Customization basics: tensors and operations

Run in Google Colab (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/cus

This is an introductory TensorFlow tutorial that shows how to:

- Import the required package
- Create and use tensors
- Use GPU acceleration
- Demonstrate **`tf.data.Dataset`** (/api_docs/python/tf/data/Dataset)

Import TensorFlow

To get started, import the **`tensorflow`** module. As of TensorFlow 2, eager execution is turned on by default. This enables a more interactive frontend to TensorFlow, the details of which we will discuss much later.

```
t tensorflow as tf
```

Tensors

A Tensor is a multi-dimensional array. Similar to NumPy **`ndarray`** objects, **`tf.Tensor`** (/api_docs/python/tf/Tensor) objects have a data type and a shape. Additionally, **`tf.Tensor`** (/api_docs/python/tf/Tensor)s can reside in accelerator memory (like a GPU). TensorFlow offers a rich library of operations (tf.add (https://www.tensorflow.org/api_docs/python/tf/add), tf.matmul (https://www.tensorflow.org/api_docs/python/tf/matmul), tf.linalg.inv (https://www.tensorflow.org/api_docs/python/tf/linalg/inv) etc.) that consume and produce tf.Tensor (/api_docs/python/tf/Tensor)s. These operations automatically convert native Python types, for example:

```
:(tf.add(1, 2))
:(tf.add([1, 2], [3, 4]))
:(tf.square(5))
:(tf.reduce_sum([1, 2, 3]))

:rator overloading is also supported
:(tf.square(2) + tf.square(3))
```

```
:nsor(3, shape=(), dtype=int32)
:nsor([4 6], shape=(2,), dtype=int32)
:nsor(25, shape=(), dtype=int32)
:nsor(6, shape=(), dtype=int32)
:nsor(13, shape=(), dtype=int32)
```

Each tf.Tensor (/api_docs/python/tf/Tensor) has a shape and a datatype:

```
:f.matmul([[1]], [[2, 3]])
:(x)
:(x.shape)
:(x.dtype)
```

```
:nsor([[2 3]], shape=(1, 2), dtype=int32)
:)
:e: 'int32'>
```

The most obvious differences between NumPy arrays and tf.Tensor (/api_docs/python/tf/Tensor)s are:

1. Tensors can be backed by accelerator memory (like GPU, TPU).
2. Tensors are immutable.

NumPy Compatibility

Converting between a TensorFlow **`tf.Tensor`** (/api_docs/python/tf/Tensor)s and a NumPy **`ndarray`** is easy:

- TensorFlow operations automatically convert NumPy ndarrays to Tensors.
- NumPy operations automatically convert Tensors to NumPy ndarrays.

Tensors are explicitly converted to NumPy ndarrays using their **`.numpy()`** method. These conversions are typically cheap since the array and **`tf.Tensor`** (/api_docs/python/tf/Tensor) share the underlying memory representation, if possible. However, sharing the underlying representation isn't always possible since the **`tf.Tensor`** (/api_docs/python/tf/Tensor) may be hosted in GPU memory while NumPy arrays are always backed by host memory, and the conversion involves a copy from GPU to host memory.

```
t numpy as np

ay = np.ones([3, 3])

("TensorFlow operations convert numpy arrays to Tensors automatically")
r = tf.multiply(ndarray, 42)
(tensor)


("And NumPy operations convert Tensors to numpy arrays automatically")
(np.add(tensor, 1))

("The .numpy() method explicitly converts a Tensor to a numpy array")
(tensor.numpy())
```

```
rFlow operations convert numpy arrays to Tensors automatically
nsor(
 42. 42.]
 42. 42.]
 42. 42.]], shape=(3, 3), dtype=float64)
lumPy operations convert Tensors to numpy arrays automatically
 43. 43.]
 43. 43.]
 43. 43.]]
numpy() method explicitly converts a Tensor to a numpy array
 42. 42.]
 42. 42.]
```

GPU acceleration

Many TensorFlow operations are accelerated using the GPU for computation. Without any annotations, TensorFlow automatically decides whether to use the GPU or CPU for an operation—copying the tensor between CPU and GPU memory, if necessary. Tensors produced by an operation are typically backed by the memory of the device on which the operation executed, for example:

```
:f.random.uniform([3, 3])

:("Is there a GPU available: "),
:(tf.config.experimental.list_physical_devices("GPU"))

:("Is the Tensor on GPU #0:  "),
:(x.device.endswith('GPU:0'))
```

```
ıere a GPU available:
;icalDevice(name='/physical_device:GPU:0', device_type='GPU')]
ıe Tensor on GPU #0:
```

Device Names

The **Tensor.device** (/api_docs/python/tf/Tensor#device) property provides a fully qualified string name of the device hosting the contents of the tensor. This name encodes many details, such as an identifier of the network address of the host on which this program is executing and the device within that host. This is required for distributed execution of a TensorFlow program. The string ends with **GPU:<N>** if the tensor is placed on the **N**-th GPU on the host.

Explicit Device Placement

In TensorFlow, *placement* refers to how individual operations are assigned (placed on) a device for execution. As mentioned, when there is no explicit guidance provided, TensorFlow automatically decides which device to execute an operation and copies tensors to that device, if needed. However, TensorFlow operations can be explicitly placed on specific devices using the **tf.device** (/api_docs/python/tf/device) context manager, for example:

```
t time

ime_matmul(x):
rt = time.time()
 loop in range(10):
f.matmul(x, x)

ult = time.time()-start

nt("10 loops: {:0.2f}ms".format(1000*result))

ce execution on CPU
("On CPU:")
tf.device("CPU:0"):
 tf.random.uniform([1000, 1000])
ert x.device.endswith("CPU:0")
e_matmul(x)

ce execution on GPU #0 if available
.config.experimental.list_physical_devices("GPU"):
nt("On GPU:")
h tf.device("GPU:0"): # Or GPU:1 for the 2nd GPU, GPU:2 for the 3rd etc.
 = tf.random.uniform([1000, 1000])
ssert x.device.endswith("GPU:0")
ime_matmul(x)
```

```
U:
ops: 96.68ms
U:
ops: 271.75ms
```

Datasets

This section uses the **tf.data.Dataset** API (https://www.tensorflow.org/guide/datasets) to build a pipeline for feeding data to your model. The **tf.data.Dataset**

(/api_docs/python/tf/data/Dataset) API is used to build performant, complex input pipelines from simple, re-usable pieces that will feed your model's training or evaluation loops.

Create a source **Dataset**

Create a *source* dataset using one of the factory functions like **Dataset.from_tensors** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensors), **Dataset.from_tensor_slices** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensor_slices), or using objects that read from files like **TextLineDataset** (https://www.tensorflow.org/api_docs/python/tf/data/TextLineDataset) or **TFRecordDataset** (https://www.tensorflow.org/api_docs/python/tf/data/TFRecordDataset). See the TensorFlow Dataset guide (https://www.tensorflow.org/guide/datasets#reading_input_data) for more information.

```
:nsors = tf.data.Dataset.from_tensor_slices([1, 2, 3, 4, 5, 6])

:ate a CSV file
't tempfile
.lename = tempfile.mkstemp()

 open(filename, 'w') as f:
'rite("""Line 1
 2
 3
 )

.le = tf.data.TextLineDataset(filename)
```

Apply transformations

Use the transformations functions like **map** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map), **batch** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch), and **shuffle** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shuffle) to apply transformations to dataset records.


```
:nsors = ds_tensors.map(tf.square).shuffle(2).batch(2)
```



```
.le = ds_file.batch(2)
```

Iterate

tf.data.Dataset (/api_docs/python/tf/data/Dataset) objects support iteration to loop over records:

```
:('Elements of ds_tensors:')
: in ds_tensors:
.nt(x)

:('\nElements in ds_file:')
: in ds_file:
.nt(x)
```

```
:nts of ds_tensors:
:nsor([1 9], shape=(2,), dtype=int32)
:nsor([16 25], shape=(2,), dtype=int32)
:nsor([ 4 36], shape=(2,), dtype=int32)

:nts in ds_file:
:nsor([b'Line 1' b'Line 2'], shape=(2,), dtype=string)
:nsor([b'Line 3' b'  '], shape=(2,), dtype=string)
```


Last updated 2020-06-12.