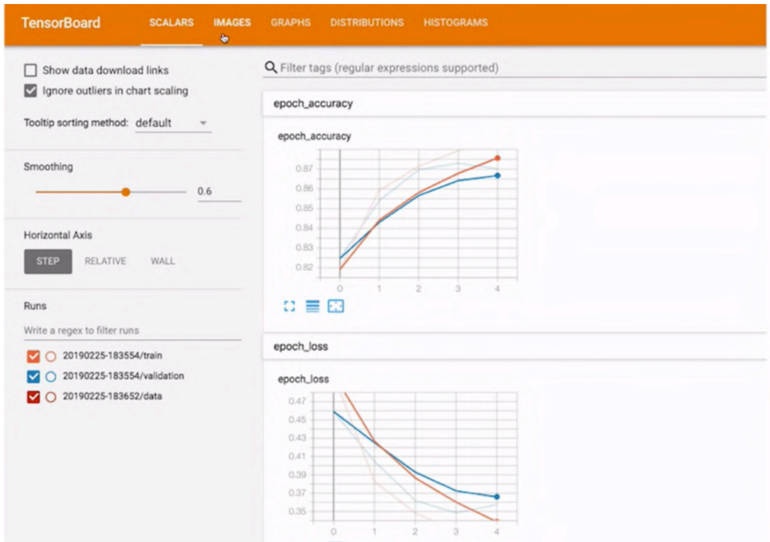
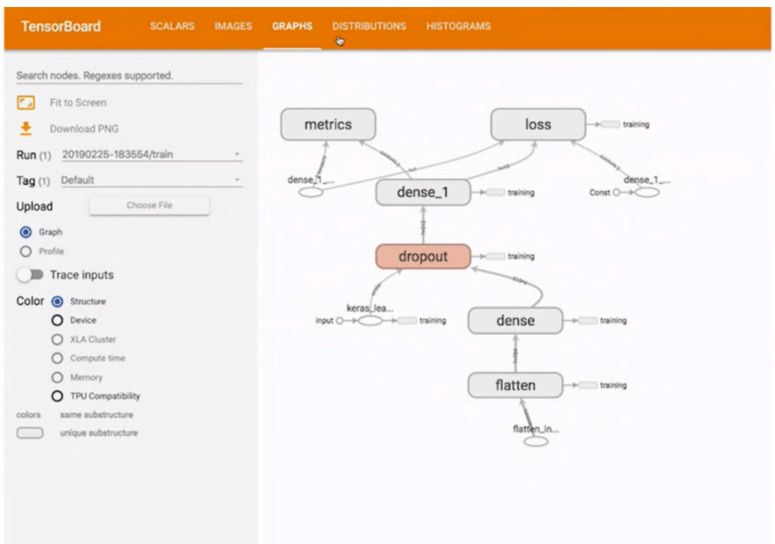
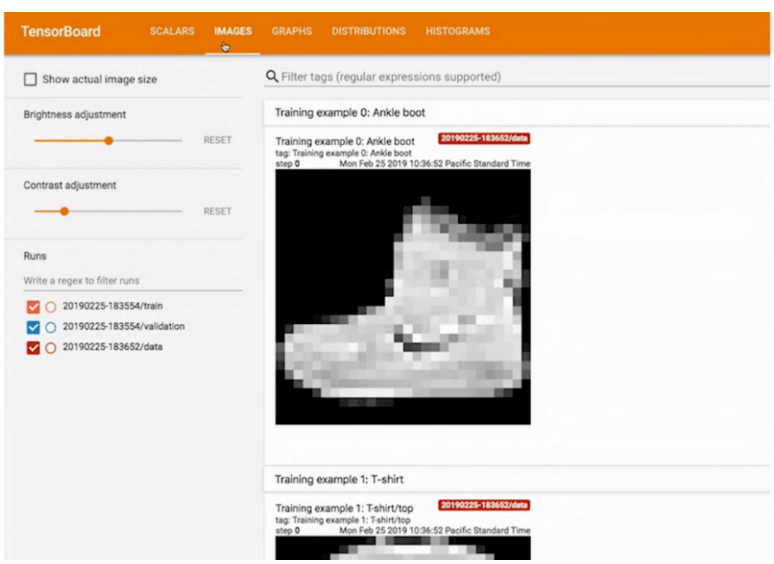






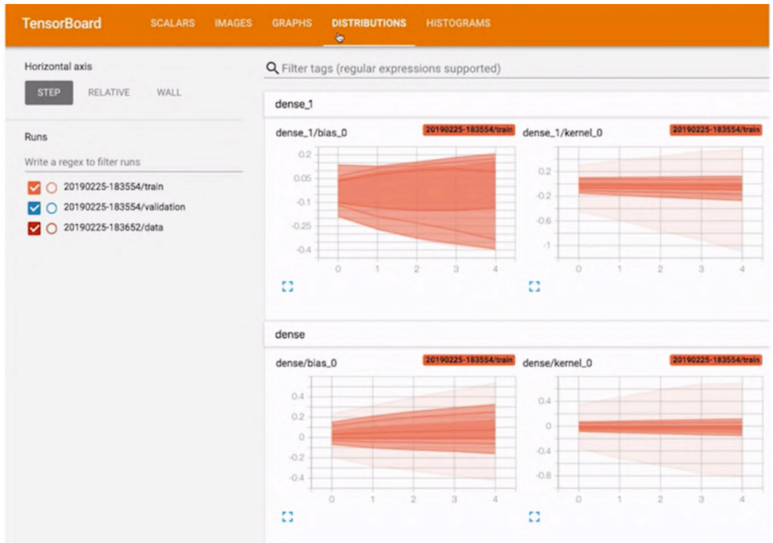
In machine learning, to improve your model, you often need to measure details about it. Maybe it's a curve of loss or accuracy over time or maybe it's visualizing the model graph, projecting learned embeddings, or just understanding the change in parameters over time.

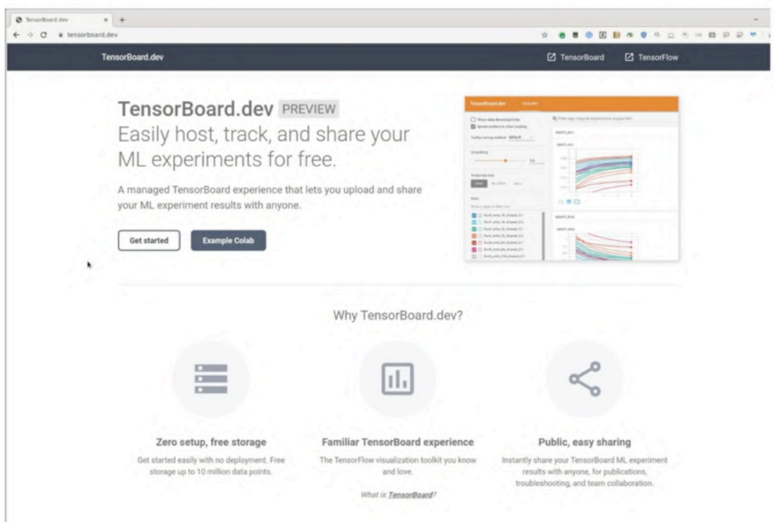
With this in mind, the TensorBoard tool is available, and with the TensorBoard.dev website, you can deploy the data of your model to the web so that you can share it with others to explore your discoveries or maybe even help debug your problems.

So this week, we'll learn all about TensorBoard, starting with that hosted service, before going into learning how to host it for yourself. Tensorboard provides you with a browser-based visualization of data about your network. This includes scalars like the accuracy and loss over time while training, as well as images, so you can visualize your data, graphs of the makeup of your neural network and much more.

In this lesson, we'll focus on the scalars to get you started. So to get started with the hosted version of TensorBoard, you can visit TensorBoard.dev.

From here, you can learn how to deploy your model data so that it can be shared with the rest of the world.





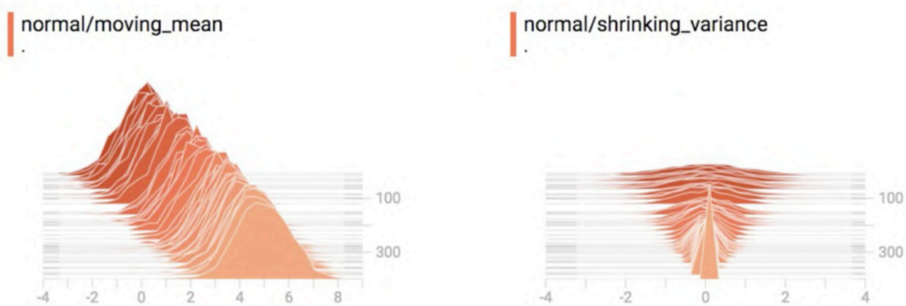

```
model = create_model()

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

log_dir="logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard_callback =
      tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

model.fit(x=x_train,
          y=y_train,
          epochs=5,
          validation_data=(x_test, y_test),
          callbacks=[tensorboard_callback])
```

It's important to understand how TensorBoard accesses the data about your model and the metadata about the training. These are all done from the log files that are written by TensorFlow.

The easiest way to do this is using callbacks. So consider a simple training scenario like this one where we'll train a model for five epochs. Note that in the model.fit, we have a TensorBoard callback, which you can see declared here.

```python
model = create_model()

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

log_dir="logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard_callback =
      tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)


model.fit(x=x_train,
          y=y_train,
          epochs=5,
          validation_data=(x_test, y_test),
          callbacks=[tensorboard_callback])
```

Instead of a Python function as you may have seen in other callbacks, it's an instance of tf.keras.callbacks.TensorBoard. This will write out the logs to the specified log directory as well as maybe some other content such as the histogram frequency.



In TensorBoard, a histogram is a visualization of weights over time. It can be a handy tool to help you see if the weights initialization or the changes because of learning are causing issues.

There are three dimensions in the diagram. As you move up the screen, you go back in time, ie, you're looking at the most recent epoch first. Then each epoch is a standard 2D chart of x against whatever value you're charting

```python
model = create_model()

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

log_dir="logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard_callback =
      tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)


model.fit(x=x_train,
          y=y_train,
          epochs=5,
          validation_data=(x_test, y_test),
          callbacks=[tensorboard_callback])
```

So when we set the histogram frequency to one as here, it's asking the TensorBoard logs for histograms to be written so that there's one chart per epoch.

```
model = create_model()

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

log_dir="logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard_callback =
      tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)


model.fit(x=x_train,
          y=y_train,
          epochs=5,
          validation_data=(x_test, y_test),
          callbacks=[tensorboard_callback])
```

You also need to specify the log directory. And you can do that by creating a directory using a timestamp so that you can keep track of different training runs and load them separately into TensorBoard should you

```
..
logs
  fit
    20191114-214911
      train
        plugins
        events.out.tfevents.1573768151.df93b15b9686.127.277.v2
        events.out.tfevents.1573768152.df93b15b9686.profile-empty
      validation
        events.out.tfevents.1573768163.df93b15b9686.127.7462.v2
```

So in this case, the logs directory will have a timestamp subdirectory and it will be creative with training and validation subdirectories written into that. When these are pointed at TensorBoard, it will chart the details in there.

```
!tensorboard dev upload --logdir ./logs
```

One of the great things with TensorBoard.dev is how it provides a hosted environment where you can publish details about your model. We'll take a look at how to do that next. To upload your logs to TensorBoard.dev, it's as simple as this from Colab, pointing the log directory switch at your logs.

```
***** TensorBoard Uploader *****

This will upload your TensorBoard logs to https://tensorboard.dev/ from
the following directory:

./logs

This TensorBoard will be visible to everyone. Do not upload sensitive
data.

Your use of this service is subject to Google's Terms of Service
<https://policies.google.com/terms> and Privacy Policy
<https://policies.google.com/privacy>, and TensorBoard.dev's Terms of Service
<https://tensorboard.dev/policy/terms/>.

This notice will not be shown again while you are logged into the uploader.
To log out, run `tensorboard dev auth revoke`.

Continue? (yes/NO)
```

```
Please visit this URL to authorize this application: https://accounts.google.com/o/oauth2/auth?response_type=code&client_
Enter the authorization code:
```

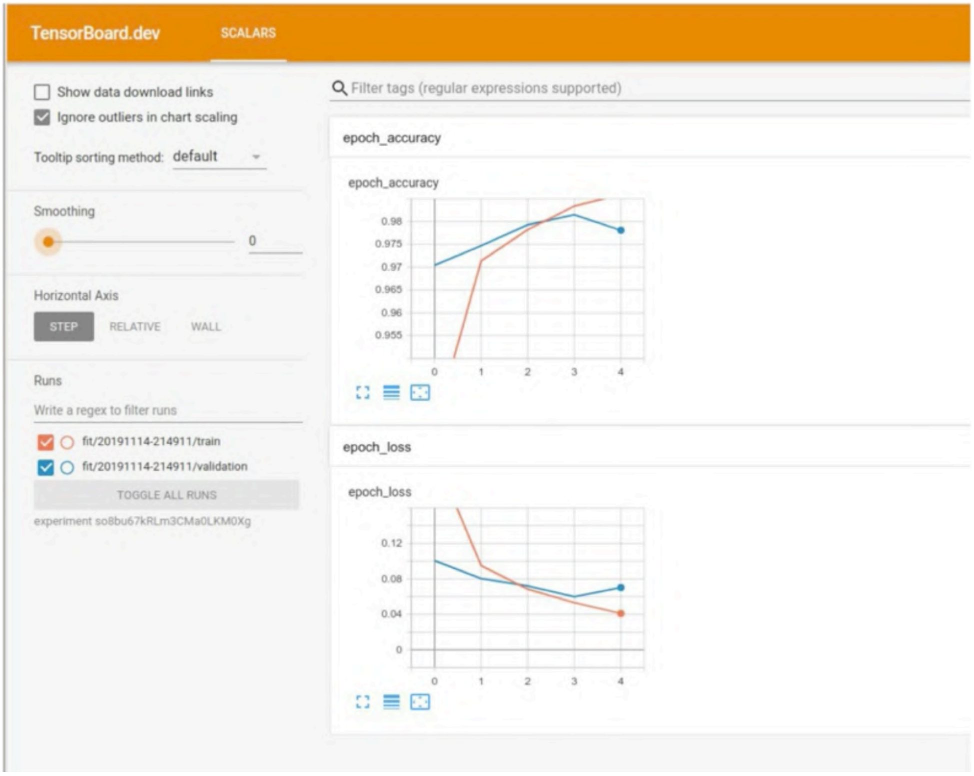
Google

Sign in

Please copy this code, switch to your application and paste it there:

```
4/tQGjxD0nqWvvj6-L5hNGA-
L8zorPiNRdYFUfVFYHDwa-jaSVQVNRGYE
```

```
Upload started and will continue reading any new data as it's added
to the logdir. To stop uploading, press Ctrl-C.
View your TensorBoard live at: https://tensorboard.dev/experiment/so8bu67kRLm3CMa0LKM0Xg
```




```
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

def create_model():
  return tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
  ])
```

We saw how he could create logs about our training and use the hosted TensorBoard service to share some of the details about that training with the world via a URL. Not all services on TensorBoard are available on TensorBoard.dev yet. So in this video, we'll take a look at the local TensorBoard so that you can explore more data. Let's start with a simple network for classifying mnist handwritten digits. We'll create some keras.layers to do it.

```
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

def create_model():
  return tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
  ])
```

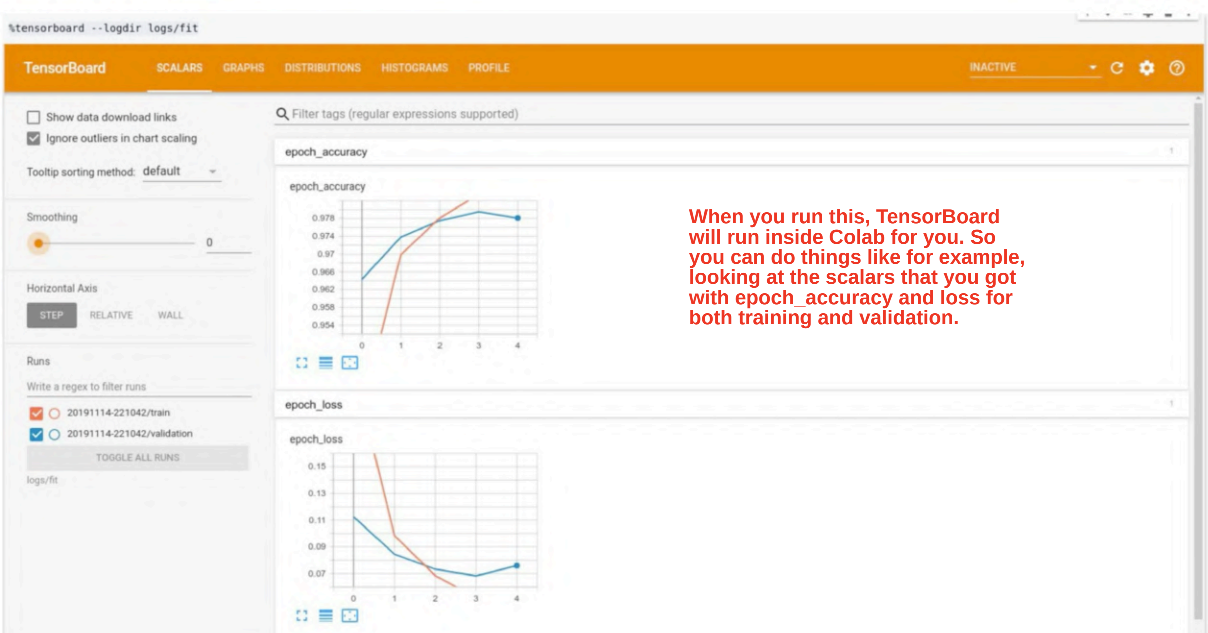
These will include some densely connected neurons using the Dense type and a Dropout layer.

```
log_dir="logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard_callback =
          tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)


model.fit(x=x_train,
          y=y_train,
          epochs=5,
          validation_data=(x_test, y_test),
          callbacks=[tensorboard_callback])
```

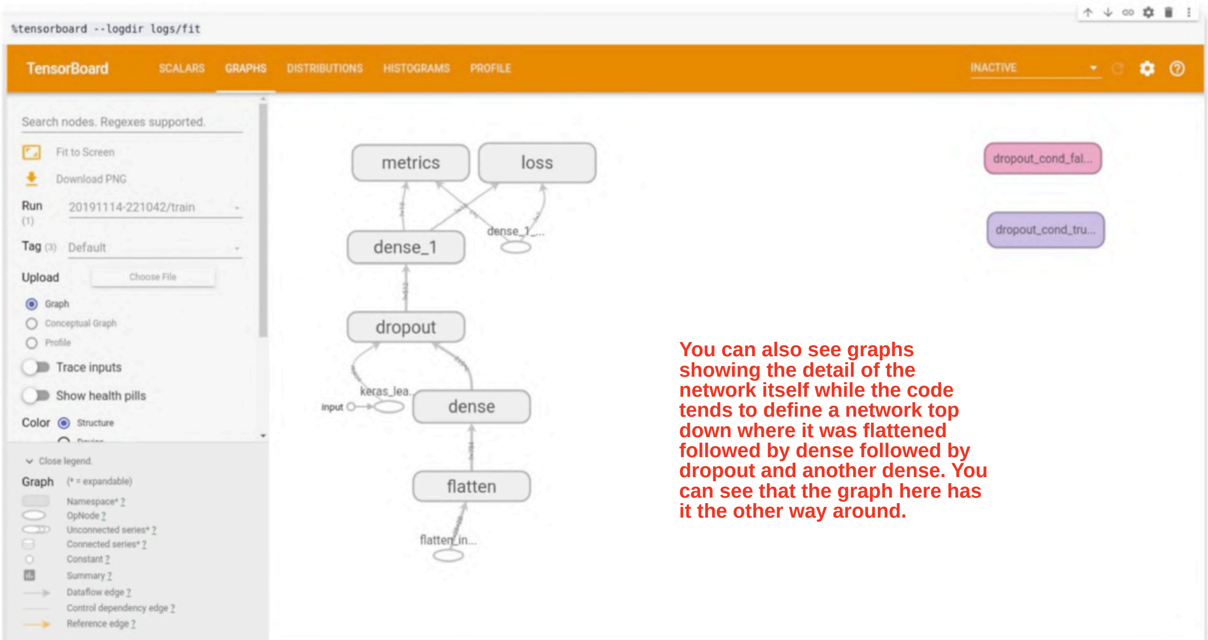
Then, when we train the network with model.fit, we can specify the tensorboard_callback to write the details to the log directories. You can see that here where we define a callback as an instance of tf.keras.callbacks.TensorBoard. And we specify this is the callback to use during model.fit.

```
log_dir="logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard_callback =
          tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)


model.fit(x=x_train,
          y=y_train,
          epochs=5,
          validation_data=(x_test, y_test),
          callbacks=[tensorboard_callback])


%tensorboard --logdir logs/fit
```

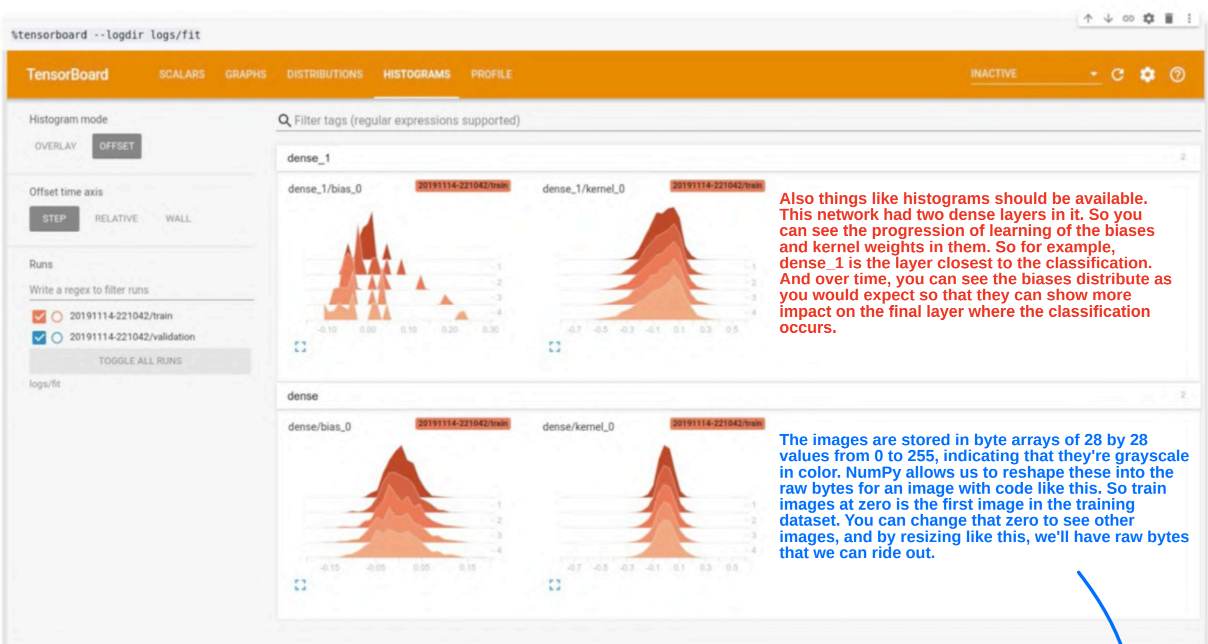
So then in order to get TensorBoard, it's as simple as calling it and specifying the directory that you created to hold the logs as you can see here.



When you run this, TensorBoard will run inside Colab for you. So you can do things like for example, looking at the scalars that you got with epoch_accuracy and loss for both training and validation.



You can also see graphs showing the detail of the network itself while the code tends to define a network top down where it was flattened followed by dense followed by dropout and another dense. You can see that the graph here has it the other way around.



Also things like histograms should be available. This network had two dense layers in it. So you can see the progression of learning of the biases and kernel weights in them. So for example, dense_1 is the layer closest to the classification. And over time, you can see the biases distribute as you would expect so that they can show more impact on the final layer where the classification occurs.

The images are stored in byte arrays of 28 by 28 values from 0 to 255, indicating that they're grayscale in color. NumPy allows us to reshape these into the raw bytes for an image with code like this. So train images at zero is the first image in the training dataset. You can change that zero to see other images, and by resizing like this, we'll have raw bytes that we can ride out.

```
fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) =
    fashion_mnist.load_data()

img = np.reshape(train_images[0], (-1, 28, 28, 1))
```

```
# Sets up a timestamped log directory.
logdir = "logs/train_data/" +
         datetime.now().strftime("%Y%m%d-%H%M%S")

# Creates a file writer for the log directory.
file_writer = tf.summary.create_file_writer(logdir)

# Using the file writer, log the reshaped image.
with file_writer.as_default():
  tf.summary.image("Training data", img, step=0)
```

To writes out the image, we need to specify a log directory, create a file writer for it, and then write the image. This is different than the callback based logging that we did earlier. We have to set up the file writer ourselves, for example.

So here we have code which just specifies a log directory as logs/training data and is then followed by a timestamp. This allows us to separate different log directories if we want, which is particularly useful when we want to track multiple training loops.

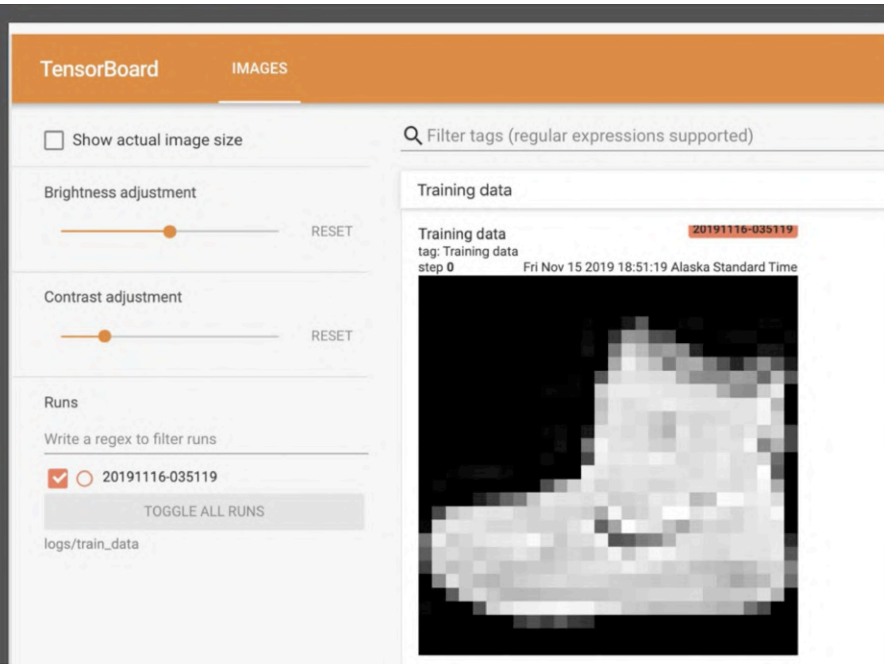
```
# Sets up a timestamped log directory.
logdir = "logs/train_data/" +
         datetime.now().strftime("%Y%m%d-%H%M%S")

# Creates a file writer for the log directory.
file_writer = tf.summary.create_file_writer(logdir)

# Using the file writer, log the reshaped image.
with file_writer.as_default():
  tf.summary.image("Training data", img, step=0)
```

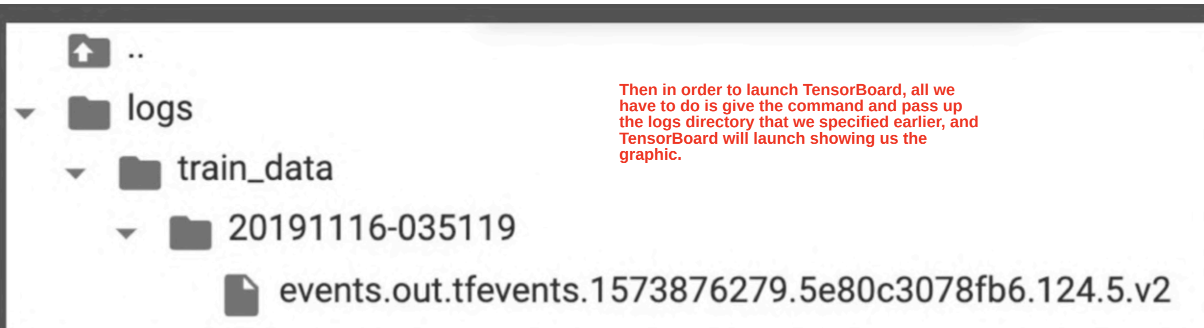
Then we'll create a tf.summary file writer and point it at this directory. The tf.summary module gives us APIs for writing summary data which can be visualized intensive board. There are lots of different data types that you can write, including scalars, audio, images, texts, and more. But for now, we're interested in images.

```
# Sets up a timestamped log directory.
logdir = "logs/train_data/" +
         datetime.now().strftime("%Y%m%d-%H%M%S")

# Creates a file writer for the log directory.
file_writer = tf.summary.create_file_writer(logdir)

# Using the file writer, log the reshaped image.
with file_writer.as_default():
  tf.summary.image("Training data", img, step=0)
```

So you can see that here that using the File Writer, we can simply output the image by calling tf.summary.image, passing it the image that we created a moment ago, by reshaping the data using NumPy.

The logs get written as a binary file containing lots of information such as the scalars we saw earlier. So don't expect the raw images to be in there. The logs will look something like this.





Then in order to launch TensorBoard, all we have to do is give the command and pass up the logs directory that we specified earlier, and TensorBoard will launch showing us the graphic.

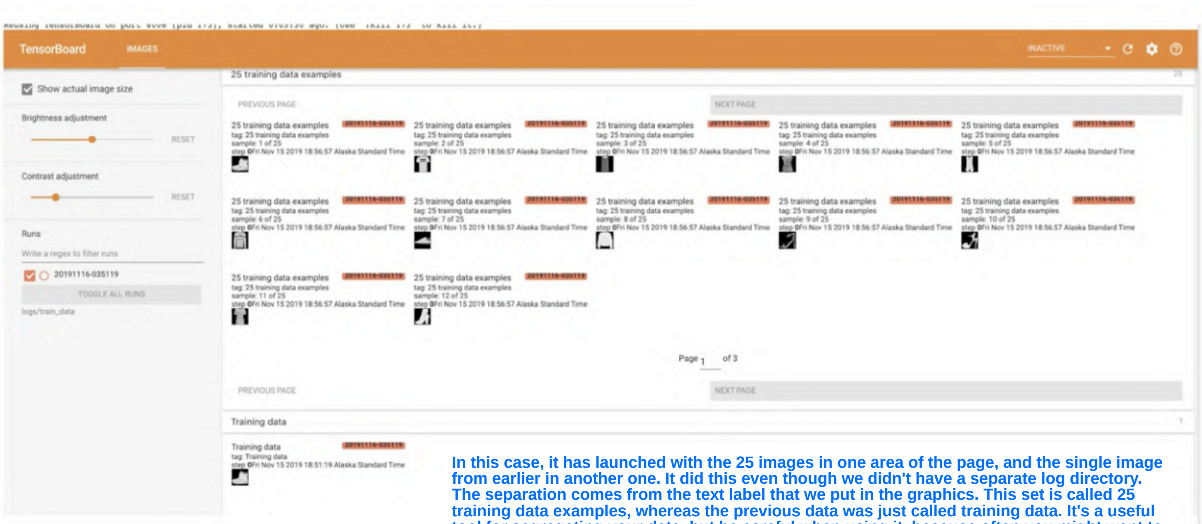
```
%tensorboard --logdir logs/train_data
```

```
with file_writer.as_default():
  # Don't forget to reshape.
  images = np.reshape(train_images[0:25], (-1, 28, 28, 1))
  tf.summary.image("25 training data examples", images,
                   max_outputs=25, step=0)

%tensorboard --logdir logs/train_data
```

if you want more than one image that's pretty easy too. NumPy arrays reshape will allow us to specify more than one image that we want to reshape. So in this case, we can take the first 25 images in the data set, and load them into the collection called images.

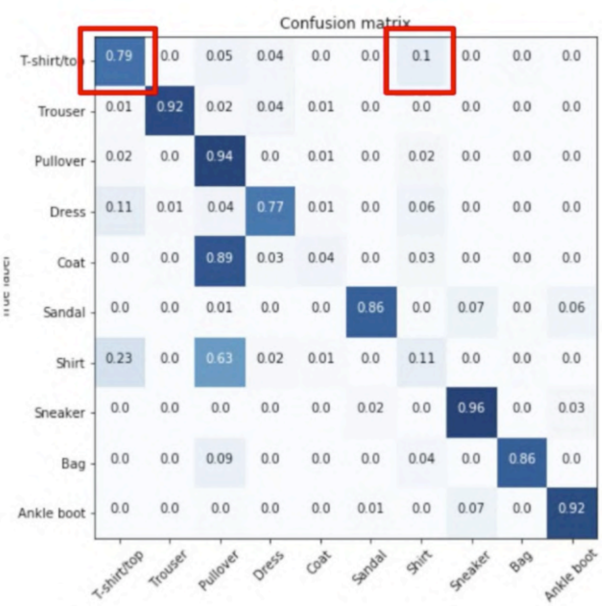
```
with file_writer.as_default():
  # Don't forget to reshape.
  images = np.reshape(train_images[0:25], (-1, 28, 28, 1))
  tf.summary.image("25 training data examples", images,
                   max_outputs=25, step=0)

%tensorboard --logdir logs/train_data
```

When we write out the images collection, we just specify the maximum number of outputs that we want to write, in this case 25. When we launched TensorBoard, it will do the rest.

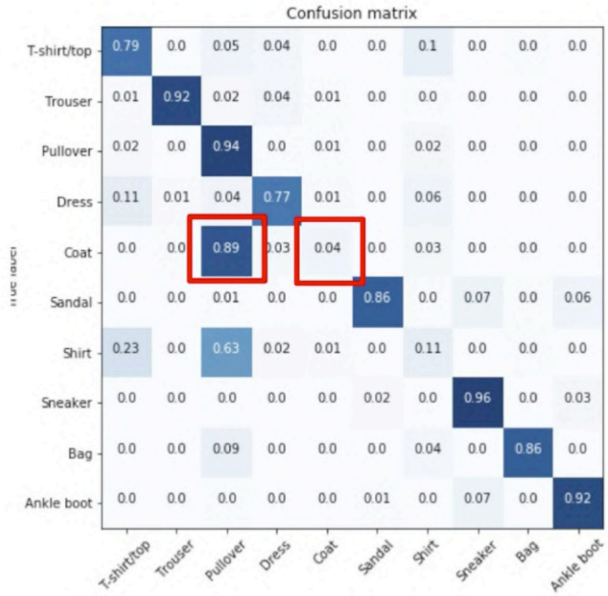


In this case, it has launched with the 25 images in one area of the page, and the single image from earlier in another one. It did this even though we didn't have a separate log directory. The separation comes from the text label that we put in the graphics. This set is called 25 training data examples, whereas the previous data was just called training data. It's a useful tool for segregating your data, but be careful when using it, because often you might want to segregate based on different training runs instead.

A confusion Matrix adds an additional layer to your debugging of your classification. It could look something like this, where the y axis is the actual label of an object and the x axis is the predicted label. A perfect system would have the diagonal from the top left to the bottom right all at 1.0 and everything else at 0. But if you have a perfect classifier, something else is likely wrong.



Here you can see that it got the T-shirt correct, 79% of the time and when it got it wrong, it confused the T-shirt with a shirt, which is pretty reasonable.



The MatPlotLib file format cannot be logged as an image, but the PNG file format can be logged. We'll convert the image into the PNG format.

```
def plot_to_image(figure):
  """Converts the matplotlib plot specified by 'figure' to a PNG image and
  returns it. The supplied figure is closed and inaccessible after this call."""
  # Save the plot to a PNG in memory.
  buf = io.BytesIO()
  plt.savefig(buf, format='png')
  # Closing the figure prevents it from being displayed directly inside
  # the notebook.
  plt.close(figure)
  buf.seek(0)
  # Convert PNG buffer to TF image
  image = tf.image.decode_png(buf.getvalue(), channels=4)
  # Add the batch dimension
  image = tf.expand_dims(image, 0)
  return image
```

The key here is the tensorflow tools to decode the raw bytes into an image using tf,image decode PNG. Once we have that, then we have to expand the dimensions on it to turn it into a raw image buffer, which can then be returned to the caller

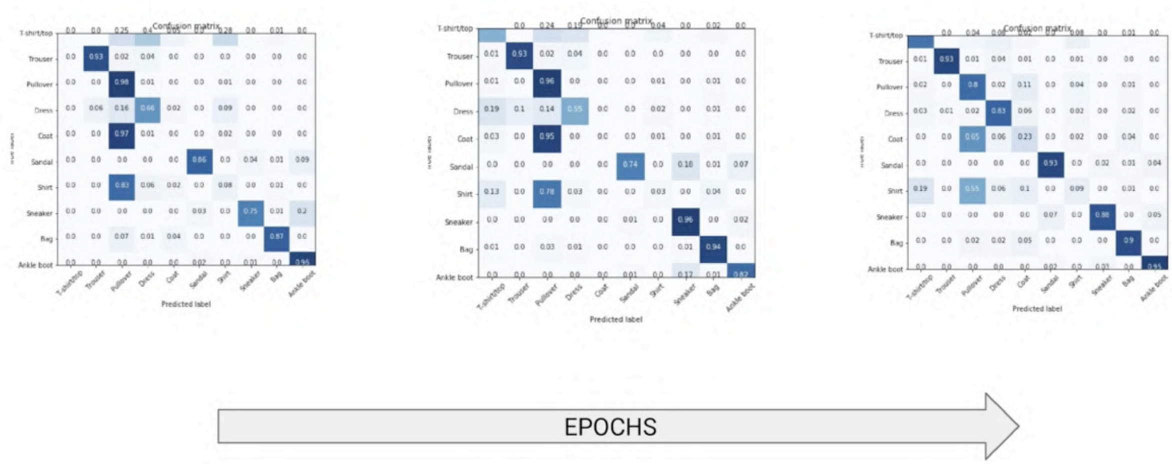
```
# Train the classifier.
model.fit(
    train_images,
    train_labels,
    epochs=5,
    verbose=0, # Suppress chatty output
    callbacks=[tensorboard_callback, cm_callback],
    validation_data=(test_images, test_labels),
)
```

So to plot the confusion matrix at the end of each epoch and log epoch based stuff to tensorboard, we'll go back to using callbacks and a really useful technique you can use is multiple_callbacks. So we'll do one for logging the scalars and another one to plot the confusion matrix and write that out and this is called cm_callback.

```python
def log_confusion_matrix(epoch, logs):
  # Use the model to predict the values from the validation dataset.
  test_pred_raw = model.predict(test_images)
  test_pred = np.argmax(test_pred_raw, axis=1)

  # Calculate the confusion matrix.
  cm = sklearn.metrics.confusion_matrix(test_labels, test_pred)
  # Log the confusion matrix as an image summary.
  figure = plot_confusion_matrix(cm, class_names=class_names)
  cm_image = plot_to_image(figure)

  # Log the confusion matrix as an image summary.
  with file_writer_cm.as_default():
    tf.summary.image("Confusion Matrix", cm_image, step=epoch)

# Define the per-epoch callback.
cm_callback = keras.callbacks.LambdaCallback(on_epoch_end=log_confusion_matrix)
```

So we'll define this callback, cm_callback, using a lambda callback type. This allows us to execute arbitrary code and we'll specify that this code would execute on epoch end in a function called log_confusion matrix,


```python
def log_confusion_matrix(epoch, logs):
  # Use the model to predict the values from the validation dataset.
  test_pred_raw = model.predict(test_images)
  test_pred = np.argmax(test_pred_raw, axis=1)

  # Calculate the confusion matrix.
  cm = sklearn.metrics.confusion_matrix(test_labels, test_pred)
  # Log the confusion matrix as an image summary.
  figure = plot_confusion_matrix(cm, class_names=class_names)
  cm_image = plot_to_image(figure)

  # Log the confusion matrix as an image summary.
  with file_writer_cm.as_default():
    tf.summary.image("Confusion Matrix", cm_image, step=epoch)

# Define the per-epoch callback.
cm_callback = keras.callbacks.LambdaCallback(on_epoch_end=log_confusion_matrix)
```



bit.ly/tensorboard-graphics

Now when we train our network using the code we saw earlier for five epochs, we'll get five confusion matrices and we can see the evolution of the confusion matrix over time, which is another great way of seeing how your network is learning and much deeper than just looking at accuracy and loss. If you want to try this notebook out for yourself where you'll plot a single image, multiple images, custom plots and finally confusion matrices, you can find it at this URL.