CONTOH IMPLEMENTASI `TensorFlow` PADA CNN

May 22, 2023

1 Contoh Aplikasi 1: Klasifikasi Angka Menggunakan CNN dan `TensorFlow`

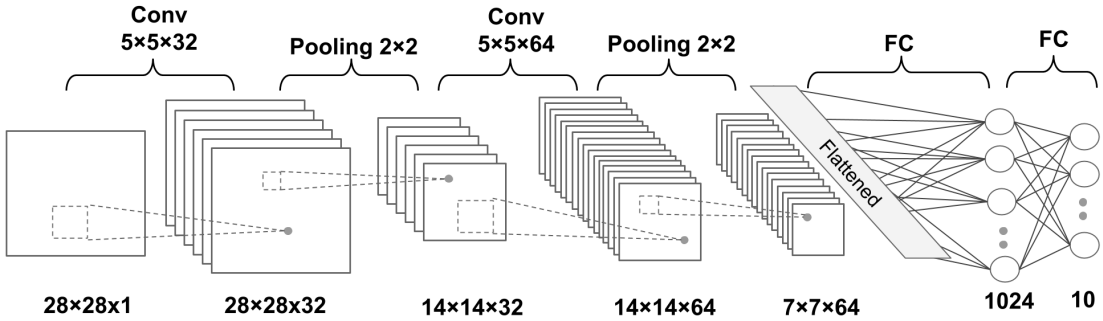
Pada penjelasan di bab sebelumnya telah diberikan contoh implementasi *Multilayer Neural Networks*. Pada bagian ini akan diberkan pemaparan implementasi API dari TansorFlow untuk CNN. Dataset yang digunakan adalah dataset tulisan tangan MNIST.

1.1 Arsitektur multilayer CNN

Arsitektur network yang akan kita implementasikan ditunjukkan pada Gambar 1. Inputnya adalah grayscale berukuran 28 x 28. Mempertimbangkan jumlah channel (yaitu 1 untuk gambar skala abu-abu) dan bacth gambar input, dimensi tensor masukan akan menjadi [*batch_size* $\times$ $28 \times 28 \times 1$].

Data input melewati dua *convolutional layer* yang memiliki ukuran kernel 5 x 5. *Convolution* pertama memiliki 32 *feature map* output, dan yang kedua memiliki 64 *feature map* output. Setiap lapisan konvolusi diikuti oleh lapisan subsampling dalam bentuk operasi max-pooling.

Kemudian sebuah *fully-connected layer* meneruskan output ke *fully-connected layer* kedua, yang bertindak sebagai lapisan keluaran softmax terakhir. Arsitektur network yang akan kita terapkan ditunjukkan pada gambar berikut:



Gambar 1: Implementasi Arsitektur CNN [batch_size X 64 X 64 X 8]

Dimensi tensor di setiap lapisan adalah sebagai berikut: >

1. **Input:** [$batchsize \times 28 \times 28 \times 1$]
2. **Conv_1:** [$batchsize \times 24 \times 24 \times 32$]
3. **Pooling_1:** [$batchsize \times 12 \times 12 \times 32$]

4. **Conv_2:** $[batchsize \times 8 \times 8 \times 64]$
5. **Pooling_1:** $[batchsize \times 4 \times 4 \times 64]$
6. **FC_1:** $[batchsize \times 1024]$
7. **FC_2 dan softmax layer:** $[batchsize \times 10]$

1.2 *Loading dan Pre-processing* Data

Pada pertemuan sebelumnya telah dijelaskan dua cara untuk melakukan loading datasets dari modul `tensorflow_dataset`, yaitu dengan menggunakan tiga tahap proses dan dengan metode lebih sederhana menggunakan fungsi `load` yang menyatukan ketiga tahap proses. Di bagian ini, kita akan menggunakan metode pertama.

- Import librari dataset tensor flow dan librari lain

```
[ ]: import tensorflow as tf
     import tensorflow_datasets as tfds
     import pandas as pd

     import matplotlib.pyplot as plt
     %matplotlib inline
```

- Tiga langkah untuk loading dataset MNIST.

```
[ ]: ## MNIST dataset

     mnist_bldr = tfds.builder('mnist') # langkah 1
     mnist_bldr.download_and_prepare()  # langkah 2
     datasets = mnist_bldr.as_dataset(shuffle_files=False) # langkah 3
     print(datasets.keys())
     mnist_train_orig, mnist_test_orig = datasets['train'], datasets['test']
```

Hasil di atas menunjukan bahwa Dataset MNIST mempunyai skema partisi untuk training dan test dataset. Tapi kita akan membuat juga data validasi dari partisi training. Pada langkah 3, digunakan `shuffle_files=False` pada metode `.as_dataset()` dengan tujuan untuk mencegah *shuffling* awal, karena kita akan membagi dataset training ke dalam dataset training yang lebih kecil dan sebuah dataset validasi (jika tidak dilakukan makan akan menghasilkan shuffling dataset setiap kali kita mengambil mini-batch dari data, sehingga kemungkinan dataset training dan validasi akan tercampur)

- *Splitting* dataset menjadi dataset training dan validasi

```
[ ]: BUFFER_SIZE = 10000
     BATCH_SIZE = 64
     NUM_EPOCHS = 20

     mnist_train = mnist_train_orig.map(
         lambda item: (tf.cast(item['image'], tf.float32)/255.0,
                       tf.cast(item['label'], tf.int32)))
```

```
mnist_test = mnist_test_orig.map(
    lambda item: (tf.cast(item['image'], tf.float32)/255.0,
                  tf.cast(item['label'], tf.int32)))

tf.random.set_seed(1)

mnist_train = mnist_train.shuffle(buffer_size=BUFFER_SIZE,
                                  reshuffle_each_iteration=False)

mnist_valid = mnist_train.take(10000).batch(BATCH_SIZE)
mnist_train = mnist_train.skip(10000).batch(BATCH_SIZE)
```

Dataset sudah dipersiapkan, langkah selanjutnya adalah implementasi CNN.

1.3 Implementasi CNN menggunakan `TensorFlow` Keras API

Untuk mengimplementasikan CNN dalam `TensorFlow`, kita menggunakan class Keras `Sequential` untuk menumpukan (*stack*) layer-layer yang berbeda, seperti *convolution*, *pooling*, *dropout*, dan juga *fully connected layers* (*dense layers*). Keras layer API menyediakan class-class berikut. * Conv2D: `tf.keras.layers.Conv2D`, untuk *convolutional layer* dua dimensi, dengan parameter berikut: * `filters`: jumlah output filter (Ekivalen dengan jumlah output feature maps) * `kernel_size`: ukuran kernel * `strides`: dengan default 1
* `padding`: bisa dengan mode valid atau same * MaxPool2D: `tf.keras.layers.MaxPool2D`, untuk subsampling (*max pooling* dan *average pooling*), dengan parameter berikut: * `pool_size`: menentukan ukuran window atau (piksel) tetangga yang akan digunakan untuk menghitung operasi `max` atau `mean` * `strides`: untuk mengkonfigurasi layer pooling * Dropout: `tf.keras.layers.Dropout`, untuk regularisasi menggunakan *dropout*, dengan parameter berikut: * `rate`: untuk menentukan probabilitas dropping unit-unit input selama proses training

Sebagai catatan, ketika membaca image, dimensi default untuk jumlah *channel* adalah dimensi terakhir dari tensor array. Format ini biasanya disebut dengan NHWC (N = jumlah image dalam batch, H = *height*, W = *width*, dan C = *channels*). Class `Conv2D` mengasumsikan input-input mempunyai format NHWC secara default. Dalam beberapa kasus dimana data dengan channel terletak pada dimensi pertama (setelah dimensi batch), maka diperlukan memindahkan sumbu-sumbu data sehingga channel berada di dimensi terakhir. Atau sebagai alternatif adalah dengan menyatakan `data_format="channels_first"`. Setelah layer dikonstruksi, maka dapat dipanggil dengan menyediakan tensor 4 dimensi dengan dimensi pertama sebagai batch, dimensi kedua tergantung setting `data_format`, dan dua dimensi lain adalah dimensi spasial.

Pada bagian ini kita akan mengkonstruksi model CNN yang terlihat pada Gambar 9 menggunakan class `Sequential`, kemudian menambahkan layer-layer convolution dan pooling. * Menambahkan 2 layer convolution dan dua layer pooling pada model.

[ ]:
```
model = tf.keras.Sequential()

model.add(tf.keras.layers.Conv2D(
    filters=32, kernel_size=(5, 5),
```

```
    strides=(1, 1), padding='same',
    data_format='channels_last',
    name='conv_1', activation='relu'))

model.add(tf.keras.layers.MaxPool2D(
    pool_size=(2, 2), name='pool_1'))

model.add(tf.keras.layers.Conv2D(
    filters=64, kernel_size=(5, 5),
    strides=(1, 1), padding='same',
    name='conv_2', activation='relu'))

model.add(tf.keras.layers.MaxPool2D(pool_size=(2, 2), name='pool_2'))
```

Layer max-pooling dengan ukuran 2×2 dan stride sama dengan 2 akan mengurangi dimensi spasial setengahnya (Jika parameter `stride` tidak dispesifikasikan pada `MaxPool2D` maka secara default ukuran stride akan sama dengan ukuran pooling)

- Pada tahap ini kita bisa menghitung ukuran feature maps secara manual dengan Keras API

[ ]:
```
model.compute_output_shape(input_shape=(16, 28, 28, 1))
```

```
TensorShape([16, 7, 7, 64])
```

Dengan menyediakan ukuran input (*input shape*) sebagai *tuple* yang dispesifikasikan pada contoh ini, metode `compute_output_shape` menghasilkan output dengan ukuran $(16, 7, 7, 64)$, yang menyatakan feature maps dengan 64 channels dan ukuran spasial 7×7 (karena `padding='same'`). Dimensi pertama menyatakan ukuran batch yaitu 16. Kita dapat menggunakan `None` juga, sehingga `input_shape=(None, 28,28,1)`. Sebagai catatan, Gambar 9 mendeskripsikan ukuran spasial output tanpa adanya padding pada convolution layer (`padding='valid'`), sehingga ukuran output menjadi $4 \times 4 \times 64$.

- Selanjutnya menambahkan *fully connected layer* (*dense layer*) untuk mengimplementasikan *classifier* setelah 2 convolution layer dan 2 pooling layer. Input pada layer ini seharusnya mempunyai rank 2 (vektor) dengan ukuran [batch_size $\times$ input_units]. Sehingga kita perlu merubah ouput dua dimensi dari layer sebelumnya menjadi vektor (*flatten*).

[ ]:
```
model.add(tf.keras.layers.Flatten())
model.compute_output_shape(input_shape=(16, 28, 28, 1))
```

```
TensorShape([16, 3136])
```

- Kemudian, kita akan menambahkan dua dense layer dengan dropout layer diantaranya.

[ ]:
```
model.add(tf.keras.layers.Dense(
    units=1024, name='fc_1',
    activation='relu'))

model.add(tf.keras.layers.Dropout(
    rate=0.5))
```

```
model.add(tf.keras.layers.Dense(
    units=10, name='fc_2',
    activation='softmax'))
```

Dense layer yang terakhir (`fc_2`) mempunyai 10 output unit untuk 10 class label pada dataset MNIST, dengan fungsi aktivasi `softmax` untuk memperoleh probabilitas class-membership tiap class.

- Selanjutnya, jika diperlukan melihat kembali ukuran output.

[ ]:
```
model.compute_output_shape(input_shape=(16, 28, 28, 1))
```

```
TensorShape([16, 10])
```

- eksekusi metode `.build()` untuk *late variable creation* dan cek model summary.

[ ]:
```
tf.random.set_seed(1)
model.build(input_shape=(None, 28, 28, 1))
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv_1 (Conv2D)             (None, 28, 28, 32)        832

 pool_1 (MaxPooling2D)       (None, 14, 14, 32)        0

 conv_2 (Conv2D)             (None, 14, 14, 64)        51264

 pool_2 (MaxPooling2D)       (None, 7, 7, 64)          0

 flatten_2 (Flatten)         (None, 3136)              0

 flatten_3 (Flatten)         (None, 3136)              0

 fc_1 (Dense)                (None, 1024)              3212288

 dropout (Dropout)           (None, 1024)              0

 fc_2 (Dense)                (None, 10)                10250

=================================================================
Total params: 3,274,634
Trainable params: 3,274,634
Non-trainable params: 0
_________________________________________________________________
```

- Kita akan menggunakan *loss function* `SparseCategorialCrossentropy` karena kasusnya adalah klasifikasi multiclass dengan label integer (*sparse*), bukan *one-hot encoded label*. Untuk optimizer, digunakan Adam Optimzer (`tf.keras.optimizer.Adam()`), yang merupakan metode berbasis gradien yang robust, cocok untuk optimisasi *non-convex* dan masalah-masalah machine learning. Metode populer lain berbasis Adam Optimizer adalan `RMSProp` dan `AdaGrad`. Dengan demikian metode `.compile()` menjadi:

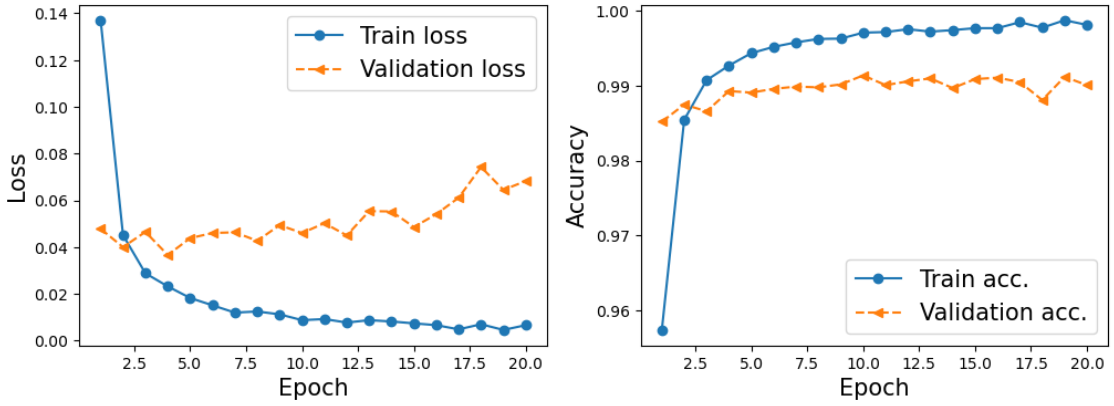
[ ]:
```
model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy']) # same as `tf.keras.metrics.
↪SparseCategoricalAccuracy(name='accuracy')`
```

- Untuk mentraining model, kita panggil metode `.fit()`. Program akan berhenti setelah jumlah epochs mencapai 20.

[ ]:
```
history = model.fit(mnist_train, epochs=NUM_EPOCHS,
                    validation_data=mnist_valid,
                    shuffle=True)
```

- Kita dapat memvisualisasikan *learning curve*.

[ ]:
```
import numpy as np
hist = history.history
x_arr = np.arange(len(hist['loss'])) + 1

fig = plt.figure(figsize=(12, 4))
ax = fig.add_subplot(1, 2, 1)
ax.plot(x_arr, hist['loss'], '-o', label='Train loss')
ax.plot(x_arr, hist['val_loss'], '--<', label='Validation loss')
ax.set_xlabel('Epoch', size=15)
ax.set_ylabel('Loss', size=15)
ax.legend(fontsize=15)
ax = fig.add_subplot(1, 2, 2)
ax.plot(x_arr, hist['accuracy'], '-o', label='Train acc.')
ax.plot(x_arr, hist['val_accuracy'], '--<', label='Validation acc.')
ax.legend(fontsize=15)
ax.set_xlabel('Epoch', size=15)
ax.set_ylabel('Accuracy', size=15)

#plt.savefig('figures/15_12.png', dpi=300)
plt.show()
```



- Evaluasi model yang telah ditraining menggunakan dataset test dapat dilakukan dengan memanggil fungsi `.evaluate()`.

[ ]:
```
test_results = model.evaluate(mnist_test.batch(20))
print('\nTest Acc. {:.2f}%'.format(test_results[1]*100))
```

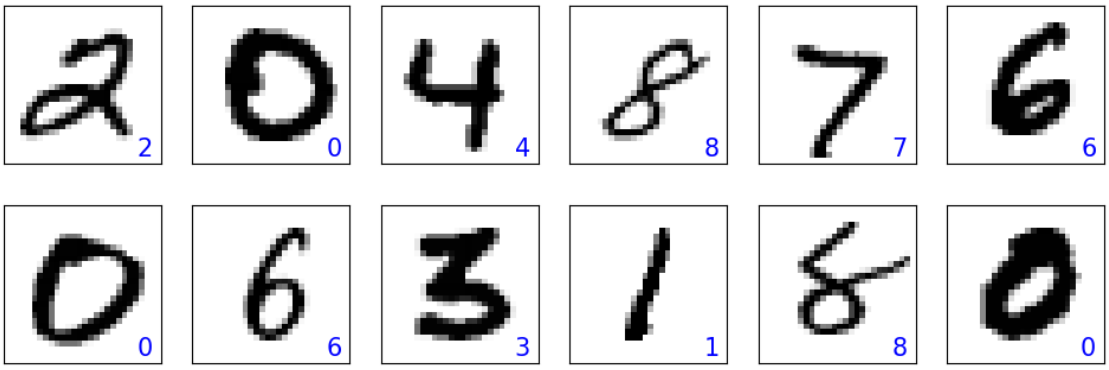
```
500/500 [==============================] - 4s 8ms/step - loss: 0.0429 -
accuracy: 0.9921

Test Acc. 99.21%
```

- Terakhir, kita akan memperoleh hasil prediksi dalam bentuk probabilitas class-membership dan mengkonversinya menjadi *predicted label* menggunakan fungsi `tf.argmax` untuk mencari elemen dengan probabilitas maksimum. Kita akan gunakan ukuran batch 12 dalam contoh ini, untuk memvisualisasikan input dan *predicted label*.

[ ]:
```
batch_test = next(iter(mnist_test.batch(12)))

preds = model(batch_test[0])

tf.print(preds.shape)
preds = tf.argmax(preds, axis=1)
print(preds)

fig = plt.figure(figsize=(12, 4))
for i in range(12):
    ax = fig.add_subplot(2, 6, i+1)
    ax.set_xticks([]); ax.set_yticks([])
    img = batch_test[0][i, :, :, 0]
    ax.imshow(img, cmap='gray_r')
    ax.text(0.9, 0.1, '{}'.format(preds[i]),
            size=15, color='blue',
            horizontalalignment='center',
```

```
            verticalalignment='center',
            transform=ax.transAxes)

#plt.savefig('figures/15_13.png', dpi=300)
plt.show()
```

```
TensorShape([12, 10])
tf.Tensor([2 0 4 8 7 6 0 6 3 1 8 0], shape=(12,), dtype=int64)
```



Hasil plot di atas menunjukan *predicted label* semuanya betul. Apakah anda bisa mencari *predicted label* yang salah seperti pada bab sebelumnya?

2 Contoh Aplikasi 2: Klasifikasi Gender berdasarkan Gambar Muka dengan CNN dan `TensorFlow`

Pada contoh kedua ini, kita akan gunakan CNN untuk klasifikasi gender berdasarkan gambar muka (*face images*) menggunakan dataset `CelebA`, yang mempunyai data sebanyak 202599 gambar muka dari para selebriti. Selain itu, terdapat 40 atribut muka bernilai biner yang tersedia untuk masing-masing gambar, termasuk gender (male dan female) dan usia (young dan old). Pada bagian ini kita hanya akan menggunakan dataset untuk training sejumlah 16000 sehingga bisa lebih cepat. Tetapi, untuk memperbaiki kinerja generalisasi dan mengurangi overfitting pada dataset yang kecil, kita akan menggunakan teknik yang disebut **data augmentation**.

2.1 *Loading* Dataset `CelebA`

- Loading dataset seperti yang kita lakukan untuk dataset MNIST sebelumnya. `CelebA` mempunyai tiga partisi: dataset training, dataset validasi dan dataset untuk test. Selanjutnya kita akan mengimplementasikan fungsi yang sederhana untuk menghitung jumlah data (*examples*) pada masing-masing partisi.

[ ]:
```
import tensorflow as tf
import tensorflow_datasets as tfds
celeba_bldr = tfds.builder('celeb_a')
```

```
celeba_bldr.download_and_prepare()
celeba = celeba_bldr.as_dataset(shuffle_files=False)
print(celeba.keys())

celeba_train = celeba['train']
celeba_valid = celeba['validation']
celeba_test = celeba['test']

def count_items(ds):
    n = 0
    for _ in ds:
        n += 1
    return n

print('Train set:  {}'.format(count_items(celeba_train)))
print('Validation: {}'.format(count_items(celeba_valid)))
print('Test set:   {}'.format(count_items(celeba_test)))
```

```
dict_keys(['train', 'validation', 'test'])
Train set:  162770
Validation: 19867
Test set:   19962
```

- Kita tidak akan menggunakan keseluruhan dataset, tetapi hanya 16000 untuk training dan 1000 untuk validasi, seperti berikut.

```
[ ]: celeba_train = celeba_train.take(16000)
celeba_valid = celeba_valid.take(1000)

print('Train set:  {}'.format(count_items(celeba_train)))
print('Validation: {}'.format(count_items(celeba_valid)))
```

```
Train set:  16000
Validation: 1000
```

Sebagai catatan, jika argumen `shuffle_files` pada `celeba_bldr.as_dataset()` di langkah sebelumnya tidak diset `False`, kita masih akan melihat sejumlah 16000 data training dan 1000 data validasi. Tetapi, setiap kali iterasi, akan *reshuffle* training data dan mengambil satu set baru sejumlah 16000 data. Maka tujuan kita untuk mentraining model menggunakan dataset kecil tidak akant tercapai. Selanjutnya kita akan mendiskusikan *data augmentation* sebagai teknik untuk meningkatkan kinerja dari deep NN.

2.2 Transformasi Image dan *Data Augmentation*

Data augmentation (DA) merangkum satu set teknik-teknik untuk berhubungan dengan kasus-kasus dimana data training terbatas. Misalkan, teknik DA tertentu dapat digunakan untuk memodifikasi atau bahkan mensintesa secara artifisial lebih banyak data dan dengan demikian akan meningkatkan kinerja dari sebuah model machine/deep learning dengan mengurangi overfitting. Meskipun DA tidak khusus untuk data-data image, tetapi ada satu set transformasi yang dapat

diaplikasikan secara unik pada data-data image, seperti *cropping* sebagian dari sebuah image, *flipping*, merubah kontras, *brightness* dan saturasi. Kita akan melihat beberapa jenis transformasi yang tersedia via modul `tf.image`.

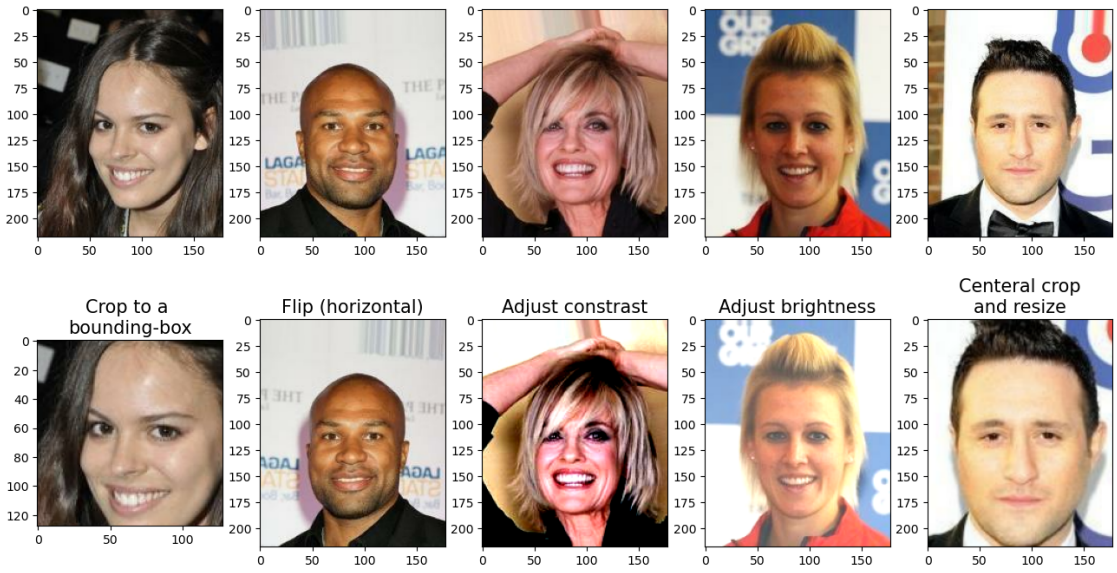
- Kita akan melihat lima contoh dari dataset `celeba_train` dan mengaplikasikan 5 jenis transformasi, yaitu:

1. *Cropping* sebuah image ke dalam sebuah *bounding box*
2. Memutar (*flipping*) image secara horizontal (*flip horizontal*)
3. Mengatur kontras (*adjust contrast*)
4. Mengatur *brightness* (*adjust brightness*)
5. *Center-cropping* sebuah image dan *resizing* hasilnya ke ukuran semula $(218, 178)$ (*central-crop and resize*).

- Visualisasi hasil dari transformasi dapat dilihat pada kode berikut.

[ ]:
```
## take 5 examples:
examples = []
for example in celeba_train.take(5):
    examples.append(example['image'])

fig = plt.figure(figsize=(16, 8.5))

## Column 1: cropping to a bounding-box
ax = fig.add_subplot(2, 5, 1)
ax.imshow(examples[0])
ax = fig.add_subplot(2, 5, 6)
ax.set_title('Crop to a \nbounding-box', size=15)
img_cropped = tf.image.crop_to_bounding_box(
    examples[0], 50, 20, 128, 128)
ax.imshow(img_cropped)

## Column 2: flipping (horizontally)
ax = fig.add_subplot(2, 5, 2)
ax.imshow(examples[1])
ax = fig.add_subplot(2, 5, 7)
ax.set_title('Flip (horizontal)', size=15)
img_flipped = tf.image.flip_left_right(examples[1])
ax.imshow(img_flipped)

## Column 3: adjust contrast
ax = fig.add_subplot(2, 5, 3)
ax.imshow(examples[2])
ax = fig.add_subplot(2, 5, 8)
ax.set_title('Adjust constrast', size=15)
img_adj_contrast = tf.image.adjust_contrast(
    examples[2], contrast_factor=2)
ax.imshow(img_adj_contrast)
```

```
## Column 4: adjust brightness
ax = fig.add_subplot(2, 5, 4)
ax.imshow(examples[3])
ax = fig.add_subplot(2, 5, 9)
ax.set_title('Adjust brightness', size=15)
img_adj_brightness = tf.image.adjust_brightness(
    examples[3], delta=0.3)
ax.imshow(img_adj_brightness)

## Column 5: cropping from image center
ax = fig.add_subplot(2, 5, 5)
ax.imshow(examples[4])
ax = fig.add_subplot(2, 5, 10)
ax.set_title('Centeral crop\nand resize', size=15)
img_center_crop = tf.image.central_crop(
    examples[4], 0.7)
img_resized = tf.image.resize(
    img_center_crop, size=(218, 178))
ax.imshow(img_resized.numpy().astype('uint8'))

# plt.savefig('figures/15_14.png', dpi=300)
plt.show()
```

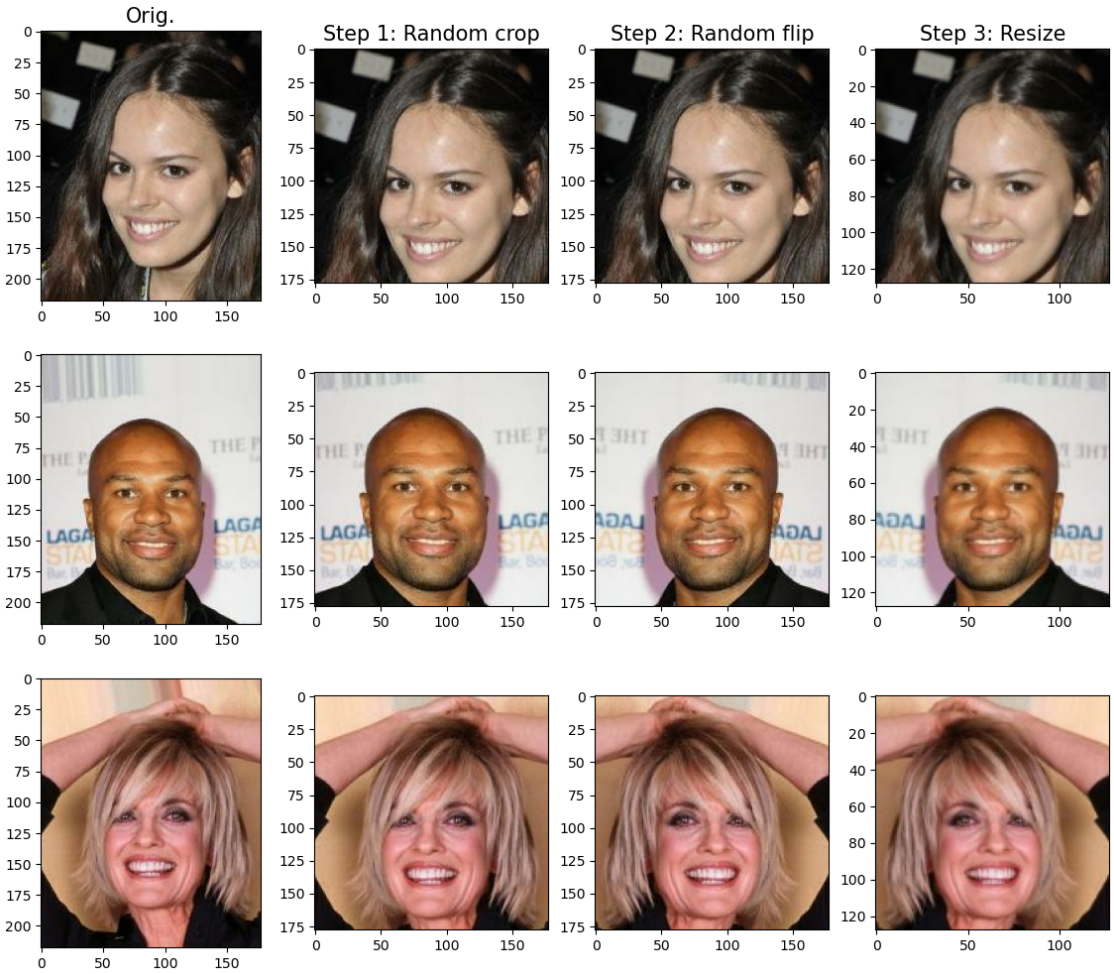


Pada gambar-gambar yang dihasilkan program di atas, baris pertama adalah image-image orsinil, dan baris kedua adalah image-image hasil 5 transformasi. Transformasi yang pertama (*crop to a bounding-box*) mempunyai spesifikasi empat angka, yaitu koordinat dari sudut kiri atas (`x=20,y=50`) serta lebar (`width=128`) dan tinggi (`height=128`). Pada `TensorFlow` posisi `(0,0)` adalah sudut kiri atas dari image.

Transformasi yang telah dilakukan pada program sebelumnya bersifat deterministik. Tetapi transformasi-transformasi tersebut dapat dibuat random (direkomendasikan untuk DA selama mentraining model).

- Sebagai contoh, program berikut melakukan *crop* image secara random, kemudian *flip* secara random, dan terakhir merubah ukuran (*resize*) ke ukuran yang diinginkan.

[ ]:
```
tf.random.set_seed(1)

fig = plt.figure(figsize=(14, 12))

for i,example in enumerate(celeba_train.take(3)):
    image = example['image']

    ax = fig.add_subplot(3, 4, i*4+1)
    ax.imshow(image)
    if i == 0:
        ax.set_title('Orig.', size=15)

    ax = fig.add_subplot(3, 4, i*4+2)
    img_crop = tf.image.random_crop(image, size=(178, 178, 3))
    ax.imshow(img_crop)
    if i == 0:
        ax.set_title('Step 1: Random crop', size=15)

    ax = fig.add_subplot(3, 4, i*4+3)
    img_flip = tf.image.random_flip_left_right(img_crop)
    ax.imshow(tf.cast(img_flip, tf.uint8))
    if i == 0:
        ax.set_title('Step 2: Random flip', size=15)

    ax = fig.add_subplot(3, 4, i*4+4)
    img_resize = tf.image.resize(img_flip, size=(128, 128))
    ax.imshow(tf.cast(img_resize, tf.uint8))
    if i == 0:
        ax.set_title('Step 3: Resize', size=15)

# plt.savefig('figures/15_15.png', dpi=300)
plt.show()
```



Sebagai catatan setiap kali eksekusi program di atas maka akan diperoleh image-image yang sedikit berbeda karena adanya transformasi random.

Kita juga dapat mendefinisikan fungsi *wrapper* dengan menggunakan *pipeline* untuk DA supaya terlihat lebih jelas dan ringkas. Kode berikut mendefinisikan fungsi `preprocess()` yang akan menerima format data *dictionary* yang berisi key `image` dan `attributes`. Fungsi akan mengembalikan sebuah tuple image yang telah ditransformasikan dan label yang telah diekstraksi dari atribut dictionary tersebut.

Kita hanya akan menerapkan DA pada data-data training, tetapi tidak untuk image-image validasi atau test.

[ ]:
```
def preprocess(example, size=(64, 64), mode='train'):
    image = example['image']
    label = example['attributes']['Male']
    if mode == 'train':
        image_cropped = tf.image.random_crop(
            image, size=(178, 178, 3))
```

```
        image_resized = tf.image.resize(
            image_cropped, size=size)
        image_flip = tf.image.random_flip_left_right(
            image_resized)
        return (image_flip/255.0, tf.cast(label, tf.int32))

    else:
        image_cropped = tf.image.crop_to_bounding_box(
            image, offset_height=20, offset_width=0,
            target_height=178, target_width=178)
        image_resized = tf.image.resize(
            image_cropped, size=size)
        return (image_resized/255.0, tf.cast(label, tf.int32))

## testing:
#item = next(iter(celeba_train))
#preprocess(item, mode='train')
```

- Untuk melihat DA, kita akan membuat subset kecil dari dataset training dan mengimplementasikan fungsi terhadap subset tersebut. Kemudian melakukan iterasi pada dataset tersebut lima kali.

[ ]:
```
tf.random.set_seed(1)

ds = celeba_train.shuffle(1000, reshuffle_each_iteration=False)
ds = ds.take(2).repeat(5)

ds = ds.map(lambda x:preprocess(x, size=(178, 178), mode='train'))

fig = plt.figure(figsize=(15, 6))
for j,example in enumerate(ds):
    ax = fig.add_subplot(2, 5, j//2+(j%2)*5+1)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.imshow(example[0])

#plt.savefig('figures/15_16.png', dpi=300)
plt.show()
```

Hasil gambar di atas menunjukan lima hasil transformasi untuk DA pada dua data image.

Kemudian, kita akan mengimplementasikan fungsi preprocessing pada dataset training dan validasi. Kita akan gunakan image dengan ukuran $(64, 64)$. Selanjutnya kita akan spesifikasikan `mode='train'` ketika bekerja dengan data training dan menggunakan `mode='eval'` untuk data validasi sehingga elemen-elemen random dari pipeline DA akan diaplikasikan hanya pada data training.

[ ]:
```
BATCH_SIZE = 32
BUFFER_SIZE = 1000
IMAGE_SIZE = (64, 64)
steps_per_epoch = np.ceil(16000/BATCH_SIZE)
print(steps_per_epoch)

ds_train = celeba_train.map(
    lambda x: preprocess(x, size=IMAGE_SIZE, mode='train'))
ds_train = ds_train.shuffle(buffer_size=BUFFER_SIZE).repeat()
ds_train = ds_train.batch(BATCH_SIZE)

ds_valid = celeba_valid.map(
    lambda x: preprocess(x, size=IMAGE_SIZE, mode='eval'))
ds_valid = ds_valid.batch(BATCH_SIZE)
```

```
500.0
```

2.3 Training CNN untuk Klasifikasi Gender

Membangun model dengan API Keras `TensorFlow` dan mentrainingnya sekarang bisa dilakukan secara langsung. Desain dari CNN kita adalah sebagai berikut: Model CNN menerima image sebagai input dengan ukuran $64 \times 64 \times 3$ (image-image mempunyai 3 channel warna, menggunakan `channels_last`). Data input melalui empat convolution layer untuk menghasilkan feature map-feature map $32, 64, 128$ dan 256 menggunakan filter-filter dengan ukuran 3×3. Tiga convolution layer yang pertama diikuti dengan `max-pooling` $P_{2\times2}$, dan dua dropout layer diterapkan

untuk regularisasi.

```
[ ]: model = tf.keras.Sequential([
         tf.keras.layers.Conv2D(
             32, (3, 3), padding='same', activation='relu'),
         tf.keras.layers.MaxPooling2D((2, 2)),
         tf.keras.layers.Dropout(rate=0.5),

         tf.keras.layers.Conv2D(
             64, (3, 3), padding='same', activation='relu'),
         tf.keras.layers.MaxPooling2D((2, 2)),
         tf.keras.layers.Dropout(rate=0.5),

         tf.keras.layers.Conv2D(
             128, (3, 3), padding='same', activation='relu'),
         tf.keras.layers.MaxPooling2D((2, 2)),

         tf.keras.layers.Conv2D(
             256, (3, 3), padding='same', activation='relu'),
     ])
```

- Kita akan lihat ukuran luaran feature maps setelah menerapkan layer-layer tersebut.

```
[ ]: model.compute_output_shape(input_shape=(None, 64, 64, 3))
```

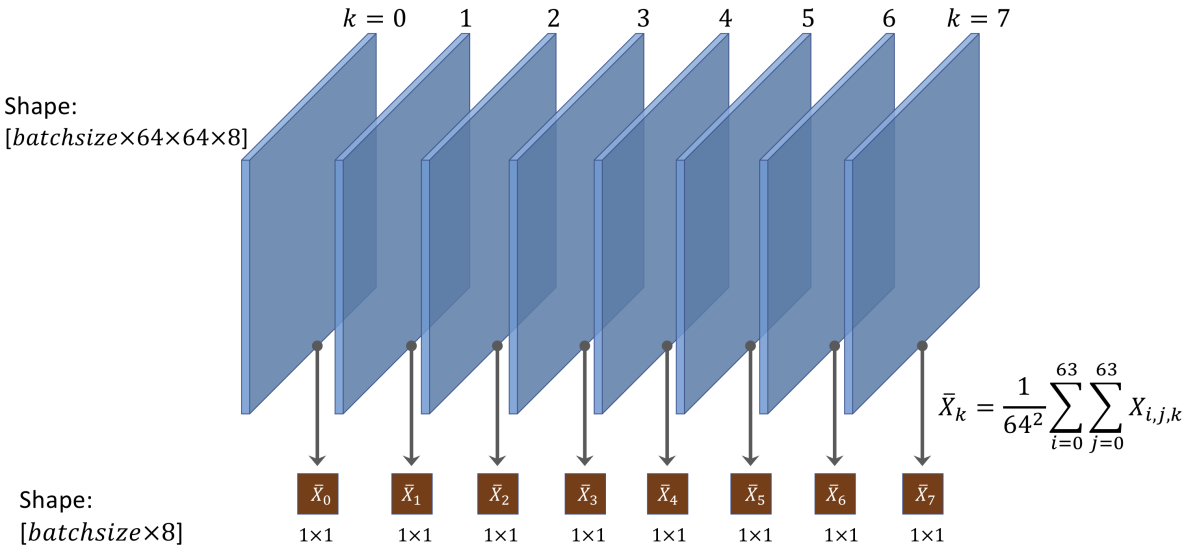
```
TensorShape([None, 8, 8, 256])
```

Terdapat 256 feature maps (*channels*) dengan ukuran 8×8. Sekarang bisa ditambahkan *fully-connected layer* untuk mendapatkan layer output dengan jumlah satu unit. Jika melakukan *flatten* dari feature map-feature map tersebut, maka jumlah unit input pada *full-connected layer* menjadi $8 \times 8 \times 256 = 16.384$. Alternatif lain adalah kita bisa mempertimbangkan layer baru yang disebut dengan *global average-pooing* (GAP) yang menghitung rata-rata masing-masing feature map secara terpisah, sehingga mengurangi *hidden unit* menjadi 256. Kemudian baru ditambahkan sebuah *fully connected layer*. GAP secara konsep sama dengan metode pooling lain. GAP dapat dilihat sebagai kasus khusus *average-pooling* dimana ukuran pooling sama dengan jumlah feature map-feature map input.

Untuk memahami lebih lanjut, perhatikan Gambar 2 yang menunjukan contoh dari input feature map dengan ukuran $batch\_size \times 64 \times 64 \times 8$. Channel-channel diberikan nomor $k = 0, 1, 2, \ldots, 7$. Operasi GAP menghitung rata-rata tiap channel sehingga ukuran output menjadi $[batch\_size \times 8]$. Sebagai catatan `GlobalAveragePooling2D` pada Keras API akan otomatis memeras outpuput menjadi berukuran $[batch\_size \times 8]$ dari sebelumnya berukuran $[batch\_size \times 1 \times 1 \times 8]$ (GAP akan mengurangi dimensi spasial 64×64 menjadi 1×1).

Pada contoh ini, ukuran feature map-feature map sebelum *dense layer* adalah $[batch\_size \times 8 \times 8 \times 256]$ dan luarannya yang diharapkan mempunyai 256 unit (atau ukuran luaran menjadi $[batch\_size \times 256]$).

- Program berikut menambahkan GAP dan menghitung kembali ukuran luaran.



Gambar 2: Ilustrasi input feature map dengan ukuran [batch_size X 64 X 64 X 8]

[ ]:
```
model.add(tf.keras.layers.GlobalAveragePooling2D())
model.compute_output_shape(input_shape=(None, 64, 64, 3))
```

```
TensorShape([None, 256])
```

Kemudian kita akan menambahkan *fully connected layer* untuk menghasilkan satu output unit. Untuk contoh ini, fungsi aktivasi yang digunakan adalah `sigmoid` atau hanya menggunakan `activation=none`. Akibatnya model akan menghasilkan logits (bukan probabilitas *class-membership*). Hasil logits ini lebih diinginkan untuk memodelkan training dengan `TensorFlow` dan Keras karena kestabilan numerik.

[ ]:
```
model.add(tf.keras.layers.Dense(1, activation=None))
```

[ ]:
```
tf.random.set_seed(1)
model.build(input_shape=(None, 64, 64, 3))
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 64, 64, 32)        896

 max_pooling2d_3 (MaxPooling  (None, 32, 32, 32)       0
 2D)

 dropout_3 (Dropout)         (None, 32, 32, 32)        0

 conv2d_5 (Conv2D)           (None, 32, 32, 64)        18496
```

```
 max_pooling2d_4 (MaxPooling  (None, 16, 16, 64)       0
 2D)

 dropout_4 (Dropout)         (None, 16, 16, 64)        0

 conv2d_6 (Conv2D)           (None, 16, 16, 128)       73856

 max_pooling2d_5 (MaxPooling  (None, 8, 8, 128)        0
 2D)

 conv2d_7 (Conv2D)           (None, 8, 8, 256)         295168

 global_average_pooling2d_2   (None, 256)              0
 (GlobalAveragePooling2D)

 dense_1 (Dense)             (None, 1)                 257

=================================================================
Total params: 388,673
Trainable params: 388,673
Non-trainable params: 0
_________________________________________________________________
```

- Langkah selanjutnya adalah mengkompilasi model dan harus menentukan *loss function* apa yang harus dipakai. Kita memiliki kasus *binary classification* dengan satu output unit sehingga akan kita gunakan `BinaryCrossEntropy`. Karena layer terakhir tidak mengimplementasikan fungsi aktivasi sigmoid (`activation=None`), maka luaran dari model adalah logits bukan probabilitas. Oleh sebab itu, kita spesifikasikan `from_logits=True`pada `BinaryCrossentropy`. Kode untuk mengkompilasi dan mentraining model adalah sebagai berikut.

[ ]:
```
model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(ds_train, validation_data=ds_valid,
                    epochs=20, steps_per_epoch=steps_per_epoch)
```

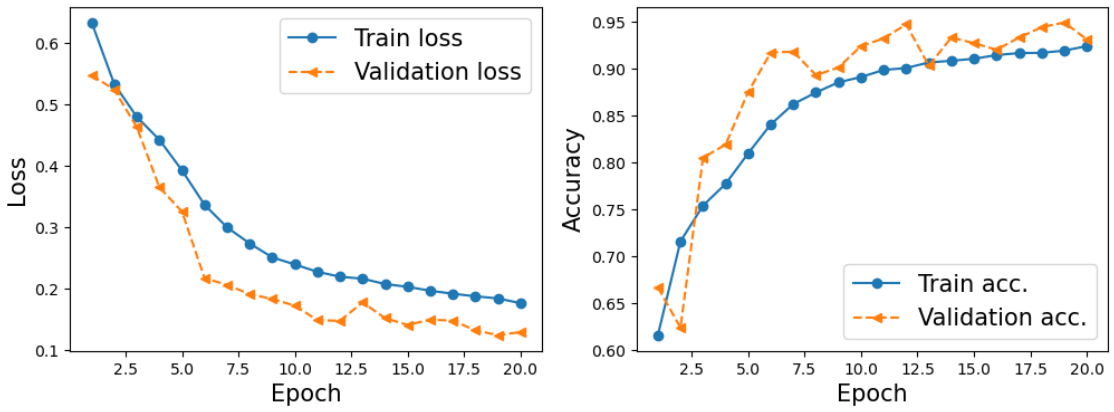
```
Epoch 1/20
500/500 [==============================] - 53s 104ms/step - loss: 0.6333 -
accuracy: 0.6161 - val_loss: 0.5477 - val_accuracy: 0.6670
Epoch 2/20
500/500 [==============================] - 52s 104ms/step - loss: 0.5330 -
accuracy: 0.7159 - val_loss: 0.5238 - val_accuracy: 0.6240
Epoch 3/20
500/500 [==============================] - 52s 104ms/step - loss: 0.4795 -
accuracy: 0.7541 - val_loss: 0.4627 - val_accuracy: 0.8050
Epoch 4/20
```

```
500/500 [==============================] - 52s 104ms/step - loss: 0.4422 -
accuracy: 0.7772 - val_loss: 0.3649 - val_accuracy: 0.8190
Epoch 5/20
500/500 [==============================] - 53s 105ms/step - loss: 0.3924 -
accuracy: 0.8097 - val_loss: 0.3253 - val_accuracy: 0.8750
Epoch 6/20
500/500 [==============================] - 52s 104ms/step - loss: 0.3363 -
accuracy: 0.8405 - val_loss: 0.2169 - val_accuracy: 0.9170
Epoch 7/20
500/500 [==============================] - 52s 104ms/step - loss: 0.2995 -
accuracy: 0.8622 - val_loss: 0.2058 - val_accuracy: 0.9180
Epoch 8/20
500/500 [==============================] - 52s 104ms/step - loss: 0.2733 -
accuracy: 0.8744 - val_loss: 0.1908 - val_accuracy: 0.8930
Epoch 9/20
500/500 [==============================] - 52s 104ms/step - loss: 0.2508 -
accuracy: 0.8853 - val_loss: 0.1829 - val_accuracy: 0.9010
Epoch 10/20
500/500 [==============================] - 52s 103ms/step - loss: 0.2391 -
accuracy: 0.8909 - val_loss: 0.1720 - val_accuracy: 0.9240
Epoch 11/20
500/500 [==============================] - 52s 103ms/step - loss: 0.2272 -
accuracy: 0.8985 - val_loss: 0.1484 - val_accuracy: 0.9320
Epoch 12/20
500/500 [==============================] - 52s 103ms/step - loss: 0.2192 -
accuracy: 0.9005 - val_loss: 0.1475 - val_accuracy: 0.9470
Epoch 13/20
500/500 [==============================] - 52s 103ms/step - loss: 0.2160 -
accuracy: 0.9064 - val_loss: 0.1773 - val_accuracy: 0.9030
Epoch 14/20
500/500 [==============================] - 52s 103ms/step - loss: 0.2072 -
accuracy: 0.9082 - val_loss: 0.1517 - val_accuracy: 0.9330
Epoch 15/20
500/500 [==============================] - 52s 103ms/step - loss: 0.2027 -
accuracy: 0.9104 - val_loss: 0.1407 - val_accuracy: 0.9270
Epoch 16/20
500/500 [==============================] - 52s 103ms/step - loss: 0.1964 -
accuracy: 0.9143 - val_loss: 0.1493 - val_accuracy: 0.9200
Epoch 17/20
500/500 [==============================] - 52s 103ms/step - loss: 0.1915 -
accuracy: 0.9164 - val_loss: 0.1477 - val_accuracy: 0.9330
Epoch 18/20
500/500 [==============================] - 52s 103ms/step - loss: 0.1873 -
accuracy: 0.9167 - val_loss: 0.1318 - val_accuracy: 0.9440
Epoch 19/20
500/500 [==============================] - 52s 103ms/step - loss: 0.1838 -
accuracy: 0.9191 - val_loss: 0.1240 - val_accuracy: 0.9490
Epoch 20/20
```

```
500/500 [==============================] - 52s 103ms/step - loss: 0.1761 -
accuracy: 0.9238 - val_loss: 0.1291 - val_accuracy: 0.9310
```

- Kemudian selanjutnya kita bisa visualisasikan *learning curve* dan membandingkan hasil loss training dan validasi serta keakuratan pada setiap epoch.

[ ]:
```
hist = history.history
x_arr = np.arange(len(hist['loss'])) + 1

fig = plt.figure(figsize=(12, 4))
ax = fig.add_subplot(1, 2, 1)
ax.plot(x_arr, hist['loss'], '-o', label='Train loss')
ax.plot(x_arr, hist['val_loss'], '--<', label='Validation loss')
ax.legend(fontsize=15)
ax.set_xlabel('Epoch', size=15)
ax.set_ylabel('Loss', size=15)

ax = fig.add_subplot(1, 2, 2)
ax.plot(x_arr, hist['accuracy'], '-o', label='Train acc.')
ax.plot(x_arr, hist['val_accuracy'], '--<', label='Validation acc.')
ax.legend(fontsize=15)
ax.set_xlabel('Epoch', size=15)
ax.set_ylabel('Accuracy', size=15)

#plt.savefig('figures/15_18.png', dpi=300)
plt.show()
```



Seperti terlihat pada learning curve, loss dari training dan validasi belum konvergen betul. Dengan kondisi ini, kita akan lihat berapa keakuratan yang telah kita peroleh.

[ ]:
```
ds_test = celeba_test.map(
    lambda x:preprocess(x, size=IMAGE_SIZE, mode='eval')).batch(32)
results = model.evaluate(ds_test, verbose=0)
```

```
print('Test Acc: {:.2f}%'.format(results[1]*100))
```

```
Test Acc: 95.26%
```

Berdasarkan hasil ini, kita dapat melanjutkan training untuk beberapa epoch tambahan sehingga kekakuratan akan meningkat. Kita bisa gunakan kembali metode `.fit()` untuk menlanjutkan menambahkan 10 epoch sebagai berikut.

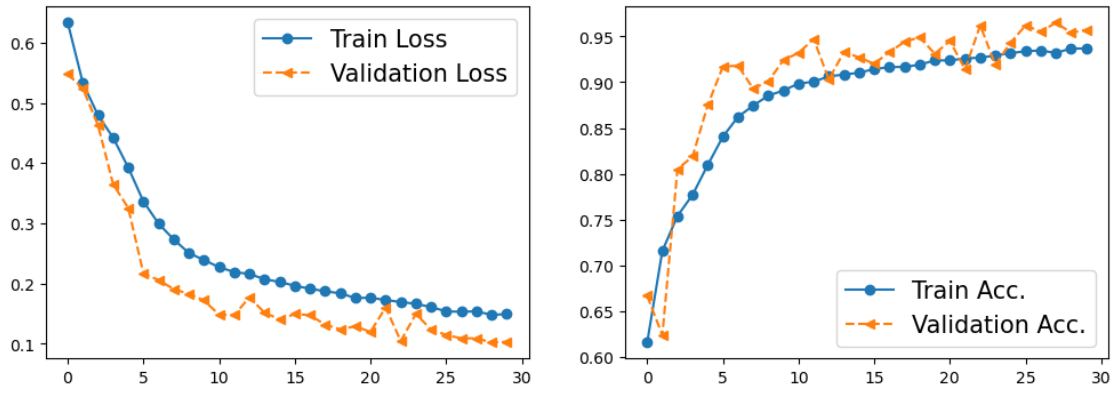
[ ]:
```
history = model.fit(ds_train, validation_data=ds_valid,
                    epochs=30, initial_epoch=20,
                    steps_per_epoch=steps_per_epoch)
```

```
Epoch 21/30
500/500 [==============================] - 52s 104ms/step - loss: 0.1765 -
accuracy: 0.9237 - val_loss: 0.1198 - val_accuracy: 0.9450
Epoch 22/30
500/500 [==============================] - 52s 104ms/step - loss: 0.1727 -
accuracy: 0.9261 - val_loss: 0.1604 - val_accuracy: 0.9140
Epoch 23/30
500/500 [==============================] - 53s 106ms/step - loss: 0.1694 -
accuracy: 0.9266 - val_loss: 0.1046 - val_accuracy: 0.9620
Epoch 24/30
500/500 [==============================] - 53s 105ms/step - loss: 0.1664 -
accuracy: 0.9292 - val_loss: 0.1496 - val_accuracy: 0.9190
Epoch 25/30
500/500 [==============================] - 54s 108ms/step - loss: 0.1610 -
accuracy: 0.9317 - val_loss: 0.1238 - val_accuracy: 0.9430
Epoch 26/30
500/500 [==============================] - 54s 107ms/step - loss: 0.1539 -
accuracy: 0.9337 - val_loss: 0.1148 - val_accuracy: 0.9620
Epoch 27/30
500/500 [==============================] - 52s 105ms/step - loss: 0.1531 -
accuracy: 0.9346 - val_loss: 0.1090 - val_accuracy: 0.9550
Epoch 28/30
500/500 [==============================] - 54s 108ms/step - loss: 0.1539 -
accuracy: 0.9318 - val_loss: 0.1092 - val_accuracy: 0.9650
Epoch 29/30
500/500 [==============================] - 53s 107ms/step - loss: 0.1480 -
accuracy: 0.9369 - val_loss: 0.1022 - val_accuracy: 0.9540
Epoch 30/30
500/500 [==============================] - 52s 104ms/step - loss: 0.1494 -
accuracy: 0.9366 - val_loss: 0.1032 - val_accuracy: 0.9570
```

- Learning curve hasil penambahan epoch dapat kita visualisasikan kembali.

[ ]:
```
hist2 = history.history
x_arr = np.arange(len(hist['loss'] + hist2['loss']))
```

```
fig = plt.figure(figsize=(12, 4))
ax = fig.add_subplot(1, 2, 1)
ax.plot(x_arr, hist['loss']+hist2['loss'],
        '-o', label='Train Loss')
ax.plot(x_arr, hist['val_loss']+hist2['val_loss'],
        '--<', label='Validation Loss')
ax.legend(fontsize=15)


ax = fig.add_subplot(1, 2, 2)
ax.plot(x_arr, hist['accuracy']+hist2['accuracy'],
        '-o', label='Train Acc.')
ax.plot(x_arr, hist['val_accuracy']+hist2['val_accuracy'],
        '--<', label='Validation Acc.')
ax.legend(fontsize=15)
plt.show()
```



- Setelah cukup puas dengan learning curve, kita bisa evaluasi model dengan dataset holdout test.

[ ]:
```
ds_test = celeba_test.map(
    lambda x:preprocess(x, size=IMAGE_SIZE, mode='eval')).batch(32)
results = model.evaluate(ds_test, verbose=0)
print('Test Acc: {:.2f}%'.format(results[1]*100))
```

```
Test Acc: 95.26%
```

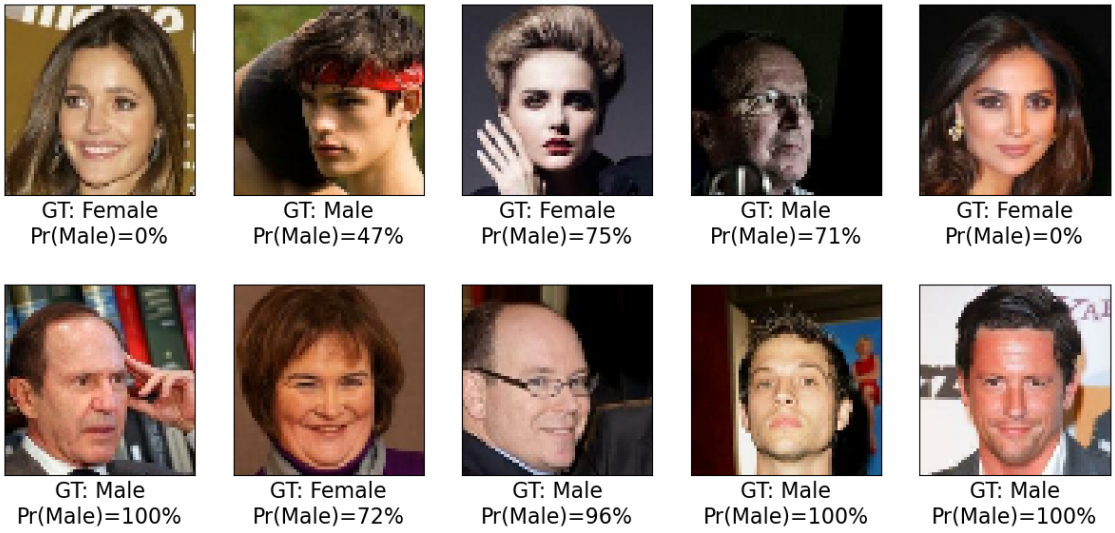
Model output yang kita gunakan adalah logits, bukan hasil probabilitas. Jika tertarik pada probabilitas *class-membership* untuk masalah klasifikasi biner dengan satu output unit ini, kita dapat gunakan fungsi `tf.sigmoid` untuk menghitung probabilitas dari *positive class* (untuk multiclass bisa digunakan `tf.math.softmax`).

- Pada program berikut akan diambil subset kecil dengan 10 sampel dari dataset test yang di-*preprocess* (`ds_test`) dan mengeksekusi `model.predict()` untuk memperoloeh logits. Ke-

mudian, kita akan menghitung probabilitas dari setiap sampel jika diasumsikan berasal dari positive class (male pada dataset `CelebA`), dan divisualisasikan 10 sampel image tersebut beserta label aktual ground truth (GT) beserta hasil prediksi hasil probabilitas. Sebagai catatan, kita gunakan `.unbatch()` pada dataset `ds_test` terlebih dahulu sebelum mengambil 10 sampel. Jika tidak dilakukan terlebih dahulu maka kita akan memperoleh 10 batch dengan ukuran sampel 32, bukan 10 sampel individual.

[ ]:
```
ds = ds_test.unbatch().take(10)

pred_logits = model.predict(ds.batch(10))
probas = tf.sigmoid(pred_logits)
probas = probas.numpy().flatten()*100

fig = plt.figure(figsize=(15, 7))
for j,example in enumerate(ds):
    ax = fig.add_subplot(2, 5, j+1)
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(example[0])
    if example[1].numpy() == 1:
        label='Male'
    else:
        label = 'Female'
    ax.text(
        0.5, -0.15,
        'GT: {:s}\nPr(Male)={:.0f}%'.format(label, probas[j]),
        size=16,
        horizontalalignment='center',
        verticalalignment='center',
        transform=ax.transAxes)

#plt.savefig('figures/figures-15_19.png', dpi=300)
plt.show()
```

```
1/1 [==============================] - 1s 1s/step
```



Kita dapat melihat 10 image beserta dengan label aktual ground truth (GT) beserta probabilitas sebagai positive class (male). Ada berapakah error yang terjadi di 10 image di atas?