Deep Learning

Practical Exercises - Week 9

Introduction to TensorBoard

ICAI

FS 2024

1 Introduction

In lab 7 and lab 8 we learned the basics of TensorFlow and implemented our first neural network which we trained using the `tf.GradientTape` context manager. As neural networks become more advanced and training steps become more complex, we need a tool to help us get an overview of our network and track the training process. This is where TensorBoard comes into play.

This lab follows this TensorBoard tutorial.

2 TensorBoard: Visualizing Learning

The computations you'll use TensorFlow for - like training a massive deep neural network - can be complex and confusing. To make it easier to understand, debug, and optimize TensorFlow programs, a suite of visualization tools called TensorBoard has been included. You can use TensorBoard to visualize your TensorFlow graph, plot quantitative metrics during the execution of your graph, and show additional data like images. When TensorBoard is fully configured, it looks similar to Figure 1.

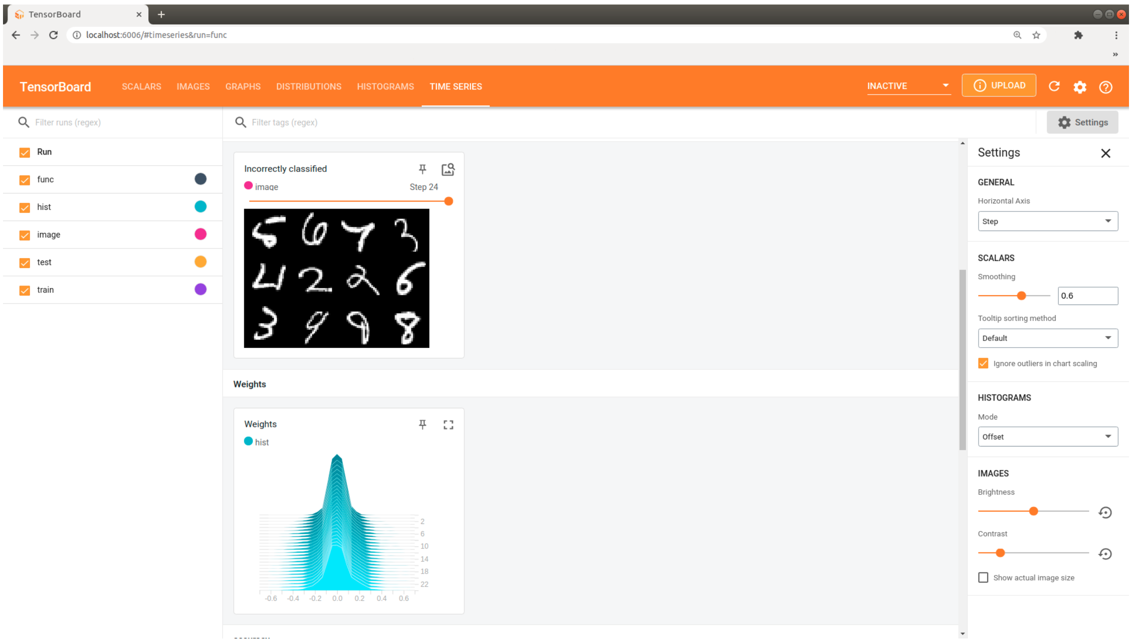

Figure 1: Fully configured TensorBoard.

This tutorial introduces you to the basic use of TensorBoard. For this purpose we use the simple softmax regression model which we created and trained in lab 8.

The script we work with is the shown below (`tensorboard_example.py`). Try to understand this training script, how it works and the meaning of each command. Please note that this is a minimal training script, containing only the necessary TensorFlow commands. Normally, a training routine also contains a validation part.

```python
import tensorflow as tf
from tensorflow.keras.layers import Dense, Softmax
from tensorflow.keras import Model

# Load and prepare data
##############################################################################

# Load MNIST data
mnist = tf.keras.datasets.mnist
(x_train, y_train), _ = mnist.load_data()

# Scale images
x_train = x_train / 255.0

# Flatten images
x_train = x_train.reshape([len(x_train), -1]).astype("float32")

# Convert labels to one-hot tensor
y_train = tf.one_hot(y_train, 10)

# Create datasets
MINI_BATCH_SIZE = 32
train_ds = (
    tf.data.Dataset.from_tensor_slices((x_train, y_train))
    .shuffle(10000)
    .batch(MINI_BATCH_SIZE)
```

```
)

# Create model, define optimizer, loss and metrics
##############################################################################


class MyModel(Model):
    def __init__(self, name=None, **kwargs):
        super(MyModel, self).__init__(name=name, **kwargs)
        self.d1 = Dense(10, use_bias=True, name="Dense_1")
        self.s1 = Softmax(name="Softmax_1")

    @tf.function
    def call(self, x, training=False):
        x = self.d1(x)
        out = self.s1(x)
        return out


# Create an instance of the model
model = MyModel(name="MNISTClassifier")

# Choose an optimizer and loss function for training
loss_object = tf.keras.losses.CategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)

# Select metrics to measure the loss
train_loss = tf.keras.metrics.Mean(name="train_loss")

# Define train step
##############################################################################


# Use tf.GradientTape to train the model
@tf.function
def train_step(images, labels):
    # Collect gradients of trainable variables
    with tf.GradientTape() as tape:
        predictions = model(images, training=True)
        loss = loss_object(labels, predictions)

    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    train_loss(loss)

# Run training
##############################################################################

EPOCHS = 25
for epoch in range(EPOCHS):
    # Reset the metrics at the start of the next epoch
    train_loss.reset_state()

    # Train
    for images, labels in train_ds:
        train_step(images, labels)

```

```
    print(
        "Epoch {:2d}, ".format(epoch + 1),
        "Loss: {:3.3f}, ".format(train_loss.result())
    )
```

2.1 Scalars and Metrics

Machine learning invariably involves understanding key metrics such as loss and how they change as training progresses. These metrics can help you understand if you're overfitting, for example, or if you're unnecessarily training for too long. You may want to compare these metrics across different training runs to help debug and improve your model.

TensorBoard's Scalars Dashboard allows you to visualize these metrics using a simple API with very little effort. This tutorial presents very basic examples to help you learn how to use these APIs with TensorBoard when developing your Keras model. You will learn how to use the TensorFlow Summary APIs to visualize custom scalars.

In general, to log a custom scalar, you need to use `tf.summary.scalar()` with a file writer. The file writer is responsible for writing data for this run to the specified directory and is used as context manager when you call the `tf.summary.scalar()`.

Set up summary writers to write the summaries to disk in a logs directory:

```
scalar_log_dir = "logs/1_run/scalar"
scalar_writer = tf.summary.create_file_writer(scalar_log_dir)
```

Start training. Use `tf.summary.scalar()` to log metrics (training loss) during training/testing within the scope of the summary writers to write the summaries to disk. You have control over which metrics to log and how often to do it.

```
# Run training
#############################################################################

EPOCHS = 25
for epoch in range(EPOCHS):
    # Reset the metrics at the start of the next epoch
    train_loss.reset_state()

    # Train
    for images, labels in train_ds:
        train_step(images, labels)

    # Write scalars to TensorBoard
    with scalar_writer.as_default():
        tf.summary.scalar("loss", train_loss.result(), step=epoch)

    print(
        "Epoch {:2d}, ".format(epoch + 1),
        "Loss: {:3.3f}, ".format(train_loss.result())
    )
```

Executing this script creates a new subdirectory named `logs` where an event file is stored. To load the event file into TensorBoard execute the following command in the shell [1]

```
tensorboard --logdir logs
```

where logdir points to the directory in which the event file is stored. Once TensorBoard is running, navigate your web browser to `localhost:6006` to view the scalar value in TensorBoard. The plot should look similar to the one in Figure 2.

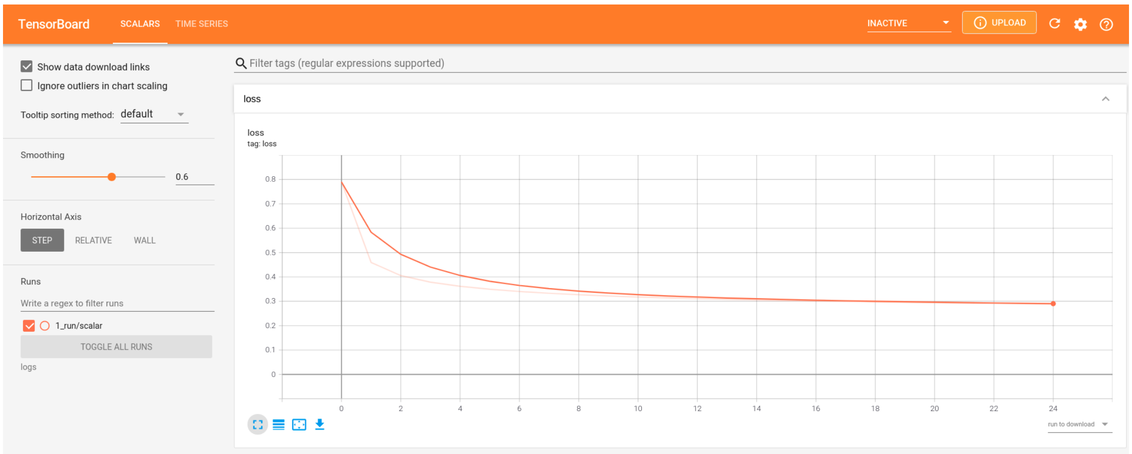

Figure 2: Line plot of training loss visualized in TensorBoard

The Scalars dashboard shows how the loss and metrics change with every epoch. You can use it to also track training speed, learning rate, and other scalar values. You can also compare this run's training and validation loss curves against your earlier runs.

2.2 Graphs of tf.functions

TensorBoard's Graphs dashboard is a powerful tool for examining your TensorFlow model. You can quickly view a conceptual graph of your model's structure and ensure it matches your intended design.

This tutorial presents a quick overview of how to generate a graph visualization of an autographed TensorFlow function. For these situations, you use TensorFlow Summary Trace API.

To use the Summary Trace API:

- Define and annotate a function with `tf.function`
- Use `tf.summary.trace_on()` immediately before your function call site (where function is traced)

[1]cmd.exe on Windows or terminal on Linux.

- With a Summary file writer, call `tf.summary.trace_export()` to save the log data
- You can then use TensorBoard to see how your function behaves

This is how we visualize our model `MNISTClassifier` using the Summary Trace API:

```
func_log_dir = "logs/1_run/func"
graph_writer = tf.summary.create_file_writer(func_log_dir)

# Create graph
tf.summary.trace_on(graph=True)

# Call only one tf.function when tracing
dummy_batch = next(train_ds.as_numpy_iterator())
model(dummy_batch[0])

with graph_writer.as_default():
    tf.summary.trace_export(name="MNISTClassifier", step=0)
tf.summary.trace_off()
```

You can now see the structure of your function as understood by TensorBoard.

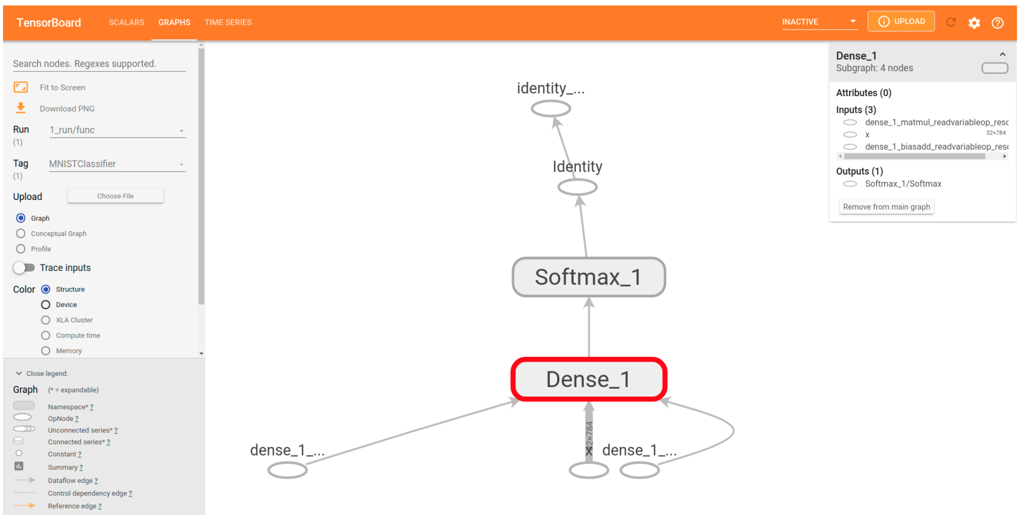

Figure 3: Graph visualization of autographed TensorFlow function

Graphs are often very large, so you can manipulate the graph visualization:

- Scroll to zoom in and out
- Drag to pan
- Double clicking toggles node expansion (a node can be a container for other nodes)
- You can also see metadata by clicking on a node. This allows you to see inputs, outputs, shapes and other details.

2.3 Histogram

In some cases it might be useful to visualize the distribution of the trainable weights / bias or the gradient magnitudes as a histogram over time. This works similar to the scalar visualization described in subsection 2.1 but with tensors of any dimension.

As a first step, you need to create a file writer that will act as a context manager when writing the value distribution of a particular tensor to an event file. This is what the code looks like:

```
hist_log_dir = "logs/1_run/hist"
hist_writer = tf.summary.create_file_writer(hist_log_dir)
```

```
EPOCHS = 25
for epoch in range(EPOCHS):
    # Reset the metrics at the start of the next epoch
    train_loss.reset_state()

    # Train
    for images, labels in train_ds:
        train_step(images, labels)

    # Write histogram to TensorBoard
    with hist_writer.as_default():
        tf.summary.histogram("Weights", model.trainable_weights[0], step=epoch)
        tf.summary.histogram("Bias", model.trainable_weights[1], step=epoch)

    print(
        "Epoch {:2d}, ".format(epoch + 1),
        "Loss: {:3.3f}, ".format(train_loss.result())
    )
```

All trainable variables of a model can be accessed by invoking the method `trainable_weights()`. In this simple example of a softmax regression model we only have two trainable tensors (weight matrix and bias vector). However, for larger neural networks it might be difficult to distinguish between weights and bias.

The TensorBoard visualization of the value distribution looks as depicted in Figure 4.

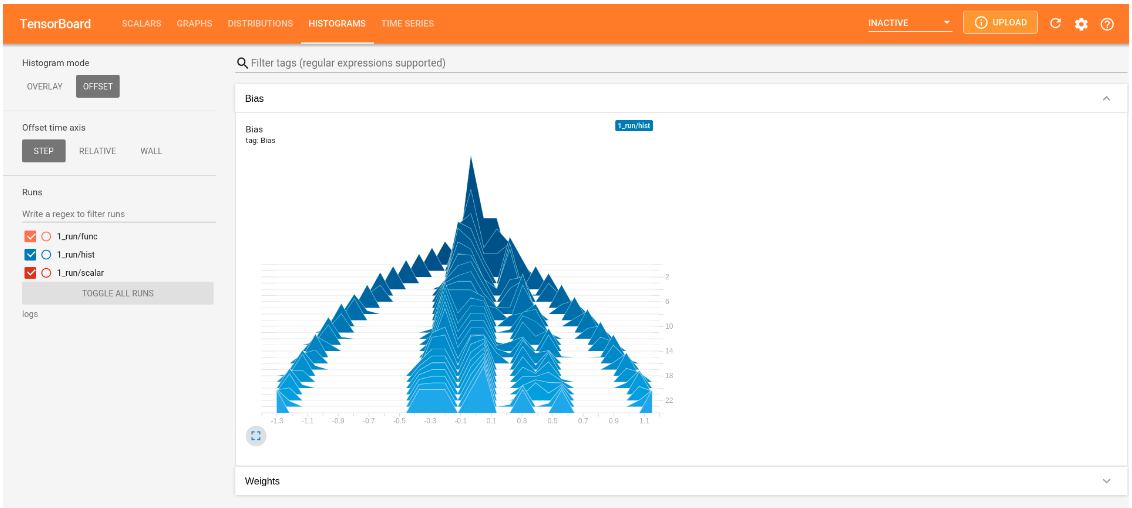

Figure 4: Visualization of a value distribution as histogram in TensorBoard

2.4 Images

Using the TensorFlow Image Summary API, you can easily log tensors and arbitrary images and view them in TensorBoard. This can be extremely helpful to sample and examine your input data, or to visualize layer weights and generated tensors. You can also log diagnostic data as images that can be helpful in the course of your model development.

In this tutorial, you will learn how to use the Image Summary API to visualize tensors as images. In the example below we store an MNIST training image in each epoch using the Image Summary API.

```
image_log_dir = "logs/1_run/image"
image_writer = tf.summary.create_file_writer(image_log_dir)
```

```
EPOCHS = 25
for epoch in range(EPOCHS):
    # Reset the metrics at the start of the next epoch
    train_loss.reset_state()

    # Train
    for images, labels in train_ds:
        train_step(images, labels)

    # Write image to TensorBoard
    with image_writer.as_default():
        image = tf.reshape(
            next(train_ds.as_numpy_iterator())[0][epoch, :], (1, 28, 28, 1)
        )
        tf.summary.image(
            "Arbitrary image",
            image,
            step=epoch,
            max_outputs=1,
        )
```

```python
    print(
        "Epoch {:2d}, ".format(epoch + 1),
        "Loss: {:3.3f}, ".format(train_loss.result())
    )
```

Admittedly, storing an arbitrary image at every epoch is not very helpful. However, one could store misclassified validation images, a confusion matrix or a generated network output (in case of a generative neural network), which supports debugging and analysis. See this tutorial for possible implementations.

The `Images` tab displays the image you just logged. It's an five.

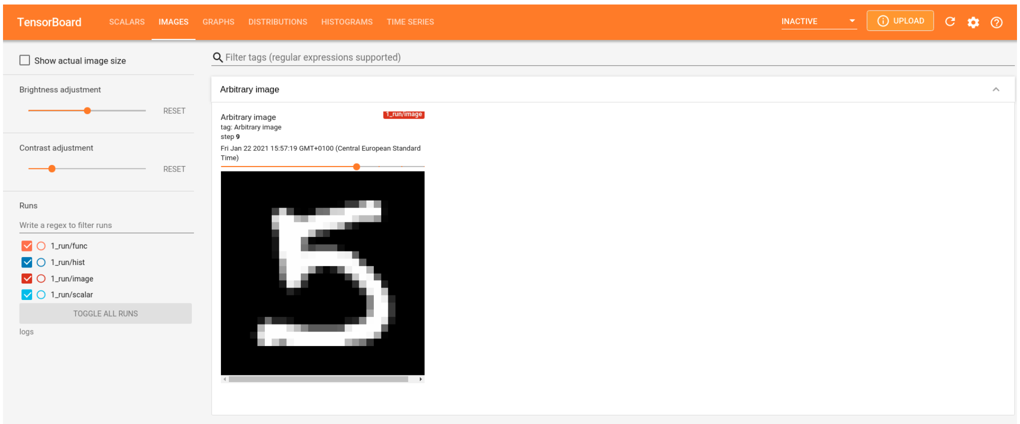

Figure 5: Stored image of step 9

TensorBoard also allows you to log and visualize embeddings and many other network properties. For more information on this topic go to TensorFlows website.

3 Exercise

Use the TensorFlow implementation of the softmax classifier from last week's lab and integrate TensorBoard commands. Use the existing code in `tensorboard_exercise.py` and log the following data:

Scalar: Training loss, training accuracy, validation loss and validation accuracy

Graph: Conceptual graph of the model `MNISTClassifier`

Histogram: Histogram of the weight matrix and bias vector

Image: Images of 12 misclassified validation images