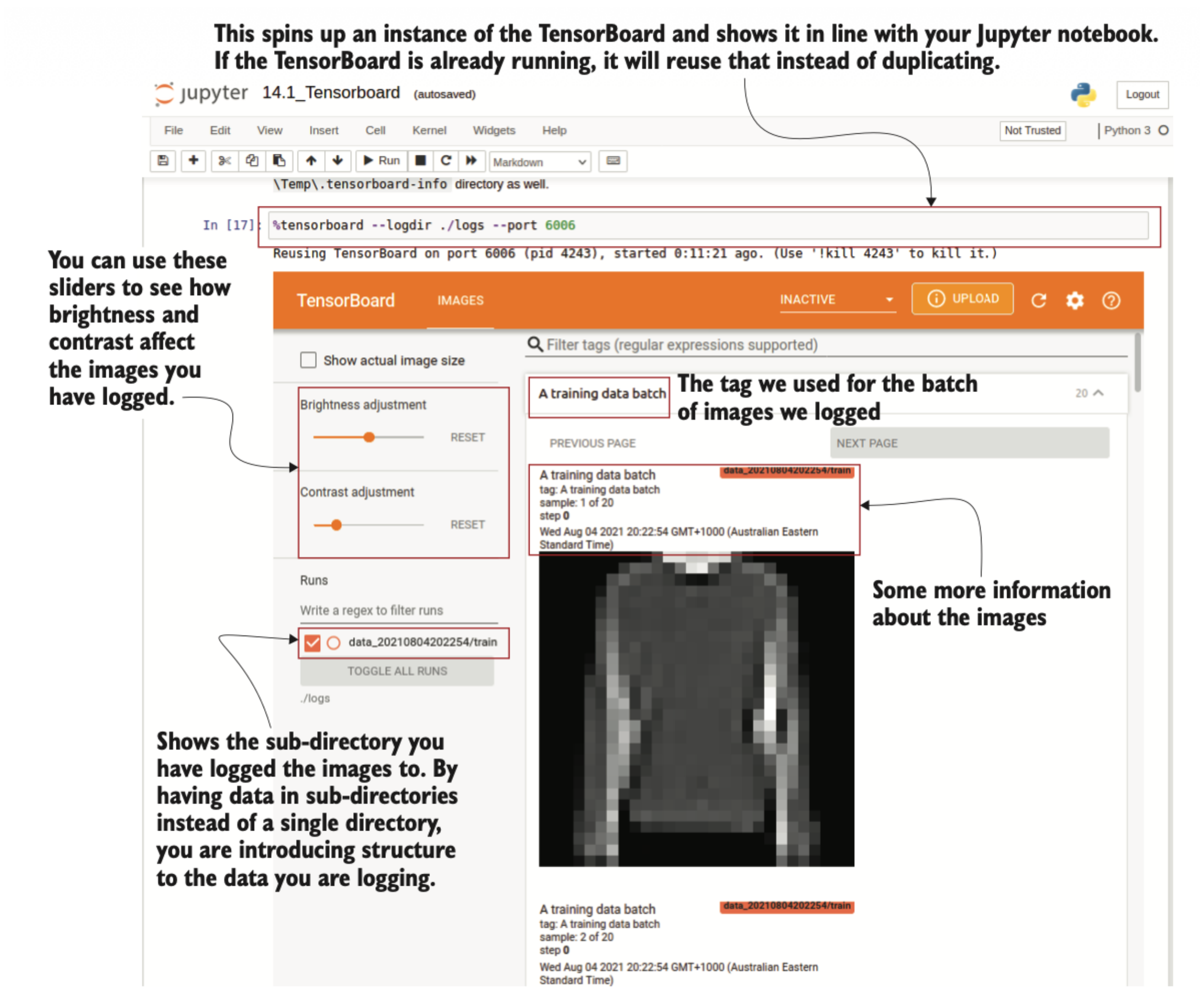
Chapter 14

TensorBoard: Big Brother of TensorFlow

The story begins with the recognition that building neural networks is only one part of the journey. Just as important is the ability to observe what is happening inside them, to track whether the data has been correctly prepared, and to detect when models drift away from learning effectively. To serve this purpose, TensorBoard emerges as a visual companion, transforming abstract tensors and training logs into interpretable dashboards. By organizing experiments into timestamped directories, it enables the researcher to compare multiple runs, follow progress over time, and spot unusual behaviors before investing too much training time.

A concrete illustration comes from image recognition of garments. The Fashion-MNIST dataset provides a set of 28×28 grayscale clothing images across ten categories, including shirts, dresses, and sneakers. After constructing training, validation, and testing datasets, images are logged through TensorBoard writers. Once opened, the dashboard presents them as labeled samples, ensuring they have been loaded with the correct categories. Beyond images, TensorBoard supports scalars, histograms, audio, and text, making it versatile for numerous data modalities. A visualization example (Figure 1) demonstrates how logged images can be browsed interactively, brightness and contrast adjusted, and data folders selectively viewed.


This spins up an instance of the TensorBoard and shows it in line with your Jupyter notebook. If the TensorBoard is already running, it will reuse that instead of duplicating.

```
\Temp\.tensorboard-info directory as well.

In [17]: %tensorboard --logdir ./logs --port 6006
Reusing TensorBoard on port 6006 (pid 4243), started 0:11:21 ago. (Use '!kill 4243' to kill it.)
```

You can use these sliders to see how brightness and contrast affect the images you have logged.

The tag we used for the batch of images we logged

Some more information about the images

A training data batch
tag: A training data batch
sample: 1 of 20
step 0
Wed Aug 04 2021 20:22:54 GMT+1000 (Australian Eastern Standard Time)

A training data batch
tag: A training data batch
sample: 2 of 20
step 0
Wed Aug 04 2021 20:22:54 GMT+1000 (Australian Eastern Standard Time)

Shows the sub-directory you have logged the images to. By having data in sub-directories instead of a single directory, you are introducing structure to the data you are logging.

Figure 1 The TensorBoard visualizing logged images, displayed inline in the Jupyter notebook. You can perform verious operations on images, such as adjusting brightness or contrast

Once the data is clear, the next focus is monitoring model training. A dense neural network with layers of 512, 256, and 10 neurons is trained to classify the clothing samples. Metrics such as accuracy and loss are streamed into TensorBoard, producing evolving line plots. With real-time updates, one can toggle between training and validation results, smooth out fluctuations, or switch the scale to logarithmic for more interpretable long-range trends. Later, a convolutional model with filters of varying kernel sizes is introduced and trained on the same data. Because both models are logged into separate directories, their curves appear on the same dashboard, allowing side-by-side performance comparison. Activation histograms deepen the analysis by showing how neuron outputs distribute over time. Initially scattered, these activations gradually converge into recognizable patterns. A snapshot (Figure 2) reveals histograms stacked across epochs, darker tones representing more recent training, clearly showing convergence behavior.

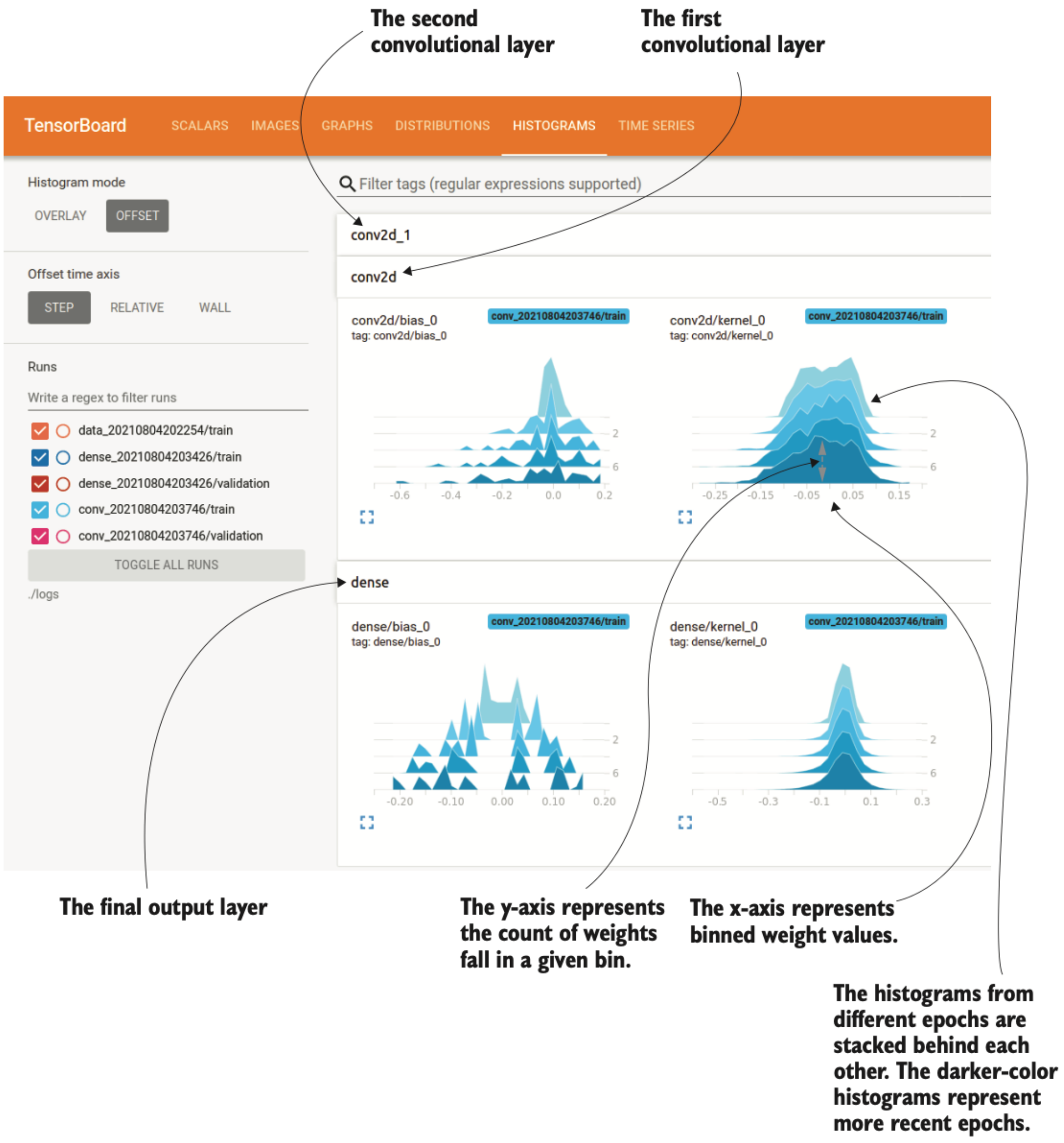
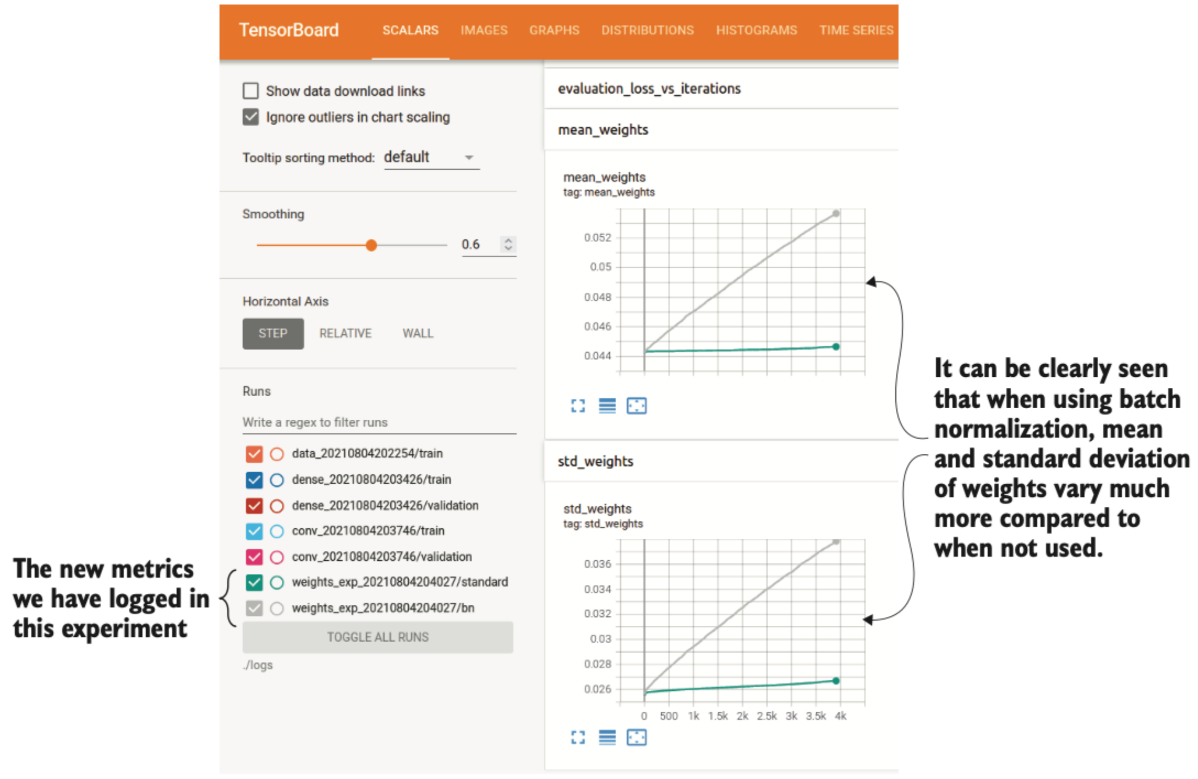

The second convolutional layer
The first convolutional layer
TensorBoard
SCALARS
IMAGES
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
TIME SERIES
Histogram mode
OVERLAY
OFFSET
Offset time axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
data_20210804202254/train
dense_20210804203426/train
dense_20210804203426/validation
conv_20210804203746/train
conv_20210804203746/validation
TOGGLE ALL RUNS
./logs
Filter tags (regular expressions supported)
conv2d_1
conv2d
conv2d/bias_0
tag: conv2d/bias_0
conv2d/kernel_0
tag: conv2d/kernel_0
dense
dense/bias_0
tag: dense/bias_0
dense/kernel_0
tag: dense/kernel_0
The final output layer
The y-axis represents the count of weights fall in a given bin.
The x-axis represents binned weight values.
The histograms from different epochs are stacked behind each other. The darker-color histograms represent more recent epochs.

Figure 2 Activation histograms dipslayed by the TensorBoard. These graphs indicate how the distribution of activations in a given layer changed over time

Not every metric comes prepackaged. Sometimes a researcher may wish to measure internal properties, such as the mean and standard deviation of weight matrices. This is achieved by writing custom summaries within the training loop. By doing so, one can contrast two otherwise identical models—one with batch normalization and one without. Visualizations show how batch normalization introduces greater variability in weights, allowing them to adapt more flexibly. This insight, plotted as separate lines for mean and standard deviation (Figure 3), uncovers dynamics that would remain hidden if only accuracy or loss were tracked.



Ignore outliers in chart scaling ✓

Tooltip sorting method: default

Smoothing 0.6

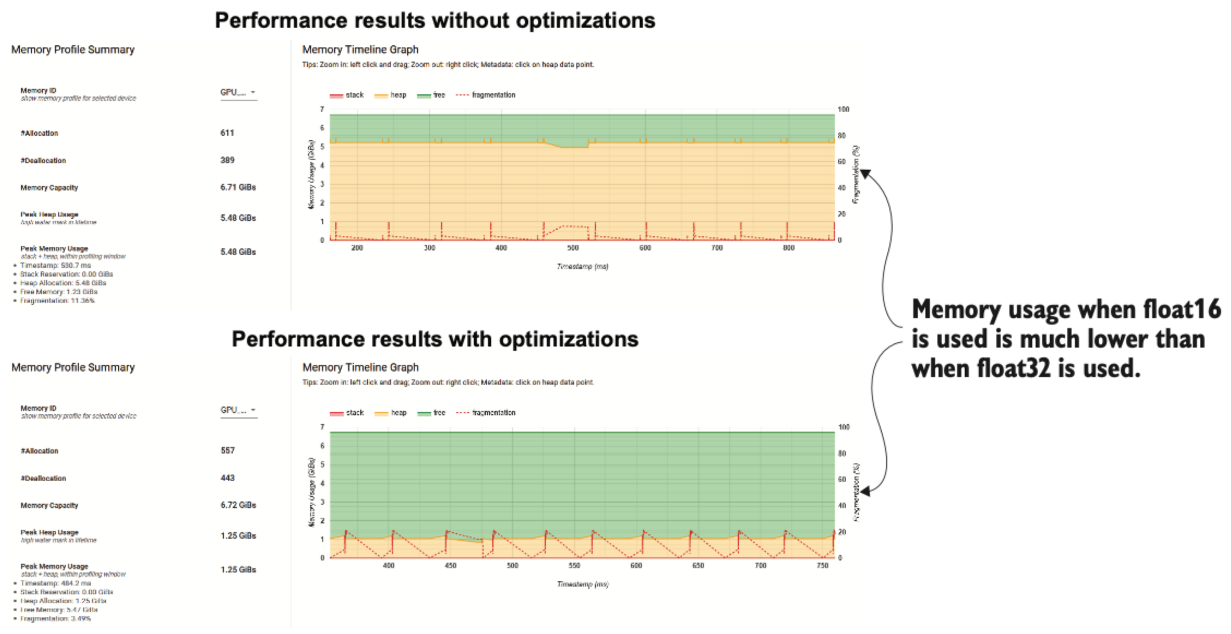
Runs:
- ✓ data_20210804202254/train
- ✓ dense_20210804203426/train
- ✓ dense_20210804203426/validation
- ✓ conv_20210804203746/train
- ✓ conv_20210804203746/validation
- ✓ weights_exp_20210804204027/standard
- weights_exp_20210804204027/bn

The new metrics we have logged in this experiment

It can be clearly seen that when using batch normalization, mean and standard deviation of weights vary much more compared to when not used.

Figure 3 The mean and standard deviation of weights plotted in the TensorBoard

Performance, however, is not just about correctness. Training large networks often consumes significant time and memory, so profiling becomes crucial. Using a flower classification task with 17 categories, a deep convolutional model is trained while TensorBoard dissects the process into subtasks: input data time, kernel launches, GPU compute cycles, and communication overhead. Profiling reveals where most time is spent and offers suggestions—optimize the input pipeline, parallelize mapping, prefetch batches, or adopt mixed precision arithmetic. The shift to 16-bit operations leads to dramatic reductions in memory usage. A comparative memory profile (Figure 4) demonstrates how consumption drops from over 5 GB to nearly 1 GB, a 76% reduction, enabling larger models to fit comfortably within hardware constraints.


Performance results without optimizations
Memory Timeline Graph
Tips: Zoom in: left click and drag; Zoom out: right click; Metadata: click on heap data point.
stack
heap
free
fragmentation
Memory Usage (GiBs)
Fragmentation (%)
Timestamp (ms)
Performance results with optimizations
Memory usage when float16 is used is much lower than when float32 is used.

Memory Profile Summary

Memory ID *show memory profile for selected device*	GPU_...
#Allocation	611
#Deallocation	389
Memory Capacity	6.71 GiBs
Peak Heap Usage *high water mark in lifetime*	5.48 GiBs
Peak Memory Usage *stack + heap, within profiling window*	5.48 GiBs

- Timestamp: 530.7 ms
- Stack Reservation: 0.00 GiBs
- Heap Allocation: 5.48 GiBs
- Free Memory: 1.23 GiBs
- Fragmentation: 11.36%

Memory Profile Summary

Memory ID *show memory profile for selected device*	GPU_...
#Allocation	557
#Deallocation	443
Memory Capacity	6.72 GiBs
Peak Heap Usage *high water mark in lifetime*	1.25 GiBs
Peak Memory Usage *stack + heap, within profiling window*	1.25 GiBs

- Timestamp: 484.2 ms
- Stack Reservation: 0.00 GiBs
- Heap Allocation: 1.25 GiBs
- Free Memory: 5.47 GiBs
- Fragmentation: 3.49%

Figure 4 Memory profile with and without the optimizations. The difference from using 16-bit operations is very clear, as it has reduced the memory consumption of the model significantly

By combining visualization, monitoring, custom logging, and profiling, TensorBoard evolves into more than a diagnostic panel—it becomes an investigative instrument. It not only confirms whether a model is improving but also reveals how weights evolve, where computation stalls, and how memory is consumed. Its interactive views guide practitioners to refine data pipelines, tune architectures, and adopt efficient training techniques. In doing so, it transforms the invisible workings of neural networks into a transparent and navigable landscape, ensuring that the complex process of deep learning remains understandable, measurable, and ultimately controllable.