# COMPILING MODEL

**Softwares Required:**

- Web browser
- Google Colab

**Steps:**

1. Save all the necessary files to Google Drive.
2. Open the Google Colab website
3. Choose the notebook with the code for compiling the model.
4. Connect to a TPU enabled runtime
5. Run the following code to import necessary packages

```
import tensorflow
from tensorflow.keras.layers import Input, BatchNormalization, ReLU, Conv2D, Conv1D, Flatten, Dense, MaxPool2D, AvgPool2D, GlobalAvgPool2D, GlobalAveragePooling1D, Concatenate, Reshap
from tensorflow.keras.models import Model
import tensorflow.keras.backend as K
import tensorflow as tf
import random
from tensorflow import keras
import time
import os
import numpy as np
import cv2
```

6. Run the code snippets to create a function that builds cffn and returns it

```
[ ] def bn_relu_conv(x, filters, kernel_size):
      x = BatchNormalization()(x)
      x = ReLU()(x)
      x = Conv2D(filters, kernel_size, padding='same')(x)
      return x
```

```
[ ] def dense_block(tensor, k, reps):
      for _ in range(reps):
        x = bn_relu_conv(tensor, k*4, 1)
        x = bn_relu_conv(x, k, 3)
        tensor = Concatenate()([tensor, x])
      return tensor
```

```
[ ] def transition_layer(x, theta):
      f = int(tensorflow.keras.backend.int_shape(x)[-1]*theta)
      x = bn_relu_conv(x, f, 1)
      x = AvgPool2D(2, strides = 2, padding= 'same')(x)
      return x
```

```
[ ] def dense_unit(x, k, num, channels, theta):
      for i in range(num-1):
        x = dense_block(x, k, channels)
        x = transition_layer(x, theta)

      x = dense_block(x, k, channels)

      return x
```

```python
def cffn():
  channels = [24, 30]
  num = [2, 3]
  k = 24
  theta = 0.5

  input = Input(shape = (64, 64, 3))
  x = Conv2D(48, 7, strides = 4, padding = 'same')(input)
  x = dense_unit(x, k, num[0], channels[0], theta)

  x = dense_unit(x, k, num[1], channels[1], theta)

  x = Dense(128)(x)
  output = Flatten()(x)

  return Model(input, output)
```

7. Run the code snipped to create a function that calculates Euclidean distance between the 2 vectors given as input

```python
def euclidean_distance(outputs):
    # unpack the vectors into separate lists
    (featsA, featsB) = outputs
    # compute the sum of squared distances between the vectors
    sumSquared = K.sum(K.square(featsA - featsB), axis=1, keepdims=True)
    # return the euclidean distance between the vectors
    return K.sqrt(K.maximum(sumSquared, K.epsilon()))
```

8. Run the code snippet that creates a function that implements contrastive loss

```python
def contrastive_loss(y, preds, margin=0.5):
    # explicitly cast the true class label data type to the predicted
    # class label data type (otherwise we run the risk of having two
    # separate data types, causing TensorFlow to error out)
    y = tf.cast(y, preds.dtype)
    # calculate the contrastive loss between the true labels and
    # the predicted labels
    squaredPreds = K.square(preds)
    squaredMargin = K.square(K.maximum(margin - preds, 0))
    loss = K.mean(y * squaredPreds + (1 - y) * squaredMargin)
    # return the computed contrastive loss to the calling function
    return loss
```

9. Run the code snippet that builds the Siamese network

```
[ ]  imgA = Input(shape=(64, 64, 3))
     imgB = Input(shape=(64, 64, 3))
     featureExtractor = cffn()
     featsA = featureExtractor(imgA)
     featsB = featureExtractor(imgB)
     # finally, construct the siamese network
     distance = Lambda(euclidean_distance)([featsA, featsB])
     model = Model(inputs=[imgA, imgB], outputs=distance)

     model.compile(loss=contrastive_loss, optimizer="adam", metrics = ['accuracy'])
```

10. Run the code snippets that extract the rar files containing real and fake images. The directories can be changed as desired

```
[ ]  pip install patool
```

```
[ ]  import patoolib
     patoolib.extract_archive("/content/drive/MyDrive/fake.rar", outdir="/content/sample_data/data")
```

```
[ ]  patoolib.extract_archive("/content/drive/MyDrive/real.rar", outdir="/content/sample_data/data")
```

11. Run the code snippet to create a function that generates pairs for the given images array and label array

```python
def make_pairs(images, labels):
    # initialize two empty lists to hold the (image, image) pairs and
    # labels to indicate if a pair is positive or negative
    pairImages = []
    pairLabels = []

    # calculate the total number of classes present in the dataset
    # and then build a list of indexes for each class label that
    # provides the indexes for all examples with a given label
    numClasses = len(np.unique(labels))
    idx = [np.where(labels == i)[0] for i in range(0, numClasses)]
    #print(idx)
        # loop over all images
    for idxA in range(len(images)):
        # grab the current image and label belonging to the current
        # iteration
        currentImage = images[idxA]
        label = labels[idxA]
        # randomly pick an image that belongs to the *same* class
        # label
        idxB = np.random.choice(idx[label])
        posImage = images[idxB]
        # prepare a positive pair and update the images and labels
        # lists, respectively
        pairImages.append([currentImage, posImage])
        pairLabels.append([1])
        # grab the indices for each of the class labels *not* equal to
        # the current label and randomly pick an image corresponding
        # to a label *not* equal to the current label
        #negIdx = np.where(labels != label)[0]
        negIdx = np.where(labels != label)[0]
        #print(negIdx)
        negImage = images[np.random.choice(negIdx)]
        #print(negImage)
        # prepare a negative pair of images and update our lists
        pairImages.append([currentImage, negImage])
        pairLabels.append([0])
    # return a 2-tuple of our image pairs and labels
    return (np.array(pairImages), np.array(pairLabels))
```

12. Run the code snippet that generates the custom ImageDataGenerator class for training the model. This class will dynamically load the images from the directory where the data is. Change the 'batch' value as required.

```python
batch = 16

idg1 = tf.keras.preprocessing.image.ImageDataGenerator()

gen1 = idg1.flow_from_directory('/content/sample_data/data',
                                shuffle=True,
                                class_mode='binary', target_size = (64, 64), batch_size = batch)


class JoinedGen(tf.keras.utils.Sequence):
    def __init__(self, input_gen1):
        self.gen1 = input_gen1

    def __len__(self):
        return len(self.gen1)

    def __getitem__(self, i):
        pairs, labels = make_pairs(gen1[i][0], gen1[i][1].astype(int))

        return [pairs[:, 0], pairs[:, 1]], labels

    def on_epoch_end(self):
        self.gen1.on_epoch_end()
```

13. Run the code snippet that calls the model.fit() function to train the model.

```
[ ] tf.config.run_functions_eagerly(True)

    mygen = JoinedGen(gen1)

    model.fit(mygen, epochs = 2, steps_per_epoch=110415//batch, use_multiprocessing = True, workers = 16)
```

14. Finally, run the code snippet that saves the model. The output directory can be changed

```
[ ] featureExtractor.save('/content/drive/MyDrive/featureExtractor')
```

15. The model is saved in the 'featureExtractor' folder or the folder specified. This folder must be downloaded and must be given as input for the load_model() function during the execution phase for importing the model.