graphs. Currently, it provides five types of visualizations: scalars, images, audio, histograms, and graphs.

When fully configured, TensorBoard window will look something like:



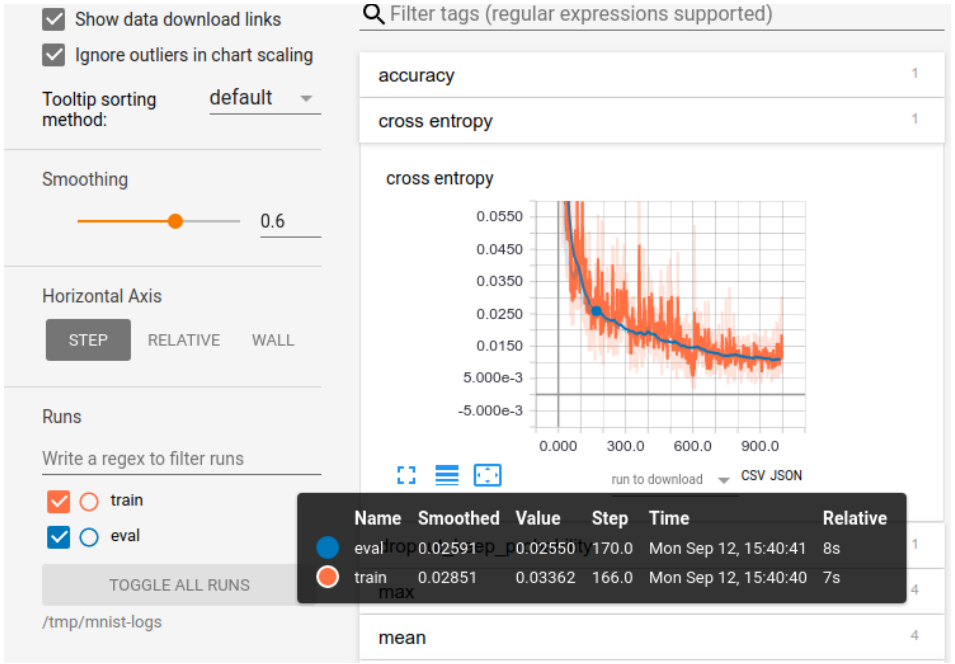


Fig. 1. TensorBoard appearance

TensorBoard was created as a way to help us understand the flow of tensors in your model so that we can debug and optimize it. It is generally used for two main purposes:

1. Visualizing the Graph

2. Writing Summaries to Visualize Learning

We'll cover the main usages of TensorBoard in this tutorial. Learning to use TensorBoard early and often will make working with TensorFlow much more enjoyable and productive.

1. Visualizing the Graph

While powerful, TensorFlow computation graphs can become extremely complicated. Visualizing the graph can help us understand and debug it. Here's an example of the visualization at work from TensorFlow website.

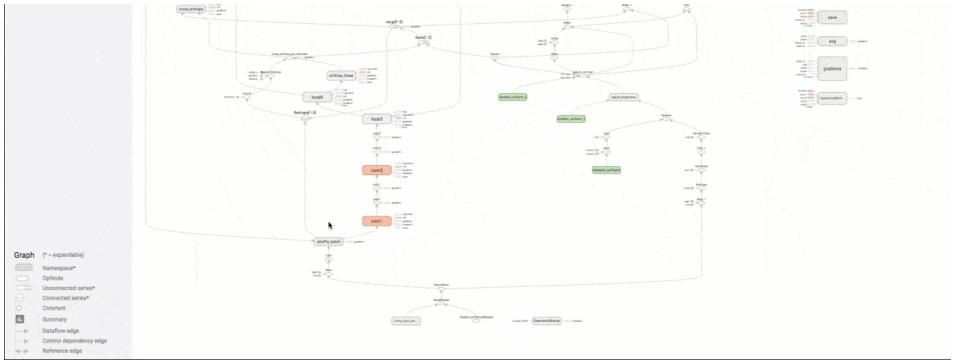


Fig. 2. Visualization of a TensorFlow graph (Source: TensorFlow website)

To make our TensorFlow program **TensorBoard-activated**, we need to add some lines of code. This will export the TensorFlow operations into a file, called **event file** (or event log file). TensorBoard is able to read this file and give some insights of the model graph and its performance.

Now let's write a simple TensorFlow program and visualize its computational graph with TensorBoard.

Example 1:

Let's create two constants and add them together. Constant tensors can be defined simply by their value:

```
import tensorflow as tf
# create graph
a = tf.constant(2)
b = tf.constant(3)
c = tf.add(a, b)
# launch the graph in a session
with tf.Session() as sess:
    print(sess.run(c))
```

01.py hosted with ♡ by GitHub view raw

To visualize the program with TensorBoard, we need to write log files of the program. To write event files, we first need to create a **writer** for those logs, using this code:

```
writer = tf.summary.FileWriter([logdir], [graph])
```

where **[logdir]** is the folder where we want to store those log files. We can also choose [logdir] to be something meaningful such as './graphs'. The second argument **[graph]** is the graph of the program we're working on. There are two ways to get the graph:

1. Call the graph using **tf.get_default_graph()**, which returns the default graph of the program
2. set it as **sess.graph** which returns the session's graph (note that this requires us to have a session created).

Lets take a look at both ways in the following example; however, the second way is more common. Either way, make sure to create a writer only after defining the graph. Otherwise, the graph visualized on TensorBoard would be incomplete.

Let's add the writer to the first example and visualize the graph.

```
import tensorflow as tf
tf.reset_default_graph()   # To clear the defined variables and operations of the previous cell
# create graph
a = tf.constant(2)
b = tf.constant(3)
c = tf.add(a, b)
# creating the writer out of the session
# writer = tf.summary.FileWriter('./graphs', tf.get_default_graph())
# launch the graph in a session
with tf.Session() as sess:
    # or creating the writer inside the session
    writer = tf.summary.FileWriter('./graphs', sess.graph)
    print(sess.run(c))
```


Now if we execute this code, TensorFlow creates a directory inside your current directory (beside your Python code file) which contains the **event file**.

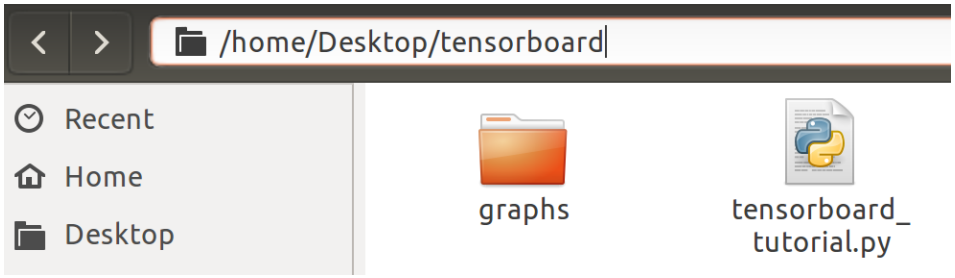


Fig. 3. Created directory which contains the event file

Next, to visualize the graph, we need to go to Terminal and make sure that the present working directory is the same as where we ran our Python code. For example, here we can switch to the directory using the commands:

$ cd ~/Desktop/tensorboard

Then run:

$ tensorboard — logdir="./graphs" — port 6006

Replace './graphs' with the name of the directory in case you choose to name it something else. This will generate a link on the command line. Control click (ctrl+left) the link to open the TensorBoard window, TensorBoard uses the web browser to show us the visualizations (or simply copy it into your browser or just open your browser and go to http://localhost:6006/). The link will direct us to the TensorBoard page, it should look similar to:

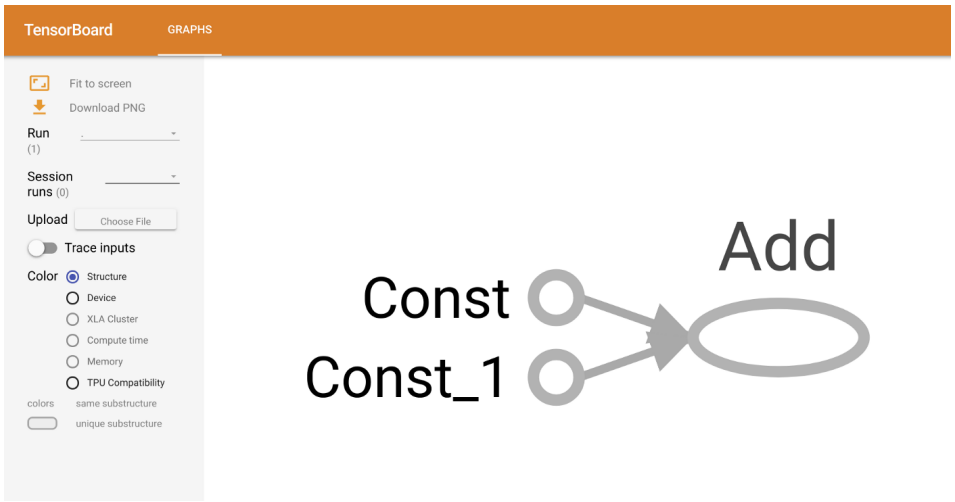
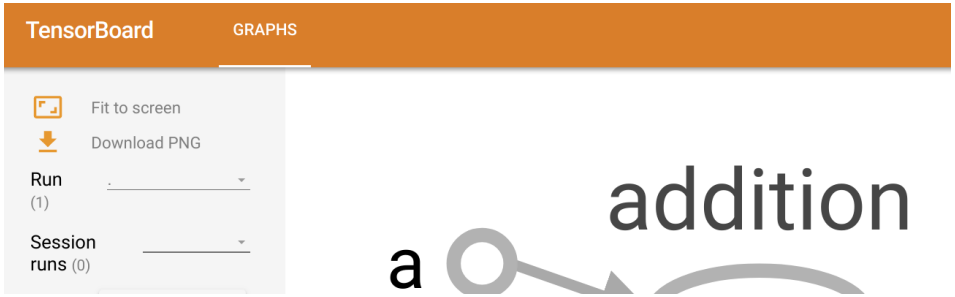


Fig. 4. TensorBoard page visualizing the graph generated in Example 1

The graph in the above picture shows us the various parts of our model. "Const" and "Const_1" in the graph correspond to a and b, and the node "Add" corresponds to c. The names given in the code (a, b, and c) are just **Python-names**, they only help us with the access while writing the code. The names mean nothing to the TensorFlow and TensorBoard. To make TensorBoard understand the names of our ops, we have to explicitly name them.

Let's modify the code one more time to add the names:

```
import tensorflow as tf
tf.reset_default_graph()   # To clear the defined variables and operations of the previous cell
# create graph
a = tf.constant(2, name="a")
b = tf.constant(3, name="b")
c = tf.add(a, b, name="addition")
# creating the writer out of the session
# writer = tf.summary.FileWriter('./graphs', tf.get_default_graph())
# launch the graph in a session
with tf.Session() as sess:
    # or creating the writer inside the session
    writer = tf.summary.FileWriter('./graphs', sess.graph)
    print(sess.run(c))
```

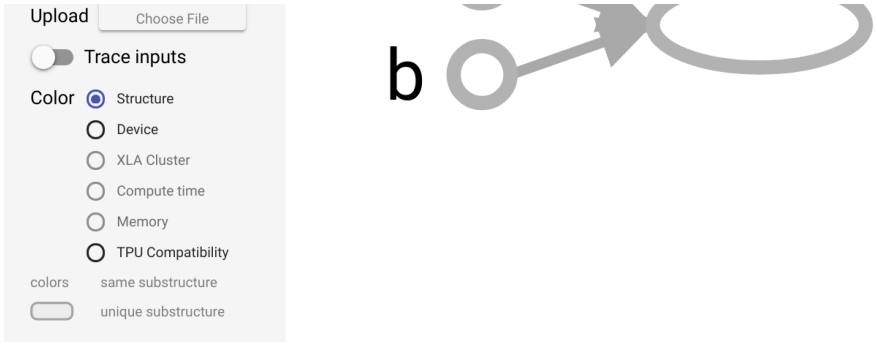


Fig. 5. TensorBoard page visualizing the graph generated in Example 1 with modified names

***Note:** If we run our code several times with the same [logdir], multiple event files will be generated in our [logdir]. TF will only show the latest version of the graph and display the warning of multiple event files. The warning can be removed by deleting the event files that we no longer need or else we can save them in a different [logdir] folder.

2. Writing Summaries to Visualize Learning

So far we only focused on how to visualize the graph in TensorBoard. Remember the other types of visualizations mentioned in the earlier part of the post that TensorBoard provides (scalars, images and histograms). In this part, we are going to use a special operation called **summary** to visualize the model parameters (like weights and biases of a neural network), metrics (like loss or accuracy value), and images (like input images to a network).

Summary is a special operation TensorBoard that takes in a regular tensor and outputs the summarized data to your disk (i.e. in the event file). Basically, there are three main types of summaries:

1. tf.summary.scalar: used to write a single scalar-valued tensor (like a classificaion loss or accuracy value)

2. tf.summary.histogram: used to plot histogram of all the values of a non-scalar tensor (can be used to visualize weight or bias matrices of a neural network)

3. tf.summary.image: used to plot images (like input images of a network, or generated output images of an autoencoder or a GAN)

In the following sections, let's go through each of the above mentioned summary types in more detail.

2.1. tf.summary.scalar:

It's for writing the values of a scalar tensor that changes over time or iterations. In the case of neural networks (say a simple network for classification task), it's usually used to monitor the changes of loss function or classification accuracy.

Let's run a simple example to understand the point.

Example 2:

Randomly pick 100 values from a standard Normal distribution, *N(0, 1)*, and plot them one after the other.

One way to do so is to simply create a variable and initialize it from a normal distribution (with mean=0 and std=1), then run a for loop in the session and initialize it 100 times. The code will be as follows and the required steps to write the summary is explained in the code:

```
import tensorflow as tf
tf.reset_default_graph()   # To clear the defined variables and operations of the previous cell
# create the scalar variable
x_scalar = tf.get_variable('x_scalar', shape=[], initializer=tf.truncated_normal_initializer(me
# ____step 1:____ create the scalar summary
first_summary = tf.summary.scalar(name='My_first_scalar_summary', tensor=x_scalar)
init = tf.global_variables_initializer()
# launch the graph in a session
with tf.Session() as sess:
    # ____step 2:____ creating the writer inside the session
    writer = tf.summary.FileWriter('./graphs', sess.graph)
    for step in range(100):
        # loop over several initializations of the variable
        sess.run(init)
        # ____step 3:____ evaluate the scalar summary
        summary = sess.run(first_summary)
        # ____step 4:____ add the summary to the writer (i.e. to the event file)
        writer.add_summary(summary, step)
    print('Done with writing the scalar summary')
```

```
Done with writing the scalar summary
```

Let's pull up TensorBoard and checkout the result. Like before, you need to open terminal and type:

$ tensorboard — logdir="./graphs" — port 6006

Here *"./graphs"* is the name of the directory we saved the event file to. In TensorBoard, we find a new tab named **"scalars"** next to the **"graphs"** tab earlier discussed (compare Fig. 5 with Fig. 6). The whole window looks like:

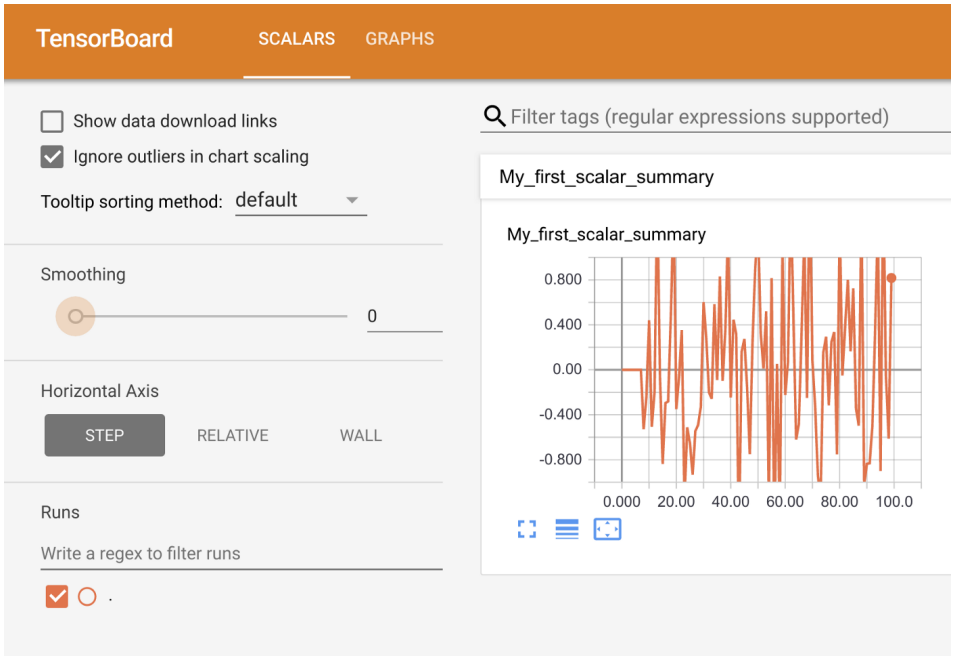


Fig. 6. TensorBoard page visualizing the written scalar summary.

In the figure, the plot panel is under the name "My_first_scalar_summary", the same name that we defined in our code. The x-axis and y-axis shows the 100 steps and the corresponding values (random values from a standard normal dist.) of the variable respectively.

2.2. tf.summary.histogram:

Histogram comes in handy if we want to observe the change of a value over time or iterations. It's used for plotting the histogram of the values of a non-scalar tensor. This provides us a view of how the histogram (and the distribution) of the tensor values change over time or iterations. In the case of neural networks, it's commonly used to monitor the changes of weight and biase distributions. It's very useful in detecting irregular behavior of the network parameters (for example, when our weights explode or shrink abnormally).

Now let's go back to our previous example and add the histogram summary to it.

Example 3:

Continue the previous example by adding a matrix of size 30x40, whose entries come from a standard normal distribution. Initialize this matrix 100 times and plot the distribution of its entries over time.

```
import tensorflow as tf
tf.reset_default_graph()   # To clear the defined variables and operations of the previous cell
# create the variables
x_scalar = tf.get_variable('x_scalar', shape=[], initializer=tf.truncated_normal_initializer(me
x_matrix = tf.get_variable('x_matrix', shape=[30, 40], initializer=tf.truncated_normal_initiali
# ____step 1:____ create the summaries
# A scalar summary for the scalar tensor
scalar_summary = tf.summary.scalar('My_scalar_summary', x_scalar)
# A histogram summary for the non-scalar (i.e. 2D or matrix) tensor
histogram_summary = tf.summary.histogram('My_histogram_summary', x_matrix)
init = tf.global_variables_initializer()
# launch the graph in a session
with tf.Session() as sess:
    # ____step 2:____ creating the writer inside the session
    writer = tf.summary.FileWriter('./graphs', sess.graph)
    for step in range(100):
        # loop over several initializations of the variable
        sess.run(init)
        # ____step 3:____ evaluate the merged summaries
        summary1, summary2 = sess.run([scalar_summary, histogram_summary])
        # s____step 4:____ add the summary to the writer (i.e. to the event file) to write on t
        writer.add_summary(summary1, step)
        # repeat steps 4 for the histogram summary
        writer.add_summary(summary2, step)
    print('Done writing the summaries')
```

```
Done writing the summaries
```

In TensorBoard, two new tabs are added to the top menu: “Distributions” and “Histograms”. The results will be as follows:

In TensorBoard, two new tabs are added to the top menu: “Distributions” and “Histograms”. The results will be as follows:

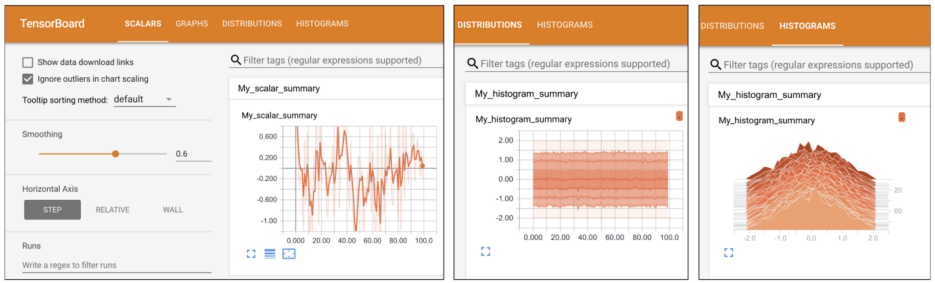


Fig. 7. (a) scalar summary, (b) distribution and © histogram of the values of the 2D-tensor over 100 steps

In the figure, the “Distributions” tab contains a plot that shows the distribution of the values of the tensor (y-axis) through steps (x-axis). You might ask what are the light and dark colors?

The answer is that each line on the chart represents a percentile in the distribution over the data. For example, the bottom line (the very light one) shows how the minimum value has changed over time, and the line in the middle shows how the median has changed. Reading from top to bottom, the lines have the following meaning: [maximum, 93%, 84%, 69%, 50%, 31%, 16%, 7%, minimum]

These percentiles can also be viewed as standard deviation boundaries on a normal distribution: [maximum, μ+1.5σ, μ+σ, μ+0.5σ, μ, μ-0.5σ, μ-σ, μ-1.5σ, minimum] so that the colored regions, read from inside to outside, have widths [σ, 2σ, 3σ] respectively.

Similarly, in the histogram panel, each chart shows temporal “slices” of data, where each slice is a histogram of the tensor at a given step. It’s organized with the oldest timestep in the back, and the most recent timestep in front.

You can easily monitor the values on the histograms at any step. Just move your cursor on the plot and see the x-y values on the histograms (Fig8 (a)). You can also change the Histogram Mode from “offset” to “overlay” (see Fig. 8- (b)) to see the histograms overlaid with one another.

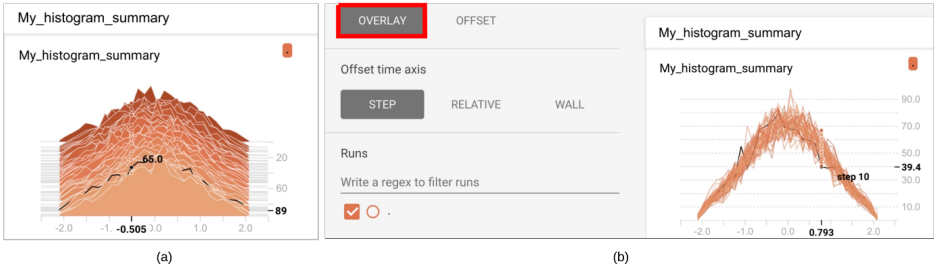


Fig. 8. (a) monitor values on the histograms, (b) overlayed histograms

As mentioned in the code, we need to run every summary (e.g. **sess.run([scalar_summary, histogram_summary])**) and then use our writer to write each of them to the disk. In practice, we can use any number of summaries to track different parameters in our model. This makes running and writing the summaries extremly inefficient. The way around it is to merge all summaries in our graph and run them at once inside your session. This can be done with **tf.summary.merge_all()** method. Let’s add it to Example 3, the code changes as follows:

```
# create the variables
x_scalar = tf.get_variable('x_scalar', shape=[], initializer=tf.truncated_normal_initializer(me
x_matrix = tf.get_variable('x_matrix', shape=[30, 40], initializer=tf.truncated_normal_initiali
# ____step 1:____ create the summaries
# A scalar summary for the scalar tensor
scalar_summary = tf.summary.scalar('My_scalar_summary', x_scalar)
# A histogram summary for the non-scalar (i.e. 2D or matrix) tensor
histogram_summary = tf.summary.histogram('My_histogram_summary', x_matrix)
# ____step 2:____ merge all summaries
merged = tf.summary.merge_all()
init = tf.global_variables_initializer()
# launch the graph in a session
with tf.Session() as sess:
    # ____step 3:____ creating the writer inside the session
    writer = tf.summary.FileWriter('./graphs', sess.graph)
    for step in range(100):
        # loop over several initializations of the variable
        sess.run(init)
```

```
        # ____step 4:____ evaluate the merged summaries
        summary = sess.run(merged)
        # ____step 5:____ add summary to the writer (i.e. to the event file) to write on the di
        writer.add_summary(summary, step)
    print('Done writing the summaries')
```

```
Done writing the summaries
```

2.2. tf.summary.image:

As the name implies, this type of summary is used for writing and visualizing tensors as images. In the case of neural networks, this is usually used for tracking the images that are either fed to the network (say in each batch) or the images generated in the output (such as the reconstructed images in an autoencoder; or the fake images made by the generator model of a Generative Adverserial Network). However, in general, this can be used for plotting any tensor. For example, we can visualize a weight matrix of size 30x40 as an image of 30x40 pixels.

An image summary can be created using:

```
tf.summary.image(name, tensor, max_outputs=3)
```

Where **name** is the name for the generated node (i.e. operation), tensor is the desired tensor to be written as an image summary (we will talk about its shape shortly), and max_outputs is the maximum number of elements from **tensor** to generate images for. but… what does it mean? The answer lies in the the shape of the **tensor**.

The **tensor** that we feed to tf.summary.image must be a 4-D tensor of shape **[batch_size, height, width, channels]** where batch_size is the number of images in the batch, the height and width determine the size of the image and finally, the channels are: 1: for Grayscale images. 3: for RGB (i.e. color) images. 4: for RGBA images (where A stands for alpha; see RGBA).

Let's look at a very simple example to get the underlying idea.

Example 4:

Let's define two variables:

1. Of size 30x10 as 3 **grayscale** images of size 10x10
2. Of size 50x30 as 5 **color** images of size 10x10

and plot them as images in TensorBoard.

```
import tensorflow as tf
tf.reset_default_graph()   # To clear the defined variables and operations of the previous cell
# create the variables
w_gs = tf.get_variable('W_Grayscale', shape=[30, 10], initializer=tf.truncated_normal_initializ
w_c = tf.get_variable('W_Color', shape=[50, 30], initializer=tf.truncated_normal_initializer(me
# ___step 0:___ reshape it to 4D-tensors
w_gs_reshaped = tf.reshape(w_gs, (3, 10, 10, 1))
w_c_reshaped = tf.reshape(w_c, (5, 10, 10, 3))
# ____step 1:____ create the summaries
gs_summary = tf.summary.image('Grayscale', w_gs_reshaped)
c_summary = tf.summary.image('Color', w_c_reshaped, max_outputs=5)
# ____step 2:____ merge all summaries
merged = tf.summary.merge_all()
# create the op for initializing all variables
init = tf.global_variables_initializer()
# launch the graph in a session
with tf.Session() as sess:
    # ____step 3:____ creating the writer inside the session
    writer = tf.summary.FileWriter('./graphs', sess.graph)
    # initialize all variables
    sess.run(init)
    # ____step 4:____ evaluate the merged op to get the summaries
    summary = sess.run(merged)
    # ____step 5:____ add summary to the writer (i.e. to the event file) to write on the disc
    writer.add_summary(summary)
    print('Done writing the summaries')
```

```
Done writing the summaries
```

Now open TensorBoard like before and switch to **IMAGES** tab. The images should be something similar to:

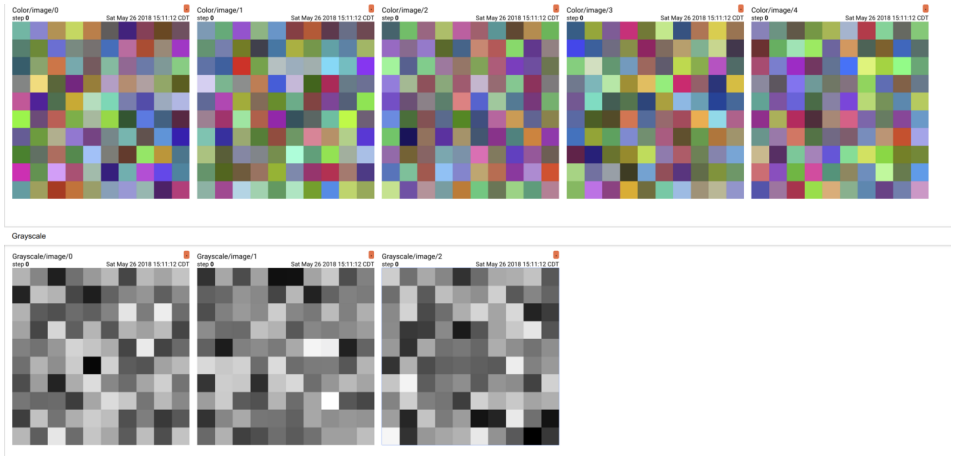


Fig. 9. generated images in TensorBoard

We can similarly add any other image of any size to our summaries and plot them in TensorBoard.

Thanks for reading this almost long tutorial. In the next tutorial, I'll run a simple neural network and visualize its graph and performance using TensorBoard.

This was my first post on Medium. So please don't hesitate to encourage me with your claps and send me your feed-backs by leaving comments below.

Machine Learning TensorFlow Tensorboard Deep Learning