ResQHub

# Machine Learning Documentation

https://github.com/ResQHub-Capstone-CH2-PS334/Machine-Learning

**Written by:**

**Raihansyah Arifin – C010BSY3731**

ResQHub Team

Part 1

# Background

*Plain*

ResQHub is an SOS safety application for everyone. This application can make a real-time call to nearby police stations or hospitals when the SOS mode is activated. The calls made to these SOS stations can be customized in the settings. By turning on the automatic call, the application will call immediately the best-recommended police stations or hospitals nearby within 3 km.

*A Hint where Machine Learning takes part*

Due to this feature, our application is prone to causing irresponsible SOS calls to nearby stations. This can lead to legal sanction for our application in the release for the reason. To solve the problem, we need to make sure that our user is authenticated and can be accountable for the calls they initiate.

On the registration page of the application, the user will be asked to fill out their NIK and full name. There are some personal data requirements as well in the process. After filling them out, the user has to take a shot of their national ID card. They will also be obliged to take a live picture of their face. The application will process the similarity score of the face in the ID and in the live. If the similarity score is high, the application will refuse to proceed. Only a similarity score below the 0.6 threshold is accepted.

*Machine Learning Outline*

The solution to calculate the similarity score of two faces is only possible with the **facial recognition** algorithm. We utilize the Celeb-A[1] dataset to train our model. In the preprocessing step, we filter the faces that satisfy some conditions to have a similar distribution to the KTP (or ID card) photo. This process is quite confronting due to the dataset taken or collected in the Celeb-A only consists of random images of diverse facial angles of celebrities across the world. Furthermore, we use the OpenCV library to get a facial landmark as our dataset. After that, each image with its corresponding landmarks will be paired. The machine learning model will receive two inputs (two images of faces) and three outputs (each face landmark embeddings and a similarity score).

*Quick conclusion*

Our machine-learning model still needs further improvement. Training the model for less than 2.5 weeks is challenging, moreover, this is about the facial recognition problem, which is not straightforward to tune and construct. We ended up having a model with a **training accuracy of 78%** (over 4416 paired images) and a **validation accuracy of 76%** (over 1472 paired images).

[1] A collection of celebrity faces across the world in a 218 x 178 resolution (for the cropped and aligned version).

Part 2:

# Setting Everything Up

*Libraries*

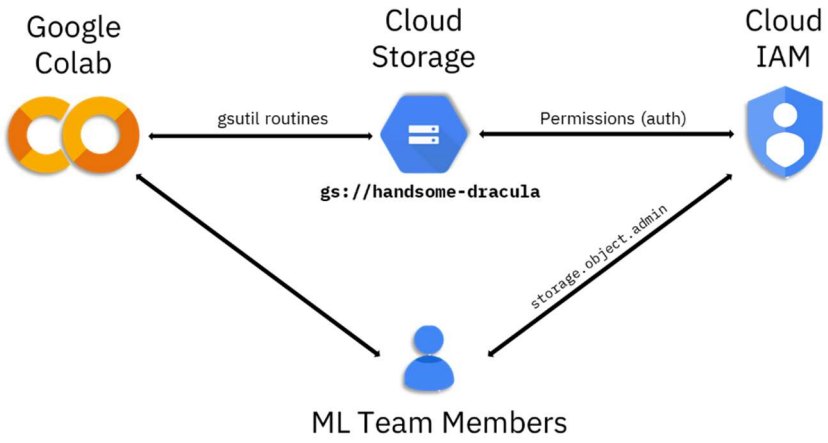
These are the libraries that we use

```
import tensorflow as tf
import pandas as pd
import math
import matplotlib.pyplot as plt
import numpy as np
import gc
import json
import os
import bz2
import cv2
import dlib
import shutil
import tensorflow.keras.layers as lyr
import tensorflow.keras.regularizers as regr
from google.colab.patches import cv2_imshow
```

Our main library is the machine learning framework. The machine learning framework that we use is TensorFlow version 2.14. On top of that, we also use pandas to ease us in querying the preprocessing stages. We also use the Google Cloud library because we want to integrate the output of each stage with the Cloud Storage service. As we have mentioned earlier, we use OpenCV to support our dataset necessity to generate 68 landmark points on a 2D plane. We also use shutil for zipping and extracting files.

*Cloud Integration*

Our preprocessed dataset, model checkpoints (weights), model training history, and dataset versioning are stored in the Cloud Storage in the `handsome-dracula` bucket. There are two Google Colabs that are connected to this Cloud Storage: First Stage Google Colab (first-stage data preprocessing)[2] and Second Stage Google Colab (second-stage data preprocessing, model training, and evaluation)[3].




[2] https://colab.research.google.com/drive/1numStv9S8URxKzatHJGDos2okCxbNKq-#scrollTo=PgBSem1IB3px
[3] https://colab.research.google.com/drive/19eEfP8c-YYz5rg27ufmJ2Ftos-VzvWiw?userstoinvite=raihansyah.arifin%40ui.ac.id&sharingaction=manageaccess&role=writer#scrollTo=wqmiYZOJ4QQ1

**Part 3:**

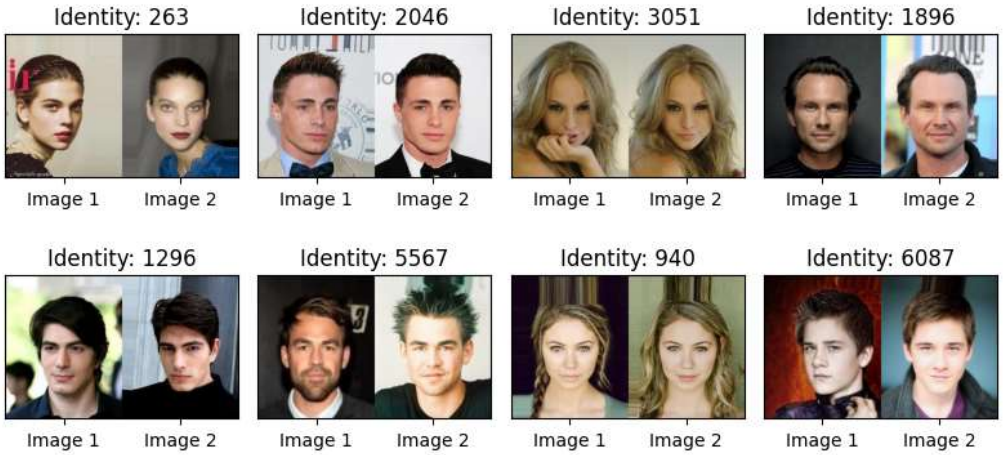
# First Stage Data Preprocessing

*Colab* (`LEGACY – MLOps 1.ipynb`)
https://colab.research.google.com/drive/1DkAf0FbrxPRDQJGRxrI_qAJ5kTo2Trrq

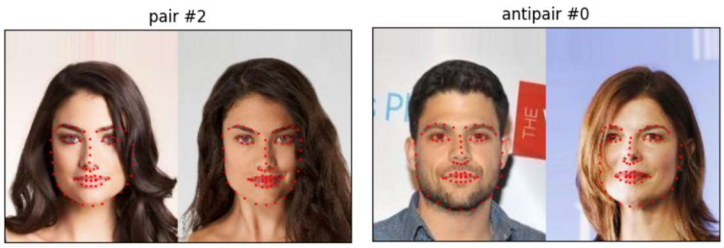
*Data Gathering*
We are using the aligned and cropped images of celebrities from Celeb-A[4]. We upload the raw dataset to the Cloud Storage along with the image labels[5] and image list of attributes[6]. Additionally, we also put the .bz2 file of the trained facial landmarks detector model to the Cloud Storage.

*Data Preprocessing*
All data is loaded using `!gsutil cp` command to copy the desired dataset stored in the Cloud Storage. Using pandas, we load the image list of attributes[6] file and filter only images that do not contain a face using these attributes: eyeglasses, bangs, hats, smiling, open mouths, and unattractive. Attractive faces are picked for the reason of image clarity and apparent strong facial features.

By loading the image labels[5] with pandas, we select identities that at least have 4 faces in the dataset. Due to the large amount of images that will be loaded into the training model, we ended up selecting a strict 4 faces per individual. After having a collection of identities and their corresponding images (4 file names listed in an array), we randomly pick and group them into two (dataset shape `(?, 2, 218, 178, 3)`)[7] with an even distribution of pairs (one identity two faces) and anti-pairs (two identities two faces). Last but not least, we are using the OpenCV facial landmarks detection model to generate 68 facial landmarks in each face. This is one of the dataset parts.




[4] https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html
[5] A CSV-like dataset that contains pairs of the image filename (column 1) and an ID number (column 2). ID number represents an individual image.
[6] A CSV-like dataset that has a binary encoding to represent what attributes exist in each face or image.
[7] This dataset shape is just a prototype. We will make a tensor dataset later in the second stage of preprocessing.




**First Stage Dataset**

The output or the result of this first stage of data preprocessing is a zip file that contains a flat namespace file system[8]. Separately, we also generate a JSON file that contains the facial landmarks of each image[9]. We save them to the Cloud Storage[10] (DEPRECATED!).

*Stage 1.5*

The dataset stored is actually still a checkpoint and does not contain a pair or non-pair image configuration. In the same preprocessing stage, we load them back again to preprocess even further. Because of this matter, we classify this stage as stage 1.5, bridging the stage 1 preprocessed dataset to the second stage of preprocessing.

The output of stage 1.5 is similar to stage 1, using a flat namespace and generating a JSON file. However, the configuration of the naming system[11] and data stored in the JSON file[12] differ. The final output of stage 1.5 is also stored in the Cloud Storage[13](DEPRECATED!). To sum up, stage 1.5 creates a dataset that has two images as features, two flattened facial landmarks of each corresponding face, and a similarity score as labels/targets. All labels are stored in the JSON file [9,12].

---

[8] For example, identity 4127 has his/her faces stored in the following naming system: 4127-0.jpeg, 4127-1.jpeg, 4127-2.jpeg, 4127-3.jpeg. Instead of using folders to group identities, we use a flat namespace. This can be beneficial when loading the dataset using `tf.keras.utils.load_imagedataset_from_directory` method.

[9] A JSON file that contains keys of image filename (ex: "4127-0.jpeg") and values of an array of rank 1 that consists of 136 numbers of landmarks (index 0 to 68 represents the *x*-axis points and 68 to 135 represents the *y*-axis points).

[10] gs://count-dracula/selected-dataset. DEPRECATED! This preprocessed dataset version will be inaccessible on January 7th, 2024. Please refer to the .tfrecord later.

[11] Still using a flat namespace naming system, but now grouped into pair and antipair. For example, identity 4127 image 0 is paired with identity 4127 image 3. These two images refer to the same person. They are grouped into the 728th pair (for instance). Hence, the naming system is pair-728-0.jpeg (for identity 4127 image 0) and pair-728-1.jpeg (for identity 4127 image 3). If two identities differ, for example identity 2583 image 0 is grouped with 7532 image 2, the naming system is antipair-944-0.jpeg (for identity 2583 image 0) and antipair-944-1.jpeg (for identity 7532 image 2) with 944 represents the 944th anti-pair in the dataset.

[12] Updating the filename to “antipair-944-1.jpeg” or so and adding one more value to the flat array of 136, making it 137. The last value or element of the array represents the similarity score in the dataset. If the face is same, the similarity score is high. If the face is different, the similarity score is low. **NOT TO BE CONFUSED** with the similarity score prediction (showing the opposite meaning).

[13] Zip file: https://storage.googleapis.com/count-dracula/selected-dataset/celeba-selected-paired-v0.zip and JSON file: https://storage.googleapis.com/count-dracula/selected-dataset/celeba-selected-paired-v0.json. DEPRECATED! This preprocessed dataset version will be inaccessible on January 7th, 2024. Please refer to the .tfrecord later.

**Part 4:**

# Second Stage Data Preprocessing and Dataset Serialization

*Colab* (LEGACY – MLOps 2.ipynb)
https://colab.research.google.com/drive/1DhE-riFYH2bnByQHmvS_8zInQvnlSDDG#scrollTo=wqmiYZOJ4QQ1

*Data Gathering*
We downloaded the dataset stored in the Cloud Storage previously[13]. Originally, the image size was 218 pixels in height and 178.0 pixels wide. We load the flat namespace files in the dataset folder[11] using `tf.keras.utils.image_dataset_from_directory` method[14].

```
__features = tf.keras.utils.image_dataset_from_directory(
  directory = f'{__stage2_dir}',
  image_size = (218, 218), #218, 178 original
  label_mode = None,
  color_mode = 'rgb',
  shuffle = False,
  batch_size = 2
)
```

The configuration above shows that we do not use labeling mode (`label_mode` set to false) as this is not a classification case. This is one of the reasons why we do not use a hierarchical file system as this method will consider folders as classes. The naming of the file in the loaded dataset is extremely crucial and cannot be shuffled (unless it is already grouped). For this reason, we set the `shuffle` to be false. The `batch_size` is set to 2 to keep intact the pairs[15]. We load all the images with RGB mode (3 channel colors).

For the label (facial landmarks and similarity score), we do the same thing. We load the JSON file and start taking data based on the filename order that `tf.keras.utils.image_dataset_from_directory` load. The same thing goes for the label, we batch them into two.

*Data Preprocessing*
Resizing the image from 218 x 178 pixels will cause a distortion in the facial landmarks points. For this reason, we need to re-map the points to fit in with the 218 x 218 pixels images. This process can be seen through this code

```
def targetMapper(x):
  resizedX = x[:, :68] * sizeWMultiplier
  resizedY = x[:, 68:-1] * sizeHMultiplier
  D = tf.expand_dims(x[:, -1], axis=1)
  return tf.concat([resizedX, resizedY, D], axis=1)

__targets = __targets.map(lambda x: targetMapper(x))
```

[14] There are another option to do this by using the ImageDataGenerator. However this method is **deprecated** in the documentation and we are advised to use `tf.keras.utils.image_dataset_from_directory` instead.
[15] `tf.keras.utils.image_dataset_from_directory` will read the images from the directory in an alphabetical order. It starts with antipair-0-0.jpeg and continues with antipair-0-1.jpeg, antipair-1-0.jpeg, antipair-1-1.jpeg, antipair-10-0.jpeg, antipair-10-1.jpeg, etc. Without batching, it will have a tensor shape of (1, 218, 218, 3) per each batch. We do not want this to happen as we want to group antipair-X-0.jpeg and antipair-X-1.jpeg together. Hence the batch size is set to 2, producing a tensor shape of (2, 218, 218, 3) per each batch.

We are leaving out the last index of the embedding as it contains the similarity score (not facial landmarks)[9]. The size multipliers are the division of the original image height or width by the target image height or width.

The facial landmarks embedding now lies in the range of (0, 0) to (218, 218). To get the best performance during the training, we need to normalize this number into range 0 and 1. To do that, the best approach is to retrieve the maximum and minimum value of each index across the entire batch. This function does the job.

```
def getMax(prev, curr):
    return tf.math.maximum(prev, curr)
def getMin(prev, curr):
    return tf.math.minimum(prev, curr)

maxEdges = (
  __targets
  .map(lambda x: x[:,:-1])
  .reduce(initial_state=0.0, reduce_func=getMax)
)
minEdges = (
  __targets
  .map(lambda x: x[:,:-1])
  .reduce(initial_state=squareSize, reduce_func=getMin)
)
```

We are going back to the images. Since `__features` is batched by 2 (shape `(?, 2, 218, 218, 3)`), we can map them using a lambda function (shape `(2, 218, 218, 3)`)[16], taking index 0 on the first rank as the face 1 tensor and taking index 1 on the first rank as the face tensor 2. We do the same thing with the labels. These lines apply this idea

```
__features0 = __features.map(lambda x: tf.cast(x[0], tf.float32) / 255.0)
__features1 = __features.map(lambda x: tf.cast(x[1], tf.float32) / 255.0)
__targets0 = __targets.map(
  lambda x: ((x[0,:-1] - minEdges[0]) / (maxEdges[0] - minEdges[0]))
)
__targets1 = __targets.map(
  lambda x: ((x[1,:-1] - minEdges[1]) / (maxEdges[1] - minEdges[1]))
)
__targetsD = __targets.map(lambda x: tf.cast(x[0, -1], tf.float32))

allX = tf.data.Dataset.zip({
  'feature0': __features0,
  'feature1':__features1
})
allY = tf.data.Dataset.zip({
  'target0': __targets0,
  'target1':__targets1,
  'targetD': __targetsD
})
allDs = tf.data.Dataset.zip({'features': allX, 'targets': allY})
```

[16] By applying .map in TensorFlow Dataset API, the result is the map of the original tensor with the lambda function or any function that is passed to it. The iteration goes by batch. Henceforth if the dataset has a shape of `(?, B, N)` with B as the batch number, only `(B, N)` is passed through the lambda function or any function that goes with it.

In total, we have 5 independent map datasets. These independent datasets are not to be shuffled as the order of the data of each dataset is still meaningful. We define `allX` as a zip dataset that concatenates two image tensors and `allY` that concatenates three labels in tensor form. Eventually, we have a unified dataset `allDs` that zip `allX` and `allX`.
One thing that is noteworthy here. The naming in the zip dataset is crucial. This dataset naming will be used in the serialization process and model fitting (training).

Now we want to split the dataset into two parts: the training dataset and the validation dataset. This is made possible with the utilization of TakeDataset and SkipDataset data types.

```
dEqu1 = 0
dEqu0 = 0
for i in __targetsD.filter(lambda x: x == 0):
  dEqu1 += 1
for i in __targetsD.filter(lambda x: x == 1):
  dEqu0 += 1

trainDsEqu0 = allDs.take(int(3 * dEqu0 / 4))
trainDsEqu1 = allDs.skip(dEqu0).take(int(3 * dEqu1 / 4))
shuffleBuffer = int(3 * dEqu0 / 4) + int(3 * dEqu1 / 4)

trainDs = (trainDsEqu0
           .concatenate(trainDsEqu1)
           .shuffle(shuffleBuffer, reshuffle_each_iteration=False)
           .batch(32))

valDsEqu0 = allDs.skip(int(3 * dEqu0 / 4)).take(int(dEqu0/4))
valDsEqu1 = allDs.skip(dEqu0).skip(int(3 * dEqu1 / 4)).take(int(dEqu1/4))
shuffleBuffer = int(dEqu0 / 4) + int(dEqu1 / 4)
valDs = (valDsEqu0
         .concatenate(valDsEqu1)
         .shuffle(shuffleBuffer, reshuffle_each_iteration=False)
         .batch(32))
```

There are some initial considerations when we want to split the dataset. We want to make sure the distribution of the labels, especially the similarity score label is equally distributed across the training set and validation set. In the code above, `dEqu1` is the total of the dataset or pair of images that are similar and otherwise for `dEqu0`[17]. We distribute 75% of the labeled-0 similarity score and 75% of the labelled-1 similarity score to the training set. On the other hand, 25% of each label of similarity score is distributed to the validation dataset. The validation dataset hence makes up a quarter part of the `allDs` dataset (1:4 val-training)

Since the `trainDs` and `valDs` dataset has been secured[18], we are safe to shuffle each dataset. The shuffle buffer[19] is set to equal the entire dataset, which means we shuffle all grouped elements. But warning, shuffling the entire grouped elements can cause memory bloat[20]. A `reshuffle_each_iteration` is set to false to prevent reshuffling every time the dataset is called.

[17] It measures how many values of *D* equals 0 or 1.
[18] Feature 0 (representing image 0), Feature 1 (representing image 1), facial landmark 0, facial landmark 1, and similarity score have been grouped and locked together, enabling it to be intact even when it gets shuffled.
[19] Group elements of the dataset to be shuffled.
[20] TensorFlow Dataset API does not read the entire dataset eagerly in the first place. When shuffling is applied, the API has to read each grouped element, loading them into the memory.

It must be emphasized that the batching in the early process does not represent batched grouped elements. We use the batch size of 32, meaning that there are 32 pairs of images with the corresponding labels in it per batch.

*Serialization*

We have `trainDs` and `valDs` in hand, ready to use. However, we might encounter a runtime crash[21] during the ML process. This can cause the dataset that we have loaded to be lost. We can reload them again, however, we are going through the shuffling process again. This process can take time and memory bloat, moreover, different shuffling results that could impact the model accuracy consistency during the training. The aim of the serialization process is to save our dataset model in TFRecord format (binary).

```
def makeTFRecord(dataset, filename):
  with tf.io.TFRecordWriter(filename) as f:
    for i in dataset:
      feature0 = tf.io.serialize_tensor(i['features']['feature0'])
      feature1 = tf.io.serialize_tensor(i['features']['feature1'])
      target0 = tf.io.serialize_tensor(i['targets']['target0'])
      target1 = tf.io.serialize_tensor(i['targets']['target1'])
      targetD = tf.io.serialize_tensor(i['targets']['targetD'])

      dataFeature = {
        'feature0': tf.train.Feature(
          bytes_list = tf.train.BytesList(value=[feature0.numpy()])
        ),
        'feature1': tf.train.Feature(
          bytes_list = tf.train.BytesList(value=[feature1.numpy()])
        ),
        'target0': tf.train.Feature(
          bytes_list = tf.train.BytesList(value=[target0.numpy()])
        ),
        'target1': tf.train.Feature(
          bytes_list = tf.train.BytesList(value=[target1.numpy()])
        ),
        'targetD': tf.train.Feature(
          bytes_list = tf.train.BytesList(value=[targetD.numpy()])
        )
      }
      collective = tf.train.Features(feature=dataFeature)
      record_bytes = tf.train.Example(features=collective).SerializeToString()

      f.write(record_bytes)
```

We are not going through the details of the code snippet above. The point is that these lines of code are doing a serialization process (`tf.io.serialize_tensor`), converting tensor data (in our case `tf.float32`) into a binary form. Each of the serialized tensors contains the binaries and gets passed to the `tf.train.BytesList` method. This method, working together with `tf.train.Feature`, is creating a binary feature space. In our case, we have five binary feature spaces, two represent the

[21] Unprecedented occurrence such as runtime error that terminates the ML session. In Colab, for instance, even with Colab Pro, we sometimes encounter an unexpected runtime termination due to the high usage of certain VMs. This could cause the loss of all the progress that has been made. Hence, we want to save the dataset that we have processed into storage. The other reason is that we want to ensure that the data is not reshuffled when it is loaded, meaning that what we have shuffled remains intact.

feature spaces for feature data and three represent the feature spaces for label data[22]. Each `Feature` is grouped inside `Features`. A `Features` is passed to the `tf.train.Example` to be serialized with `SerializeToString()` method. This way, we are generating a binary file for the dataset[23].

We verge on the end of the preprocessing process. The last thing we do in the preprocessing process is to upload the dataset into Cloud Storage[23] to be used later.

*We terminated our runtime before proceeding to the training process.*[24]

[22] Notice that we have `tf.train.Feature` and `tf.train.Features` (with S). `Feature` contains a list of a certain datatype. The list that is incorporated is flat (rank 1). In other words, Feature can be thought of as a list. On the contrary, `Feataues` contains `Feature`, a collection of lists labeled with a certain key per each. `Features` can be imagined as a dictionary.

[23] https://storage.cloud.google.com/handsome-dracula/machine-learning/dataset-tfrecord/trainDs.tfrecord (train binary dataset). https://storage.cloud.google.com/handsome-dracula/machine-learning/dataset-tfrecord/trainDs.tfrecord (validation binary dataset)

[24] The memory bloat from the shuffling process is still present and it is arduous to drop down the amount of garbage memory. Hence, to ensure training safety (providing against unprecedented termination during training), we terminate our current runtime to get a fresh start.

Part 5:

# Deserialization, Model Design, Model Training, and Evaluation

*Data Gathering: Data Deserialization*

We load the data in the Cloud Storage[23] using the `!gsutil` command. The data loaded is a serialized tensor in a binary form of our preprocessed dataset. The following method that we create does the job.

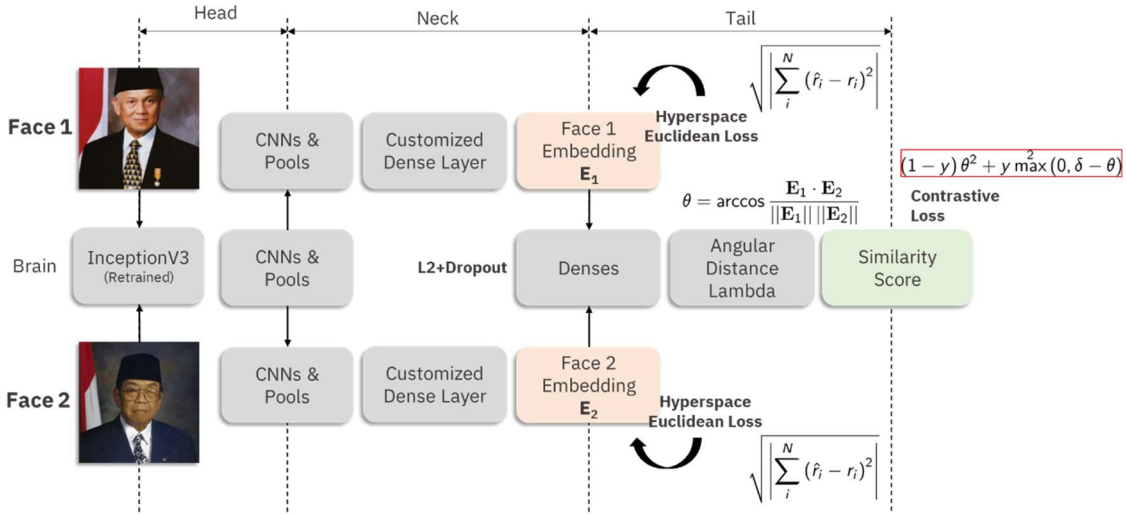
```
def readTFRecord(filename):
  features = {
  'feature0': tf.io.FixedLenFeature([], tf.string),
  'feature1': tf.io.FixedLenFeature([], tf.string),
  'target0': tf.io.FixedLenFeature([], tf.string),
  'target1': tf.io.FixedLenFeature([], tf.string),
  'targetD': tf.io.FixedLenFeature([], tf.string),
  }

  def decoder(n):
    singleParser = tf.io.parse_single_example(n, features)
    feature0 = tf.io.parse_tensor(singleParser['feature0'], tf.float32)
    feature1 = tf.io.parse_tensor(singleParser['feature1'], tf.float32)
    target0 = tf.io.parse_tensor(singleParser['target0'], tf.float32)
    target1 = tf.io.parse_tensor(singleParser['target1'], tf.float32)
    targetD = tf.io.parse_tensor(singleParser['targetD'], tf.float32)
    dataStructure = {
      'features': {
        'feature0': feature0,
        'feature1': feature1
      },
      'targets': {
        'target0': target0,
        'target1': target1,
        'targetD': targetD
      }
    }
    #return ((feature0, feature1), (target0, target1, targetD))
    return (dataStructure)

  return tf.data.TFRecordDataset(filename).map(decoder)

trainDs = readTFRecord('trainDs.tfrecord').prefetch(tf.data.AUTOTUNE)
valDs = readTFRecord('valDs.tfrecord').prefetch(tf.data.AUTOTUNE)
```

We read the recorded TFRecord using `tf.data` API, passing the record file to the `TFRecordDataset` method. The `decoder` is the mapping function according to the `features`[22] that is created earlier. The last two lines are the ready-to-use PrefetchDataset. At this point, we have done dealing with the dataset.

*Modeling*

Our modeling idea is to create a neural network model that takes two images as an input and produces three outputs as we designed the dataset to be so. A regular conventional sequential model cannot do this. This type of model needs special care by using the Functional API. Essentially, a sequential model

is part of the Functional API, yet it is limited to only creating a linear model (one input one output). In our case, we use the Functional API to merge two inputs together into the same neural network model[25]. This type of model is also called **Siamese Neural Network**.




We divide our model into four sections: brain, head, neck, and tail. The brain is the transfer learning model. We use the InceptionV3-trained model[26] as our transfer learning, clipping from the first CNN layer down to the `mixed5` layer. To do the fine-tuning process, we do not freeze the layers. The second part is the head. The output of `mixed5` is then carried along to our CNN and Max/Average Pooling layers. The last CNN-Pooling layer and the Flatten layer are the beginning parts of the neck section, connected all the way down to the first and second output (each face embedding)[27]. In the neck section, we define our customized dense layer, named NormedDense.

```
@tf.keras.saving.register_keras_serializable(
  package="resqhubModel",
  name="NormedDense"
)
class NormedDense(tf.keras.layers.Layer):
  def __init__(self, units=32, **kwargs):
    super(NormedDense, self).__init__(**kwargs)
    self.units = units

  def build(self, input_shape):
    __init_w = tf.random_uniform_initializer()
    __init_b = tf.random_uniform_initializer()

    self.w = tf.Variable(
      name = 'weights',
      initial_value = __init_w(
        shape = (input_shape[-1], self.units),
        dtype = 'float32'
      ),
      trainable = True
    )
    self.b = tf.Variable(
```

[25] It is not actually merged together (merged in a metaphoric way). We create two neural network models that share the same weights and biases, perfectly identical. In reality, it must be emphasized that they are not merged literally.
[26] This might be not the best choice for face recognition. We should have picked up ResNet, for instance. InceptionV3 is only good at classification cases (sparse (categorical / non categorial) cross entropy), not contrastive.
[27] Each embedding output shape is `(?, 136)`.

```
      name = 'bias',
      initial_value = __init_b(
        shape = (self.units,),
        dtype='float32'
      ),
      trainable = True
    )

  def get_config(self):
    config = super().get_config()
    config.update({
        'units': self.units
    })
    return config

  def call(self, inputs):
    matrix = tf.linalg.matmul(inputs, self.w) + self.b
    return matrix
```

However, using the NormedDense layer, disables us to predict a single image during the application. The reason behind this is that the operation that we use in NormedDense[28] is initially:

```
  # !deprecated
  def call(self, inputs):
    wx = tf.linalg.matmul(inputs, self.w)
    wxMax = tf.reduce_max(wx, axis=1, keepdims=True)
    wxMin = tf.reduce_min(wx, axis=1, keepdims=True)
    wxNormed = (wx - wxMin)/(wxMax - wxMin)
    matrix = wxNormed + self.b
    return matrix
```

This causes a NaN (undefined behavior) output when we pass one data only as the divider goes to zero[29]. A defined behavior will appear if the data passed to the model is more than one. This is not our case, hence we revoke this method and use the regular expression of dense layer operation but with a different initializer.

The output of this NormedDense is matrices (if using batch) or a vector (for singular data)[30] of each facial embedding (shape `(?, 136)`)

The last section of our model is the tail. The embedding input of each face gets merged into a new Siamese Neural Network using regular dense layers. In this process, at the end tip of the model, we put a lambda layer that calculates the hyperspace angular distance of two hyperspace vectors. This can be calculated with the following equation

$$\theta = \arccos \frac{\mathbf{E}_1 \cdot \mathbf{E}_2}{\|\mathbf{E}_1\| \, \|\mathbf{E}_2\|} \tag{1.1}$$

With $\mathbf{E}_1$ as the landmark embedding for image 1 and $\mathbf{E}_2$ as the landmark embedding for image 2. The value of $\theta$ is the similarity score that we are talking about.

---

[28] This customized layer creates a normalized regular dense layer matrix output. The normalization process is done through each feature point across the batch (columns), creating a range of numbers from 0 to 1.
[29] `wMax` equals `wMin` for a single data.
[30] A vector of rank 2, shape = `(1, 136)`.

The landmark embedding output is trained with a hyperspace Euclidean distance (or in simpler terms, a vector distance for higher dimensions). Hence the loss can be formulated as following

$$L_E(\hat{\mathbf{r}}, \mathbf{r}) = \sqrt{\left|\sum_i^N (\hat{r}_i - r_i)^2\right|}, (\hat{\mathbf{r}}, \mathbf{r}) \in \mathrm{R}^{136} \tag{1.2}$$

With $\hat{\mathbf{r}}$ is the predicted landmark embedding and $\mathbf{r}$ is the dataset landmark embedding. As the $\mathbf{E}_1$ and $\mathbf{E}_2$ produced and merged into the second Siamese NN (the tail) and the extracted feature embedding space has been weighed using equation 1.1 (a scalar[31] for singular data and a matrix for batch data), we use a **contrastive loss function** to calculate each loss with the following equation.

$$L_C(\boldsymbol{\theta}, \mathbf{y}) = (1 - \mathbf{y})\boldsymbol{\theta}^2 + \mathbf{y}(\max(0, \delta - \boldsymbol{\theta}))^2 \tag{1.3}$$

With $\boldsymbol{\theta}$ is the collection of hyperspace angular distance calculated throughout the dataset, $\mathbf{y}$ being the true value for similarity score, and $\delta$ as the margin. Since we have three outputs in our model, we distribute each loss with a certain portion,

$$\begin{aligned} L(\boldsymbol{\theta}, \mathbf{y}, \mathbf{E}_1, \mathbf{E}_2) &= \alpha L_E(\widehat{\mathbf{E}}_1, \mathbf{E}_1) + \beta L_E(\widehat{\mathbf{E}}_2, \mathbf{E}_2) + \gamma L_C(\boldsymbol{\theta}, \mathbf{y}) \\ &= \alpha \sqrt{\left|\sum_i^N \left(\widehat{E_i^{(1)}} - E_i^{(1)}\right)^2\right|} + \beta \sqrt{\left|\sum_i^N \left(\widehat{E_i^{(2)}} - E_i^{(2)}\right)^2\right|} + (1 - \mathbf{y})\boldsymbol{\theta}^2 + \mathbf{y}(\max(0, \delta - \boldsymbol{\theta}))^2 \end{aligned} \tag{1.4}$$

The value of $\alpha$, $\beta$, and $\gamma$ can be tuned in the training process. These are the lines to show the full code that represents equations 1.1 – 1.4.

```
alpha, beta, gamma = 0.5, 0.5, 1
tf.random.set_seed(2356)
@tf.keras.saving.register_keras_serializable(
  package="resqhubModel",
  name="Loss2"
)
class Loss2(tf.keras.losses.Loss):
  def __init__(self, delta=1):
    super().__init__()
    self.delta = delta

  def call (self, y_true, y_pred):
    a2 = y_true * tf.math.square(y_pred) #? x 1
    b2 = (1 - y_true) * tf.math.square(
        tf.math.maximum(self.delta - y_pred, 0)
    )
    c2 = a2 + b2
    return c2 * gamma

@tf.keras.saving.register_keras_serializable(
  package="resqhubModel",
  name="Loss1"
)
```

[31] A scalar of tensor rank 2, shape = `(1,1)`.

```
class Loss1(tf.keras.losses.Loss):
  def __init__(self):
    super().__init__()

  def call (self, y_true, y_pred):
    a1 = y_true # ? x 136
    b1 = y_pred # ? x 136
    c1 = tf.math.sqrt(
      tf.math.reduce_sum(
        tf.math.square(a1 - b1),
        axis = -1,
        keepdims = True),
    ) / tf.cast(tf.shape(y_pred)[0], dtype=tf.float32)
    return c1 * beta

@tf.keras.saving.register_keras_serializable(
  package="resqhubModel",
  name="Loss0"
)
class Loss0(tf.keras.losses.Loss):
  def __init__(self):
    super().__init__()

  def call (self, y_true, y_pred):
    a0 = y_true # ? x 136
    b0 = y_pred # ? x 136
    c0 = tf.math.sqrt(
      tf.math.reduce_sum(
        tf.math.square(a0 - b0),
        axis = -1,
        keepdims = True),
    ) / tf.cast(tf.shape(y_pred)[0], dtype=tf.float32)
    return c0 * alpha

@tf.keras.saving.register_keras_serializable(
  package="resqhubModel",
  name="angularDist"
)
def angularDist(vects):
  r0, r1 = vects
  dotProduct = tf.math.reduce_sum(r0 * r1, axis=-1) + tf.keras.backend.epsilon()
  lengthR0 = tf.math.sqrt(tf.math.reduce_sum(tf.math.square(r0), axis=-1)) +
tf.keras.backend.epsilon()
  lengthR1 = tf.math.sqrt(tf.math.reduce_sum(tf.math.square(r1), axis=-1)) +
tf.keras.backend.epsilon()
  return tf.math.acos(dotProduct/(lengthR0 * lengthR1))

@tf.keras.saving.register_keras_serializable(
  package="resqhubModel",
  name="lambdaNormalizer"
)
def lambdaNormalizer(max=1, min=-1):
  def do(x):
    return x / 218
  return do
```

*Model Training*

We use the customized loss function as shown in equation 1.4. We also use a customized accuracy metric to satisfy the contrastive loss accuracy with the following lines.

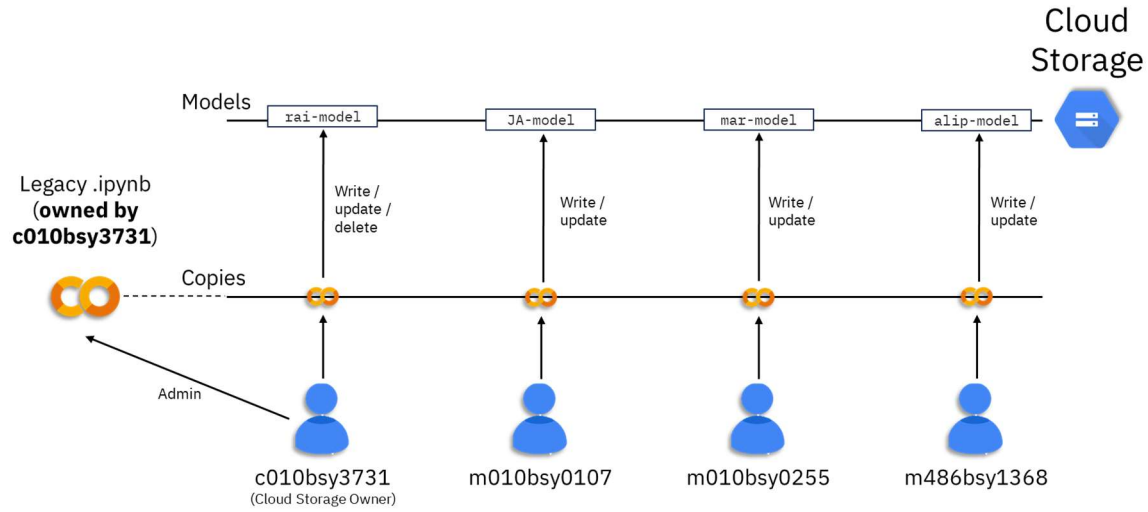
```
def pairwiseAcc(y_true, y_pred):
  threshold = 0.5
  y_true = tf.cast(y_true, dtype=tf.float32)
  y_pred_binary = tf.cast(tf.math.less(y_pred, threshold), dtype=tf.float32)
  total = tf.shape(y_true)[0]
  validity = tf.shape(tf.where(y_true == y_pred_binary))[0] / total
  return validity
```

For model training

```
def trainModel(model, lr=3e-6):
  model.compile(
    loss = [Loss0(), Loss1(), Loss2()],
    optimizer = tf.keras.optimizers.RMSprop(learning_rate=lr),
    metrics={'targetD': [pairwiseAcc], 'target0': [], 'target1': []}
  )

  hist = model.fit(
      trainDs.map(lambda x: (x['features'], x['targets'])),
      epochs = 20,
      validation_data = valDs.map(lambda x: (x['features'], x['targets'])),
      shuffle = False
  )
  return hist
```

There is some note-taking here. Since we do not use or feed a raw tensor directly to the fitting method (`tf.keras.Model.fit`), we need to map our dataset to tell the fitting method which part is the feature and which part is the target[32]. We use the regular RMSprop optimizers with a learning rate of 3e-6. We do not reshuffle the batch by setting the parameter `shuffle` to false.

*Training Approach*




[32] This process can be automatically detected by `tf.keras.Model.fit` method when passing a dataset from `tf.data.Dataset` API. For example when we use `tf.data.Dataset.from_tensor_slices(x,y)`, the tensor `x` is automatically assigned as feature and `y` the target. However, upon passing a dictionary tensor, we must specify the value name of the dictionary.

Our Machine Learning model case is arduous to train with good accuracy. For the sake of time management, we use an ensemble training approach[33]. These are the members that participate in the model training: c010bsy3731, m010bsy0107, m010bsy0255, m486bsy1368.

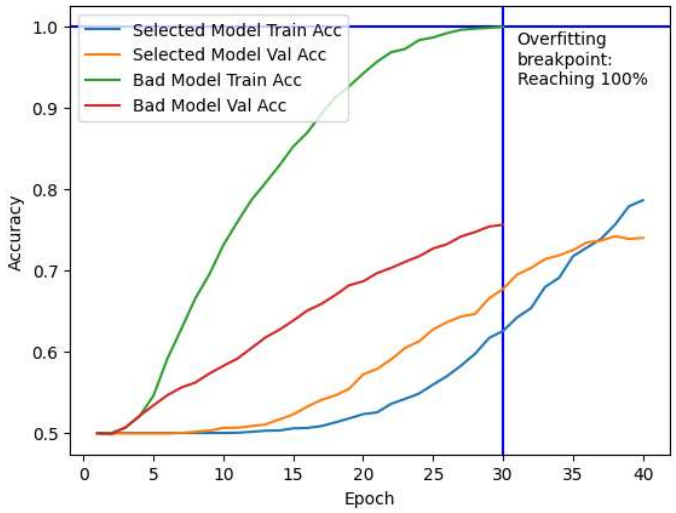
As the picture shows above, the legacy Colab file (owned by c010bsy3731) is distributed across the members as a copy (changes they made did not affect the legacy). Each member is granted to write and update their model access to Cloud Storage by c010bsy3731[34].

During the training process, all of the team members use the legacy model[35]. Each team member is tuning the parameters of dropout layers, L2 regularization $\lambda$, making small changes in the CNN filter number or the units of the dense layers, and tuning the value of the value of $\alpha$, $\beta$, and $\gamma$ in the loss distribution.

In the end, we use an ensemble approach to incorporate our models:

$$M_{ensemble} = 0.7M_{ja} + 0.1M_{mar} + 0.1M_{alip} + 0.1M_{rai} \tag{1.5}$$

The reason why we put different weights across the trained model is because the model shows the most significant ideal model compared to the others (overfit).




The selected model is the JA-model and the Bad model is the combination of the other models. For JA-model model, it passes a **training accuracy of 78%** (over 4416 paired images) and a **validation accuracy of 76%** (over 1472 paired images). The other models stuck between 74% validation accuracy while training accuracy keeps creeping up to 100%.

[33] Where the model is trained by each individuals.
[34] Here is the link to our model checkpoints and versioning: https://console.cloud.google.com/storage/browser/handsome-dracula/machine-learning/model-checkpoints?pageState=(%22StorageObjectListTable%22:(%22f%22:%22%255B%255D%22))&project=resqhub-capstone&prefix=&forceOnObjectsSortingFiltering=false
[35] The model that has been constructed in the legacy Colab/.ipynb file.