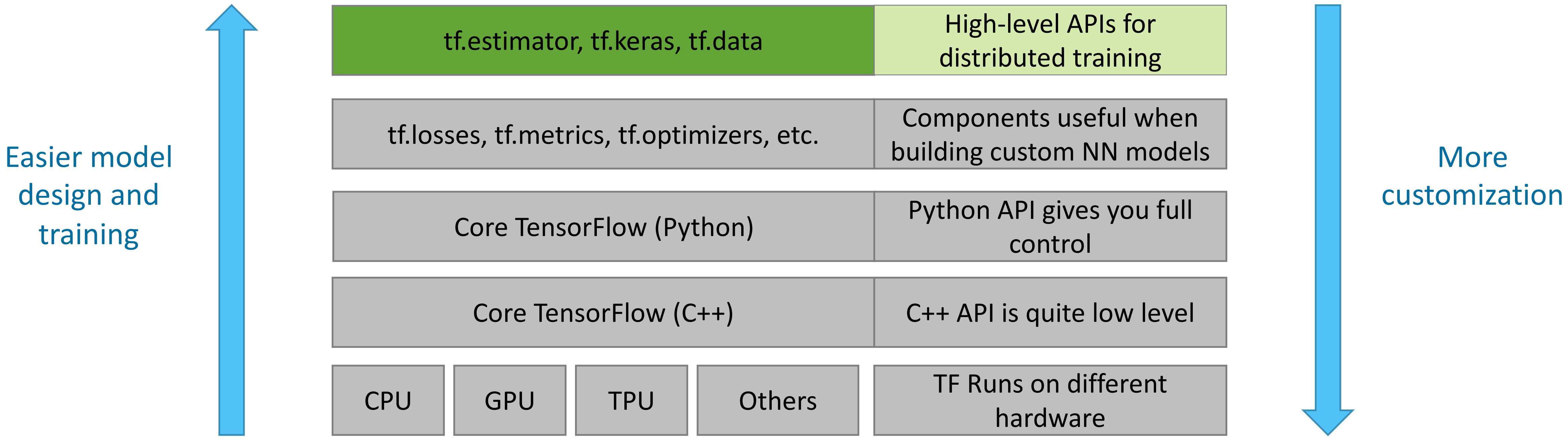
Tensorflow 2.0

The Dataset API

Tensorflow API Hierarchy

- Tensorflow exposes APIs at multiple abstraction levels



The Dataset API

- The `tf.data` API is used to build complex input pipelines
- It allows to read data from multiple sources and perform complex transformations
- Examples:
  - Aggregate data from files in a distributed file system
  - Apply random perturbations to each image (augmentation)
  - Create batches for training
  - Extract tokens from raw text
  - Apply look-up embeddings
  - Etc.

The Dataset API

- Main class: `tf.data.Dataset:` represents a sequence of elements, each consisting of one or more components.

Input	Label	Element 0
Input	Label	Element 1

- Example: for a supervised classification problem, each element could be a training example, composed of two tensors (input, label)

- A dataset can be created from:
  - One of different **sources** (in-memory data, files)
  - A **transformation** of one or more other Datasets

Create a Dataset (From Tensors)

- Two distinct methods:

- **from_tensors()**: creates a dataset containing a single element taking a set of TF tensors, or convertible types (list, np.array) as input

- **from_tensor_slices()**: creates a dataset whose elements are slices of the input tensors (over the first dimension)

Create a Dataset (From Tensors)

```
x = [1, 2, 3]
ds = tf.data.Dataset.from_tensors(x)
print(len(ds))
# Output: 1

for elem in ds:
    print(elem.numpy())
# Output:
# [1 2 3]
```

```
x = [1, 2, 3]
ds = tf.data.Dataset.from_tensor_slices(x)
print(len(ds))
# Output: 3

for elem in ds:
    print(elem.numpy())
# Output:
# 1
# 2
# 3
```

Create a Dataset (From Tensors)

- You can pass multiple tensors (e.g. input and labels) to both functions.
  - You can use a Python dictionary to give names to the components of each element.
  - In case of from_tensor_slices(), all components must have the same first dimension

```
dataset = tf.data.Dataset.from_tensor_slices({"a": [1, 2], "b": [3, 4]})

for elem in dataset:
      print([‘{}: {}'.format(_, elem[_].numpy()) for _ in elem])

# Output:
# ['a: 1', 'b: 3’]
# ['a: 2', 'b: 4']
```

Create a Dataset (from Tensors)

- Here's a most realistic example which consumes two numpy arrays, returned by `keras.datasets`

```
train, test = tf.keras.datasets.fashion_mnist.load_data()
images, labels = train
print(type(images))
# Output: np.ndarray

train_ds = tf.data.Dataset.from_tensor_slices((images, labels))
```

Create a Dataset (from Tensors)

- Using in-memory tensors is the simplest (and often the most efficient) way to create datasets when your training inputs entirely fit in memory.

Create a Dataset (from Text Files)

- The `tf.data` API supports several file formats to allow processing large datasets that do not fit in memory.

- The simplest type of file that can be process is a text file

```
dataset = tf.data.TextLineDataset(file_paths)
```

One or more text file names, each line becomes a dataset element (of type string)

Create a Dataset (from Text Files)

- If the file contains lines that you don't want to include in your dataset you can use the `skip()` and `filter()` transformations

```
# keep only lines not starting with a '0'
def my_filter(line):
      return tf.not_equal(tf.strings.substr(line, 0, 1, "0"))


dataset = dataset.skip(1).filter(my_filter)
```

Skip first line

Apply filter

Create a Dataset (from Text Files)

- CSV are very common among text files (especially for structured datasets)
- Using the TextLineDataset with CSVs is not really convenient (you have to write code to split each line into its values and convert them to the appropriate type)
- The easiest way to manage CSV files is through the Pandas package.
  - Pandas dataframes can be converted to Python dictionaries, which in turn can be digested by from_tensor_slices()

```
import pandas as pd

df = pd.read_csv(my_file, index_col=None) #read in Pandas
ds = tf.data.Dataset.from_tensor_slices(dict(df)) # convert to DS
```

Create a Dataset (from Text Files)

- Alternatively, `tf.data.experimental.make_csv_dataset()` can handle CSVs that do not fit in memory.

- Several advanced options (but we'll skip them here…)

Create a Dataset (from TFRecords)

- `TFRecord` is a simple record-oriented binary format that many TensorFlow applications use for training data.

- The API provides a direct way to read TFRecord sources.

```
dataset = tf.data.TFRecordDataset(filenames)
```

Dataset Transformations

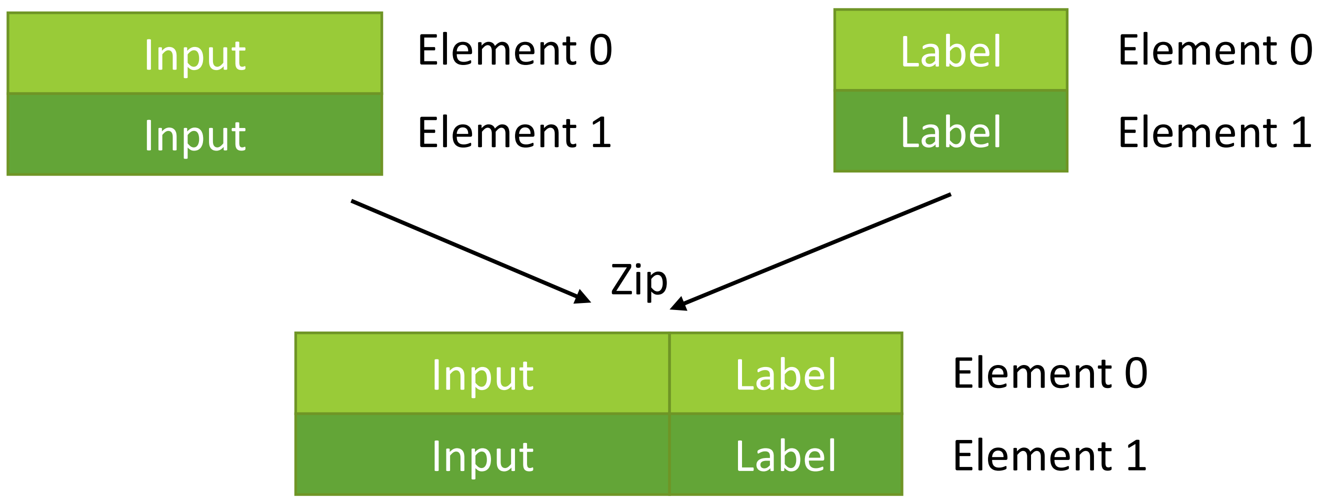
- Enough for sources (there are many more)

- Let’s see how we can use the `tf.data` API to easily transform data.

Dataset Zipping

- The new dataset will contain a component for each element of the original ones.

```
dataset = tf.data.Dataset.zip((inputs, labels))
```



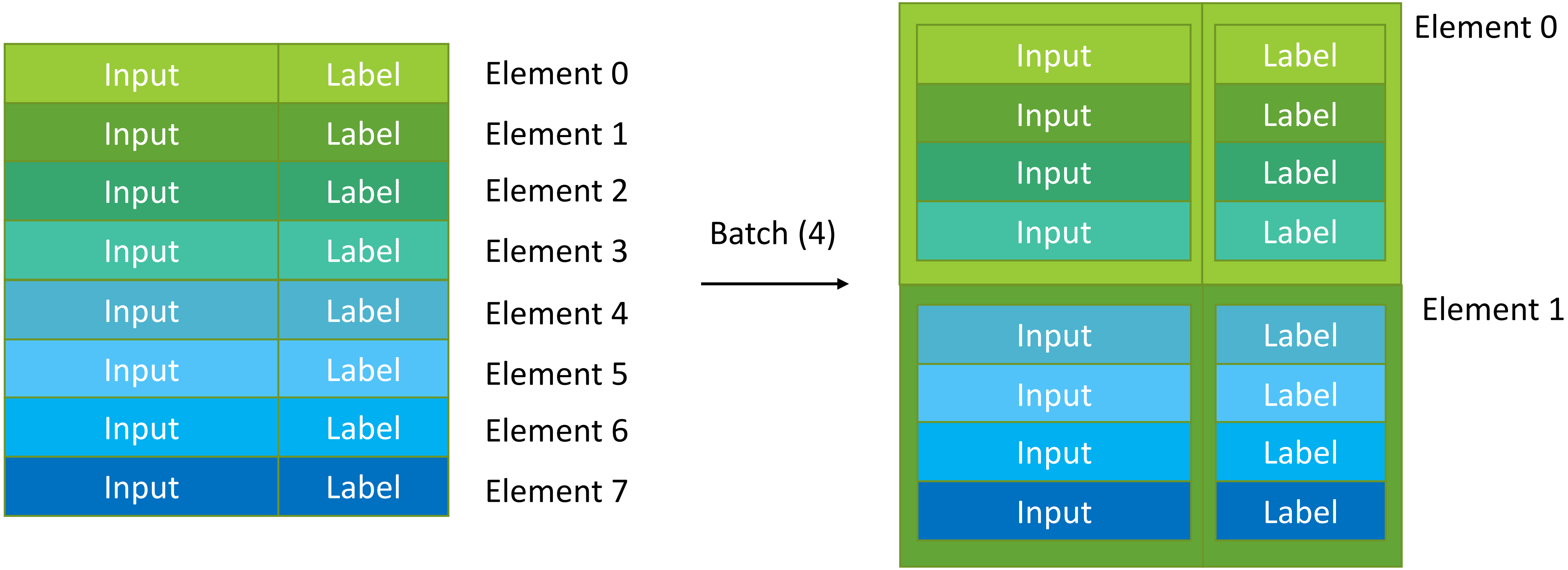
Dataset Batching

- Each element of the new dataset will be composed of `batch_size` <u>stacked</u> elements of the original one
- So, elements will have the same dimensions of the original Dataset with an additional outer dimension equal to batch_size

```
batched_dataset = dataset.batch(
        batch_size,
        drop_remainder=False
)
```

Set it to True to drop the latest batch, which might include less than batch_size elements elements.
Avoid that the outer dimension becomes “None”

Dataset Batching



Dataset Batching

- Generate **padded** batches (for variable length inputs):

```
dataset = dataset.padded_batch(
    batch_size,
    padded_shapes=None,
    padding_values=None,
    drop_remainder=True
)
```

A tuple specifies the shape of each element after padding. None lets the function use the maximum value of each dimension **in that batch**.

Default is 0.

Dataset Repetition

- Used with custom training loops to train for multiple epochs

```
train_dataset = dataset.batch(32).repeat(5)
```

Repeat with no argument repeats infinite times.

Dataset Shuffling

- Maintains a fixed-size buffer and chooses the next element uniformly at random from that buffer.
- Larger buffers → "More random" shuffling → Higher memory requirements

```
# select randomly among 100 elements
dataset = dataset.shuffle(buffer_size=100)
```

More Advanced Preprocessing

- One of the most flexible ways to preprocess data stored in a `tf.data.Dataset` is to use the `map()` transformation.
- This transformation takes a user-defined fuction as input and applies it to each element of the dataset

Create a Dataset from a List of FIlenames

- Create a dataset whose elements are file names

```
list_ds = tf.data.Dataset.list_files(str(root_path/'*/*’))
```

- Write a function to read from file & preprocess a single image

```
# read and preprocess an image
def parse_image(filename):
  image = tf.io.read_file(filename)
  image = tf.image.decode_jpeg(image)
  image = tf.image.convert_image_dtype(image, tf.float32)
  image = tf.image.resize(image, [128, 128])
  return image
```

Create a Dataset from a List of FIlenames

- Apply the function to all elements of the filenames Dataset

```
image_ds = list_ds.map(parse_image)
```

Reduce and Filter

- Reduce and filter are two other very useful data transformation functions.
- Of course, there can't be a `map()` without a `reduce()`

```
ds = tf.data.Dataset.from_tensor_slices(
      tf.constant(range(10), dtype=np.float32))

def my_sum(state, input):
      return state + input

sum = ds.reduce(0.0, my_sum)
```

Initial state   Function

Reduce and Filter

- Filter returns a dataset containing only the lines that match the predicate function

```
def is_odd(x):
    return x % 2 != 0

ds_odd = ds.filter(is_odd)
```

Optimizing Input Pipeline Performance

- Input pipelines can often be the bottleneck during training (remember our example of profiling in TensorBoard)
- The `tf.data` API offers several facilities to speed-up the reading and preprocessing of inputs.

Prefecthing Data

- Prefetching is a technique in which data reading/preprocessing and model execution are <u>overlapped</u> during training.
- It is a sort of pipelining, in which while the training process is running on batch N, the input process is already fetching batch N+1
- Calling the prefetch() transformation on a dataset creates an <u>additional thread</u> and an <u>internal buffer</u> to implement this
- The number of elements (batches) to prefetch can be set manually or automatically at runtime (suggested)

```
ds = ds.prefetch(tf.data.experimental.AUTOTUNE)
```

Applying Map in Parallel

- One of the advantages of the `map()` funciton is that it is naturally parallelizable. You simply need to add a parameter (in case this is your bottleneck).

```
ds = ds.map(
    my_function,
    num_parallel_calls=tf.data.experimental.AUTOTUNE
)
```

Caching Data

- The cache() transformation can be used to cache (partially) preprocessed data in memory during the first training epoch, to avoid re-loading them multiple times
  - What is cached is the output of all transformations that are applied **before** cache()
  - What is applied after cache() is re-done everytime
  - Of course, the output of cache() must fit in memory

```
ds = ds.batch(32)
ds = ds.map(my_expensive_function)
ds = ds.cache() # cache after map to avoid repeating the expensive function
ds = ds.shuffle() # shuffle after cache to have a different shuffle at every iteration
```

Other Transformation Functions

- Of course, there are many more transformations (and optimizations) that we didn't look into.

- The documentation is your friend

Dataset Notebook

- **Notebook:** 02_TF_Data.ipynb

Additional Modules for Input Processing

- TensorFlow offers a set of additional utility modules to handle different types of input data.
- The most relevant for the labs of this course (and for the IoT world) are:
  - tf.io
  - tf.image
  - tf.audio
  - tf.signal
- Here we briefly overview some of the functions included in each module, without getting into the details
  - You'll see some of these in action during the labs

tf.io

- Utility functions to read and write different types of formats to/from disk and to-from string tensors to other types of tensors. Examples:
  - `read_file()` → Reads the content of a file into a string tensor (also `write_file()`)
  - `decode_jpeg()` → Takes a string tensor read with `read_file()` and returns a 2D or 3D tensor of type uint8 representing the pixels of the image (also `encode_jpeg()`).
  - `is_jpeg()` → Convenience function to check if a string tensor encodes a JPEG image.
  - `encode_base64() / decode_base64()` → Convert between web-safe base64 string tensors and binary string tensors.

tf.image

- Contains various functions for image pre-processing:
  - Image resizing: `resize()`, `resize_with_pad()`, `resize_with_crop_or_pad()`
  - Color space conversion: `rgb_to_grayscale()`, `grayscale_to_rgb()`, etc.
  - Adjustments: `adjust_brightness()`, `adjust_contrast()`, etc.
  - Rotation/flipping: `flip_left_right()`, `flip_up_down()`, `rot90()`, etc.

tf.audio

- Contains two functions for reading/writing wav files to/from tensors:

  - `decode_wav(...)`: Decode a 16-bit PCM WAV file to a float tensor.

  - `encode_wav(...)`: Encode audio data using the WAV file format.

tf.signal

- Contains various functions for generic signal processing:

  - `fft()` / `fft2d()` / `fft3d()`: 1/2/3D Fast Fourier Transform
  - `ifft()` / `ifft2d()` / `ifft3d()` : 1/2/3D Inverse Fast Fourier Transform
  - `dct()` / `idct()` : Direct and Inverse Discrete Cosine Transform
  - Etc.