Università degli Studi di Pavia

Deep Learning

08-A Few Relevant Asides

Marco Piastra & Andrea Pedrini(*)
(thanks are due to Mirto Musci and Gianluca Gerard as well)

(*) Dipartimento di Matematica F. Casorati

This presentation can be downloaded at:
http://vision.unipv.it/DL

Hardware for Deep Learning

GPU vs. CPU

- ***The GPU resides on a separate board***
  - *Almost an independent computer*

Model is here

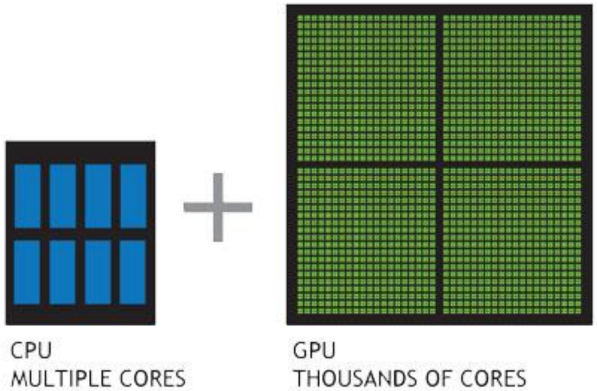
Data is here

[image http://cs231n.stanford.edu/slides/2021/lecture_6.pdf]

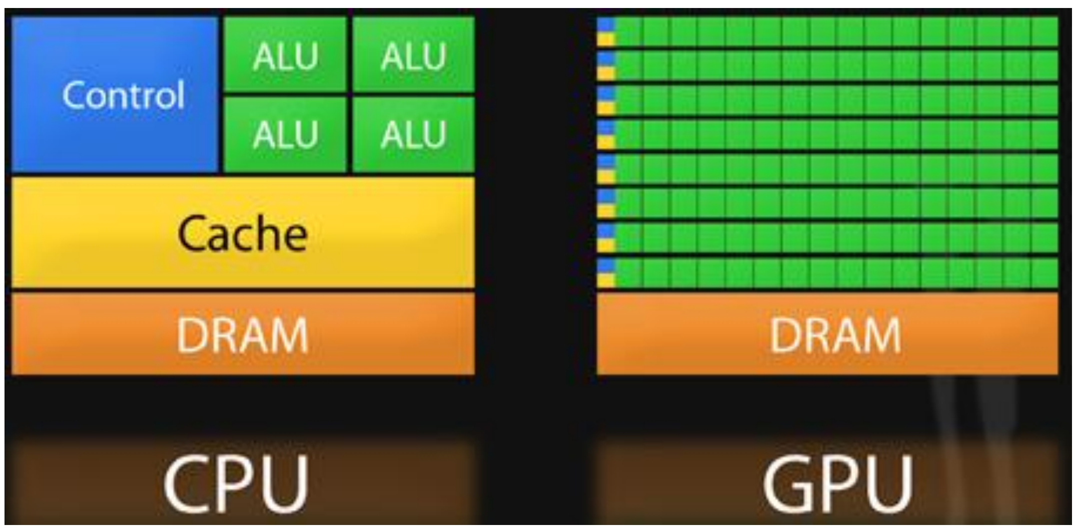
GPU vs. CPU

- ***Different hardware architectures***

  For different computing paradigms



[images from http://www.nvidia.com/docs/]



GPU vs. CPU

- ***Different hardware architectures***

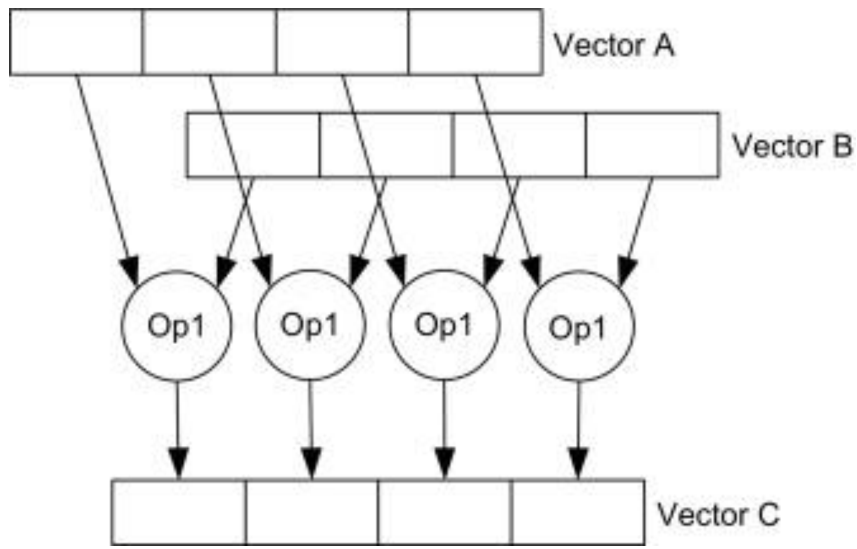
  For different computing paradigms

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	10	4.3 GHz	System RAM	$385	~640 **G**FLOPs FP32
GPU (NVIDIA RTX 3090)	10496	1.6 GHz	24 GB GDDR 6X	$1499	~35.6 **T**FLOPs FP32
GPU (Data Center) NVIDIA A100	6912 CUDA, 432 Tensor	1.5 GHz	40/80 GB HBM2	$3/hr (GCP)	~9.7 TFLOPs FP64 ~20 TFLOPs FP32 ~312 TFLOPs FP16
TPU Google Cloud TPUv3	2 Matrix Units (MXUs) per core, 4 cores	?	128 GB HBM	$8/hr (GCP)	~420 TFLOPs (non-standard FP)

[image http://cs231n.stanford.edu/slides/2021/lecture_6.pdf]

SIMT Parallelism

- ***Single Instruction, Multiple Data*** (SIMD)
  - Execution is parallel
  - All cores are executing <u>the same instruction</u>, in sync
  - Each core works on specific data



[images from https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data]

SIMT Parallelism

- ***Single Instruction, Multiple Threads*** (SIMT)

  Execution is parallel

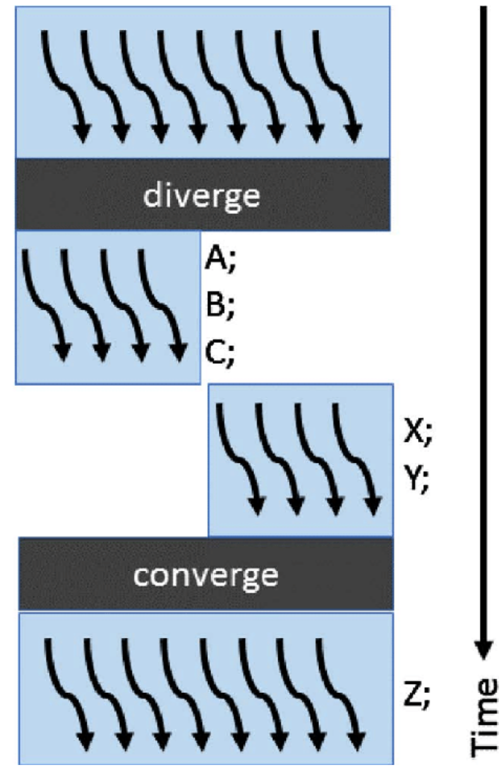
  All ***active*** cores are executing the same instruction, in sync

  Each core works on specific data

  The control system activates and deactivates cores on each *execution branch*

  Moral: any computation might be performed, but divergent ones will be sequentialized

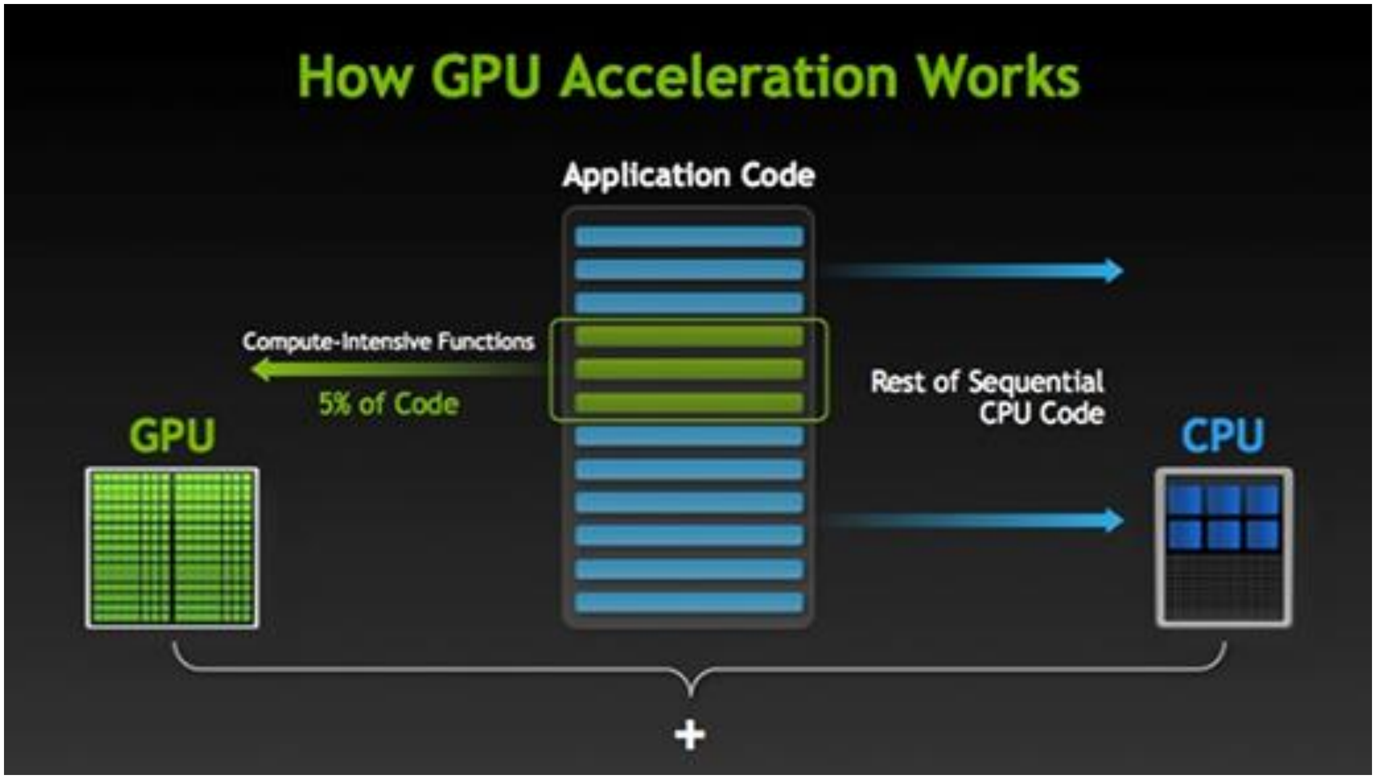
```
if (condition <= 0) {
        A;
        B;
        C;
} else {
        X;
        Y;
}
Z;
```



[images from https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data]

Selective parallelization

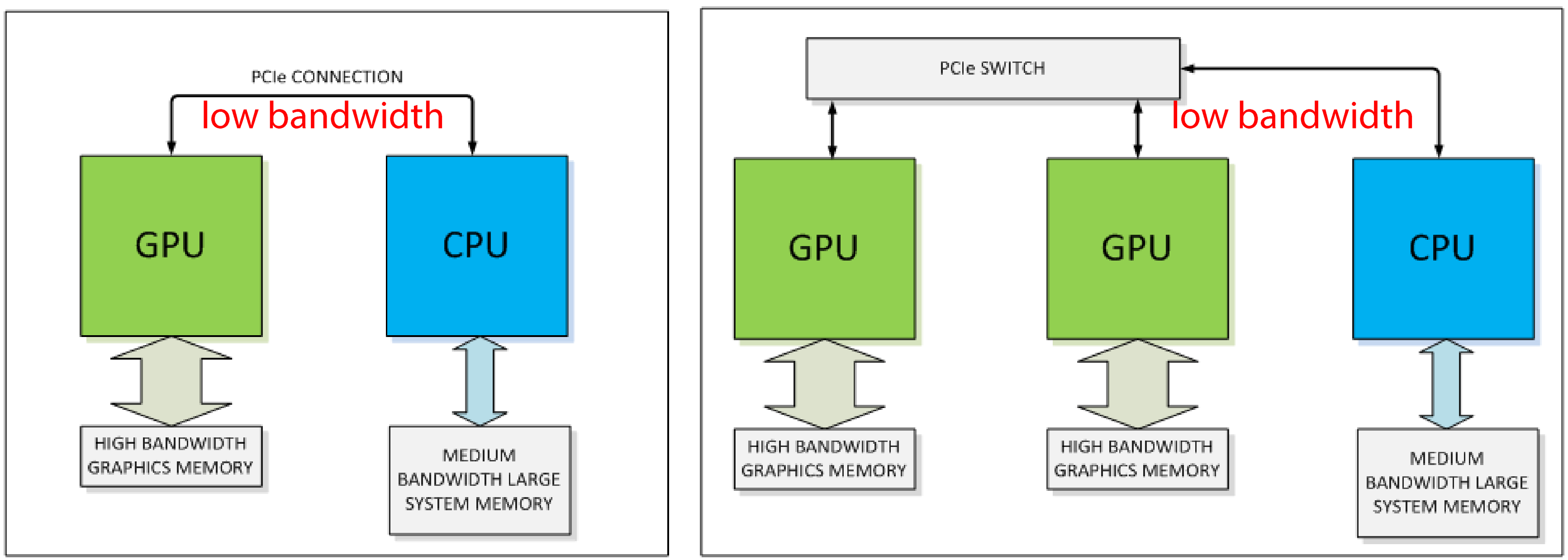
Not all parts of a program are worth executing in parallel...



[images from http://www.nvidia.com/docs/]

TensorFlow and GPUs

- TF computations are optimized to be run on **GPUs**
  - For the programmer, these implementation details are (mostly) **transparent**
  - TF can also run on the CPU only, but with lower performance.
- TF automatically manages **memory transfers** to/from GPUs
  - Memory transfers are very costly, due to low bandwidth PCIe



[NVIDIA.com]

Tensor transformations: slicing

Slicing

- A tensor is an **n-dimensional** array
  - You can even use the .numpy() method to return a numpy version of the tensor
- **To access a single cell** you need to specify **n indices**
  - **rank 0** (scalar): no indices are necessary (it is already a single number)
  - **rank 1** (vector): passing a single index allows you to access a number

```
my_scalar = my_vector[2]
```

  - **rank 2 or higher**: passing two or more numbers returns a scalar

```
my_scalar = my_matrix[1, 2]
```

- **A single number** returns **a subtensor**
  - The example below is for a matrix (a 2-D tensor)

```
my_row_vector = my_matrix[2]
my_column_vector = my_matrix[:, 3]
```

The **: notation** means "leave this dimension as is"

TensorFlow slicing and NumPy slicing

- The **[ ] notation** overloads `Tensor.getitem`

  This **operation** extracts the specified region from the tensor

  Very similar behavior w.r.t. numpy

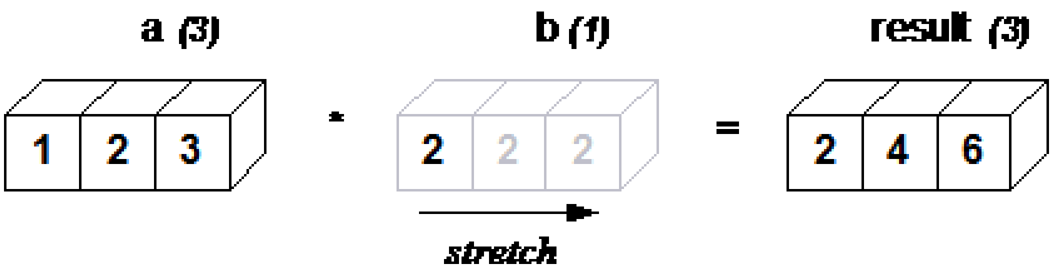
- **Interesting Examples**

```
foo = tf.constant([  [1,2,3],
                     [4,5,6],
                     [7,8,9]  ])
# skip every row and reverse every column
tf.print(foo[::2,::-1]) # => [[3,2,1], [9,8,7]]
# Insert another dimension
tf.print(foo[:, tf.newaxis, :]) # => [[[1,2,3]], [[4,5,6]], [[7,8,9]]]
# Ellipses (the following lines are equivalent)
tf.print(foo[tf.newaxis, ...]) # => [[[1,2,3], [4,5,6], [7,8,9]]]
tf.print(foo[tf.newaxis]) # => [[[1,2,3], [4,5,6], [7,8,9]]]
```

Tensor transformations: broadcasting

Broadcasting: an example with TensorFlow

```
# Create a three-element vector (1-D tensor).
a = tf.constant([1, 2, 3], dtype=tf.int32, name='a')
# Create a constant scalar with value 2.
b = tf.constant(2, dtype=tf.int32, name='b')

# Multiply the two tensors element-wise.
result = tf.multiply(a, b)
tf.print(result)
```



- Vector a is multiplied, **element-wise**, with scalar b
- Before multiplying, scalar b is **stretched** to get the same shape as vector a
- The final result is a vector with the **same shape** as vector a

The General Broadcasting Rules

- TensorFlow adopts the general broadcasting rules of NumPy

  When operating on two arrays, NumPy compares their shapes element-wise

  It starts with the **trailing** dimensions, and works its way forward

- Two dimensions are **compatible** when
  1. they are equal, or
  2. one of them is 1
- The size of the resulting array is the **maximum size** along each dimension of the input arrays

- When a tensor is broadcast, its entries are **conceptually copied**

  Broadcasting is a performance optimization, thus,
  for performance reasons, **no actual copying occurs**

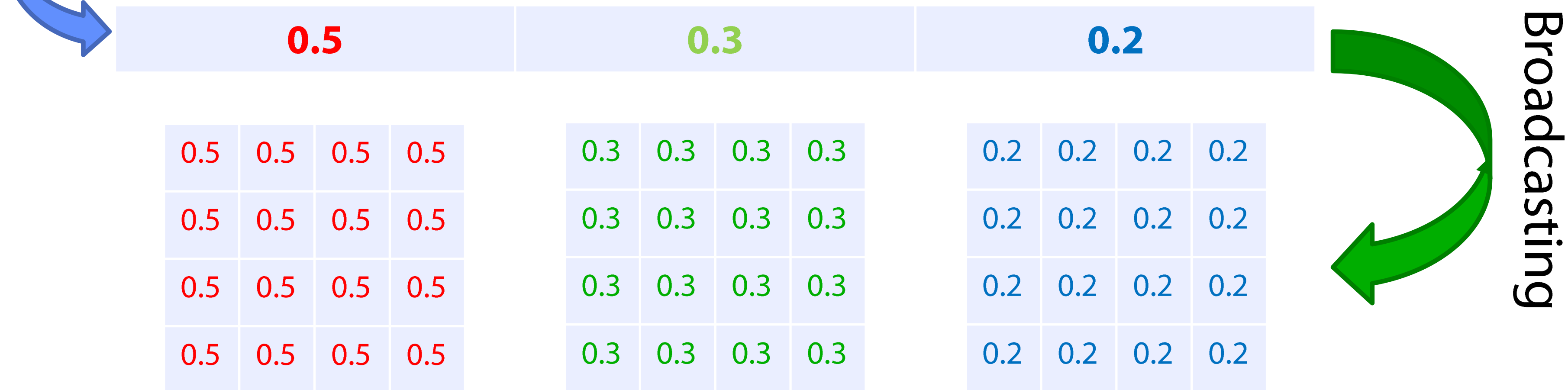
Applying the General Broadcasting Rule

```
A      (2d array):   5 x 4
B      (1d array):       1
Result (2d array):   5 x 4

A      (3d array):  15 x 3 x 1
B      (2d array):       3 x 5
Result (3d array):  15 x 3 x 5

A      (4d array):   8 x 1 x 6 x 5
B      (3d array):       7 x 1 x 5
Result (4d array):   8 x 7 x 6 x 5
```

Broadcasting: another example

- Each channel of an RGB image can be scaled by multiplying the image by a 1-D array (vector) with 3 values.

```
Image  (3d array): 4 x 4 x 3
Scale  (1d array):         3
Result (3d array): 4 x 4 x 3
```



Tensor transformations: reshaping

Reshaping: examples

- In the previous example, to create the matrix ***A*** we wrote

```
a = tf.transpose(tf.constant([[0.0,1.0,2.0,3.0]]))
```

We could have written instead:

```
a = tf.reshape(tensor = [0.0,1.0,2.0,3.0],
               shape  = [-1,1])
```

The second instruction reshapes the original 1-D Tensor with 4 values into a 2-D Tensor still with **the same** 4 values

We used the **special value -1** for shape so that we didn't have to specify how many values `tensor` has

- Another example: **flatten** a 2-D Tensor

```
a = tf.ones([4,3])                      # A 2-D (4,3) tensor
b = tf.reshape(tf.range(1.0,5.0),[-1,1]) # A 2-D (4,1) tensor
t = a*b (and NOT tf.matmul(a,b))        # A 2-D (4,3) tensor
t_1d = tf.reshape(t,[-1])               # A 1-D (12) tensor
tf.print(t_1d)
```

Tensors reshaping

- As we just saw, the function to reshape a tensor is the following

```
tf.reshape(tensor, shape, name=None)
```

- The operation returns a tensor with shape `shape`, filled with the values of the original `tensor`

  The number of elements implied by shape must be the same as the number of elements in tensor

  e.g. shape [4,3] must be reshaped in something with a total shape of 12

- If one component of `shape` is the special value -1, the size of that dimension is computed so that the total size remains the same

  A shape of [-1] flattens into 1-D; at most one component of `shape` can be -1

> Reshaping is often used **to flatten the output of the last convolutional layer** of a CNN so that it can be used as the input of the first dense layer

Tensor transformations: stacking

Stacking

- Tensors can be stacked together by using the function

```
tf.stack([tensor0, tensor1, …], axis=0, name='stack’)
```

It packs the list of tensors, along the axis dimension,
into a tensor with rank one higher than each tensor in the list

- Example:

```
x = tf.constant([1, 4])      # Shape (2,)
y = tf.constant([2, 5])      # Shape (2,)
z = tf.constant([3, 6])      # Shape (2,)
tf.stack([x, y, z], axis=0) # Shape (3,2): [[1,4],[2,5],[3,6]]
tf.stack([x, y, z], axis=1) # Shape (2,3): [[1,2,3],[4,5,6]]
```

- Given a list of *n* tensors of shapes [(a, b, c), …, (a, b, c)]:
  - if `axis == 0` then the output tensor will have the shape (*n*, a, b, c)
  - if `axis == 1` then the output tensor will have the shape (a, *n*, b, c)

Splitting

- A tensor can be split into multiple tensors with the function

  `tf.split()`

- Examples:

```
# 'value' is a tensor with shape [6, 30]
# split 'value' into 2 tensors along dimension 0
split0_0, split0_1 = tf.split(value, 2, axis=0)
tf.shape(split0_0)  # [3, 30]
# split 'value' into 3 tensors along dimension 1
split1_0, split1_1, split1_2 = tf.split(value, 3, axis=1)
tf.shape(split1_0)  # [6, 10]

# Split 'value' into 3 tensors with sizes [4, 15, 11]
# along dimension 1 (note that 4+15+11 = 30)
split0, split1, split2 = tf.split(value, [4, 15, 11], 1)
tf.shape(split0)  # [6, 4]
tf.shape(split1)  # [6, 15]
tf.shape(split2)  # [6, 11]
```