Chapter 3:
Tensor Flow-1: Basic Utility

March 26, 2023

Pada bab ini, fokus kita adalah pada `TensorFlow` yang merupakan salah satu library populer saat ini. Kita akan mengimplementasikan neural networks (NNs) secara lebih efisien menggunakan `TensorFlow` dibandingkan dengan `NumPy`, dan melihat bagaimana `TensorFlow` membawa keuntungan pada kinerja training.

Topik-topik yang akan dibahas pada bab ini diantaranya adalah sebagai berikut:

- Bagaimana `TensorFlow` akan memperbaiki kinerja training
- Bekerja dengan Dataset API (`tf.data`) pada `TensorFlow` untuk membangun *pipelines* dan model training yang efisien
- Bekerja dengan `TensorFlow` untuk mengoptimasi kode machine learning
- Menggunakan *high level API* dari `TensorFlow` untuk membangun multilayer NN
- Memilih fungsi aktivasi untuk NNs artifisial
- Memperkenalkan `Keras` (`tf.keras`) sebagai *wrapper* untuk `TensorFlow` yang dapat digunakan untuk mengimplementasikan arsitektur *deep learning* dengan baik

1 `TensorFlow` dan Kinerja Training

`TensorFlow` dapat mempercepat pekerjaan machine learning secara signifikan. Untuk memahami cara kerjanya, kita akan memulai dengan mendiskusikan tantangan yang sering terjadi saat mengeksekusi perhitungan yang sangat kompleks pada perangkat keras.

1.1 Tantangan Kinerja

Kinerja prosessor komputer saat ini terus meningkat tiap tahun, sehingga dapat digunakan untuk melakukan training pada sistem pembelajar (*learning systems*) yang lebih kompleks. Meskipun komputer desktop termurah yang tersedia sekarang datang dengan kekuatan unit-unit pemrosesan dengan *multiple cores*.

Fungsi-fungsi Library machine learning `ScikitLearn` (kita pelajari di Mata Kuliah Pembelajaran Mesin) melakukan komputasi yang dapat disebar pada unit-unit *multi processing*. Tetapi, secara default pada `Python`, eksekusi terbatas hanya pada satu *core* karena ***global interpreter lock*** (GIL). Dengan demikian, meskipun kita dapat memanfaatkan library *multiprocessing* dari `Python` untuk mendistribusikan komputasi ke bebarapa *multi cores*, kita harus berhadapan dengan kenyataan bahwa desktop-desktop paling canggih jarang dilengkapi dengan *core* lebih dari 8 atau 16.

Pada bab sebelumnya, *multilayer perceptron* (MLP) yang sangat sederhana telah diimplementasikan dengan hanya satu *hidden layer* yang terdiri dari 100 unit. Kita harus mengoptimasi sekitar 80.000 parameter pembobot ($[784 \times 100 + 100] + [100 \times 10] + 10 = 79.510$) untuk *learn* model klasifikasi image yang sangat sederhana. Image-image pada MNIST cenderung ukurannya kecil-kecil (28×28). Jika image-image yang digunakan mempunyai kepadatan pixel yang lebih tinggi, maka akan terjadi ledakan jumlah parameter menjadi sangat besar. Kondisi tersebut tidak memungkinkan lagi dilakukan oleh unit pemrosesan tunggal.

Salah satu solusi yang memungkinkan adalah menggunakan *Graphical Processing Unit* (GPUs) yang dapat dianggap juga sebagai tenaga penggerak(*work horses*). GPU bisa dilihat sebagai kluster komputer kecil di dalam sebuah mesin dengan kemampuan komputasi yang tinggi. Keuntungan lain, GPU lebih murah dibandingkan dengan *Central Processing Units* (CPU), seperti terlihat pada Gambar 3.1

Specifications	Intel® Core™ i9-9960X X-series Processor	NVIDIA GeForce® RTX™ 2080 Ti
Base Clock Frequency	3.1 GHz	1.35 GHz
Cores	16 (32 threads)	4352
Memory Bandwidth	79.47 GB/s	616 GB/s
Floating-Point Calculations	1290 GFLOPS	13400 GFLOPS
Cost	~ $1700.00	~ $1100.00

Gambar 3.1: Perbedaan CPU dan GPU

Dengan harga GPU lebih murah, kira-kira 65% dari harga CPU, GPU mempunyai 272 kali lebih banyak jumlah core-nya dan mampu melakukan 10 kali lebih cepat dalam kalkulasi *floating point* tiap detik. Tantangannya adalah menuliskan kode untuk target GPU tidak sesederhana mengeksekusi kode `Python` dalam interpreter. Terdapat banyak *package* spesial, seperti `CUDA` dan `OpenCL` untuk kita gunakan pada GPU. Tetapi, menulis kode pada `CUDA` atau `OpenCL` mungkin bukan *environment* yang paling nyaman dalam mengimplementasikan dan *running* algoritma-algoritma Machine Learning. Kabar baiknya adalah `TensorFlow` dibuat dengan tujuan tersebut.

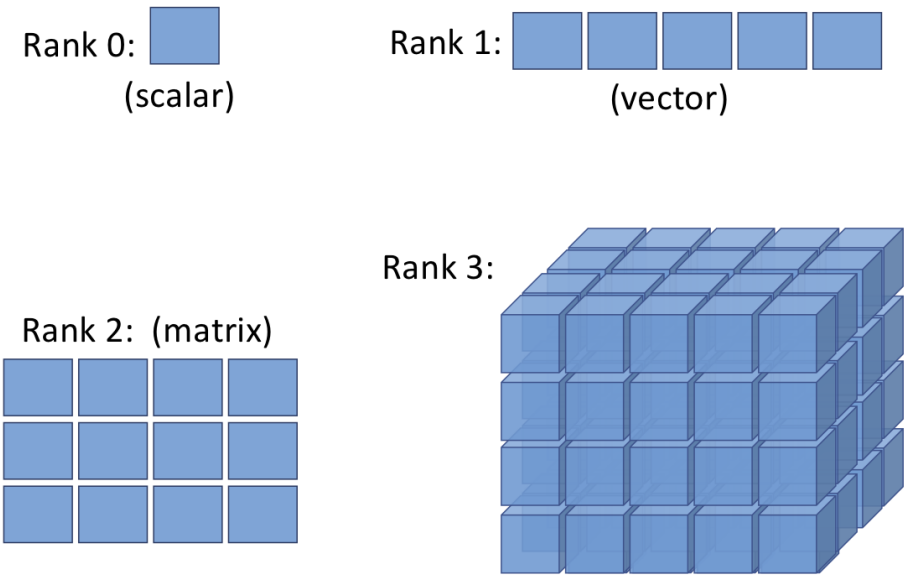
1.2 Apa itu `TensorFlow`?

`TensorFlow` merupakan antarmuka (*interface*) pemrograman yang *scalable* dan *multiplatform* untuk mengimplementasikan dan *running* algoritma-algoritma machinge learning, termasuk *wrappers* untuk deep learning. `TensorFlow` dibangun dan dipimpin oleh para peneliti dan insinyur dari Tim `Google` Brain, meskipun banyak kontribusi juga dari komunitas *open source*. `TensorFlow` pada awalnya dibangun untuk penggunaan internal di `Google`, tetapi kemudian di-*release* di November 2015 dengan lisensi *permissive open source*. Banyak peneliti dan praktisi dari akademia dan industri mengadaptasikan `TensorFlow` untuk membangung solusi-solusi deep learning.

Untuk memperbaiki kinerja model machine learning, `TensorFlow` dapat dieksekusi pada CPUs dan GPUs. Tetapi, kemampuan terbaiknya dapat diperoleh jika digunakan pada GPUs. `TensorFlow` secara resmi mendukung GPU berbasis `CUDA`, sedangkan dukungnan untuk devais-devais berbasis `OpenCL` masih dalam tarap eksperimental. Tetapi `OpenCL` juga akan disupport secara resmi dalam waktu dekat. Saat ini `TensorFlow` mendukung antarmuka-antarmuka *frontend‘ untuk banyak bahasa pemrograman.

Untuk kita pengguna `Python`, API `Python` dari `TensorFlow` saat ini merupakan API paling komplit sehingga menarik perhatian praktisi-praktisi machine learning dan deep learning. Lebih jauh, `TensorFlow` mempunyai API resmi juga pada `C++`. Beberapa tool baru di `TensorFlow` telah di-*release* diantaranya adalah `TensorFlow.js` dan `TensorFlow Lite` dengan fokus pada *running* dan *deploying* model-model machine learning pada Web Browser dan aplikasi mobile, serta devais-devais Internet of Things (IoT). API untuk bahasa pemrograman lain, seperti `Java`, `Haskell`, `Node.js` dan `Go`, masih belum stabil. Tetapi komunitas *open source* dan developer-developer `TensorFlow` secara kontinyu melakukan perbaikan-perbaikan.

`TensorFlow` dibangun sekitar graph komputasi yang terdiri dari satu set node-node. Setiap node merepresentasikan sebuah operasi yang boleh jadi mempunyai nol atau lebih input atau output. Sebuah tensor tercipta sebagai handel simbolik yang mengacu pada input dan output dari operasi-operasi ini. Secara matematis, tensor-tensor dapat dipandang sebagai generalisasi dari skalar, vektor, matriks dan selanjutnya. Secara konkrit, skalar dapat didefinisikan sebagai tensor dengan rank-0, vektor sebagai rank-1 tensor, matriks sebagai rank-2 tensor, dan matriks-matriks yang ditumpuk pada dimensi ketiga dapat didefinisikan sebagai rank-3 tensor. Tetapi, dengan catatan pada `TensorFlow`, harga-harga di simpan pada array-array `Numpy`, dan tensor-tensor menyediakan referensi pada array-array tersebut. Gambar 3.2 menunjukan konsep tensor secara visual.



Gambar 3.2: Representasi tensor-tensor rank-0, rank-1, rank-2 dan rank-3

2 Langkah-Langkah Awal untuk `TensorFlow`

Pada bagian ini akan dipelajari langkah-langkah awal dalam menggunakan API *low level* dari `TensorFlow`, diantaranya bagaimana membuat tensor pada `TensorFlow` dan cara-cara berbeda untuk memanipulasi tensor, seperti merubah ukuran, tipe data dan lain-lain. Untuk instalasi `TensorFlow` dapat dilihat pada link berikut: Link Install `TensorFlow`.

2.1 Membuat Tensor pada TensorFlow

- Sebagai langkah pertama kita bisa membuat sebuah tensor dari list atau array NumPy menggunakan fungsi `tf.convert_to` sebagai berikut

[79]:
```
import tensorflow as tf
print('TensorFlow version:', tf.__version__)
import numpy as np

np.set_printoptions(precision=3)
```

```
TensorFlow version: 2.9.1
```

[80]:
```
a = np.array([1, 2, 3], dtype=np.int32)
b = [4, 5, 6]

t_a = tf.convert_to_tensor(a)
t_b = tf.convert_to_tensor(b)

print(t_a)
print(t_b)
```

```
tf.Tensor([1 2 3], shape=(3,), dtype=int32)
tf.Tensor([4 5 6], shape=(3,), dtype=int32)
```

Hasilnya berupa tensor-tensor `t_a` dan `t_b` dengan *properties* `shape=(3,0)` dan `dtype=int32`. Serupa dengan NumPy, kita bisa melihat *properties*

[81]:
```
t_ones=tf.ones((2,3))
t_ones.shape
```

[81]:
```
TensorShape([2, 3])
```

- Untuk mendapatkan akses terhadap harga-harga kemana tensor menunjuk, maka gunakan metode `.numpy()` pada sebuah tensor

[82]:
```
t_a.numpy()
```

[82]:
```
array([1, 2, 3], dtype=int32)
```

- Membuat tensor dengan harga-harga konstan dapat dilakukan dengan

[83]:
```
const_tensor=tf.constant([1.2,5,np.pi],dtype=tf.float32)
print(const_tensor)
```

```
tf.Tensor([1.2   5.    3.142], shape=(3,), dtype=float32)
```

[84]:
```
const_tensor.numpy()
```

[84]:
```
array([1.2  , 5.   , 3.142], dtype=float32)
```

2.2 Memanipulasi tipe data dan ukuran tensor

Pada bagian ini akan dipelajari bagaimana memanipulasi tipe data dan ukuran (*shape*) dari tensor menggunakan beberapa fungsi `TensorFlow` diantaranya `cast`, `reshape`, `transpose` dan `squeeze`.

- Fungsi `tf.cast()` bisa digunakan untuk merubah tipe data dari tensor ke dalam tipe yang diinginkan.

[85]:
```
t_a_new = tf.cast(t_a,tf.int64)
print(t_a_new.dtype)
```

```
<dtype: 'int64'>
```

Beberapa operasi membutuhkan tensor-tensor input mempunyai jumlah dimensi tertentu (*rank*) dengan jumlah elemen-elemen tertentu (*shape*). Sehingga kadang kita perlu merubah ukuran sebuah tensor, menambahkan dimensi atau menghapus dimensi.

- Transpose sebuah tensor

[86]:
```
t = tf.random.uniform(shape=(3, 5))

t_tr = tf.transpose(t)
print(t.shape, ' --> ', t_tr.shape)
```

```
(3, 5)  -->  (5, 3)
```

- Reshape sebuah tensor (mis. dari vektor 1D ke 2D)

[87]:
```
t = tf.zeros((30,))

t_reshape = tf.reshape(t, shape=(5, 6))

print(t_reshape.shape)
```

```
(5, 6)
```

- Mengurangi dimensi yang tidak diperlukan (dimensi dengan ukuran 1 akan dihapus pada contoh ini)

[88]:
```
t = tf.zeros((1, 2, 1, 4, 1))

t_sqz = tf.squeeze(t, axis=(2, 4))

print(t.shape, ' --> ', t_sqz.shape)
```

```
(1, 2, 1, 4, 1)  -->  (1, 2, 4)
```

2.3 Operasi matematika pada tensor

Menerapkan operasi matematika pada tensor terutama aljabar linier sangat diperlukan untuk membangun model-model machine learning. Pada bagian ini akan dibahas penggunaan operasi aljabar

linier yang populer seperti perkalian per elemen (*element-wise product*), perkalian matriks, dan menghitung norm sebuah tensor.

- Menginisiasi dua tensor random, satu dengan distribusi uniform pada range $[-1, 1)$ dan satu lagi dengan distribusi normal

[89]:
```
tf.random.set_seed(1)
t1 = tf.random.uniform(shape=(5, 2), minval=-1.0, maxval=1.0)
t2 = tf.random.normal(shape=(5, 2), mean=0.0, stddev=1.0)
```

Perhatikan bahwa `t2` dan `t2` mempunyai ukuran (*shape*) sama

- Berikut perkalian antar elemen kedua matriks tersebut

[90]:
```
t3 = tf.multiply(t1, t2).numpy()
print(t3)
```

```
[[-0.27  -0.874]
 [-0.017 -0.175]
 [-0.296 -0.139]
 [-0.727  0.135]
 [-0.401  0.004]]
```

- Untuk menghitung mean sepanjang sumbu tertentu, maka gunakan `tf.math.reduce_mean()`. Misalkan menghitung mean untuk tiap kolom (sepanjang sumbu baris axis=0) pada `t1`

[91]:
```
t4 = tf.math.reduce_mean(t1, axis=0) # untuk mean sepanjang kolom gunakan axis=1
print(t4)
```

```
tf.Tensor([0.09  0.207], shape=(2,), dtype=float32)
```

- Perkalian matriks antara `t1` dan `t2` yaitu $t_1 \times t_2^T$, dimana $(\cdot)^T$ menyatakan transpose, akan dihasilkan matriks 5×5

[92]:
```
t5 = tf.linalg.matmul(t1, t2, transpose_b=True)
print(t5.numpy())
```

```
[[-1.144  1.115 -0.87  -0.321  0.856]
 [ 0.248 -0.191  0.25  -0.064 -0.331]
 [-0.478  0.407 -0.436  0.022  0.527]
 [ 0.525 -0.234  0.741 -0.593 -1.194]
 [-0.099  0.26   0.125 -0.462 -0.396]]
```

Sedangkan untuk perkalian $t_1^T \times t_2$ akan menghasilkan ukuran matriks 2×2

[93]:
```
t6 = tf.linalg.matmul(t1, t2, transpose_a=True)
print(t6.numpy())
```

```
[[-1.711  0.302]
 [ 0.371 -1.049]]
```

- Kemudian, fungsi `tf.norm()` dapat digunakan untuk menghitung norm L^p sebuah tensor. Misalkan menghitung norm L^2 dari `t1`

[94]:
```
norm_t1 = tf.norm(t1, ord=2, axis=1).numpy()
print(norm_t1)
```

```
[1.046 0.293 0.504 0.96  0.383]
```

Untuk memastikan hasil di atas betul, bandingkan dengan hasil dari fungsi `NumPy`berikut

[95]:
```
np.sqrt(np.sum(np.square(t1), axis=1))
```

[95]:
```
array([1.046, 0.293, 0.504, 0.96 , 0.383], dtype=float32)
```

2.4 *Split*, *stack* dan *concatenate* tensor-tensor

Pada bagian ini kita akan melihat operasi `TensorFlow` untuk memisahkan (*split*) sebuah tensor menjadi beberapa tensor, atau kebalikannya menumpuk (*stack*) dan menyambung urutkan (*concatenate*) beberapa tensor menjadi satu tensor.

Diasumsikan kita mempunyai satu tensor dan ingin dibagi menjadi 2 atau lebih tensor-tensor. Untuk melakukannya `TensorFlow` menyediakan metode `tf.split()` yang membagi sebuah tensor input ke dalam tensor-tensor dengan ukuran sama. Kita dapat menentukan jumlah tensor hasil pemisahan dengan menggunakan argumen `num_or_size_splits` untuk *split* tensor sepanjang dimensi tertentu yang dinyatakan dengan argumen `axis`. Dalam kasus ini, ukuran total input tensor sepanjang dimensi yang telah ditentukan melalui argumen `axis` harus dapat dibagi dengan jumlah *split* yang diinginkan. Alternatif lain, kita dapat menyediakan ukuran yang diinginkan melalui sebuah `list`.

- Pada contoh berikut, sebuah tensor dengan ukuran (*size*) 6, dibagi ke dalam sebuah list dengan 3 tensor dengan ukuran masing-masing 2.

[96]:
```
tf.random.set_seed(1)
t = tf.random.uniform((6,))
print(t.numpy())
```

```
[0.165 0.901 0.631 0.435 0.292 0.643]
```

[97]:
```
t_splits = tf.split(t,  num_or_size_splits =3)
[item.numpy() for item in t_splits]
```

[97]:
```
[array([0.165, 0.901], dtype=float32),
 array([0.631, 0.435], dtype=float32),
 array([0.292, 0.643], dtype=float32)]
```

Jika ukuran 6×2 dan dibagi menjadi 2 tensor sepanjang baris maka

[98]:
```
t = tf.random.uniform((6,2))
print(t.numpy())
t_splits = tf.split(t,  num_or_size_splits =2, axis=0)
[item.numpy() for item in t_splits]
```

```
[[0.51  0.444]
 [0.409 0.992]
 [0.689 0.346]
 [0.436 0.601]
 [0.457 0.753]
 [0.188 0.549]]
```

[98]:
```
[array([[0.51 , 0.444],
        [0.409, 0.992],
        [0.689, 0.346]], dtype=float32),
 array([[0.436, 0.601],
        [0.457, 0.753],
        [0.188, 0.549]], dtype=float32)]
```

- Kita juga dapat menyediakan ukuran-ukuran split yang berbeda. Daripada mendefinisikan jumlah split, kita dapat menspesifikasikan ukuran-ukuran tensor output secara langsung. Berikut kita membagi sebuah tensor dengan ukuran 5 menjadi dua tensor dengan ukuran 3 dan 2.

[99]:
```
tf.random.set_seed(1)
t = tf.random.uniform((5,4))
print(t.numpy())
t_splits = tf.split(t, num_or_size_splits=[2,2], axis=1) #untuk split kolom
# t_splits= tf.split(t, num_or_size_splits=[3,2], axis=0) #untuk split baris
[item.numpy() for item in t_splits]
```

```
[[0.165 0.901 0.631 0.435]
 [0.292 0.643 0.976 0.435]
 [0.66  0.605 0.637 0.614]
 [0.889 0.628 0.532 0.026]
 [0.441 0.253 0.886 0.887]]
```

[99]:
```
[array([[0.165, 0.901],
        [0.292, 0.643],
        [0.66 , 0.605],
        [0.889, 0.628],
        [0.441, 0.253]], dtype=float32),
 array([[0.631, 0.435],
        [0.976, 0.435],
        [0.637, 0.614],
        [0.532, 0.026],
        [0.886, 0.887]], dtype=float32)]
```

- Kadang-kadang, kita bekerja dengan banyak tensor dan perlu untuk menyambung urut atau menumpuk tensor-tensor tersebut ke dalam satu tensor. Maka kita dapat gunakan metode `tf.stack()` dan `tf.concat()`. Sebagai contoh, jika terdapat tensor 1D **A** berisi semua 1 dengan ukuran 3 dan tensor 1D **B** berisi semua 0 dengan ukuran 2, maka bisa disambung urutkan menjadi tensor 1D **C** dengan ukuran 5:

[100]:
```
A = tf.ones((3,))
B = tf.zeros((3,))
C = tf.concat([A, B], axis=0)
print(C.numpy())
```

```
[1. 1. 1. 0. 0. 0.]
```

- Jika kita mempunyai tensor **A** dan **B** dengan ukuran 3, maka kita bisa menumpukan (*stacking*) untuk membentuk tensor 2D **S***

[101]:
```
A = tf.ones((3,))
B = tf.zeros((3,))
S = tf.stack([A, B], axis=1)
print(S.numpy())
```

```
[[1. 0.]
 [1. 0.]
 [1. 0.]]
```

API dari `TensorFlow` mempunyai banyak operasi yang dapat digunakan untuk mebuat sebuah model, memproses data, dan lain-lain. Untuk full operasi dan fungsi yang bisa digunakan, maka bisa dilihat di dokumentasi `TensorFlow` pada link berikut: Dokumentasi `TensorFlow`

3 Membangun *input pipelines* menggunakan `tf.data`

Ketika training sebuah model *deep neural networks* (DNN), kita men-training model secara beratahap misalkan menggunakan algoritma *stochastic gradient descent*. Sebagai informasi `Keras` API merupakan *wrapper* pada `TensorFlow` untuk membangun model-model NN. `Keras` API menyediakan sebuah metode `.fit()` untuk men-training model-model. Pada kasus dimana dataset training kecil dan dapat di-*load* sebagai sebuah tensor ke dalam memori, model-model `TensorFlow` (yang dibangun dengan `Keras` API) dapat menggunakan langsung tensor ini melalui metode `.fit()` untuk melakukan training. Tetapi pada umumnya, ketika dataset sangat besar untuk sebuah memori komputer, maka diperlukan untuk me-*load* data dari divais penyimpanan utama (mis. *hard drive* atau *solid-state drive*) dalam bentuk *chunks*, atau per-batch. Selain itu, kita boleh jadi perlu mengkonstruksi *pipeline* pemrosesan data untuk mengimplementasikan transformasi-transformasi tertentu dan langkah-langkah *preprocessing* pada data, seperti pemusatan mean (*mean centering*), penskalaan (*scaling*), atau menambahkan noise untuk mengaugmentasi prosedur training dan menghindari *overfitting*.

Menerapkan fungsi-fungsi preprocessing secara manual setiap saat akan membuat tidak praktis. Untungnya, `TensorFlow` menyediakan *class* spesial untuk melakukan konstruksi *pipeline* dari preprocessing secara efisien dan mudah. Pada bagian ini, akan membahas *overview* dari metode-metode berbeda untuk mengkonstruksi `TensorFlow Dataset`, termasuk transformasi dataset dan langkah-langkah umum pada preprocessing.

3.1 Membuat dataset `TensorFlow` dari tensor-tensor yang ada

- Jika data telah ada dalam bentuk objek-objek tensor, list-list `Python`, atau array-array `NumPy`, kita dapat membuat dataset secara mudah menggunakan fungsi

`tf.data.Dataset.from_tensor_slices()`. Fungsi ini mengembalikan sebuah objek dengan class `Dataset`, dimana elemen-elemen individunya dapat diiterasi pada input dataset. Contoh sederhana dapat dilihat pada kode berikut, yang membuat dataset dari sebuah list yang berisi harga-harga terentu.

[102]:
```
a = [1.2, 3.4, 7.5, 4.1, 5.0, 1.0]
ds = tf.data.Dataset.from_tensor_slices(a)
print(ds)
```

```
<TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.float32,
name=None)>
```

- Kita dapat melakukan iterasi melalui dataset pada setiap entri-nya seperti kode berikut

[103]:
```
for item in ds:
    print(item)
```

```
tf.Tensor(1.2, shape=(), dtype=float32)
tf.Tensor(3.4, shape=(), dtype=float32)
tf.Tensor(7.5, shape=(), dtype=float32)
tf.Tensor(4.1, shape=(), dtype=float32)
tf.Tensor(5.0, shape=(), dtype=float32)
tf.Tensor(1.0, shape=(), dtype=float32)
```

- Jika ingin dibuat batch-batch dari dataset ini, dengan ukuran 3, kita bisa melakukannya dengan kode berikut

[104]:
```
ds_batch = ds.batch(3)
for i, elem in enumerate(ds_batch, 1):
    print('batch {}:'.format(i), elem.numpy())
```

```
batch 1: [1.2 3.4 7.5]
batch 2: [4.1 5.  1. ]
```

Kode di atas membuat dua batch dari dataset tersebut, dimana tiga elemen pertama masuk ke batch #1 dan elemen-elemen sisa masuk ke batch #2. Metode `.batch()`mempunyai opsi argumen `drop_remainder` (dengan harga default `False`) yang berguna ketika jumlah elemen-elemen pada tensor tidak bisa dibagi dengan ukuran batch yang diinginkan.

3.2 Mengkombinasikan dua tensor kedalam sebuah *joint dataset*

Seringkali kita mempunyai data dalam dua atau lebih tensor. Misalkan, satu tensor untuk *feature* dan satu tensor untuk label. Pada kasus tersebut, kita perlu untuk membuat dataset yang mengkombinasikan tensor-tensor tersebut yang membuat kita dapat mengambil elemen-elemen dari tensor ini dalam bentuk tuple-tuple.

- Jika diasumsikan kita mempunyai dua tensor `t_x` dan `t_y`. Tensor `t_x` menyimpan harga-harga *feature* dengan ukuran 3 sedangkan `t_y` berisi label-label class. Pertama kita buat 2 tensor sebagai contoh.

[105]:
```
tf.random.set_seed(1)
t_x = tf.random.uniform([4, 3], dtype=tf.float32)
t_y = tf.range(4)
```

Sekarang, kita ingin membuat *joint dataset* dari kedua tensor tersebut. Perhatikan bahwa terdapat korespondensi satu-ke-satu antara elemen-elemen dari dua tensor tersebut.

[106]:
```
ds_x = tf.data.Dataset.from_tensor_slices(t_x)
ds_y = tf.data.Dataset.from_tensor_slices(t_y)
ds_joint = tf.data.Dataset.zip((ds_x, ds_y))
for example in ds_joint:
    print('  x: ', example[0].numpy(),
          '  y: ', example[1].numpy())
```

```
  x:  [0.165 0.901 0.631]   y:  0
  x:  [0.435 0.292 0.643]   y:  1
  x:  [0.976 0.435 0.66 ]   y:  2
  x:  [0.605 0.637 0.614]   y:  3
```

Alternatif lain adalah kita membuat *joint dataset* menggunakan `tf.data.Dataset.from_tensor_slices()` sebagai berikut, yang akan menghasilkan output yang sama dengan metode sebelumnya.

[107]:
```
## metode alternatif:
ds_joint = tf.data.Dataset.from_tensor_slices((t_x, t_y))
for example in ds_joint:
    print('  x: ', example[0].numpy(), '  y: ', example[1].numpy())
```

```
  x:  [0.165 0.901 0.631]   y:  0
  x:  [0.435 0.292 0.643]   y:  1
  x:  [0.976 0.435 0.66 ]   y:  2
  x:  [0.605 0.637 0.614]   y:  3
```

- Selanjutnya kita akan melihat bagaimana mengimplementasikan transformasi pada setiap elemen individu dari sebuah dataset. Kita akan gunakan dataset `ds_joint` dari langkah sebelumnya, dan mengaplikasikan *feature-scaling* untuk menskalakan harga-harga pada range $[-1, 1)$, dimana sebelumnya `t_x` berada pada range $[0, 1)$ berdasarkan distribusi uniform.

[108]:
```
ds_trans = ds_joint.map(lambda x, y: (x*2-1.0, y))

for example in ds_trans:
    print('  x: ', example[0].numpy(),'  y: ', example[1].numpy())
```

```
  x:  [-0.67   0.803  0.262]   y:  0
  x:  [-0.131 -0.416  0.285]   y:  1
  x:  [ 0.952 -0.13   0.32 ]   y:  2
  x:  [0.21  0.273 0.229]   y:  3
```

Menerapkan transformasi semacam ini dapat digunakan untuk fungsi-fungsi yang terdefinisi oleh user. Misalkan kita mempunyai dataset yang dibuat dari list file-file image dalam sebuah disk, kita

dapat mendefinisikan sebuah fungsi untuk *load* image-image dari nama file-nama file dan mengimplementasikan fungsi tersebut dengan memanggil metode `.map()`. Bagian-bagian selanjutnya akan menunjukan bagaimana transformasi jamak diimplementasikan pada dataset.

3.3 *Shuffle, batch* dan *repeat*

Pada bab sebelumnya disampaikan bahwa untuk melakukan training model NN menggunakan *stochastic gradient descent*, maka penting untuk memberikan data training menggunakan batch-batch yang dikocok secara acak (*randomly shuffled batches*). Sekarang, selain membuat batch yang telah kita pelajari, akan kita lihat bagaimana mengocok (*shuffle*) dan re-iterasi pada dataset. Kita akan tetap bekerja menggunakan dataset `ds_joint` dari bagian sebelumnya.

- Pertama, membuat versi yang telah di-*shuffle* (dikocok) dari dataset `ds_joint`

[109]:
```
tf.random.set_seed(1)
ds = ds_joint.shuffle(buffer_size=len(t_x))
for example in ds:
    print('  x: ', example[0].numpy(),'  y: ', example[1].numpy())
```

```
  x:  [0.976 0.435 0.66 ]   y:  2
  x:  [0.435 0.292 0.643]   y:  1
  x:  [0.165 0.901 0.631]   y:  0
  x:  [0.605 0.637 0.614]   y:  3
```

Baris telah di-*shuffle* tanpa mempengaruhi korespondensi satu-ke-satu antara entri-entri pada x dan y. Metode `.shuffle()` memerlukan argumen `buffer_size` yang menyatakan berapa elemen-elemen pada dataset yang digrupkan secara bersamaan sebelum di-*shuffle*. Untuk menjamin randomisasi *shuffle* secara komplit pada tiap epoch, kita bisa memilih ukuran buffer sama dengan jumlah *training examples*, seperti pada kode di atas kita menggunakan `buffer_size=len(t_x)`.

- Membagi dataset pada beberapa batch untuk model training dapat dilakukan dengan metode `.batch()`. Kita akan membuat batch berdasarkan dataset `ds_joint` dan melihat seperti apakah sebuah batch itu.

[110]:
```
ds = ds_joint.batch(batch_size=3, drop_remainder=False)
batch_x, batch_y = next(iter(ds))
print('Batch-x: \n', batch_x.numpy())
```

```
Batch-x:
 [[0.165 0.901 0.631]
 [0.435 0.292 0.643]
 [0.976 0.435 0.66 ]]
```

[111]:
```
print('Batch-y:   ', batch_y.numpy())
```

```
Batch-y:    [0 1 2]
```

- Ketika mentraining sebuah model untuk beberapa epoch, diperlukan melakukan *shuffle* dan iterasi melalui dataset dengan jumlah epoch yang diinginkan. Kita ulangi dataset yang telah ter-*batch* sebanyak dua kali.

[112]:
```
ds = ds_joint.batch(3).repeat(count=2)
for i,(batch_x, batch_y) in enumerate(ds):
    print(i, batch_x.shape, batch_y.numpy())
```

```
0 (3, 3) [0 1 2]
1 (1, 3) [3]
2 (3, 3) [0 1 2]
3 (1, 3) [3]
```

Kode di atas menghasilkan dua *copy* dari setiap batch. Jika kita merubah urutan dua operasi di atas, pertama kali `batch` kemudian `repeat` maka hasilnya akan berbeda.

[113]:
```
ds = ds_joint.repeat(count=2).batch(3)
for i,(batch_x, batch_y) in enumerate(ds):
    print(i, batch_x.shape, batch_y.numpy())
```

```
0 (3, 3) [0 1 2]
1 (3, 3) [3 0 1]
2 (2, 3) [2 3]
```

Perhatikan hasil di atas yang menunjukan perbedaan antara batch-batch. Ketika `batch` pertama dan `repeat`, maka akan diperoleh empat batch. Sebaliknya, ketika `repeat` dilakukan terlebih dahulu, akan dihasilkan tiga batch.

- Untuk memahami lebih jauh bagaimana tiga operasi (`shuffle`, `batch` dan `repeat`) berprilaku, akan dibuat eksperimen dengan urutan yang berbeda-beda. Eksperimen berikut mempunyai urutan (1)`shuffle`, (2)`batch`, (3) `repeat`.

[114]:
```
tf.random.set_seed(1)
## Order 1: shuffle -> batch -> repeat
ds = ds_joint.shuffle(4).batch(2).repeat(3)

for i,(batch_x, batch_y) in enumerate(ds):
    print(i, batch_x.shape, batch_y.numpy())
```

```
0 (2, 3) [2 1]
1 (2, 3) [0 3]
2 (2, 3) [0 3]
3 (2, 3) [1 2]
4 (2, 3) [3 0]
5 (2, 3) [1 2]
```

[115]:
```
tf.random.set_seed(1)
## Order 1: shuffle -> batch -> repeat
ds = ds_joint.shuffle(4).batch(2).repeat(20)

for i,(batch_x, batch_y) in enumerate(ds):
    print(i, batch_x.shape, batch_y.numpy())
```

```
0 (2, 3) [2 1]
1 (2, 3) [0 3]
2 (2, 3) [0 3]
3 (2, 3) [1 2]
4 (2, 3) [3 0]
5 (2, 3) [1 2]
6 (2, 3) [1 3]
7 (2, 3) [2 0]
8 (2, 3) [1 2]
9 (2, 3) [3 0]
10 (2, 3) [3 0]
11 (2, 3) [2 1]
12 (2, 3) [3 0]
13 (2, 3) [1 2]
14 (2, 3) [3 0]
15 (2, 3) [2 1]
16 (2, 3) [2 3]
17 (2, 3) [0 1]
18 (2, 3) [1 2]
19 (2, 3) [0 3]
20 (2, 3) [0 1]
21 (2, 3) [2 3]
22 (2, 3) [3 2]
23 (2, 3) [0 1]
24 (2, 3) [3 0]
25 (2, 3) [1 2]
26 (2, 3) [1 3]
27 (2, 3) [2 0]
28 (2, 3) [2 1]
29 (2, 3) [0 3]
30 (2, 3) [2 3]
31 (2, 3) [0 1]
32 (2, 3) [3 1]
33 (2, 3) [2 0]
34 (2, 3) [3 2]
35 (2, 3) [1 0]
36 (2, 3) [3 0]
37 (2, 3) [2 1]
38 (2, 3) [0 2]
39 (2, 3) [3 1]
```

- Untuk urutan berbeda: (2)`batch`, (1)`shuffle`, (3) `repeat`

[116]:
```
tf.random.set_seed(1)
## Order 2: batch -> shuffle -> repeat
ds = ds_joint.batch(2).shuffle(4).repeat(3)
for i,(batch_x, batch_y) in enumerate(ds):
    print(i, batch_x.shape, batch_y.numpy())
```

```
0 (2, 3) [0 1]
1 (2, 3) [2 3]
2 (2, 3) [0 1]
3 (2, 3) [2 3]
4 (2, 3) [2 3]
5 (2, 3) [0 1]
```

[117]:
```
tf.random.set_seed(1)
## Order 2: batch -> shuffle -> repeat
ds = ds_joint.batch(2).shuffle(4).repeat(20)
for i,(batch_x, batch_y) in enumerate(ds):
    print(i, batch_x.shape, batch_y.numpy())
```

```
0 (2, 3) [0 1]
1 (2, 3) [2 3]
2 (2, 3) [0 1]
3 (2, 3) [2 3]
4 (2, 3) [2 3]
5 (2, 3) [0 1]
6 (2, 3) [2 3]
7 (2, 3) [0 1]
8 (2, 3) [2 3]
9 (2, 3) [0 1]
10 (2, 3) [2 3]
11 (2, 3) [0 1]
12 (2, 3) [2 3]
13 (2, 3) [0 1]
14 (2, 3) [2 3]
15 (2, 3) [0 1]
16 (2, 3) [0 1]
17 (2, 3) [2 3]
18 (2, 3) [2 3]
19 (2, 3) [0 1]
20 (2, 3) [0 1]
21 (2, 3) [2 3]
22 (2, 3) [2 3]
23 (2, 3) [0 1]
24 (2, 3) [2 3]
25 (2, 3) [0 1]
26 (2, 3) [2 3]
27 (2, 3) [0 1]
28 (2, 3) [0 1]
29 (2, 3) [2 3]
30 (2, 3) [0 1]
31 (2, 3) [2 3]
32 (2, 3) [2 3]
33 (2, 3) [0 1]
34 (2, 3) [2 3]
```

```
35 (2, 3) [0 1]
36 (2, 3) [2 3]
37 (2, 3) [0 1]
38 (2, 3) [0 1]
39 (2, 3) [2 3]
```

Untuk contoh kode pertama dengan urutan (1)`shuffle`, (2)`batch`, (3) `repeat`, terlihat bahwa dataset telah ter-*shuffle* seperti yang diharapkan. Sedangkan kasus kedua dengan urutan (2)`batch`, (1)`shuffle`, (3) `repeat`maka elemen-elemen dalam sebuah batch tidak ter-*shuffle* sama sekali. Kekurangan *shuffling* tersebut dapat kita teliti dengan melihat tensor-nya yang mengandung harga-harga target y. Semua batch berisi pasangan harga $[y = 0, y = 1]$ atau pasangan harga sisa $[y = 2, y = 3]$. Kita tidak melihat kemungkinan permutasi lain $[y = 2, y = 0]$, $[y = 1, y = 3]$, dan selanjutnya. Perhatikan bahwa untuk menjamin hasil-hasil ini bukan kebetulan, kita dapat mengulangnya dengan jumlah yang lebih besar dari 3, misalkan `.repeat(20)`.

> **Pertanyaan 3.1.** Coba anda prediksikan apa yang akan terjadi jika digunakan operasi `shuffle` setelah `repeat`? Misalkan (2)`batch`, (3) `repeat`, (1)`shuffle`

3.4 Membuat sebuah dataset dari file-file pada disk lokal

Pada bagian ini, kita akan membuat dataset berasal dari file-file image yang disimpan pada disk. Pastikan pada folder tempat file ini berada terdapat folder dengan nama `cat_dog_images` yang berisi 6 image dari kucing dan anjing dengan format `JPEG`. Dataset kecil ini akan menunjukan bagaimana membuat dataset dari file-file yang disimpan secara umum dapat bekerja. Untuk membuatnya, kita akan menggunakan dua modul tambahan pada `TensorFlow` yaitu`tf.io` untuk membaca isi file image, dan `tf.image` untuk mendekodekan isi mentah dan *resize* image.

> **Catatan 3.1.** Modul `tf.io` dan `tf.image` menyediakan banyak fungsi-fungsi tambahan yang tidak akan dibahas di sini. Untuk mempelajari lebih jauh dapat browsing ke alamat berikut yaitu `tf.io` pada link https://www.tensorflow.org/api_docs/python/tf/io](https://www.tensorflow.org/api_docs/python/tf/io) dan `tf.image` pada link https://www.tensorflow.org/api_docs/python/tf/image.

- Sebelum memulai, kita terlibih dahulu akan melihat isi folder `cat_dog_images`. Dengan menggunakan library `pathlib`, akan diperoleh list dari file-file image yang ada pada folder tersebut.

[118]:
```
import pathlib
imgdir_path = pathlib.Path('cat_dog_images')
file_list = sorted([str(path) for path in imgdir_path.glob('*.jpg')])
print(file_list)
```

```
['cat_dog_images/cat-01.jpg', 'cat_dog_images/cat-02.jpg',
'cat_dog_images/cat-03.jpg', 'cat_dog_images/dog-01.jpg',
'cat_dog_images/dog-02.jpg', 'cat_dog_images/dog-03.jpg']
```

- Sedangkan untuk memvisualisasikan file-file image tersebut adalah dengan kode berikut.

[119]:
```
import matplotlib.pyplot as plt
import os

fig = plt.figure(figsize=(10, 5))
```

```
for i,file in enumerate(file_list):
    img_raw = tf.io.read_file(file)
    img = tf.image.decode_image(img_raw)
    print('Image shape: ', img.shape)
    ax = fig.add_subplot(2, 3, i+1)
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(img)
    ax.set_title(os.path.basename(file), size=15)

# plt.savefig('ch13-catdot-examples.pdf')
plt.tight_layout()
plt.show()
```

```
Image shape:  (900, 1200, 3)
Image shape:  (900, 1200, 3)
Image shape:  (900, 742, 3)
Image shape:  (800, 1200, 3)
Image shape:  (800, 1200, 3)
Image shape:  (900, 1200, 3)
```

Terlihat image-image tersebut mempunyai *aspect ratio* yang berbeda-beda. Kita ingin membuat *aspect ratio* menjadi sama untuk setiap image, dan label untuk image-image tersebut disediakan di dalam nama file-nama filenya. Sehingga, kita dapat mengekstrak label-label dari list nama file yang ada, memberikan label 1 untuk image anjing dan angka 0 untuk image kucing.

[120]:
```
labels = [1 if 'dog' in os.path.basename(file) else 0
          for file in file_list]
print(labels)
```

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

- Sekarang kita telah mempunyai dua list: list nama file (atau *path* dari setiap image) dan list dari label-label image tersebut. Pada bagian sebelunnya kita telah pelajari bagaimana menggabungkan *joint dataset* dari dua tensor, dan di sini kita akan gunakan pendekatan kedua. Dataset yang terbentuk akan dinamai `df_files_labels`, karena mempunyai nama file dan label.

[121]:
```
ds_files_labels = tf.data.Dataset.from_tensor_slices((file_list, labels))

for item in ds_files_labels:
    print(item[0].numpy(), item[1].numpy())
```

```
b'cat_dog_images/cat-01.jpg' 0
b'cat_dog_images/cat-02.jpg' 0
b'cat_dog_images/cat-03.jpg' 0
b'cat_dog_images/dog-01.jpg' 1
b'cat_dog_images/dog-02.jpg' 1
b'cat_dog_images/dog-03.jpg' 1
```

- Selanjutnya kita akan mengaplikasikan transformasi pada dataset tersebut: *load* isi image dari *path*-nya, dekodekan isi mentah, dan *resize* gambar ke ukuran yang diinginkan, misalkan 80×120. Sebelumnya kita sudah melihat bagaimana mengimplementasikan fungsi lambda dengan metode `.map()`. Tetapi karena kita perlu mengaplikasikan beberapa preprocessing sekaligus, maka akan dibuat sebuah *helper function* dan menggunakannya ketika memanggil `.map()`. Kode berikut akan menghasilkan visualisasi image-image dengan ukuran sama beserta dengan labelnya.

[122]:
```
def load_and_preprocess(path, label):
    image = tf.io.read_file(path)
    image = tf.image.decode_jpeg(image, channels=3)
    image = tf.image.resize(image, [img_height, img_width])
    image /= 255.0

    return image, label

img_width, img_height = 120, 80

ds_images_labels = ds_files_labels.map(load_and_preprocess)

fig = plt.figure(figsize=(10, 5))
for i,example in enumerate(ds_images_labels):
    print(example[0].shape, example[1].numpy())
    ax = fig.add_subplot(2, 3, i+1)
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(example[0])
    ax.set_title('{}'.format(example[1].numpy()),
                 size=15)
```

```
plt.tight_layout()
#plt.savefig('ch13-catdog-dataset.pdf')
plt.show()
```

```
(80, 120, 3) 0
(80, 120, 3) 0
(80, 120, 3) 0
(80, 120, 3) 1
(80, 120, 3) 1
(80, 120, 3) 1
```

3.5 *Fetching* dataset dari librari `tensorflow_datasets`

Librari `tensorflow_datasets` menyediakan banyak koleksi yang bagus dan tersedia secara gratis untuk melakukan training atau mengevaluasi model-model *deep learning*. Dataset-dataset tersebut sudah diformat dengan baik dan mempunyai deskripsi yang informatif, termasuk format *feature* dan label, tipe dan dimensinya, termasuk sitasi paper original yang memperkenalkan datasetnya dalam format BibTe$\mathcal{X}$. Keuntungan lain adalah dataset-dataset ini semuanya dipersiapkan dan siap digunakan sebagai objek `tf.data.Datasets`, sehingga semua fungsi yang disampaikan di bagian-bagian sebelumnya dapat digunakan langsung terhadap objek tersebut. Overview datasets yang disediakan oleh `TensorFlow` dapat diakses pada link berikut: Overview datasets `TensorFlow`.

- Pertama, kita perlu menginstall library `tensorflow_datasets` menggunakan `pip` (atau `conda`) dengan perintah berikut (**Catatan** Untuk dapat menggunakan dataset-dataset ini maka `TensorFlow` yang digunakan harus versi $>$ 2.1.0).

[49]:
```
! pip install tensorflow-datasets
```

- Kemudian meng-import module dan juga melihat list dataset yang disediakan.

[123]:
```
import tensorflow_datasets as tfds
print(len(tfds.list_builders()))
print(tfds.list_builders()[:5])
```

```
1139
['abstract_reasoning', 'accentdb', 'aeslc', 'aflw2k3d', 'ag_news_subset']
```

Kode di atas menghasilkan jumlah 1139 datasets (bisa jadi akan bertambah di masa depan), kemudian kita menampilkan 5 nama dataset teratas yang ada. Ada dua cara untuk *fetching* dataset, yang akan kita tunjukan dengan dua kali *fetching* dataset yang berbeda: `CelebA (celeb_a)` dan `MNIST`.

3.5.1 Cara Pertama

Pendekatan pertama terdiri dari 3 cara:

1. Memanggil fungsi *builder*
2. Mengeksekusi metode `download_and_prepare()`
3. Memanggil metode `as_dataset()`

Pertama akan kita lihat full list dataset yang ada, dengan kode berikut.

[124]:
```
## Run this to see the full list:
tfds.list_builders()
```

[124]:

- Pertama kita akan menggunakan dataset `CelebA` dan mencetak deskripsi terkait yang disediakan oleh librari.

[125]:
```
print(celeba_bldr.info.features)
```

```
FeaturesDict({
    'attributes': FeaturesDict({
        '5_o_Clock_Shadow': bool,
        'Arched_Eyebrows': bool,
        'Attractive': bool,
        'Bags_Under_Eyes': bool,
        'Bald': bool,
        'Bangs': bool,
        'Big_Lips': bool,
        'Big_Nose': bool,
        'Black_Hair': bool,
        'Blond_Hair': bool,
        'Blurry': bool,
        'Brown_Hair': bool,
        'Bushy_Eyebrows': bool,
        'Chubby': bool,
        'Double_Chin': bool,
        'Eyeglasses': bool,
        'Goatee': bool,
```

```
            'Gray_Hair': bool,
            'Heavy_Makeup': bool,
            'High_Cheekbones': bool,
            'Male': bool,
            'Mouth_Slightly_Open': bool,
            'Mustache': bool,
            'Narrow_Eyes': bool,
            'No_Beard': bool,
            'Oval_Face': bool,
            'Pale_Skin': bool,
            'Pointy_Nose': bool,
            'Receding_Hairline': bool,
            'Rosy_Cheeks': bool,
            'Sideburns': bool,
            'Smiling': bool,
            'Straight_Hair': bool,
            'Wavy_Hair': bool,
            'Wearing_Earrings': bool,
            'Wearing_Hat': bool,
            'Wearing_Lipstick': bool,
            'Wearing_Necklace': bool,
            'Wearing_Necktie': bool,
            'Young': bool,
        }),
        'image': Image(shape=(218, 178, 3), dtype=uint8),
        'landmarks': FeaturesDict({
            'lefteye_x': int64,
            'lefteye_y': int64,
            'leftmouth_x': int64,
            'leftmouth_y': int64,
            'nose_x': int64,
            'nose_y': int64,
            'righteye_x': int64,
            'righteye_y': int64,
            'rightmouth_x': int64,
            'rightmouth_y': int64,
        }),
    })
```

[126]:
```
print(celeba_bldr.info.features.keys())
```

```
dict_keys(['image', 'landmarks', 'attributes'])
```

[127]:
```
print(celeba_bldr.info.features['image'])
```

```
Image(shape=(218, 178, 3), dtype=uint8)
```

[128]:
```
print(celeba_bldr.info.features['attributes'].keys())
```

```
dict_keys(['5_o_Clock_Shadow', 'Arched_Eyebrows', 'Attractive',
'Bags_Under_Eyes', 'Bald', 'Bangs', 'Big_Lips', 'Big_Nose', 'Black_Hair',
'Blond_Hair', 'Blurry', 'Brown_Hair', 'Bushy_Eyebrows', 'Chubby', 'Double_Chin',
'Eyeglasses', 'Goatee', 'Gray_Hair', 'Heavy_Makeup', 'High_Cheekbones', 'Male',
'Mouth_Slightly_Open', 'Mustache', 'Narrow_Eyes', 'No_Beard', 'Oval_Face',
'Pale_Skin', 'Pointy_Nose', 'Receding_Hairline', 'Rosy_Cheeks', 'Sideburns',
'Smiling', 'Straight_Hair', 'Wavy_Hair', 'Wearing_Earrings', 'Wearing_Hat',
'Wearing_Lipstick', 'Wearing_Necklace', 'Wearing_Necktie', 'Young'])
```

[129]:
```
print(celeba_bldr.info.citation)
```

```
@inproceedings{conf/iccv/LiuLWT15,
  added-at = {2018-10-09T00:00:00.000+0200},
  author = {Liu, Ziwei and Luo, Ping and Wang, Xiaogang and Tang, Xiaoou},
  biburl =
{https://www.bibsonomy.org/bibtex/250e4959be61db325d2f02c1d8cd7bfbb/dblp},
  booktitle = {ICCV},
  crossref = {conf/iccv/2015},
  ee = {http://doi.ieeecomputersociety.org/10.1109/ICCV.2015.425},
  interhash = {3f735aaa11957e73914bbe2ca9d5e702},
  intrahash = {50e4959be61db325d2f02c1d8cd7bfbb},
  isbn = {978-1-4673-8391-2},
  keywords = {dblp},
  pages = {3730-3738},
  publisher = {IEEE Computer Society},
  timestamp = {2018-10-11T11:43:28.000+0200},
  title = {Deep Learning Face Attributes in the Wild.},
  url = {http://dblp.uni-trier.de/db/conf/iccv/iccv2015.html#LiuLWT15},
  year = 2015
}
```

Perintah-perintah di atas menghasilkan informasi yang berguna untuk mengetahui struktur dari dataset. *Feature* yang tersimpan pada sebuah *dictionary* terdiri dari tiga key: `image`, `landmarks` dan `attributes`. Entri `image` menyatakan image muka dari selebriti, `landmarks` menyatakan *dictionary* dari titik esktraksi wajah, seperti posisi mata, hidung dan sebagainya, sedangkan `attributes` adalah *dictionary* dari 40 atribut orang pada image, seperti ekspresi wajah, makeup, properti dari rambut, dan sebagainya.

- Selanjutnya memanggil metode `download_and_prepare()`. Perintah ini akan mendownload data dan menyimpanannya pada folder yang telah ditentukan untuk semua dataset `TensorFlow`. Jika kita telah melakukannya sekali, perintah ini hanya akan mencek apakah data sudah didownload, sehingga tidak akan mendownload kembali apabila data sudah ada pada folder tersebut.

[130]:
```
# Download the data, prepare it, and write it to disk
celeba_bldr.download_and_prepare()
```

- Kemudian melakukan *instantiate* (membuat objek) dari dataset dengan nama `CelebData`

[131]:
```
# Load data from disk as tf.data.Datasets
CelebData = celeba_bldr.as_dataset(shuffle_files=False)
CelebData.keys()
```

[131]:
```
dict_keys(['train', 'validation', 'test'])
```

Terlihat bahwa data sudah terbagi menjadi *train*, *test* dan *validation*.

- Untuk melihat bagaimana image terlihat, kode berikut dapat dieksekusi.

[132]:
```
ds_train = CelebData['train']
assert isinstance(ds_train, tf.data.Dataset)

example = next(iter(ds_train))
print(type(example))
print(example.keys())
```

```
<class 'dict'>
dict_keys(['attributes', 'image', 'landmarks'])
```

Perhatikan bahwa elemen-elemen pada dataset ini berbentuk sebuah *dictionary*. Jika kita ingin menggunakan dataset ini untuk sebuah model *supervised deep learning* pada saat training, maka harus diformat kembali ke dalam bentuk *tuple* dari (`features`, `label`). Untuk label kita akan gunakan kategori 'Male' dari attribut dengan menggunakan transformasi via `map()`:

[133]:
```
ds_train = ds_train.map(lambda item:
     (item['image'], tf.cast(item['attributes']['Male'], tf.int32)))
```

Selanjutnya, kita akan mem-*batch* dataset dan mengambil sebuah *batch* dari 18 contoh darinya untuk divisualisasikan dengan label masing-masing.

[134]:
```
ds_train = ds_train.batch(18)
images, labels = next(iter(ds_train))

print(images.shape, labels)
```

```
(18, 218, 178, 3) tf.Tensor([0 1 0 0 1 1 1 1 1 0 0 0 1 0 0 1 1 1], shape=(18,),
dtype=int32)
```

[135]:
```
fig = plt.figure(figsize=(12, 8))
for i,(image,label) in enumerate(zip(images, labels)):
    ax = fig.add_subplot(3, 6, i+1)
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(image)
    ax.set_title('{}'.format(label), size=15)


plt.show()
```

Maka akan diperoleh image 18 selebriti dengan masing-masing label 1 untuk laki-laki, dan 0 untuk perempuan. Dengan demikian proses *fetching* dan *using* dataset image `CelebA` telah dilakukan.

3.5.2 Cara Kedua

Selanjutnya kita akan menggunakan pendekatan kedua untuk *fetching* dataset `MNIST` dari `tensorflow_datasets`

- Terdapat fungsi *wrapper* yang disebut `load()` yang mengkombinasikan tiga langkah sebelumnya untuk *fetching* dataset.

[136]:
```
mnist, mnist_info = tfds.load('mnist', with_info=True, shuffle_files=False)
print(mnist_info)
print(mnist.keys())
```

```
tfds.core.DatasetInfo(
    name='mnist',
    full_name='mnist/3.0.1',
    description="""
    The MNIST database of handwritten digits.
    """,
    homepage='http://yann.lecun.com/exdb/mnist/',
    data_path='/Users/fikysuratman/tensorflow_datasets/mnist/3.0.1',
    file_format=tfrecord,
    download_size=11.06 MiB,
```

```
    dataset_size=21.00 MiB,
    features=FeaturesDict({
        'image': Image(shape=(28, 28, 1), dtype=uint8),
        'label': ClassLabel(shape=(), dtype=int64, num_classes=10),
    }),
    supervised_keys=('image', 'label'),
    disable_shuffling=False,
    splits={
        'test': <SplitInfo num_examples=10000, num_shards=1>,
        'train': <SplitInfo num_examples=60000, num_shards=1>,
    },
    citation="""@article{lecun2010mnist,
      title={MNIST handwritten digit database},
      author={LeCun, Yann and Cortes, Corinna and Burges, CJ},
      journal={ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist},
      volume={2},
      year={2010}
    }""",
)
dict_keys(['test', 'train'])
```

Terlihat bahwa, dataset `MNIST` terbagi menjadi dua partisi, yaitu `train` dan `test`.

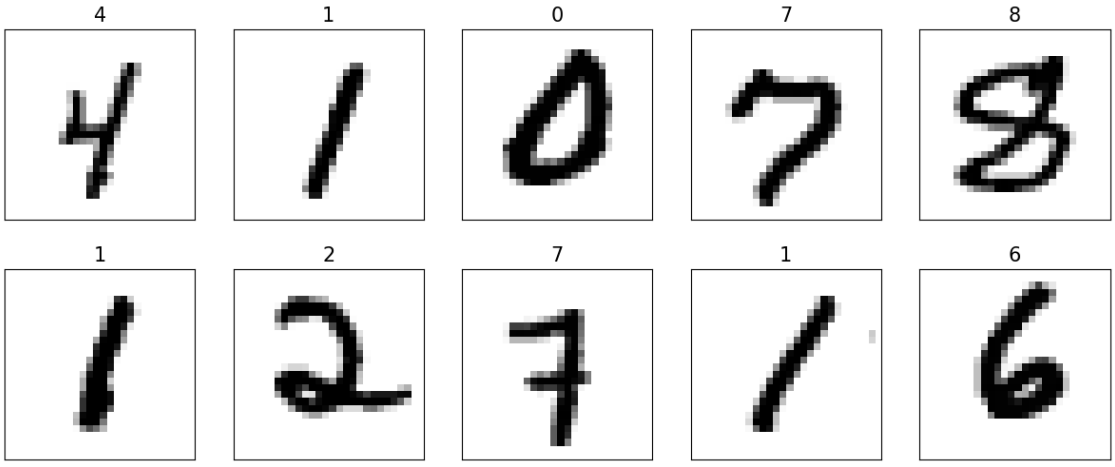
- Selanjutnya kita akan mengambil partisi `train`, mentransformasikannya untuk mengkonversi elemen-elemen yang berbentuk *dictionary* menjadi sebuah *tuple*, kemudian memvisualisasikan 10 contoh data.

[137]:
```
ds_train = mnist['train']
assert isinstance(ds_train, tf.data.Dataset)

ds_train = ds_train.map(lambda item: (item['image'], item['label']))

ds_train = ds_train.batch(10)
batch = next(iter(ds_train))
print(batch[0].shape, batch[1])

fig = plt.figure(figsize=(15, 6))
for i,(image,label) in enumerate(zip(batch[0], batch[1])):
    ax = fig.add_subplot(2, 5, i+1)
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(image[:, :, 0], cmap='gray_r')
    ax.set_title('{}'.format(label), size=15)

plt.show()
```

```
2023-03-26 08:59:33.128025: W
tensorflow/core/kernels/data/cache_dataset_ops.cc:856] The calling iterator did
not fully read the dataset being cached. In order to avoid unexpected truncation
of the dataset, the partially cached contents of the dataset  will be discarded.
```

```
This can happen if you have an input pipeline similar to
`dataset.cache().take(k).repeat()`. You should use
`dataset.take(k).cache().repeat()` instead.
```

```
(10, 28, 28, 1) tf.Tensor([4 1 0 7 8 1 2 7 1 6], shape=(10,), dtype=int64)
```



Dengan demikian pembahasan mengenai membangun, memanipulasi dataset dan *fetching* dataset dari librari `tensorflow_datasets` telah selesai. Pada bagian selanjutnya akan kita lihat bagaimana membangun model Neural Networks di `TensorFlow`