MACHINE LEARNING

TENSORS

Tensors

- Understanding tensors, the basic data structure in PyTorch
- Indexing and operating on tensors
- Interoperating with NumPy multidimensional arrays
- Moving computations to the GPU for speed

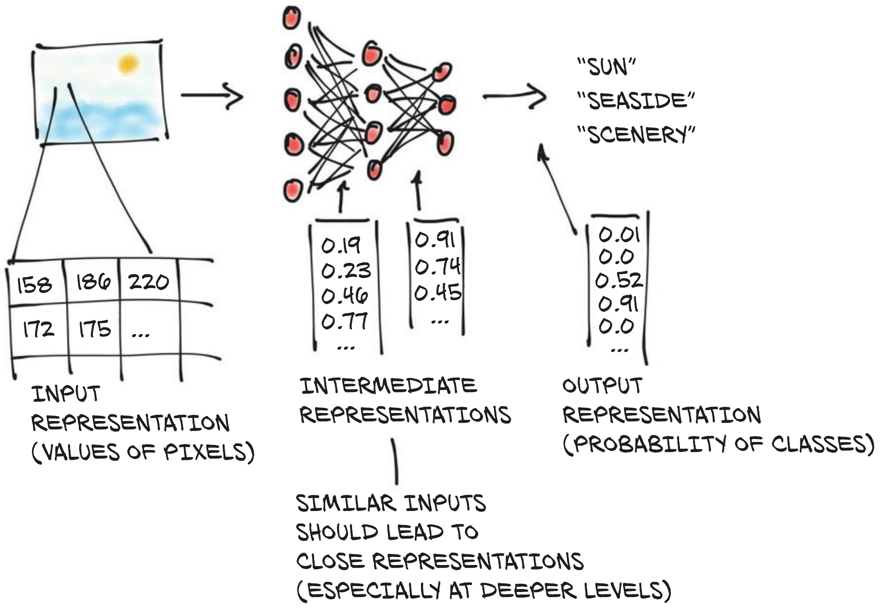
Representation as floating-point numbers

Neural nets work with floating-point (FP) numbers.

We need a way

- to **encode real-world data** into a FP representation, and then
- to **decode the FP output** into something we can interpret

Neural nets learns the transformation in stages (sequence of intermediate representations). Also FP numbers.



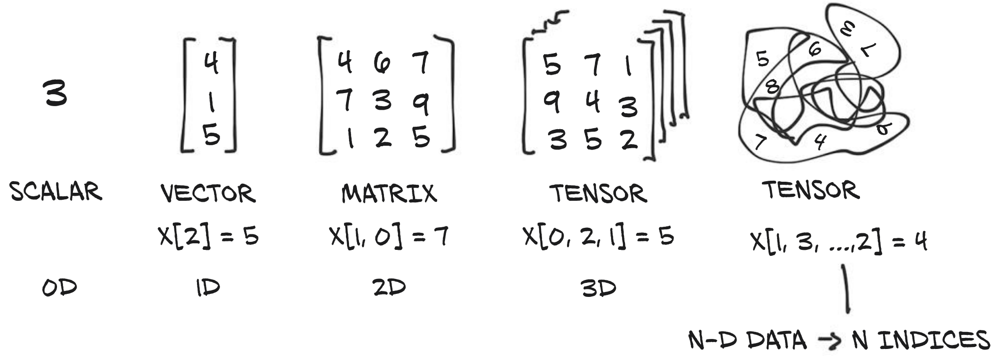
"SUN"
"SEASIDE"
"SCENERY"
158
186
220
172
175
...
0.19
0.23
0.46
0.77
...
0.91
0.74
0.45
...
0.01
0.0
0.52
0.91
0.0
...
INPUT REPRESENTATION (VALUES OF PIXELS)
INTERMEDIATE REPRESENTATIONS
OUTPUT REPRESENTATION (PROBABILITY OF CLASSES)
SIMILAR INPUTS SHOULD LEAD TO CLOSE REPRESENTATIONS (ESPECIALLY AT DEEPER LEVELS)

Multidimensional arrays

Multidimensional array → generalization of matrices to an **arbitrary number of dimensions.**

In terms of data, a tensor is an array: a data structure that stores a collection of numbers that are accessible individually using an index, and that can be indexed with multiple indices.

We extend this notion of tensor to include in this data structure other attributes to use them as building blocks of **computational graphs**.



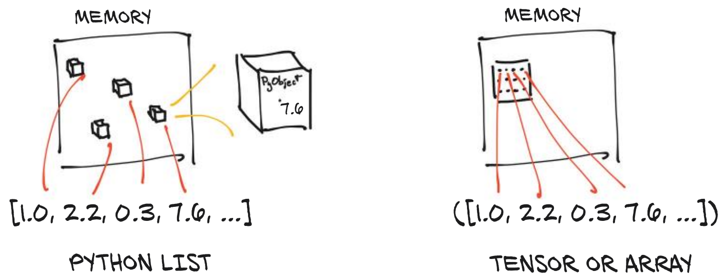
TENSORS

- *Understanding tensors*
- *Building tensors*

Constructing tensors

Import torch module. Creation of a **one-dimensional array, size 3, all values 1**

```
import torch
a = torch.ones(3)
a[1]
float(a[1])
a[2] = 2.0
a
```



List are sequences of objects that are individually allocated in memory (container of objects, like a dictionary).

NumPy arrays or PyTorch tensors are views of contiguous memory blocks containing unboxed C numeric types (32 bit – 4 byte – float in this case), **plus some metadata**

Building tensors, higher dimensionality

Starting from an empty array or using a lists (or lists of lists).

```
points = torch.zeros(6)
points[0] = 4.0
points[1] = 1.0
points[2] = 5.0


points = torch.tensor([4.0, 1.0, 5.0, 3.0, 2.0, 1.0])
points


float(points[0]), float(points[1])


points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points

points.shape
```

Views, indexing

```
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
```

```
points[0, 1]
```

The output `points[0, 1]` represents a different view of the same underlying data. A portion of the same chunk of memory.

Tensors are indexed and sliced as other sequences (lists, tuples) in Python.

```
points[:]

points[1:]

points[1:, :]

points[0, :]

points[1:, 0]
```

Tensor element types

The dtype argument to tensor constructors (that is, functions like tensor, zeros, and ones) specifies the numerical data (d) type that will be contained in the tensor.

Similar to the standard NumPy argument of the same name

- **torch.float32** or torch.float: 32-bit floating-point (typical representation)
- torch.float64 or torch.double: 64-bit, double-precision floating-point (rarely justified)
- torch.float16 or torch.half: 16-bit, half-precision floating-point (offered in modern GPU, TPU)
- ….
- **torch.int64** or torch.long: signed 64-bit integers (required for indexing, default for int)
- **torch.bool**: Boolean

Dtype specification and casting

Creating tensors of a specific dtype and casting

```
short_points = torch.tensor([[1, 2], [3, 4]], dtype=torch.short)

short_points = torch.ones(10, 2).to(dtype=torch.short)

double_points = torch.zeros(10, 2).to(torch.double)
```

TENSORS

- *Understanding tensors*
- *Building tensors*
- *Tensor API*

The tensor API

https://pytorch.org/docs

Most **operations** on and between tensors are available in the ***torch*** **module** and can **also be called as methods** of a tensor object. For instance, the transpose:

```
a = torch.ones(3, 2)
a_t = torch.transpose(a, 0, 1)
```

$\longleftarrow$ **Function from torch module**

```
a.shape, a_t.shape
```

```
a = torch.ones(3, 2)
a_t = a.transpose(0, 1)
```

$\longleftarrow$ **Method on the object**

```
a.shape, a_t.shape
```

Tensor operations, summary

- ***Creation ops*** —Functions for constructing a tensor, like ones and from_numpy
- ***Indexing, slicing, joining, mutating ops***—Functions for changing the shape, stride, or content of a tensor, like transpose
- ***Math ops***—Functions for manipulating the content of the tensor through computations
- ***Random sampling***—Functions for generating values by drawing randomly from probability distributions, like rand and normal
- ***Serialization*** —Functions for saving and loading tensors, like load and save
- ***Parallelism***—Functions for controlling the number of threads for parallel CPU execution, like set_num_threads

Math ops

- ***Math ops***—Functions for manipulating the content of the tensor through computations
  - *Pointwise ops*—Functions for obtaining a new tensor by applying a function to each element independently, like abs and cos
  - *Reduction ops*—Functions for computing aggregate values by iterating through tensors, like mean, std, and norm
  - *Comparison ops*—Functions for evaluating numerical predicates over tensors, like equal and max
  - *Spectral ops*—Functions for transforming in and operating in the frequency domain
  - *Other operations* —Special functions operating on vectors, like cross, or matrices, like trace
  - *BLAS and LAPACK operations*—Functions following the Basic Linear Algebra Subprograms (BLAS) specification for scalar, vector-vector, matrix-vector, and matrix-matrix operations

Examples, scalar multiplication

If *k* is a scalar and **A** is a *n*-dimensional vector, then

$$k\mathbf{A} = k\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} ka_1 \\ ka_2 \\ \vdots \\ ka_n \end{bmatrix}$$

$$2\begin{bmatrix} 25 \\ 20 \\ 5 \end{bmatrix} = \begin{bmatrix} 2\times 25 \\ 2\times 20 \\ 2\times 5 \end{bmatrix} = \begin{bmatrix} 50 \\ 40 \\ 10 \end{bmatrix}$$

```
a = torch.tensor([[1, 2]], dtype=torch.int64)

a * 3


""

Out[77]: tensor([[3, 6]])

""
```

Example, pointwise summation

Each element of the tensor input is summed with the corresponding element of the Tensor other. The resulting tensor is returned.

Example: addition/substraction with vectors (also multiplication, division, etc).

$$\mathbf{A} \pm \mathbf{B} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \pm \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 \pm b_1 \\ a_2 \pm b_2 \\ \vdots \\ a_n \pm b_n \end{bmatrix}$$

$$\begin{bmatrix} 2 \\ 3 \\ 4 \\ 1 \end{bmatrix} + \begin{bmatrix} 5 \\ -2 \\ 3 \\ 7 \end{bmatrix} = \begin{bmatrix} 2+5 \\ 3-2 \\ 4+3 \\ 1+7 \end{bmatrix} = \begin{bmatrix} 7 \\ 1 \\ 7 \\ 8 \end{bmatrix}$$

```
a = torch.tensor([[1, 2]], dtype=torch.int64)

b = torch.tensor([0, 4], dtype=torch.int64)


a + b
"""
Out[78]: tensor([[1, 6]])
"""
```

Broadcastability

For many operations, like pointwise multiplications, t shapes of the inputs must be equal or, in general, broadcastable.

```
a = torch.randn(2, 1)
b = torch.randn(2, 1)
a + b
```

```
a
tensor([[-1.4422],
        [-0.5490]])
b
tensor([[ 1.0545],
        [-0.7491]])

a + b
Out[3]:
tensor([[-0.3878],
        [-1.2982]])
```

```
a = torch.randn(2, 1)
b = torch.randn(1, 2)
a + b
```

```
a
tensor([[-1.4422],
        [-0.5490]])
b
tensor([[-0.4344, -0.7714]])

a + b
Out[3]:
tensor([[-1.8766, -2.2136],
        [-0.9835, -1.3204]])
```

Broadcasting and broadcasting semantics

https://pytorch.org/docs/stable/notes/broadcasting.html#broadcasting-semantics

Concept

If a PyTorch operation supports broadcast, then its Tensor arguments can be automatically expanded to be of equal sizes (without making copies of the data).

General semantics

Two tensors are “broadcastable” if the following rules hold:

- Each tensor has at least one dimension.
- When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist.

Broadcasting and broadcasting semantics

Calculation

If two tensors x, y are **“broadcastable”**, the resulting tensor size is calculated as follows:

- If the number of dimensions of x and y are not equal, **prepend 1** to the dimensions of the tensor with fewer dimensions to make them equal length.
- Then, for each dimension size, **the resulting dimension size is the max** of the sizes of x and y along that dimension.

Broadcasting and broadcasting semantics

```
>>> x=torch.empty(5,7,3)
>>> y=torch.empty(5,7,3)
# same shapes are always broadcastable (i.e. the above rules always hold)

>>> x=torch.empty((0,))
>>> y=torch.empty(2,2)
# x and y are not broadcastable, because x does not have at least 1 dimension

# can line up trailing dimensions
>>> x=torch.empty(5,3,4,1)
>>> y=torch.empty(  3,1,1)
# x and y are broadcastable.
# 1st trailing dimension: both have size 1
# 2nd trailing dimension: y has size 1
# 3rd trailing dimension: x size == y size
# 4th trailing dimension: y dimension doesn't exist

# but:
>>> x=torch.empty(5,2,4,1)
>>> y=torch.empty(  3,1,1)
# x and y are not broadcastable, because in the 3rd trailing dimension 2 != 3
```

Broadcasting example with add

Example of boardcasting pointwise add operation

```
# can line up trailing dimensions to make reading easier
>>> x=torch.empty(5,1,4,1)
>>> y=torch.empty(  3,1,1)
>>> (x+y).size()
torch.Size([5, 3, 4, 1])

# but not necessary:
>>> x=torch.empty(1)
>>> y=torch.empty(3,1,7)
>>> (x+y).size()
torch.Size([3, 1, 7])

>>> x=torch.empty(5,2,4,1)
>>> y=torch.empty(3,1,1)
>>> (x+y).size()
RuntimeError: The size of tensor a (2) must match the size of tensor b (3) at non-singleton
dimension 1
```

Broadcast and compute

```
import torch
a = torch.tensor([[1, 2]], dtype=torch.int64)
b = torch.tensor([[3], [4]], dtype=torch.int64)

torch.mul(a, b)
```

Are a and b broadcastable?

What is the shape of torch.mul(a, b)?

Obtain result.

Math ops, reduction and comparison, examples

Examples of reduction (aggregation) and comparison.

```
a = torch.tensor([[1, 2]], dtype=torch.int64)
a.sum()

b = torch.tensor([0, 4], dtype=torch.int64)
a < b
```

Math ops, matrix product of two tensors

Matrix product of two tensors.

The behavior depends on the dimensionality of the tensors as follows:

- If both tensors are 1-dimensional, the dot product (scalar) is returned.
- If both arguments are 2-dimensional, the matrix-matrix product is returned.
- If the first argument is 1-dimensional and the second argument is 2-dimensional, a 1 is prepended to its dimension for the purpose of the matrix multiply. After the matrix multiply, the prepended dimension is removed.
- If the first argument is 2-dimensional and the second argument is 1-dimensional, the matrix-vector product is returned.
- If both arguments are at least 1-dimensional and at least one argument is N-dimensional (where N > 2), then a batched matrix multiply is returned. I

Dot (scalar) product

Let $\mathbf{u} = [u_1, u_2, \ldots, u_n]$ $\mathbf{v} = [v_1, v_2, \ldots, v_n]$

be two vectors of length n. Then the dot product of the two vectors u and v is defined as

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + \ldots + u_n v_n = \sum_{i=1}^{n} u_i v_i$$

```
# If both tensors are 1-dimensional, the dot product (scalar) is returned.


a = torch.tensor([1, 2])

b = torch.tensor([3, 4])

torch.matmul(a, b)


""

Out[65]: tensor(11)

""
```

Matrix-matrix multiplication

Matrix Product AB: Number of columns in the first matrix must equal the number of rows in the second matrix, e.g., [m × k][k × n] = [m × n]

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & & & \\ a_{3,1} & & a_{3,3} & & \vdots \\ \vdots & & & \ddots & \\ a_{m,1} & & \cdots & & a_{m,n} \end{bmatrix} \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & \cdots & b_{1,m} \\ b_{2,1} & b_{2,2} & & & \\ b_{3,1} & & b_{3,3} & & \vdots \\ \vdots & & & \ddots & \\ b_{n,1} & & \cdots & & b_{n,m} \end{bmatrix}$$

= [m × n] matrix whose (i,j) entry is the dot product of the ith row of A and the jth column of B.

Matrix-matrix multiplication

Example in Pytorch

```
# If both arguments are 2-dimensional, the matrix-matrix product is returned.

a = torch.tensor([[1, 2], [3, 2]])
b = torch.tensor([[3, 4], [3, 4]])
torch.matmul(a, b)
""""
Out[66]:
tensor([[ 9, 12],
        [15, 20]])
""""
```

Vector-matrix multiplication

Broadcasat in vector, then matrix-matrix multplication, then prepended dimension removed.

```
# If the first argument is 1-dimensional and the second argument is 2-dimensional, a 1 is
# prepended to its dimension for the purpose of the matrix multiply. After the matrix multiply,
# the prepended dimension is removed.

a = torch.tensor([1, 2])
b = torch.tensor([[3, 4], [3, 4]])
torch.matmul(a, b)

"""
Out[67]: tensor([ 9, 12])
"""
```

Matrix-vector multiplication

$$A\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}$$

```
# If the first argument is 2-dimensional and the second argument is 1-dimensional,
# the matrix-vector product is returned.

b = torch.tensor([[3, 4], [3, 4]])
a = torch.tensor([1, 2])
torch.matmul(b, a)

"""
Out[68]: tensor([11, 11])
"""
```

Batched matrix multiplication

Extension in Python

```
# If both arguments are at least 1-dimensional and at least one argument is N-dimensional
# (where N > 2), then a batched matrix multiply is returned.

a = torch.ones(3, 2, 2)
b = torch.tensor([[[2, 2]], [[3, 3]], [[4, 4]]], dtype=torch.float32)
torch.matmul(b, a)

"""
Out[76]:
tensor([[[4., 4.]],
        [[6., 6.]],
        [[8., 8.]]])
"""
```

TENSORS

- *Understanding tensors*
- *Building tensors*
- *Tensor API*
- *Storage, offset and stride*

PyTorch implementation of tensor data

Values in tensors are allocated in **contiguous chunks of memory managed by *torch.Storage* instances**.

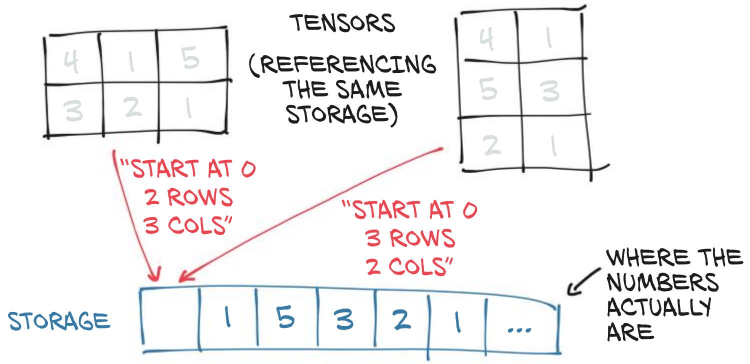
A **storage** is a **one-dimensional array of numerical data**: that is, a contiguous block of memory containing numbers of a given type, such as float (32 bits representing a floating-point number) or int64 (64 bits representing an integer).

A PyTorch Tensor instance is a view of such a Storage instance that is capable of indexing into that storage using an **offset** and per-dimension **strides**.

torch.Storage, offset and stride

Multiple tensors can index the same storage even if they index into the data differently.

The underlying memory is allocated only once, however, so creating alternate **tensor-views** of the data can be **done quickly regardless of the size of the data** managed by the Storage instance.



torch.Storage, offset and stride

In order to index into a storage, tensors rely on a few pieces of information that, together with their storage, unequivocally define them: size, offset, and stride.

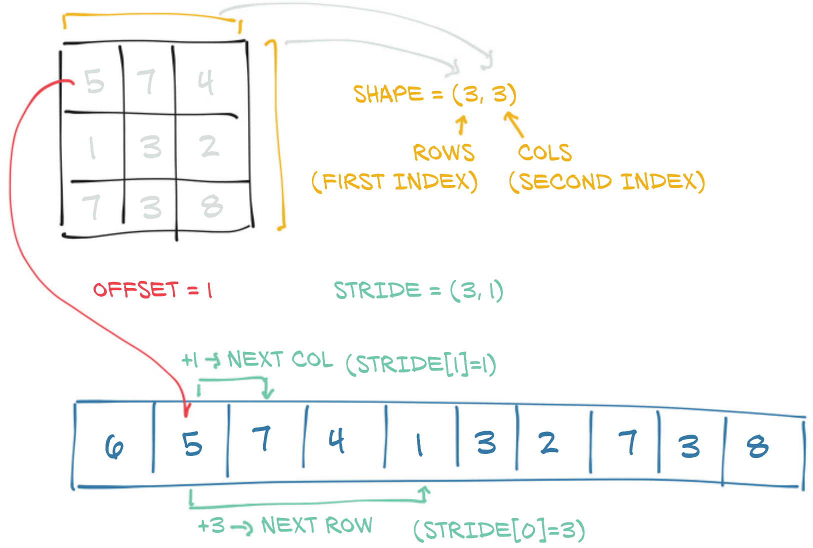
The **size** (or shape, in NumPy parlance) is a tuple indicating how many elements across each dimension the tensor represents.

The storage **offset** is the index in the storage corresponding to the first element in the tensor.

The **stride** is the number of elements in the storage that need to be skipped over to obtain the next element along each dimension.

Shape, offset and stride

Relationship between a tensor's offset, size, and stride. Here the tensor is a view of a larger storage, like one that might have been allocated when creating a larger tensor.



Shape, offset and stride

Accessing an element i, j in a 2D tensor results in accessing the storage_offset + stride[0] * i + stride[1] * j element in the storage.

The offset will usually be zero; if this tensor is a view of a storage created to hold a larger tensor, the offset might be a positive value.

```
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points.size()
"""
Out[92]: torch.Size([3, 2])
"""

second_point = points[1]
second_point.storage_offset()
"""
Out[93]: 2
"""

second_point.size()
"""
Out[94]: torch.Size([2])
"""
```

Shape, offset and stride

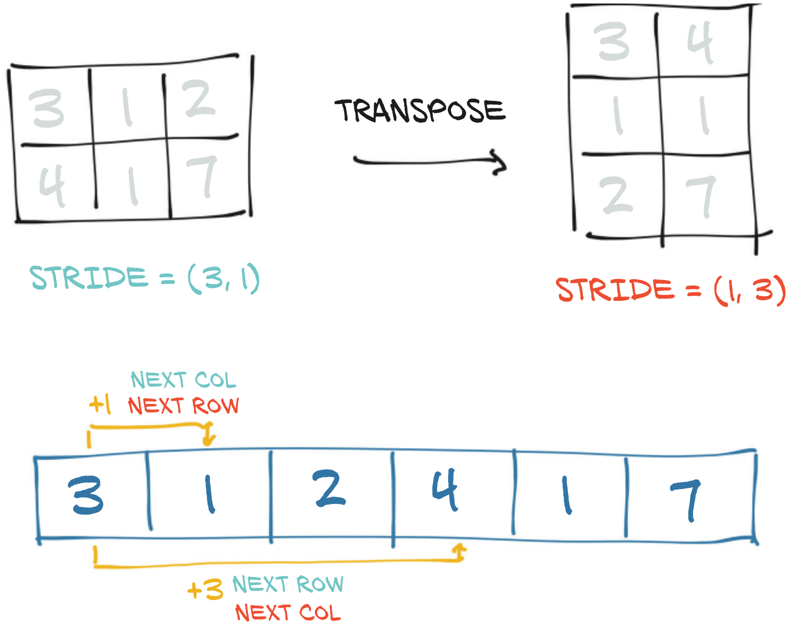
Anticipate output:

```
points = torch.tensor([[4.0, 1.0, 5.0], [5.0, 3.0, 6.0], [2.0, 1.0, 7.0]])
points.size()
"""
Out[1]: torch.Size([_, _])
"""


point_view = points[2]
point_view.storage_offset()
"""
Out[2]: _
"""


point_view.size()
"""
Out[3]: torch.Size([_])
```

Operations without data copy

Example, transpose.

```
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points_t = points.t()
```



Tensors

- *Understanding tensors*
- *Building tensors*
- *Tensor API*
- *Storage, offset and stride*
- *Processors: CPUs, GPUs, TPUs and CUDA*

GPU and TPU

Until know, we have been looking mostly at the data stored in the tensor, and how we accesss it using indices.

Another characteristic of the tensor, beyond data, is that **it resides in a processor**.

So far when we’ve talked about storage, we’ve meant memory on the CPU.

PyTorch tensors also can be stored on a different kinds of processors: a **graphics processing unit (GPU)** and a **tensor processing unit (TPU).**

- Tensors can be transferred to (one of) the GPU(s)/TPU(s) in order to perform massively parallel, fast computations.

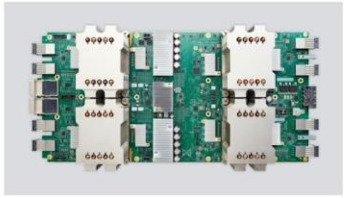
ASIC (application-specific integrated circuits)

Universidad Francisco de Vitoria
UFV Madrid

Both GPU and TPU as are custom application-specific integrated circuits.

GPUs are ASIC devices designed designed to improve the performance of **graphics manipulation and output.**

TPUs are ASIC devices designed specifically to handle the computational demands of ML, customized for **very fast tensor operations (matrix and vector multiplications)**


Cloud TPU v2
180 teraflops
64 GB High Bandwidth Memory (HBM)

Cloud TPU v3
420 teraflops
128 GB HBM

Cloud TPU v2 Pod
11.5 petaflops
4 TB HBM
2-D toroidal mesh network

Cloud TPU v3 Pod
100+ petaflops
32 TB HBM
2-D toroidal mesh network

CPU

The CPU is a general-purpose processor based on the von Neumann architecture. Memory stores data and program.

CPU is flexible (it can run any type of program). But the hardware doesn't always know what the next calculation is until it reads the next instruction.

Each CPU's Arithmetic Logic Units (ALUs), which are the components that hold and control multipliers and adders, can execute only one calculation at a time. Each time, the CPU must access memory, which limits the total throughput and consumes significant energy.

GPU

To gain higher throughput than a CPU, a **GPU employs thousands of ALUs in a single processor** (modern ones, 2,500–5,000 ALUs in a single processor). This enables **large amounts of parallel multiplications** and additions.

In 2007, NVIDIA developed a parallel computing platform and **application programming interface called CUDA**, which **enabled developers to use GPUs for almost all types of vector, scalar, or matrix multiplication and addition**.

- On a typical ML training: GPU ~ order of magnitude higher throughput than a CPU.

But for every single calculation in the thousands of ALUs, a GPU must access registers or shared (very fast) memory to read and store the intermediate calculation results.

TPU

Tensor Processing Unit (TPU) is an AI accelerator application-specific integrated circuit (ASIC) developed by Google **specifically for neural network machine learning.**

Multipliers and adders are connected to each other directly to form a large physical matrix of those operators.

- As each multiplication is executed, the result will be passed to the next multipliers while taking the summation at the same time.
- No memory access is required at all during the parallel computations

Tensors

- *Understanding tensors*
- *Building tensors*
- *Tensor API*
- *Storage, offset and stride*
- *Processors: CPUs, GPUs, TPUs and CUDA*
- *Access to GPUs-TPUs*

Access to GPU and TPU

- Own a machine with a cuda enabled GPU
- Cloud hosted Jupyter notebook services with GPU/TPU (Google Colab)
- Cloud virtual machines

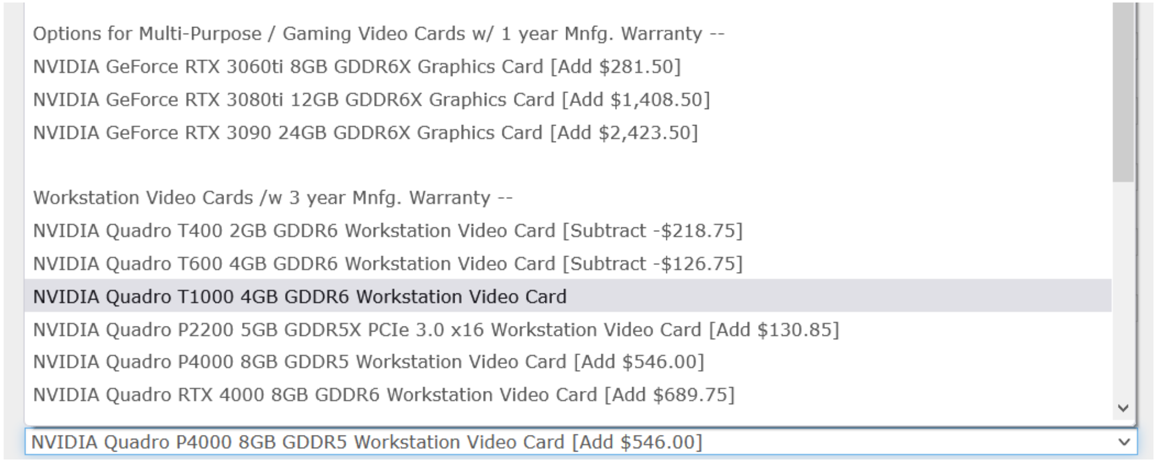
Tensors

- *Understanding tensors*
- *Building tensors*
- *Tensor API*
- *Storage, offset and stride*
- *Processors: CPUs, GPUs, TPUs and CUDA*
- *Access to GPUs-TPUs*
  - Local GPUs

CUDA GPU Workstations

Expensive solution

Also external GPU casings.



Options for Multi-Purpose / Gaming Video Cards w/ 1 year Mnfg. Warranty --
NVIDIA GeForce RTX 3060ti 8GB GDDR6X Graphics Card [Add $281.50]
NVIDIA GeForce RTX 3080ti 12GB GDDR6X Graphics Card [Add $1,408.50]
NVIDIA GeForce RTX 3090 24GB GDDR6X Graphics Card [Add $2,423.50]

Workstation Video Cards /w 3 year Mnfg. Warranty --
NVIDIA Quadro T400 2GB GDDR6 Workstation Video Card [Subtract -$218.75]
NVIDIA Quadro T600 4GB GDDR6 Workstation Video Card [Subtract -$126.75]
NVIDIA Quadro T1000 4GB GDDR6 Workstation Video Card
NVIDIA Quadro P2200 5GB GDDR5X PCIe 3.0 x16 Workstation Video Card [Add $130.85]
NVIDIA Quadro P4000 8GB GDDR5 Workstation Video Card [Add $546.00]
NVIDIA Quadro RTX 4000 8GB GDDR6 Workstation Video Card [Add $689.75]

NVIDIA Quadro P4000 8GB GDDR5 Workstation Video Card [Add $546.00]

https://www.titancomputers.com/Multi-CUDA-GPU-Workstation-s/954.htm

CUDA semantics for machines with local GPU

Universidad Francisco de Vitoria
UFV Madrid

Multiple ways of creating tensors in CUDA devices (GPU):

- Create the device object using torch.device and then pass it as *device* argument.
- Using the .cuda method
- Within a with construct
- .to(device=cuda) to move a tensor

```
d = torch.randn(2, device=cuda2)

e = torch.randn(2).to(cuda2)

f = torch.randn(2).cuda(cuda2)
```

```
import torch

cuda = torch.device('cuda')     # Default CUDA device
cuda0 = torch.device('cuda:0')
cuda2 = torch.device('cuda:2')  # GPU 2 (these are 0-indexed)

x = torch.tensor([1., 2.], device=cuda0)
print(f'x: {x}')
# x.device is device(type='cuda', index=0)
y = torch.tensor([1., 2.]).cuda()
print(f'y: {y}')
# y.device is device(type='cuda', index=0)

with torch.cuda.device(0):
   # allocates a tensor on GPU 0
   a = torch.tensor([1., 2.], device=cuda)
   print(f'a: {a}')
```

TENSORS

- *Understanding tensors*
- *Building tensors*
- *Tensor API*
- *Storage, offset and stride*
- *Processors: CPUs, GPUs, TPUs and CUDA*
- *Access to GPUs-TPUs*
  - Local GPUs
  - Google Colab

Jupyter Notebooks

Jupyter Notebook: a page in the browser through which we can run code interactively.

Install with pip/conda (assisted by PyCharm) like any other package:

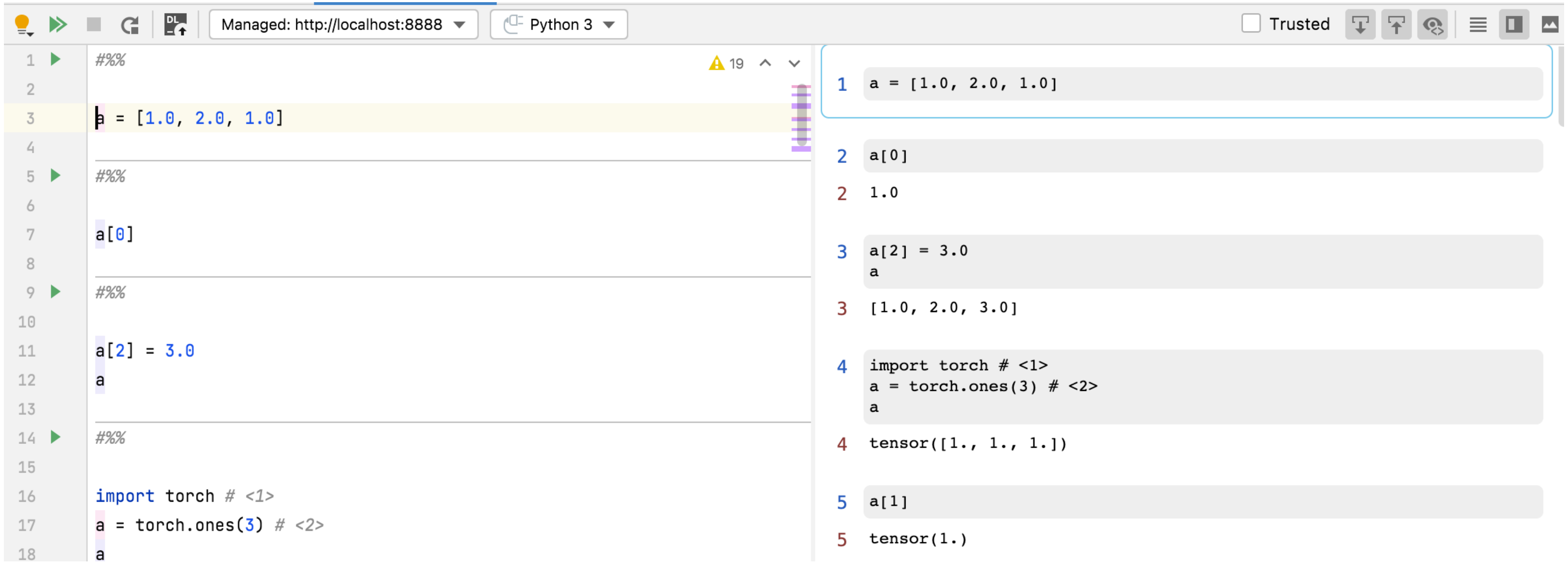
- https://jupyter.org/install

The code is evaluated by a *kernel*, a process running on a server that is ready to receive code to execute and send back the results, which are then **rendered inline on the page**.

The fundamental unit with which we interact with a notebook is a ***cell***: a box on the page where we can type code and have the kernel evaluate it

Jupyter Notebooks on Pycharm

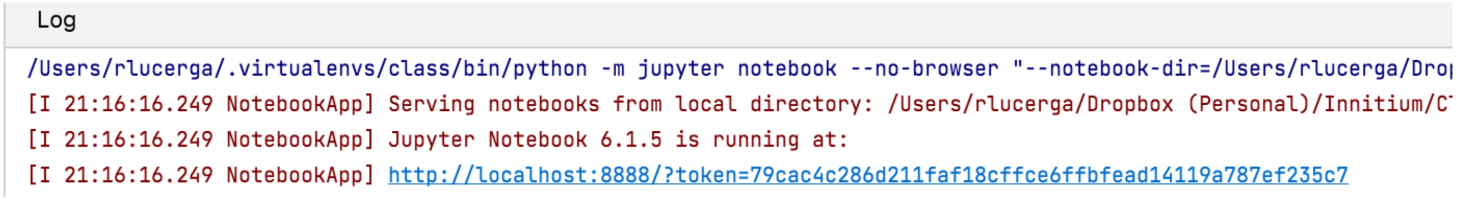
Install Jupyter library



```
#%%

a = [1.0, 2.0, 1.0]

#%%

a[0]

#%%

a[2] = 3.0
a

#%%

import torch # <1>
a = torch.ones(3) # <2>
a
```

```
1  a = [1.0, 2.0, 1.0]

2  a[0]
2  1.0

3  a[2] = 3.0
   a
3  [1.0, 2.0, 3.0]

4  import torch # <1>
   a = torch.ones(3) # <2>
   a
4  tensor([1., 1., 1.])

5  a[1]
5  tensor(1.)
```

Jupyter Notebooks on PyCharm, in a browser

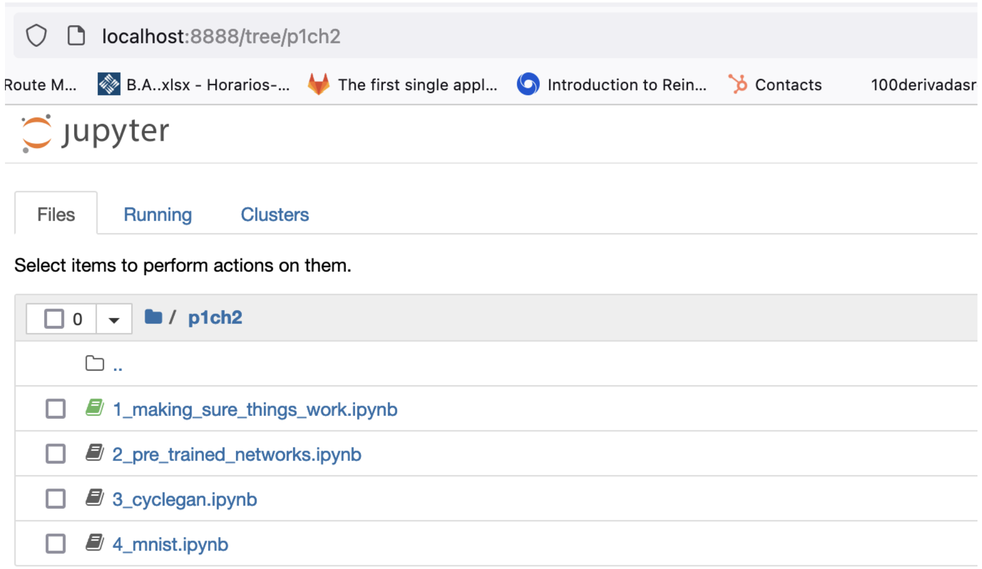


Log

```
/Users/rlucerga/.virtualenvs/class/bin/python -m jupyter notebook --no-browser "--notebook-dir=/Users/rlucerga/Dro
[I 21:16:16.249 NotebookApp] Serving notebooks from local directory: /Users/rlucerga/Dropbox (Personal)/Innitium/C
[I 21:16:16.249 NotebookApp] Jupyter Notebook 6.1.5 is running at:
[I 21:16:16.249 NotebookApp] http://localhost:8888/?token=79cac4c286d211faf18cffce6ffbfead14119a787ef235c7
```

Jupyter tab

Link with token



localhost:8888/tree/p1ch2

Select items to perform actions on them.

0 / p1ch2

..

1_making_sure_things_work.ipynb

2_pre_trained_networks.ipynb

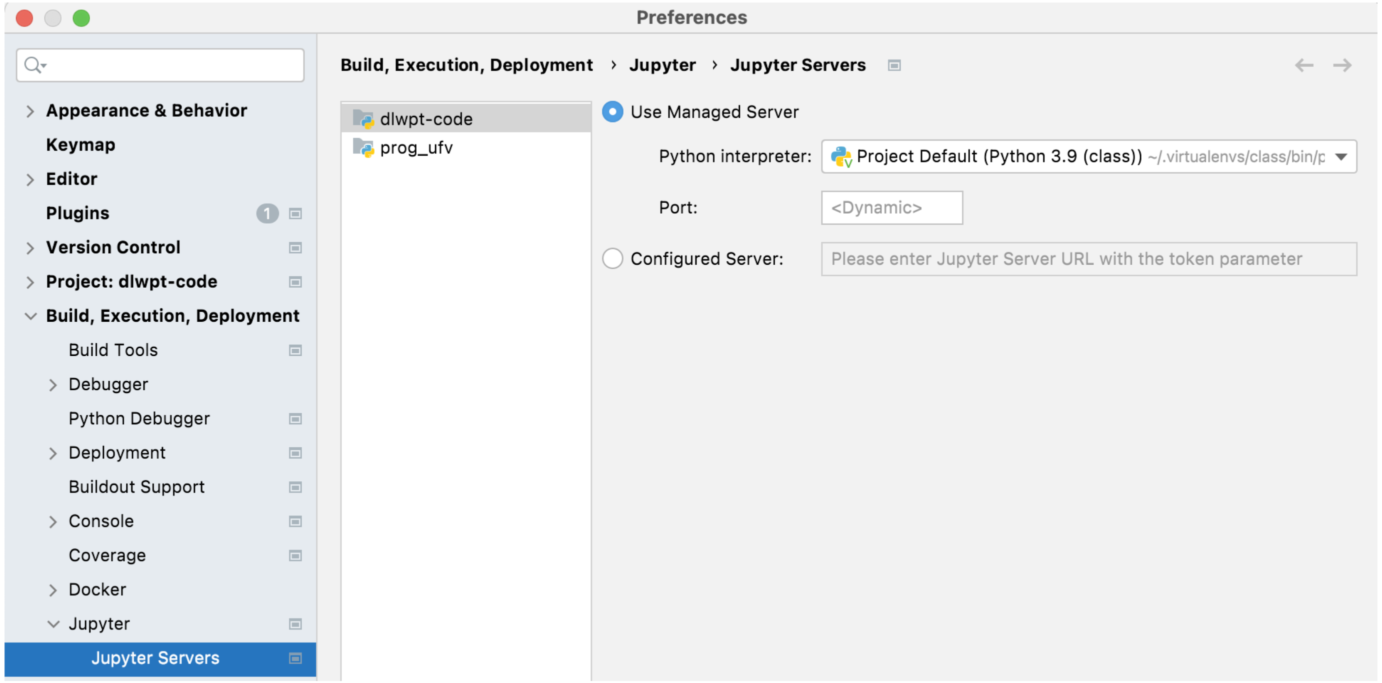
3_cyclegan.ipynb

4_mnist.ipynb

Jupyter Notebooks on PyCharm, interpreter

Universidad Francisco de Vitoria
UFV Madrid

Configuring interpreter used by the Jupyter server


Preferences

Build, Execution, Deployment › Jupyter › Jupyter Servers

- Appearance & Behavior
- Keymap
- Editor
- Plugins 1
- Version Control
- Project: dlwpt-code
- Build, Execution, Deployment
  - Build Tools
  - Debugger
  - Python Debugger
  - Deployment
  - Buildout Support
  - Console
  - Coverage
  - Docker
  - Jupyter
    - Jupyter Servers

dlwpt-code
prog_ufv

[✓ Use Managed Server]

Python interpreter: Project Default (Python 3.9 (class)) ~/.virtualenvs/class/bin/p

Port: <Dynamic>

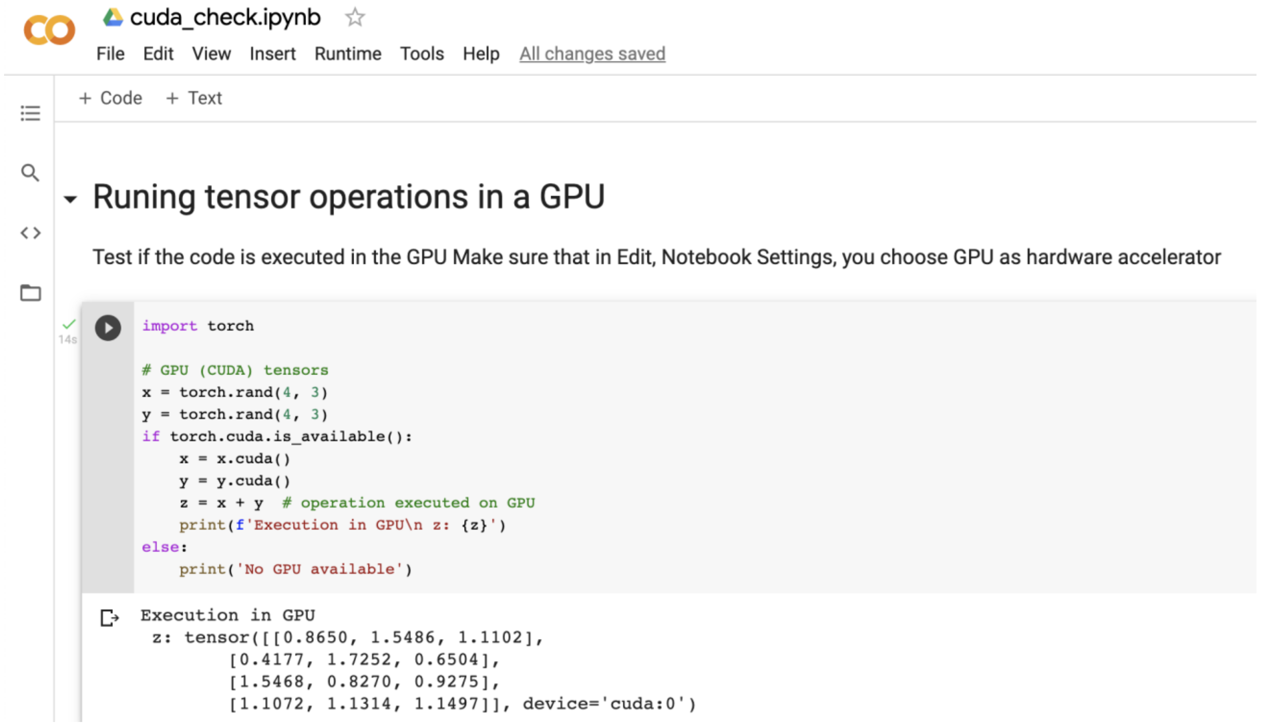
Configured Server: Please enter Jupyter Server URL with the token parameter

Google Colaboratory (Google Colab) access to GPUs

Google Colab (https://colab.research.google.com)

Cloud platform, GPU-enabled Jupyter Notebooks with PyTorch preinstalled, with a free quota

Note: Activate GPU or TPU in Edit → Notebook Settings.



Runing tensor operations in a GPU

Test if the code is executed in the GPU Make sure that in Edit, Notebook Settings, you choose GPU as hardware accelerator

```
import torch

# GPU (CUDA) tensors
x = torch.rand(4, 3)
y = torch.rand(4, 3)
if torch.cuda.is_available():
    x = x.cuda()
    y = y.cuda()
    z = x + y  # operation executed on GPU
    print(f'Execution in GPU\n z: {z}')
else:
    print('No GPU available')
```

```
Execution in GPU
 z: tensor([[0.8650, 1.5486, 1.1102],
        [0.4177, 1.7252, 0.6504],
        [1.5468, 0.8270, 0.9275],
        [1.1072, 1.1314, 1.1497]], device='cuda:0')
```

Using TPUs in Google Colab

Tutorial: https://colab.research.google.com/github/pytorch/xla/blob/master/contrib/colab/getting-started.ipynb

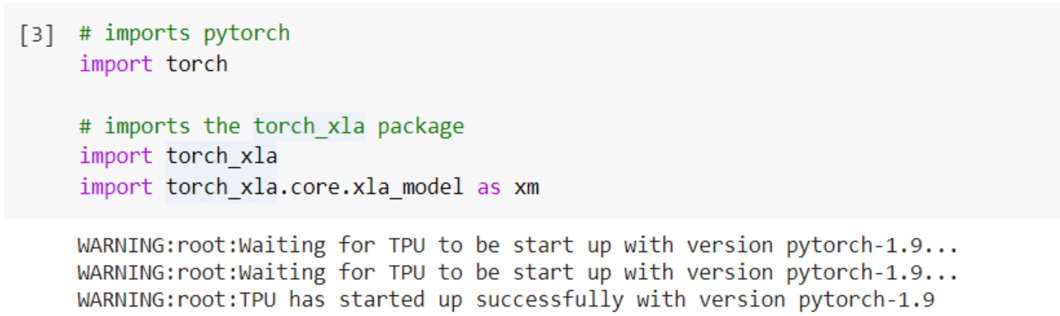
- Need to install PyTorch/XLA on Colab (1)
  - PyTorch/XLA: package that lets PyTorch connect to Cloud TPUs.
  - XLA is the name of the TPU compiler
- PyTorch uses Cloud TPUs just like it uses CPU or CUDA devices.
  - Each core of a Cloud TPU is treated as a different PyTorch device.

```
(1) !pip install cloud-tpu-client==0.10 https://storage.googleapis.com/tpu-pytorch/wheels/torch_xla-1.9-cp37-cp37m-linux_x86_64.whl
```

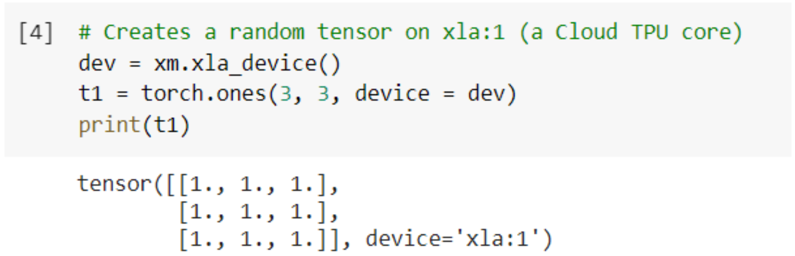
Using TPUs in Google Collab

This lets PyTorch create and manipulate tensors on TPUs.

```
# imports pytorch
import torch

# imports the torch_xla package
import torch_xla
import torch_xla.core.xla_model as xm

# Creates a random tensor on xla:1 (a Cloud TPU core)
dev = xm.xla_device()

a = torch.randn(2, 2, device = dev)
b = torch.randn(2, 2, device = dev)
print(a + b)
print(b * 2)
print(torch.matmul(a, b))
```



```
[3] # imports pytorch
    import torch

    # imports the torch_xla package
    import torch_xla
    import torch_xla.core.xla_model as xm

    WARNING:root:Waiting for TPU to be start up with version pytorch-1.9...
    WARNING:root:Waiting for TPU to be start up with version pytorch-1.9...
    WARNING:root:TPU has started up successfully with version pytorch-1.9
```



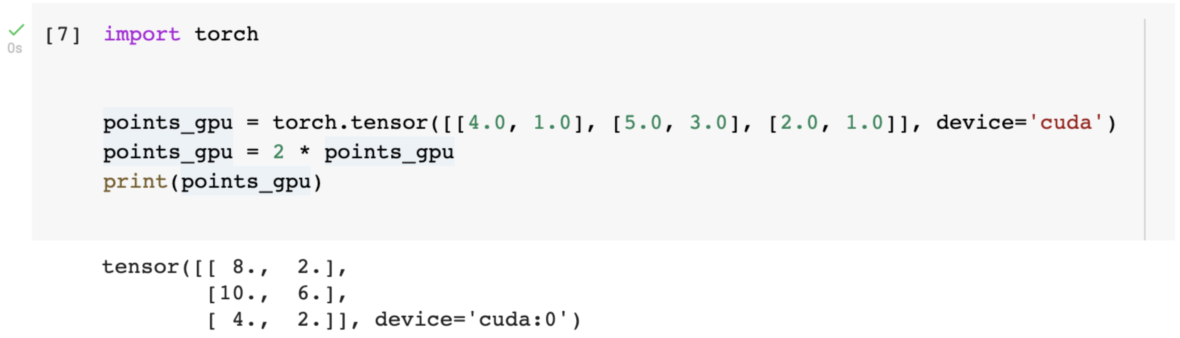
```
[4] # Creates a random tensor on xla:1 (a Cloud TPU core)
    dev = xm.xla_device()
    t1 = torch.ones(3, 3, device = dev)
    print(t1)

    tensor([[1., 1., 1.],
            [1., 1., 1.],
            [1., 1., 1.]], device='xla:1')
```

Move a tensor to a GPU

- Access Colab and create an account.
- https://colab.research.google.com
- If you have a Google Drive, give access to the drive.
- Create a new notebook and replicate the code in the slide above.
- Activate GPU or TPU acceleration: Edit, Notebook Settings, Hardware Accelerator
- Create two tensors directly in the GPU and compute their product
- Create a tensor in the CPU, move it to the GPU with the .to() method and compute a new product there.

GPU/TPU through Google Collab

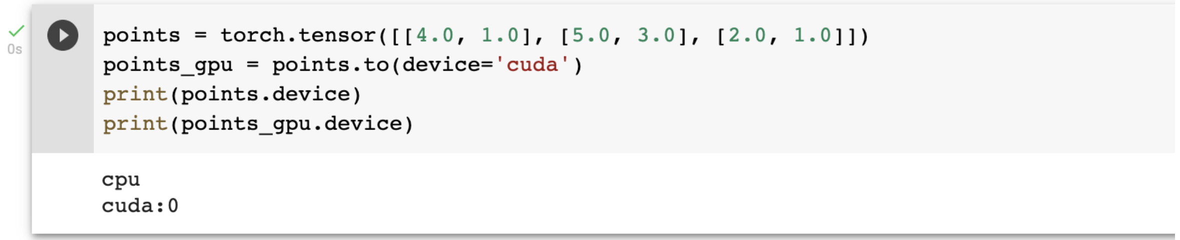
Example: create tensors is CPU/GPU



```
[7] import torch


    points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device='cuda')
    points_gpu = 2 * points_gpu
    print(points_gpu)
```

```
tensor([[ 8.,  2.],
        [10.,  6.],
        [ 4.,  2.]], device='cuda:0')
```

Another way, sending to GPU after creating in CPU



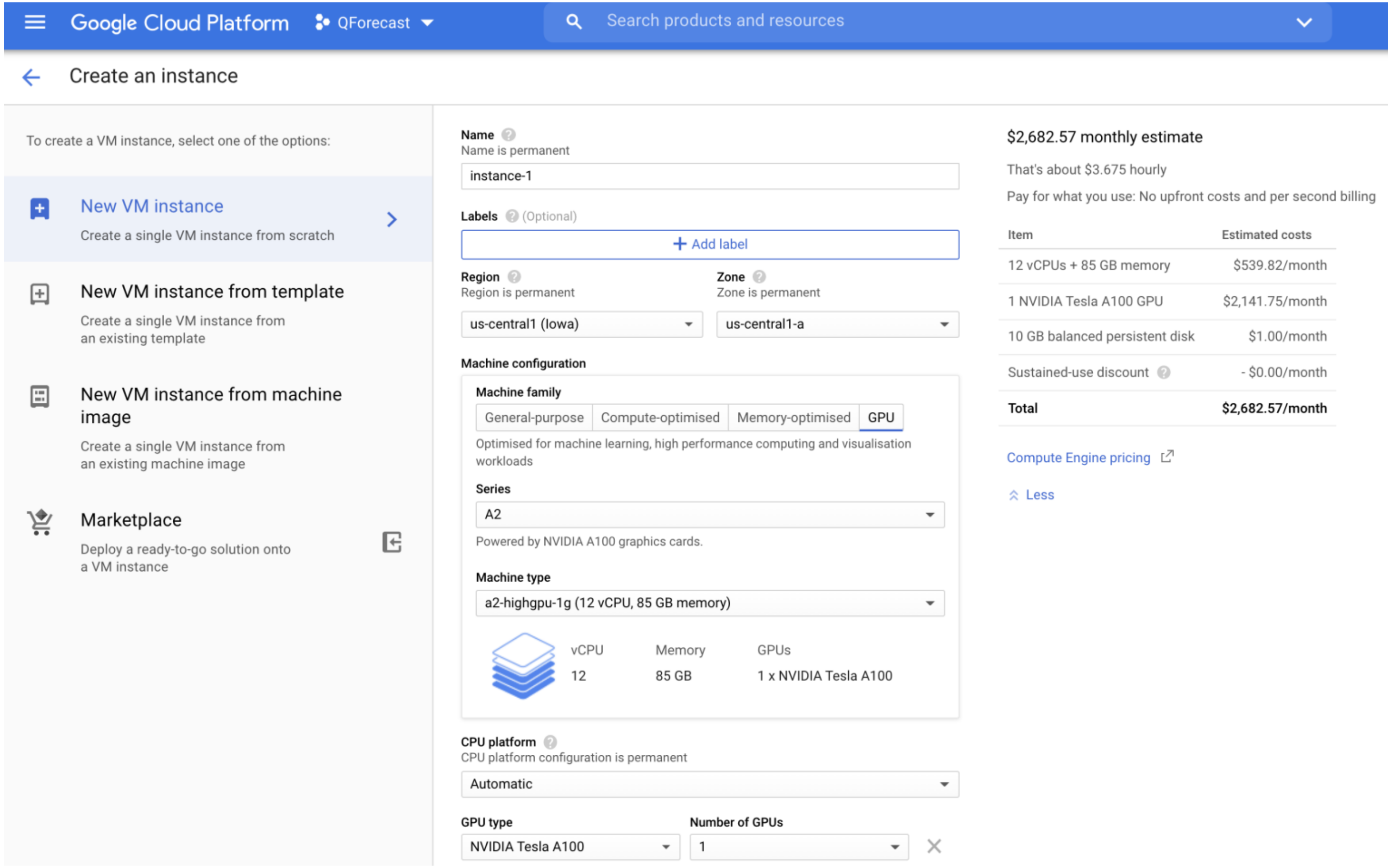
```
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points_gpu = points.to(device='cuda')
print(points.device)
print(points_gpu.device)
```

```
cpu
cuda:0
```

TENSORS

- *Understanding tensors*
- *Building tensors*
- *Tensor API*
- *Storage, offset and stride*
- *Processors: CPUs, GPUs, TPUs and CUDA*
- *Access to GPUs-TPUs*
  - Local GPUs
  - Google Colab
  - Cloud Computing

Cloud server



Google Cloud Platform | QForecast | Search products and resources

Create an instance

To create a VM instance, select one of the options:

New VM instance
Create a single VM instance from scratch

New VM instance from template
Create a single VM instance from an existing template

New VM instance from machine image
Create a single VM instance from an existing machine image

Marketplace
Deploy a ready-to-go solution onto a VM instance

Name
Name is permanent
instance-1

Labels (Optional)
+ Add label

Region
Region is permanent
us-central1 (Iowa)

Zone
Zone is permanent
us-central1-a

Machine configuration

Machine family
General-purpose | Compute-optimised | Memory-optimised | GPU

Optimised for machine learning, high performance computing and visualisation workloads

Series
A2
Powered by NVIDIA A100 graphics cards.

Machine type
a2-highgpu-1g (12 vCPU, 85 GB memory)

vCPU	Memory	GPUs
12	85 GB	1 x NVIDIA Tesla A100

CPU platform
CPU platform configuration is permanent
Automatic

GPU type
NVIDIA Tesla A100

Number of GPUs
1

$2,682.57 monthly estimate

That's about $3.675 hourly

Pay for what you use: No upfront costs and per second billing

Item	Estimated costs
12 vCPUs + 85 GB memory	$539.82/month
1 NVIDIA Tesla A100 GPU	$2,141.75/month
10 GB balanced persistent disk	$1.00/month
Sustained-use discount	- $0.00/month
Total	**$2,682.57/month**

Compute Engine pricing

Less

TENSORS

Summary

- Floating point numbers for input, internal representation and outputs of NN.
- Tensor objects as building blocks of computational graphs
- Tensor data: multidimensional array, with a comprehensive API (creation, indexing, math ops, serialization, parallelism)
- Broadcasting rules
- Storage: size, offset, stride
- Tensors in devices: Application-specific integrated circuits (GPU and TPU)