**1**

**Build TensorFlow Input Pipelines**

We introduce readers to TensorFlow input pipelines with the tf.data API, which enables them to build complex input pipelines from simple, reusable pieces. Input pipelines are the lifeblood of any deep learning experiment because learning models expect data in a TensorFlow consumable form. It is very easy to create high performance pipelines with the tf.data.Dataset abstraction (a component of the tf.data API) because it represents a sequence of elements from a dataset in a simple format.

Notebooks for chapters are located at the following URL: *<https://github.com/paperd/tensorflow>*.

So, what is deep learning? **Deep learning** is a machine learning technique that provides insights from data through automated learning algorithms with the purpose of informing decision making. Deep learning algorithms use successive layers to progressively extract higher level features from raw input. Whew, that’s a mouthful. Let’s break it down a bit. Deep learning emphasizes learning successive layers of increasingly meaningful representations from the data. Each layer of a deep learning model learns from the data. So, each layer passes down what it learns to the next layer. In image processing, lower layers may identify edges, while higher layers may identify concepts relevant to a human such as digits, letters or faces. Don’t worry if this is confusing because we have yet to define the basics, which we are about to do now.

# Neural Networks

Successive layers are almost always learned by models called neural networks. **Neural networks** are a set of algorithms modeled loosely after the human brain that are designed to recognize patterns. These networks interpret sensory data through a kind of machine perception based on labeling or clustering raw input.

A layer is the core building block in deep learning. A **layer** is a container of artificial neurons that usually receives weighted input, transforms it with a set of mostly non-linear functions, and then passes these values as output to the next layer. **Artificial neurons** are elementary units in a neural network that receive one or more inputs and sums them to produce an output. Every layer in a neural network is composed of artificial neurons.

A layer can be thought of as a data processing module that acts as a filter for data. Data goes into a layer and it comes out in a more useful form. That is, layers extract representations out of the data fed into them. Of course, we hope that the representations are meaningful to help us solve the problem at hand. It takes a lot of *practice* and *experimentation* to reap benefits that are meaningful. So, we demonstrate and explain numerous code examples to help you gain insights. But, we recommend working through the examples many times because deep learning is a very complex and intricate subject.

A very common deep learning problem is the identification of digits 0 through 9. We can solve this problem by creating a neural network composed of successive layers to help us *automatically* predict a digit from its image data.

For example, suppose we have an image of the digit 8 in our dataset. If our neural network is robust, it should be able to correctly predict that the digit is 8 from the image data without human intervention! That is, the network model is able to *predict* with a high degree of accuracy images of digits. Of course, humans can easily distinguish digits between 0 and 9, but the ability of a computer model to do this is amazing and at the heart of what deep learning is all about.

A neural network is a collection of *neurons* with *synapses* connecting them. It is organized into three main parts:

* input layer
* hidden layer
* output layer

When training a neural network, data is initially passed to the input layer. So, the **input layer** brings the initial data into the system for further processing by subsequent layers of artificial neurons. It then passes the data through the activation function before passing it on to the first hidden layer. An **activation function** is an algorithm that defines the output of a neuron given an input or set of inputs.

A **hidden layer** is in-between input layers and output layers where artificial neurons take in a set of weighted inputs and produce an output through an activation function. A network can have multiple hidden layers. The **output layer** produces the result for given inputs. It is the place where all the computation is done. Neurons tend to be remarkably simple with nothing but a floating point value, an input, and an output. That float point value is what we refer to as the *weight* of a neuron.

So, neurons take inputs from their previous layer, transform them to keep values within a manageable range with an activation function, and send the transformed inputs along with their weights to neurons in the next layer. Since values at the input layer are generally centered at zero and have already been appropriately scaled, they don’t need transformation with an activation function.

# Learning Representation from Data

Machine learning algorithms discover rules to execute a data processing task. So, to conduct machine learning we need three things:

1. input data points
2. examples of expected output
3. a way to measure algorithm performance

*Input data points* are data of some kind. An example of input data could be pictures. Image recognition in deep learning requires pictures. Of course deep learning models require numeric data. So, how do the models interpret images? The pictures must be transformed in some way! Don’t worry about how this is done because we cover it in the next chapter.

**Note:** All data, not just image data, must be fit to a numerical representation before it can be consumed by a deep learning model.

To make predictions from data in deep learning, we need *examples of expected output*. So, the data must contain a representation of each data example and what each data example represents. We can better understand with an example. When predicting digits, the data must contain representations of each digit and what the digit represents. If an example from the data is the digit 9, we must have the representation of the digit 9 and a target value of 9. We cover how to represent a digit and its target value in the next chapter.

Finally, we need to gauge algorithmic performance. We do this by determining the distance between the *algorithm’s current output and its expected output*. This distance is often called loss or error. The *loss* is used as feedback to adjust the way the algorithm works. Such an adjustment is called *learning*. For example, if our neural network model predicts that a digit is 3 but it is really 8, our model has at least some loss. That is, there is some distance between what the model predicts and its expected output.

# TensorFlow 2.x

**TensorFlow** is a *Python* open source library for numerical computation created to facilitate machine learning and deep learning problem solving. TensorFlow bundles together machine learning modules, deep learning modules, and associated algorithms into a common programming environment. TensorFlow 2.x is the most current version of the software. We use the 2.x designation because the software is changing so rapidly.

# Google Colab

**Google Colab** (short for Google Colaboratory) is a cloud service that offers a data science work space for *Python* very similar to the *Juypter Notebook*. Actually, any Jupyter Notebook can be directly loaded into the Colab cloud service.

Colab notebooks are stored in Google Drive and can be shared as you would with Google Docs or Sheets. Simply click the Share button at the top right of any Colaboratory notebook or follow the Google Drive sharing instructions.

Peruse [*https://colab.research.google.com/notebooks/welcome.ipynb*](https://colab.research.google.com/notebooks/welcome.ipynb) to get started. The site offers a nice tutorial, but you can browse YouTube videos or other tutorials to deepen your Colab skills. Of course, we walk you through the basics.

# Google Drive

**Google Drive** is a cloud-based file storage and synchronization service that you are most likely already familiar with, so we won’t spend much time on it. But, we need to show how to connect Google Colab with Google Drive. It just takes a few simple steps:

1. Sign into your Google email account
2. Open a new browser tab and browse to *Google Colab*
3. Click the *Google Colab* link
4. Click *Google Drive* in the top menu from the pop-up window

All notebooks on your Google Drive account appear in the window. You should see no notebooks appear unless you’ve worked with Colab in the past. Notebooks are saved on Google Drive My Drive inside the Colab Notebooks directory. This directory is automatically created when Colab is connected to Google Drive.

If you want to create a new notebook, click on NEW NOTEBOOK. Or, click on CANCEL, which takes you to the Welcome To Colaboratory screen. This screen offers the main menu for Google Colab as well as the table of contents that helps you get started.

**Note:** The connection we just established between Google Colab and Google drive is *persistent*. That is, we only need to establish this connection *once* unless browser history is cleared.

# Create a New Notebook

Within the Colab environment, it is easy to create a new notebook. Open Google Colab in a browser (if not already open). From the pop-up window, click NEW NOTEBOOK. If already in the Colab environment, click *File* in the top left menu under *Welcome To Colaboratory*. Click *New notebook* from the drop-down menu. A code cell is now ready for executing Python code! Add code or text cells by clicking on the *+ Code* or *+ Text* buttons. For more options, click on *Insert* from the menu.

To create your first piece of code, add the following in the code cell:

string = 'Peter picked a pail of pickeled peppers'

string

To execute code, click the *little arrow to the left*. The output from the code cell shows the contents of the *string* variable.

**Tip:** We recommend copying and pasting code from the website.

# GPU Hardware Accelerator

To vastly speed up processing, we use the GPU available from the Google Colab cloud service. Colab provides a free Tesla K80 GPU of about 12 GB. It’s very easy to enable the GPU in a Colab notebook:

1. click *Runtime* in the top left menu
2. click *Change runtime type* from the drop-down menu
3. choose *GPU* from the *Hardware accelerator* drop-down menu
4. click *SAVE*

**Note:** The GPU must be enabled in *each* notebook. But, it only has to be enabled once.

Test if GPU is active:

import tensorflow as tf

# display tf version and test if GPU is active

tf.\_\_version\_\_, tf.test.gpu\_device\_name()

Import the tensorflow library and display the version of TensorFlow as well as the status of the GPU. If '/device:GPU:0' is displayed, the GPU is active. If '..' is displayed, the regular CPU is active.

# Download a File from a URL

Let’s get to work! We can directly download a file from a URL with the *tf.keras.utils.get\_file* utility. We need the tensorflow library, but we already imported it. We recommend creating a new code cell by clicking *+ Code*.

The following code cell downloads a CSV file from a URL:

# import keras module

from tensorflow import keras

ds = 'auto-mpg.data'

url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data'

dataset\_path = tf.keras.utils.get\_file(ds, url)

dataset\_path

Import the keras module from the tensorflow library. Use *tf.keras.utils.get\_file* to download a dataset from the *UCI Machine Learning Repository*. This repository is a collection of databases, domain theories, and data generators for the empirical analysis of machine learning algorithms.

**Tip:** We highly recommend testing small pieces of code in their own code cells to reduce debugging time and effort.

# Prepare the Dataset

As is, the dataset needs some preprocessing. For example, it is without feature headings. We recommend creating a new code cell for each example. Do so now by clicking *+ Code*.

The following code cell creates a pandas dataframe by accessing the CSV file from the path we created in the previous code cell:

# import the pandas library

import pandas as pd

cols = ['MPG','Cylinders','Displacement','Horsepower','Weight',

        'Acceleration', 'Model Year', 'Origin']

raw\_dataset = pd.read\_csv(dataset\_path, names=cols,

                      na\_values = "?", comment='\t',

                      sep=" ", skipinitialspace=True)

Import pandas and create a list to hold feature names. Use the *read\_csv()* function to place the CSV data into a pandas dataframe. A **pandas dataframe** is a two-dimensional (or 2D) data structure with data aligned in a tabular row and column fashion. Let’s look at the last five records:

raw\_dataset.tail()

The Pandas *tail* method returns the last n rows. By default, it returns the last five rows. It is useful for quickly verifying data.

We can save the original data by creating a copy:

# create a copy

data = raw\_dataset.copy()

# verify contents by displaying some data

data.tail()

# Colab Abends

An AbEnd (also abnormal end or abend) is an abnormal termination of software or a program crash. When we run Colab for a long time (several hours), read a large dataset into memory, and process said data or create a really large notebook, it may crash. When this happens, we have two choices:

1. Restart runtime
2. Close the notebook and restart from scratch

To restart runtime, click *Runtime* in the top menu, click *Restart runtime* from the drop-down menu, and click *YES* when prompted. And, rerun your notebook from the beginning. Colab recommends this option. To restart from scratch, clear browser history and open Colab.

# Colab Strange Results

Sometimes unexpected errors or other strange results arise when working with Colab. If this happens, restart runtime or open Colab from scratch as described in the *Colab Abends* section for the notebook you are working on.

# Tensors

A **tensor** is a container for *numeric* data. Tensors can contain an arbitrary number of dimensions. A dimension is often called an *axis*.

In deep learning, tensors are considered a generalization of matrices represented by *n-dimensional* arrays. The dimensionality of a tensor is often described by its number of axes. So, tensors are defined by how many axes they have in total. The **rank** is the number of axes represented by a tensor.

The best way to understand tensors is through examples. So, let’s start with the simplest type.

# Scalars (0D tensors)

A **scalar** is a tensor of only one number. So, a scalar is considered a zero-dimensional (or 0D) tensor. Examples include a numpy float32 or float64 number.

Let’s look at an example:

import numpy as np

# create numpy scalar 9

scalar = np.array(9)

scalar

Import the numpy module. Assign a numpy array containing 9 to a variable and display the result.

Signal the rank of the scalar tensor as so:

# signal its rank

print (str(scalar.ndim) + 'D')

The *ndim* attribute signals the number of axes (or dimensionality) of a numpy tensor. Since our variable is a scalar, we see 0D displayed.

# Vectors (1D tensors)

A **vector** is an array of numbers. So, a vector is a one-dimensional (or 1D tensor). A 1D tensor has exactly one axis.

Create a numpy vector containing six elements:

# create numpy vector [0, 1, 0, 0, 0, 0]

vector = np.array([0, 1, 0, 0, 0, 0])

vector

We see [0, 1, 0, 0, 0, 0] displayed.

Signal the rank of the vector:

# signal its rank

print (str(vector.ndim) + 'D')

We see 1D displayed.

# Matrices (2D tensors)

A **matrix** is an array of vectors. The simplest matrix is a two-dimensional (or 2D) tensor. A 2D tensor has two axes. Its axes are generally referred to as *rows* and *columns*.

Create a numpy matrix:

# create a numpy matrix

matrix = np.array([[0, 1, 0, 0, 0, 0],

                   [0, 0, 1, 0, 0, 0],

                   [0, 0, 0, 1, 0, 0],

                   [0, 0, 0, 0, 1, 0],

                   [0, 0, 0, 0, 0, 1]])

matrix

Rows are entries from the first axis and columns are entries from the second axis. So, the first row from our example is [0, 1, 0, 0, 0, 0] and the first column is [0, 0, 0, 0, 0, 0]. Since the matrix has five rows and six columns, it is considered a 5 x 6 matrix.

Signal the rank of the matrix:

# signal its rank

print (str(matrix.ndim) + 'D')

We see 2D displayed.

# 3D Matrices (3D tensors)

We can create a 3D tensor by packing 2D tensors into a new array. A 3D tensor can be visually interpreted as a cube of numbers.

Let’s look at an example:

# create a 3D tensor

D3 = np.array([[[0, 1, 2]],

               [[3, 4, 5]],

               [[6, 7, 8]]])

# signal its rank

print (str(D3.ndim) + 'D')

By packing 3D tensors into an array, we can create 4D tensors and so on. In deep learning, we generally manipulate tensors that are 0D to 4D. With video processing, we can go up to 5D.

# Key Attributes of Tensors

1. rank
2. shape
3. data type

As discussed earlier, **rank** is the number of axes and it is displayed with the *ndim* attribute. A 0D tensor has zero axes, a 1D tensor has one axis, a 2D tensor has two axes, and a 3D tensor has three axes.

**Shape** is a tuple of integers that describe the number of dimensions along each axis. So, our 3D matrix has shape (3, 1, 3), 2D matrix has shape (5, 6), vector has shape (6,) and scalar has an empty shape (). We can display shape with the *shape* attribute.

Let’s see the shape of our 3D matrix:

# 3 instances of 1 x 3 matrices

D3.shape

Display the shape of our matrix:

# 5 rows and 6 columns (or 5 x 6 matrix)

matrix.shape

Display the shape of our vector:

# 6 element vector

vector.shape

Display the shape of our scalar:

# just a scalar number

scalar.shape

**Data type** is the description of data contained in a tensor. Use the *dtype* attribute to display data type.

Display data types of our tensors:

# dtype of tensors

print (scalar.dtype)

print (vector.dtype)

print (matrix.dtype)

print (D3.dtype)

We see that all of our tensors contain int64 values.

# Input Pipelines

An **input pipeline** is a sequence of data processing components that manipulate and apply data transformations. Pipelines are very common in machine learning and deep learning systems because these systems are *data-rich*. That is, they demand large volumes of data to perform well. Input pipelines are the best way to transform large datasets because they break data down into manageable components.

Each component of an input pipeline pulls in a large amount of data, processes it in some manner, and spits out the result. The next component pulls in the resultant data, processes it in another manner, and spits out its own output. The pipeline continues until all of its components have finished their work.

# The tf.data API

TensorFlow application revolves around the concept of a dataset encapsulated in the tf.data API. The *tf.data API* enables you to build fast, flexible, and easy-to-use input pipelines from simple, reusable pieces. It is the recommended API for building input pipelines in TensorFlow.

Let’s create a simple TensorFlow tensor:

import tensorflow as tf

X = tf.range(5)

X

Import the tensorflow library if necessary. Create a TensorFlow tensor with shape (5,) and data type int32. The vector contains values [0, 1, 2, 3, 4], which corresponds to shape (5,).

**Tip:** Compare your output with website code output to verify results.

Use the *numpy()* attribute to display values from a TensorFlow tensor:

X.numpy()

We can also access individual elements from a tensor:

# first element from tensor

X[0].numpy()

Or, access multiple elements:

# 2nd, 3rd, and 4th elements from tensor

X[1:4].numpy()

The code slices the second, third, and fourth elements from the tensor.

# Function from\_tensor\_slices

Function *from\_tensor\_slices* creates a tf.data.Dataset from all slices of a tensor. In our case, it creates a dataset whose elements are all the slices of *X* (along the first dimension). A **tf.data.Dataset** is the standard TensorFlow API to build input pipelines. It represents a potentially large set of elements.

Let’s demonstrate with an example:

dataset = tf.data.Dataset.from\_tensor\_slices(X)

dataset

We just created a TensorSliceDataset object sliced from X with shapes () and data types tf.int32.

# Iterate a tf.data.Dataset

Iterate over a *tf.data.Dataset* with a simple loop:

for item in dataset:

  print (item)

We see each value along with its shape and data type.

# Tensors and numpy

TensorFlow tensors play nice with numpy. We can create a TensorFlow tensor from a numpy array and vice versa. We can even apply TensorFlow operations to numpy arrays and numpy operations to TensorFlow tensors.

Create a numpy array from the TensorFlow dataset we just created as shown in Listing 1-1.

# create a variable to hold a line break

br = '\n' # this is just a convenient way to include a line break

# import numpy

import numpy as np

# technique 1

ls = []

for item in dataset:

  e = item.numpy()

  ls.append(e)

np\_arr = np.asarray(ls, dtype=np.float32)

print (type(np\_arr))

print (np\_arr, br)

# technique 2

ls = [item.numpy() for item in dataset]

np\_arr = np.asarray(ls, dtype=np.float32)

print (type(np\_arr))

print (np\_arr)

***Listing 1-1. Create a numpy array from a TensorFlow dataset***

We show two techniques to create a numpy array from a TensorFlow dataset. Technique 1 initializes a list, iterates the tensor, converts each value to numpy, and appends it to a list. The list is then converted to a numpy array. Technique 2 performs the same logic, but uses list comprehension for more compact code.

We can convert the numpy array back to a TensorFlow dataset with the *constant* method:

tf\_arr = tf.constant(np\_arr)

tf\_arr

However, constants are immutable. That is, their values cannot be modified. So, we can use the *variable* method if we need to modify values.

Use the variable method:

tf\_arr = tf.Variable(np\_arr)

tf\_arr

# Chaining Transformations

We can apply transformations to a tf.data.Dataset by calling its transformation methods. Each method returns a *new* dataset, which allows us to chain transformations. Let’s start with a single transformation.

Create a tf.data.Dataset and show its values:

dataset = tf.data.Dataset.range(5)

for item in dataset:

  print (item)

The dataset contains values [0, 1, 2, 3, 4].

Use the *repeat()* transformation method to repeat elements in a tensor:

data\_rep = dataset.repeat(3)

for item in data\_rep:

  print (item)

The new dataset contains three sets of the original. We repeat data to enlarge a dataset for better model performance without getting new data.

Now, let’s chain transformations:

data\_batch = dataset.repeat(3).batch(7)

for item in data\_batch:

  print (item)

What happened? The first transformation, *repeat(3)*, creates three copies of the original dataset. We chain the first transformation into the second with *batch(7)*, which creates batches of seven elements each.

So, the new dataset consists of three tensors. The first tensor contains [0, 1, 2, 3, 4, 0, 1], the second tensor contains [2, 3, 4, 0, 1, 2, 3], and the third tensor contains [4]. By the time we get to the third batch, we run out of data.

We can drop the final batch:

data\_drop = dataset.repeat(3).batch(7, drop\_remainder=True)

for item in data\_drop:

  print (item)

The new dataset consists of two tensors. The first tensor contains [0, 1, 2, 3, 4, 0, 1] and the second tensor contains [2, 3, 4, 0, 1, 2, 3]. Transformation methods don’t modify datasets. They create new ones so we can keep track of each dataset by naming them differently.

Create equal batches:

data\_equal = dataset.repeat(3).batch(5)

for item in data\_equal:

  print (item)

The new dataset consists of three tensors. Each tensor contains [0, 1, 2, 3, 4].

# Mapping Tensors

Use the *map* method to transform elements in a tensor. The **map** function returns a map object of the results after applying the given function to each element of a given iterator. The returned object is an iterator. An **iterator** is a Python object capable of returning its members one at a time. Lists, tuples, and strings are common iterators in Python.

Whew! Let’s look at the example in Listing 1-2 to help us understand how the function works.

# create a dataset

dataset = tf.data.Dataset.range(7)

# repeat and batch it

data\_batch = dataset.repeat(3).batch(7)

# display the batched dataset

for row in data\_batch:

  print (row)

# map() a function on it

data\_map = data\_batch.map(lambda x: x \*\* 2)

# display the first batch

print ()

for item in data\_map.take(1):

  print (item)

***Listing 1-2. A simple map function example***

We create a new dataset with values [0, 1, 2, 3, 4, 5, 6]. We chain the repeat transformation to the batch transformation to create a new dataset with three tensors. Each tensor contains [0, 1, 2, 3, 4, 5, 6]. We square each element by mapping with a lambda function. A **lambda function** is a single-line function declared with no name that can have any number of arguments, but can only have one expression. Instead of iterating the entire dataset, we can take one or more samples with the *take* method. In our case, we just take the first sample (or first tensor) from the dataset.

So, the mapped dataset contains three tensors. Each tensor contains [0, 1, 4, 9, 16, 25, 36] because the lambda function squares a value and the map function maps the lambda function expression onto each element in a tensor.

# Filter a tf.data.Dataset

What if we want to filter a dataset? The **filter** method filters a given sequence with the help of a function that tests whether each element in the sequence is true or not.

Let’s look at the example in Listing 1-3.

# create a dataset

dataset = tf.data.Dataset.range(7)

# display the dataset

for row in dataset:

  print (row)

# apply a filter

data\_filter = dataset.filter(lambda x: x < 6 and x > 3)

print ()

for item in data\_filter:

  print (item)

***Listing 1-3. A simple filter() function example***

We create a new dataset with values [0, 1, 2, 3, 4, 5, 6]. We filter the dataset to extract values between 3 and 6. Since we use less than and greater than in the lambda function, we don’t include either 3 or 6 values. So, the filtered dataset contains [4, 5] because the lambda function extracts values between 3 and 6 non-inclusive.

# Shuffling a Dataset

Deep learning algorithms work best when instances in the training set are independent and identically distributed. A simple way to ensure this is to shuffle instances with the *shuffle* method.

Create a tf.data.Dataset to shuffle:

# create a dataset

dataset = tf.data.Dataset.range(10).repeat(3)

print ('dataset has', len(list(dataset)), 'elements')

Shuffle the dataset:

# shuffle data into batches of 7

ds = dataset.shuffle(buffer\_size=5).batch(7)

for item in ds:

  print (item)

We get tensors of seven elements. Notice that the last tensor has only two elements. We have four tensors of size seven, which equals 28. Since the dataset has 30 elements, we have two elements left over.

We set buffer size to 5. So, TensorFlow keeps a buffer of the next five samples (or tensors) and randomly selects one of those five samples. It then adds the next element to the buffer. Each sample contains a batch of data. So, each sample in our example contains seven elements because we set batch size to 7. Performance can be improved by experimenting with different batch and buffer sizes, but getting it right takes time and energy.

Once a dataset is shuffled, each dataset iteration creates a new shuffle:

# rerun to get a different shuffle

for item in ds:

  print (item)

# TensorFlow Math

TensorFlow provides several operations for math computations with the *tf.math* module. Peruse [*https://www.tensorflow.org/api\_docs/python/tf/math*](https://www.tensorflow.org/api_docs/python/tf/math) for all possible math operations. To perform math operations, tensors must have the same shape.

## Vector Tensors

Let’s create some data:

# create data

v1 = np.array([0, 1, 4, 8, 16])

v2 = np.array([0, 3, 9, 27, 81])

Convert numpy arrays to tensor constants and add:

conv1 = tf.constant(v1)

conv2 = tf.constant(v2)

result = tf.add(conv1, conv2)

result

The result is [0, 4, 13, 35, 97].

Convert numpy arrays to tensor variables and add:

varv1 = tf.Variable(v1)

varv2 = tf.Variable(v2)

result = tf.add(varv1, varv2)

result

We get the same result. The only difference between constants and variables is that constant values cannot be changed. That is, they are immutable.

Subtract tensor variables:

result = tf.subtract(varv2, varv1)

result

Subtracting var1 from var2 is [0, 2, 5, 19, 65].

Mix constants and variables:

result = tf.add(conv1, varv2)

result

The result is [0, 4, 13, 35, 97].

Test for equivalency:

result = tf.equal(varv1, varv2)

result

The result is [True, False, False, False, False].

Multiply tensor constants:

result = tf.multiply(conv1, conv2)

result

The result is [0, 3, 36, 216, 1296].

Divide a tensor by a value:

result = tf.divide(conv2, 3)

result

The result is [0, 1, 3, 9, 27].

## Matrix Tensors

Math operations work on n-dimensional tensors. So, let’s perform math operations on matrix tensors.

Create some data:

# create data

m1 = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])

m2 = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])

m1.shape, m2.shape

We just created a pair of 3 x 3 matrices. That is, both have three rows and three columns.

Convert numpy matrices to tensors and add:

conm1 = tf.constant(m1)

conm2 = tf.constant(m2)

result = tf.add(conm1, conm2)

result

The result is [[2, 0, 0], [0, 2, 0], [0, 0, 2]].

Test for equivalency:

result = tf.equal(conm1, conm2)

result

The result is [[True, True, True], [True, True, True], [True, True, True]] because the tensors are equivalent.

## tf.data.Dataset Tensors

We can perform math operations on tf.data.Dataset tensors.

Create a dataset:

# create a dataset

m = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

m

We just created a 3 x 4 matrix. So, the matrix consists of three rows and four columns.

Convert the dataset to a tf.data.Dataset:

dataset = tf.data.Dataset.from\_tensor\_slices(m)

dataset

The tf.data.Dataset is a *TensorSliceDataset* because we transformed the numpy dataset with the *from\_tensor\_slices* method. The dataset contains three tensors. The shape of each tensor is (4,), which means that each has four elements. Elements are data type int64.

Display the tensors:

for t in dataset:

  print (t)

The tf.data.Dataset contains three tensors with values [1, 2, 3, 4], [5, 6, 7, 8], and [9, 10, 11, 12].

Transform tensor values:

squared\_data = dataset.map(lambda x: x \*\* 2)

# display tensors

for item in squared\_data:

  print (item)

We map the lambda function to tensor elements. So, each element is squared.

# Save a Notebook

Although *Autosave* is implemented in Google Colab, there is a delay between the moment you execute a cell and when the save occurs. So, we recommend periodically saving.

Manually saving a notebook just takes two steps:

1. Click *File* in the top left menu
2. Click *Save* from the drop-down menu

The notebook is saved in Google Drive My Drive in the *Colab Notebooks* directory.

# Download a Notebook to a Local Drive

Google Drive is an excellent place to store Colab notebooks. But, we also like to save notebooks to a local drive.

Download a notebook to a local drive:

1. Be sure to save the notebook
2. Click *File* in the top left menu
3. Click *Download .ipynb* from the drop-down menu

The notebook is now located in the *Downloads* directory.

# Load a Notebook from a Local Drive

We use Google Drive as backup because it only provides 15 GB of free space. If you work for a company, they may provide extra storage. Given this case, you may want to use Google drive for primary storage.

To load a notebook from a local drive:

1. Open *Google Colab*
2. From the pop-up menu, click *Upload*
3. Click *Choose File*
4. Locate the notebook on your local drive and open it