TensorFlow Part 1

Foundation of Deep Learning

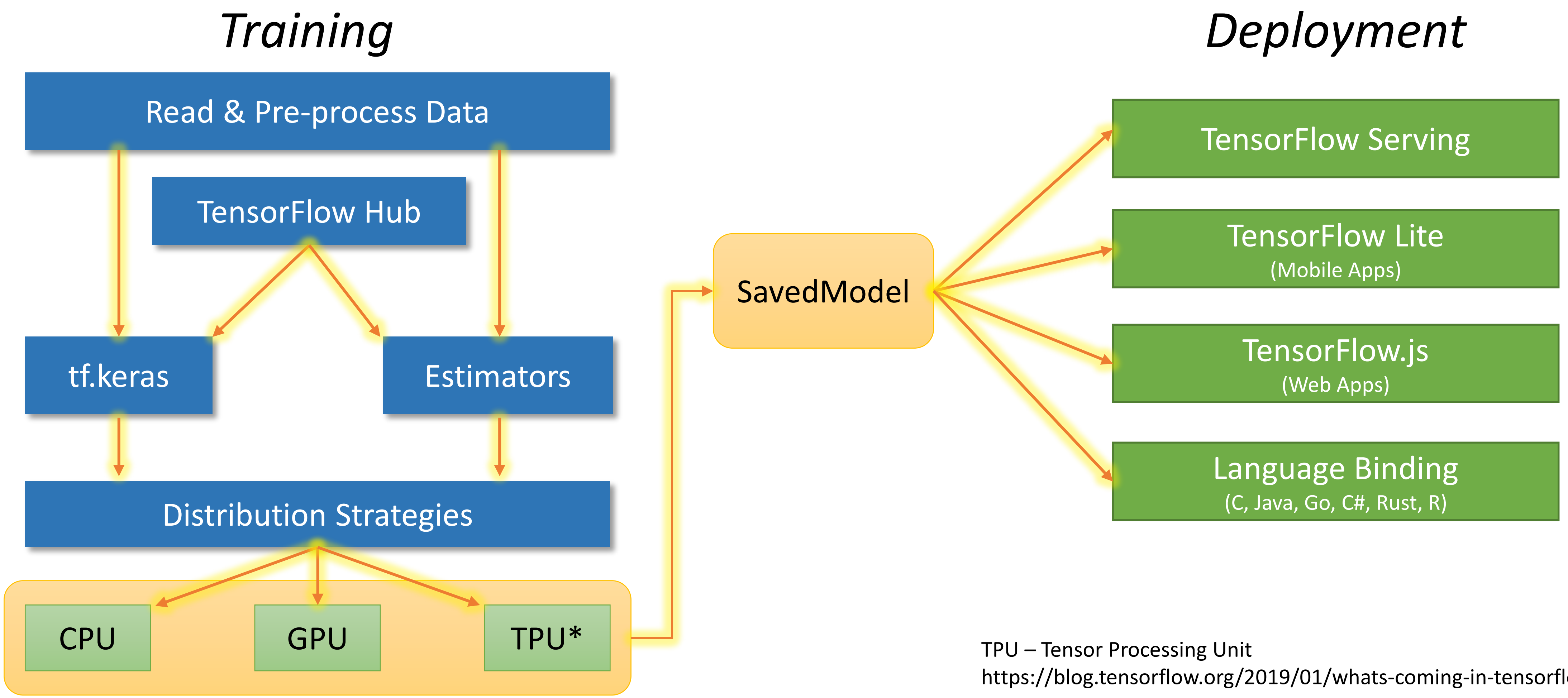
Deep Learning Frameworks

- Theano – considered grandfather of DL frameworks, developed by the Universite de Montreal in 2007
- TensorFlow - end-to-end open-source deep learning framework developed by Google and released in 2015, scalable and widely used in industry
- PyTorch - based on Torch developed by Facebook’s AI research group, open-sourced on GitHub in 2017, limited visualization
- Keras - high-level neural network API written in Python can run on top of TensorFlow, CNTK and Theano.
- Others such as Apache MXNet and Microsoft CNTK

TensorFlow

- TensorFlow is free and supports the following programming languages:
  - Python
  - JavaScript
  - C/C++
  - Java
  - Go
- Python is the preferred language. Backward compatibility is not guaranteed for languages other then Python and C. [https://www.tensorflow.org/guide/versions]
- As can be seen from its history, it is primarily designed for large scale distributed training and inference.
- Although very popular for deep learning, it is also useful for other machine learning algorithms and computations.
- We can use Low-level (`tf.function`) and High-level (Keras) TensorFlow APIs
  - In this module we will cover both the low-level and high-level TensorFlow APIs.

TensorFlow 2.0 Platform



TPU – Tensor Processing Unit
https://blog.tensorflow.org/2019/01/whats-coming-in-tensorflow-2-0.html

TensorFlow Hub

- Repository for publishing and using pre-trained models
- URL: https://www.tensorflow.org/hub

TensorFlow Hub is a repository of trained machine learning models.

TensorFlow Hub is a repository of trained machine learning models ready for fine-tuning and deployable anywhere. Reuse trained models like BERT and Faster R-CNN with just a few lines of code.

```
!pip install --upgrade tensorflow_hub

import tensorflow_hub as hub

model = hub.KerasLayer("https://tfhub.dev/google/nnlm-en-dim128/2"
embeddings = model(["The rain in Spain.", "falls",
                    "mainly", "In the plain!"])

print(embeddings.shape)  #(4,128)
```

See the guide
Learn about how to use TensorFlow Hub and how it works.

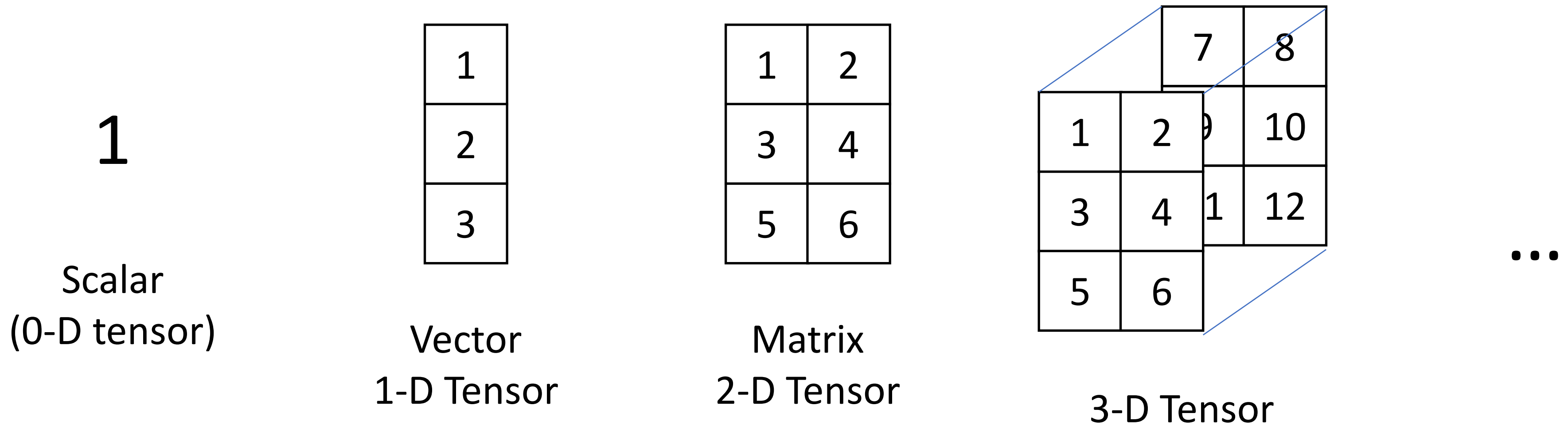
See tutorials
Tutorials show you end-to-end examples using TensorFlow Hub.

See models
Find trained TF, TFLite, and TF.js models for your use case.

Tensors

- TensorFlow works exclusively on tensors.
  - We have actually met tensors in NumPy previously.
- Tensors are data that are constructed in multiple dimensions
  - Dimension 0: scalar
  - Dimension 1: vector
  - Dimension 2 : matrix
- The number of dimensions is called the rank.
- We usually refer to the dimensions as axes.
- The number of axes is the same as the rank. For example, a rank 3 tensor (3-D Tensor) will have 3 axes.

Tensors



<u>NOTE</u>
Do not be confused about the dimension.
This is a 1-D tensor but a 3-D vector.
This is a 3-D vector because it has 3 items and can represent a point in 3-D space.

Tensors Shape

- The shape of a tensor is a tuple of integers indicating the length along each axis.
- Example
  - The tensor on the right is of shape (2, 3, 4)
  - There are 3 numbers, so it is a 3D tensor (rank 3)
  - Each number indicates how many elements there are along each axis
    - Axis 0 -> 2 elements
    - Axis 1 -> 3 elements
    - Axis 2 -> 4 elements

```
m = np.array(
[
    [
        [1, 2, 3, 4],
        [5, 6, 7, 8],
        [9, 10, 11, 12]
    ],
    [
        [13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24]
    ]
])
```

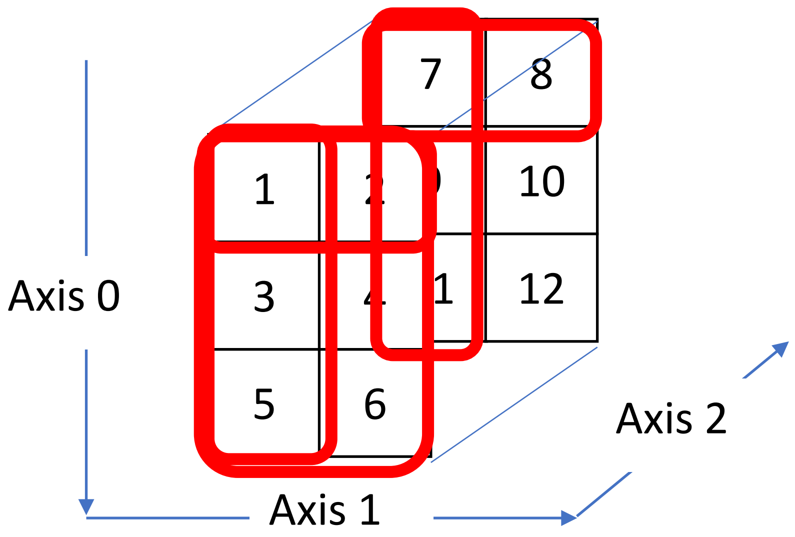
Shape of m is (2, 3, 4)

A tuple of 3 numbers,
hence it is a 3-D tensor,
there will be 3 axes

Tensors

```
[
    [
        [1, 7], [2, 8]
    ],
    [
        [3, 9], [4, 10]
    ],
    [
        [5, 11], [6, 12]
    ]
]
```

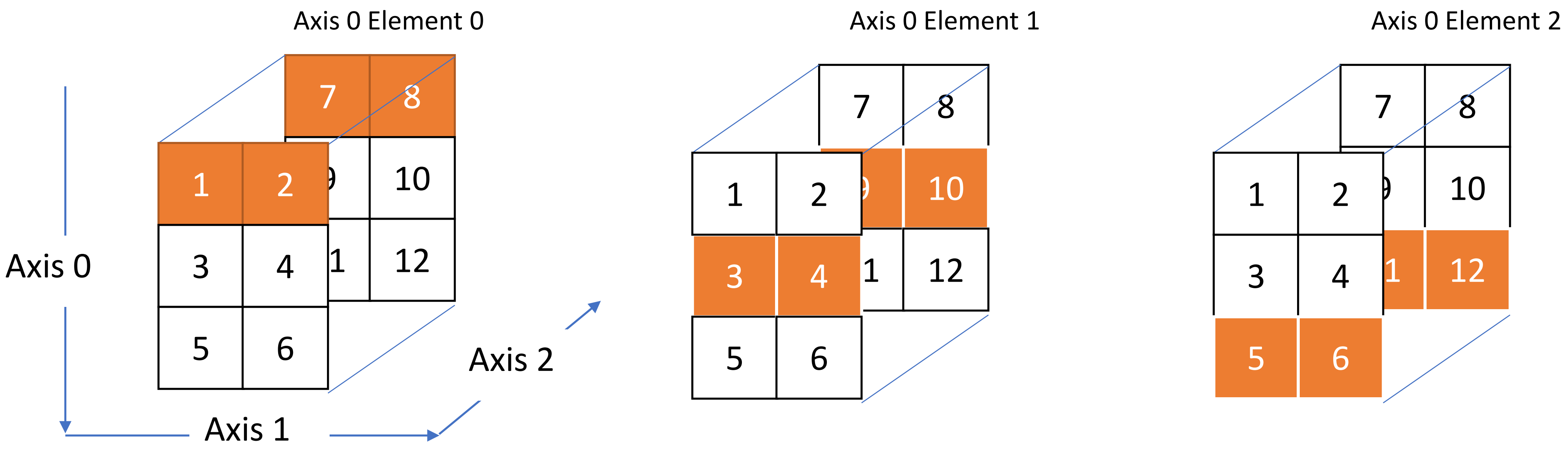
3-D Tensor



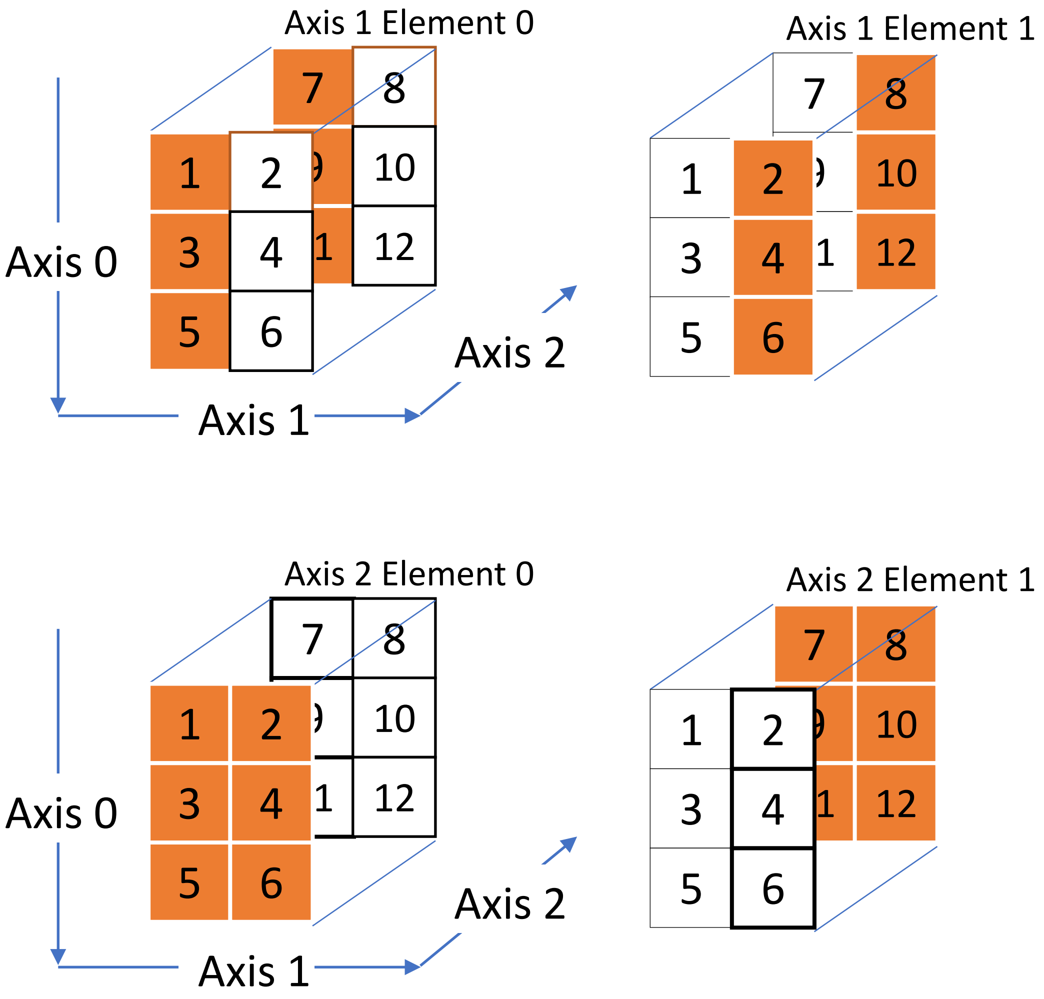

```
tf.gather(tensor1, 0, axis=0)
```

- First element of axis 0 will be [1, 7] and [2, 8]
- First element of axis 1 will be [1, 7], [3, 9], [5,11]
- First element of axis 2 will be [1, 2], [3, 4] and [5, 6]

Tensors


Axis 0 Element 0
Axis 0 Element 1
Axis 0 Element 2
Axis 0
Axis 1
Axis 2
7 8
1 2 10
3 4 12
5 6

Tensors


Axis 1 Element 0
Axis 1 Element 1
Axis 2 Element 0
Axis 2 Element 1
Axis 0
Axis 1
Axis 2
1
2
3
4
5
6
7
8
10
12

Tensors

```
m = np.array([
    [
        [1, 7], [2, 8]
    ],
    [
        [3, 9], [4, 10]
    ],
    [
        [5, 11], [6, 12]
    ]
])
t = convert_to_tensor(m)
print(tf.gather(t, 0, axis=0)) #param - tensor, index, axis
print(tf.gather(t, 0, axis=1))
print(tf.gather(t, 0, axis=2))
```

Result

```
[[1 7]
 [2 8]]

[[ 1  7]
 [ 3  9]
 [ 5 11]]

[[1 2]
 [3 4]
 [5 6]]
```

Tensors –Axis 0

- There are certain conventions in deep learning when using tensors.
- Axis 0 is usually the samples axis. Each element along axis 0 is a sample of data (row in the traditional feature vector)
- Sometimes is it also called the batch axis or batch dimension. This is because data are usually processed in batches. We break samples into batches along axis 0.

```
#first batch of 128 samples.
batch = training_data[:128]
```

Axis 0

ASG	AGE	LOS	CLAIM
0	23	2	3619
1	20	3	4590.6
1	29	1	3850
1	27	2	11187.4
1	20	2	4368
0	25	3	5763.8
0	26	3	5108.6
1	37	2	4582.2
0	37	2	4677.4
0	30	2	3904.6

Tensors – Features and Examples

- Deep learning has been applied to many use cases and there are many different types of data.
- So far, we have been working almost exclusively with data samples and its features, see table on the right.
- In such cases, it is sufficient to use a 2-D tensor for the data
- Deep learning has been very successful especially in areas like time series, images and even video sequences. So a 2D tensor is not sufficient, the following slides show how data are represented using tensors.

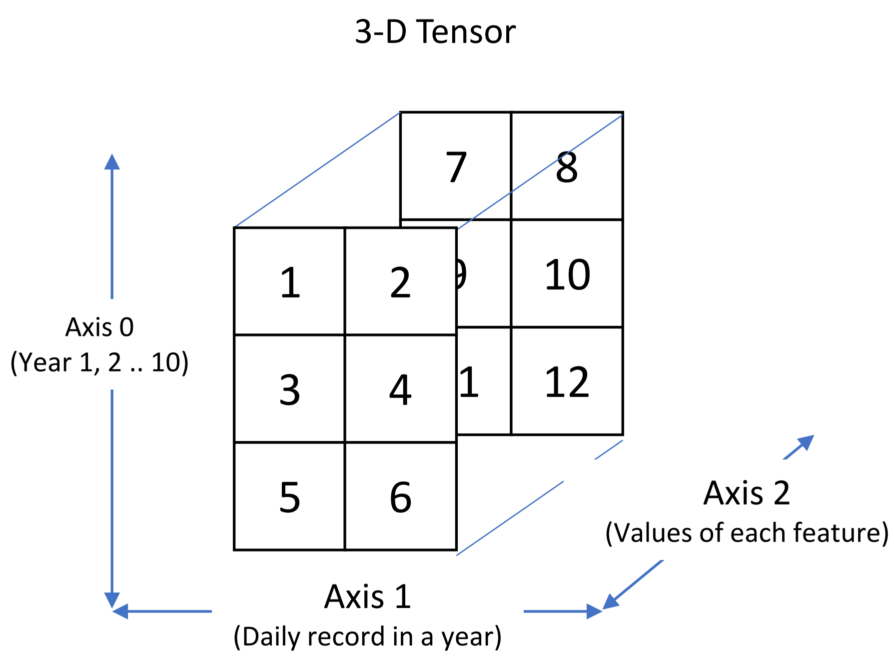
Age	Income	Years Employed
22	2500	2
45	5000	6
60	6000	20

$$\begin{bmatrix} 22 & 2500 & 2 \\ 45 & 5000 & 6 \\ 60 & 6000 & 20 \end{bmatrix}$$

2-D Tensor

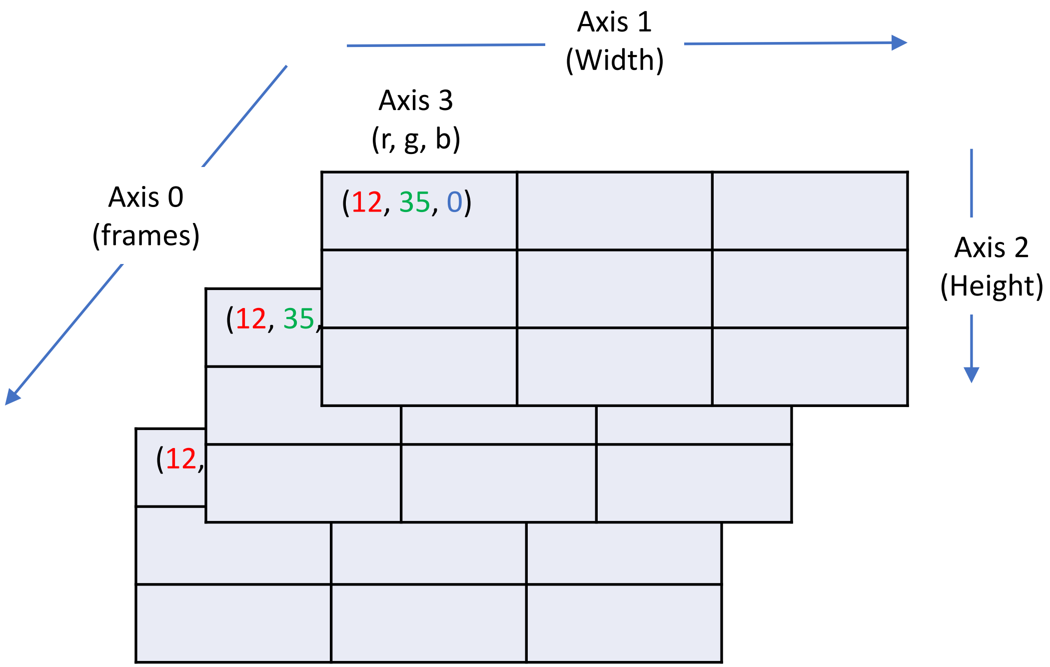
Tensors – Time Series

- Time series data uses 3D tensors
  - axis 0 – samples
  - axis 1 – timesteps
  - axis 2 - features
- Timestep refers to fixed duration, e.g. every 5 minutes.
- Example: if we store daily record of temperature (high, low, average) over 10 years, we will have a tensor of shape (10, 365, 3)
  - *Samples* - We have 10 samples where each sample is one year's worth of daily record.
  - *Timestep* - Each year we have 365 daily record.
  - *Features* - Each daily record has 3 features (high, low, average).



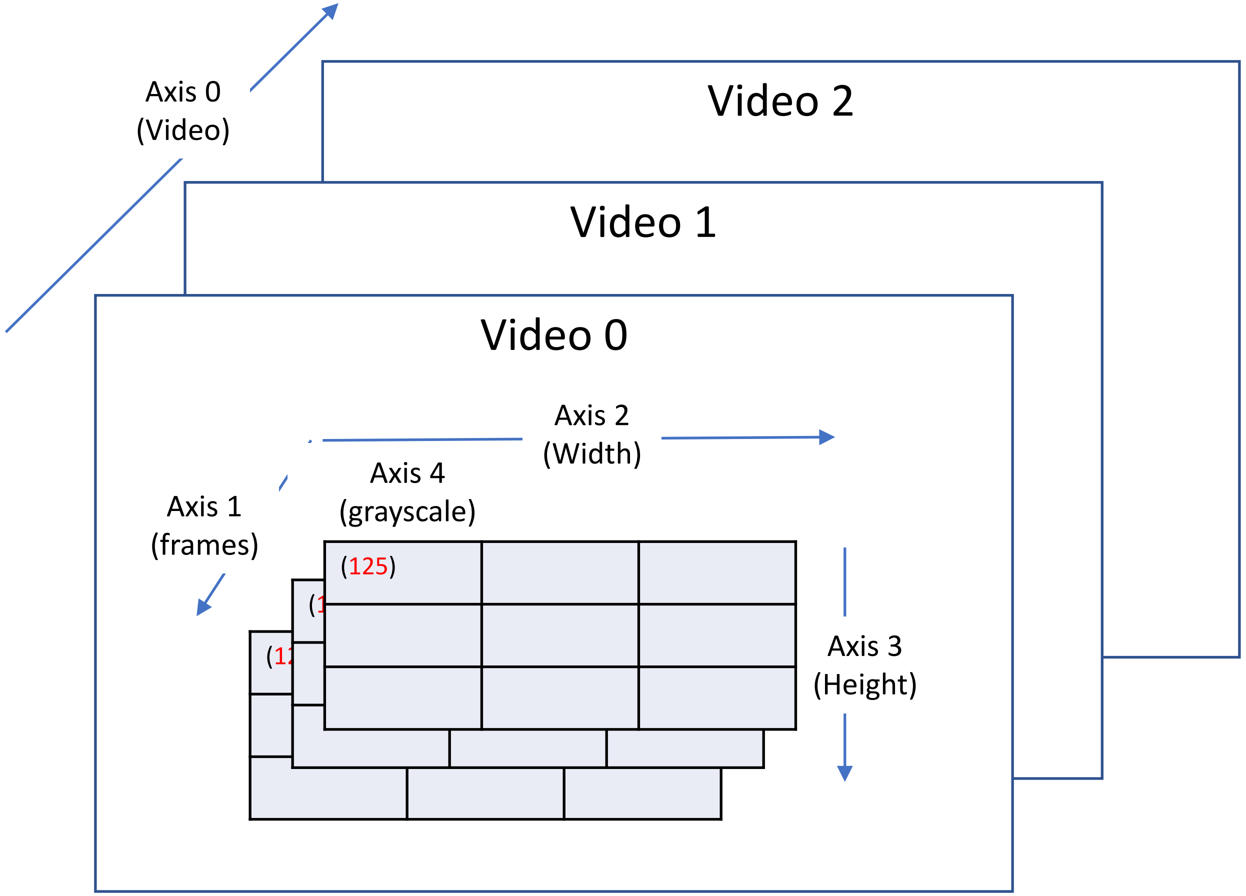
Tensors – Image Data

- TensorFlow uses 4D tensors to represent image data (multiple images).
  - Axis 0 – samples
  - Axis 1 – width,
  - Axis 2 – height
  - Axis 3 – channels. Channel refers to the colour information of each pixel.
- Note that other platforms might use different format for image data. For example, *Theano* uses a different format - samples, channels , width, height
- Each sample is a frame of image
- Each frame has width and height (number of pixels)
- Each channel can be 3 values (red, green and blue).
- Example
  - If you have 100 images, each image has resolution 1024 by 768 pixels. Each pixel has 3 values (red, green and blue).
  - Shape will be (100, 1024, 768, 3)



Tensors – Video Data

- Video can be considered a collection of image frames.
- 5D tensors (samples, frame, width, height, channels)
- Example
  - If you have 10 videos, each video is 10 seconds long and plays at 30 frames per second.
  - Each frame has resolution 176 by 144 pixels. Each pixel is greyscale with values from 0 to 255.
  - Shape will be (10, 300, 176, 144, 1)



Types of Tensors

- In TensorFlow, we can create tensors using either a constants or a variable.
- *Constants* are tensors and the values are not meant to change during the training process.
- Variables acts very much like a tensor. Strictly speaking, it is actually a type of data structure backed by a tensor. We use it to hold data that will vary during the training process.
  - If you use `tf.debugging.is_numeric_tensor()` on a variable, it will return false.
- We need to distinguish the difference between trainable and non-trainable variables.
- Trainable variables
  - The variables that the training process is trying to find the best value of.
  - For example, the β values in the linear regression algorithm ($y = \beta_1 x_1 + \beta_2 x_2 + \dots \beta_n x_n$) or the weights in a neural network.
- Non-Trainable variables
  - Other variables
  - For example, the hyper-parameter k used in k-NN.

Examples - Constants

```
import numpy as np
import tensorflow as tf
m = np.array([[[1, 7], [2, 8]], [ [3, 9], [4, 10]], [ [5, 11], [6, 12]]])
c = tf.constant(m)
print(c)
```

When you create a constant using TensorFlow, it is no longer a Numpy array, it will become a Tensor.

```
tf.Tensor(
[[[ 1  7]
  [ 2  8]]

 [[ 3  9]
  [ 4 10]]

 [[ 5 11]
  [ 6 12]]], shape=(3, 2, 2), dtype=int32)
```

Attributes like shape and data types are still applicable.

Example – Variables

```
import tensorflow as tf
k = tf.Variable(5, dtype=tf.int32, name="k")
param = tf.Variable(1.5, dtype=tf.float32, trainable=False, name="param1")
print(k)
print(param)
k.assign(10)
print(k)
```

```
<tf.Variable 'k:0' shape=() dtype=int32, numpy=5>
<tf.Variable 'param1:0' shape=() dtype=float32, numpy=1.5>
<tf.Variable 'k:0' shape=() dtype=int32, numpy=10>
```

We have created trainable and non-trainable variables. If not specified, a trainable variable will be created.

It will be better if we name the variables. This allows us to better identify the variables if necessary.
After creation, a variable’s value can be changed using the `assign()` method but the shape and data type are fixed.

Operations

- In addition to variables and constants, dataflow graph contains operations.
- Any defined operations become nodes within dataflow graph.
- Operations represent units of computation while tensors represent data.
- Unlike some other deep learning libraries, TensorFlow comes with a wide set of operations and is flexible enough to be used for other purposes.
- Most of the mathematical operations are already provided by TensorFlow
    - Mathematics (Example add and multiply functions)
    - Array Manipulations
    - Control Flow
    - Statement Managements

Module	Available Python API
tf	tf.abs
	tf.acosh
	tf.add
	tf.add_n
	tf.angle
	tf.arg_max
	tf.arg_min
	tf.asinh
	tf.assign
	tf.assign_add
	tf.assign_sub
	tf.atan
	tf.atan2
	tf.atanh
	tf.batch_to_space
	tf.batch_to_space_nd
	tf.broadcast_dynamic_shape
	tf.broadcast_static_shape

TensorFlow Operations

- TensorFlow operations has different syntax to achieve the same effect:

```
import tensorflow as tf

var1=tf.Variable([1.,2.],tf.float32, name='var_1')
var2=tf.Variable([3.,4.],tf.float32, name='var_2')

print(tf.add(var1, var2))
print(var1 + var2)
```

The result is the same:

```
tf.Tensor([4. 6.], shape=(2,), dtype=float32)
tf.Tensor([4. 6.], shape=(2,), dtype=float32)
```

TensorFlow Operations

- Very often, when constructing operations in TensorFlow, we need to work with tensor object instead of data structures like Numpy arrays, in such cases, we can use the `tf.convert_to_tensor()` function to perform the conversion.
- Example of `convert_to_tensor()`:

```
python_list = [1, 2, 3]
numpy_array = np.array([4, 5, 6])

tensor1 = tf.convert_to_tensor(python_list)
tensor2 = tf.convert_to_tensor(numpy_array)
```

TensorFlow `tf.function`

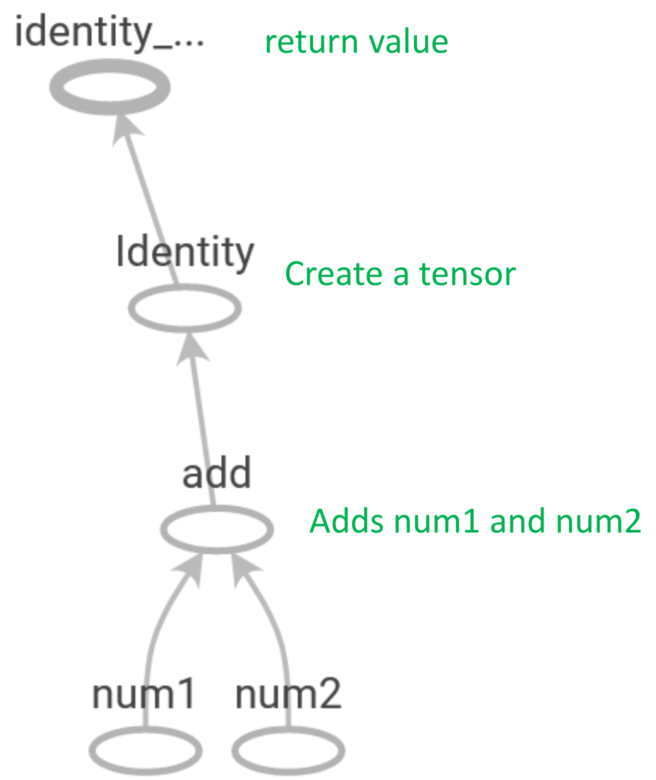
- In TensorFlow 2.x, Google encourages the use of the `tf.function` annotation when training our model.
- After we have debugged and ensured that the training functions are working properly, we should use the `tf.function` annotation for the computational intensive training process.
- The `tf.function` annotation is a tool that helps us generated Python-independent dataflow graphs out of our Python code, it uses the AutoGraph feature of TensorFlow.
- Basically AutoGraph converts `tf.function` annotated Python function into dataflow graph.
- These graphs are wrapped around by a *concrete function* that allows us to use the graph as if it is a function.
- The `tf.function` annotation also helps us create portable models, and it is required to use *SavedModel*.

TensorFlow `tf.function`

Use the @ symbol to annotate the Python function as a tf.function

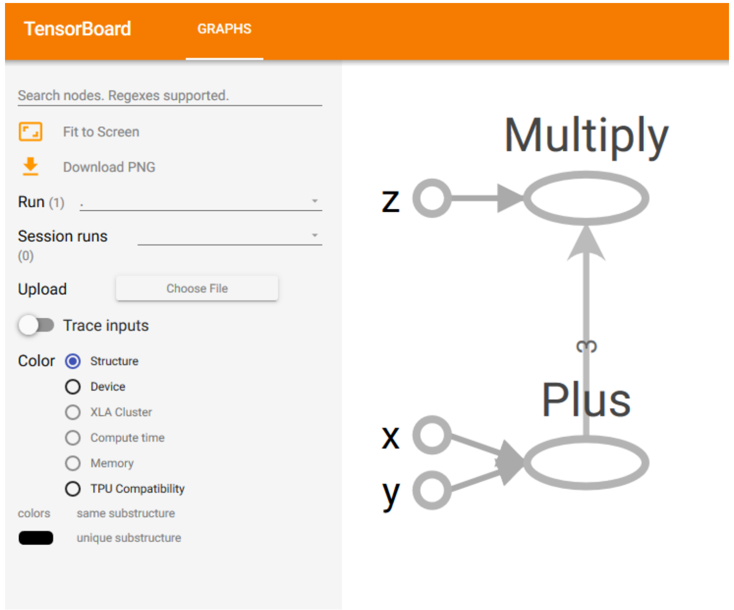
```
@tf.function
def add(num1, num2):
      return num1 + num2

x1 = tf.constant([1, 2])
x2 = tf.constant([3, 4])
add(x1, x2)
```

The annotated Python function will be compiled into a dataflow graph



TensorBoard

- Very often, training with large amount of data can take a long time.
- It can be instructive for us to view how values change over time during the training phase.
- Google provides the TensorBoard for visualization and makes using TensorFlow easier.
- With TensorBoard, we can see
  - Flowchart of the layout of the dataflow graph
  - How the data flows and are processed
  - How the operations are connected
  - Summary logs
  - Performance traces



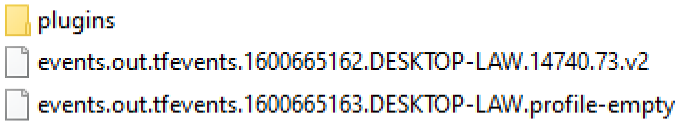
The graph created by the codes in the previous slide is visualized as shown:

TensorBoard

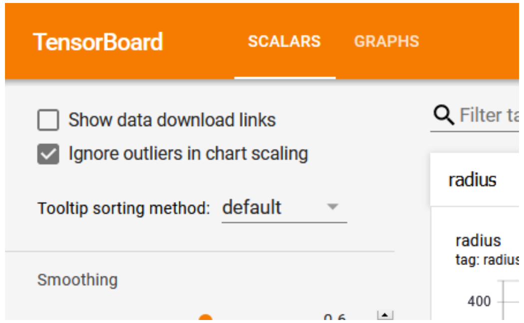
```
log_dir = "\log\myfiles"
writer = tf.summary.create_file_writer(log_dir)
with writer.as_default():
        tf.summary.scaler("myvalues", 100, step)
```

- TensorBoard also allows us to visualize scalar, histogram, images and even precision-recall curves using TensorBoard.
- In our TensorFlow codes, we write data to a log directory.
    - Data to be visualized are written with the help of the `tf.summary` module.
- We then run the TensorBoard pointing to the same log directory.
- On anaconda, you need to stop your Jupyter Notebook kernel before running TensorBoard. Otherwise, as the Jupyter Notebook is still holding on to the file, TensorBoard will not be able to load the data!

"\log\myfiles"




```
Tensorboard --logdir \log\myfiles
```



Optimizers

- TensorFlow optimizers allow us to derive parameters that minimize a certain loss or cost functions.
- Some optimizers included are:
  - Gradient Descent with Momentum
  - Adadelta
  - Adagrad Dual Averaging
  - Proximal Adagrad
  - Adam
  - Adamax
  - NAdam
  - FTRL

Optimizers

To use optimizers, we

1. Define the cost function
2. Instantiate an instance of the required optimizer
3. Set the relevant parameters
4. Call the `minimize()` function iteratively.

Defines a cost function

Creates an optimizer

```
loss = lambda: (actual - predicted)**2
opt = tf.keras.optimizers.SGD(learning_rate=0.01)

for in in range(10):
       opt.minimize(loss, [weight, bias])
```

Calls the `minimize()` function

List of variables that will be updated to minimize the loss function.

Saving Models

- Sometimes, we might need to save our models either during training or for deployment.
- We can either do a *checkpoint* or *SavedModel*.
- Checkpoint
  - Captures all parameters used by a model.
  - Does not contain any description of the computation defined by the model
  - Only useful when source code that will use the saved parameter values is available.
- SavedModel
  - Includes the checkpoint data and a serialized description of the computation defined by the model
  - Does not need the source codes in order to perform inference, and thus useful for deployment.

Keras Checkpoint/SavedModel

- The `tf.keras.Model` provides
  - A `save_weights()` function that makes it easy to save a checkpoint.
  - A `save()` function that makes it easy to save as a TensorFlow SavedModel.

```
class Net(tf.keras.Model):
  """A simple linear model."""

  def __init__(self):
    super(Net, self).__init__()
    self.l1 = tf.keras.layers.Dense(5)

  def call(self, x):
    return self.l1(x)

net = Net()

net.save_weights('myfile')

net.save('myfile', save_format='tf')
```

Save as checkpoint

Save as SavedModel

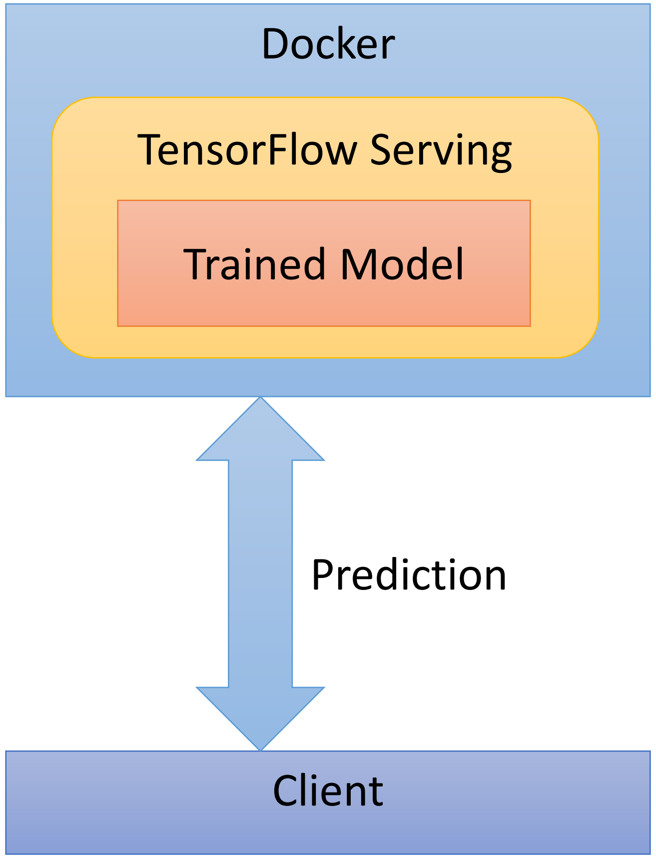
Manual Checkpoint

- We can use `tf.train.Checkpoint` to manually write variables (models data are stored as variables) to files.
- The two functions to use are `save()` and `restore()`.
- An example is shown on the right to illustrate the use of the two functions.
- The `assert_consumed()` function checks if the loading is complete and there are no errors (e.g. mismatch variables)

```
import tensorflow as tf
import os

#var is the variable to save
var = tf.Variable(0, name="var")

#Create a checkpoint
#and indicate the variable to track (i.e. var)
checkpoint = tf.train.Checkpoint(var=var)

#Save
checkpoint.save("/tmp/cp1")

#Restore
status = checkpoint.restore("/tmp/cp1").assert_consumed()
```

TensorFlow Serving

- After we have trained our model, we want to use it for prediction.
- One possibility is to use TensorFlow Serving
  - Designed for production environments
  - Makes it easy to deploy new algorithms and experiments
  - Provides out-of-the-box integration with TensorFlow models
  - Can be easily extended to serve other types of models and data.
- Refer to https://www.tensorflow.org/tfx/serving/serving_basic for tutorial on deploying and serving a TensorFlow model.



Summary

- Deep Learning Frameworks
- Tensorflow Platform and Hub
- Tensors, dimension, ranks and axes
- Trainable vs Non-trainable variables
- Tensorflow Operations
- The tf.function annotation
- TensorBoard
- Optimizers
- Saving and Deployment