In [1]:
```
def create_model():
    inputs = Input((64, 64, 1))
    x = Conv2D(96, (11, 11), padding="same", activation="relu")(inputs)
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = Dropout(0.3)(x)

    x = Conv2D(256, (5, 5), padding="same", activation="relu")(x)
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = Dropout(0.3)(x)

    x = Conv2D(384, (3, 3), padding="same", activation="relu")(x)
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = Dropout(0.3)(x)

    pooledOutput = GlobalAveragePooling2D()(x)
    pooledOutput = Dense(1024)(pooledOutput)
    outputs = Dense(128)(pooledOutput)

    model = Model(inputs, outputs)
    return model
```

The network takes images of shape 64x64. Then there are three batches of Conv-Pool-Dropout present in the network . The network ends with the 128 node Fully Connected Embedding Layer

In [2]:
```
from keras.layers import Input, Conv2D, MaxPooling2D, Dropout, GlobalAveragePool
from keras.models import Model
feature_extractor = create_model()
imgA = Input(shape=(64, 64, 1))
imgB = Input(shape=(64, 64, 1))
featA = feature_extractor(imgA)
featB = feature_extractor(imgB)
```

The fact that the network uses the same structure twice with two different images, it actually can be achieved with a single instance of the network. With this the parameter updating also becomes easier, as the weights and the biases will be updated in the same instance only. Two images are provided to the network, and the network produces the embedding layers or the features, hence the network also acts as a Feature Extractor.

In [3]:
```
from keras.layers import Lambda
from keras import backend as k
def euclidean_distance(vectors):
    (featA, featB) = vectors
    sum_squared = k.sum(k.square(featA - featB), axis=1, keepdims=True)
    return k.sqrt(k.maximum(sum_squared, k.epsilon()))

distance = Lambda(euclidean_distance)([featA, featB])
outputs = Dense(1, activation="sigmoid")(distance)
model = Model(inputs=[imgA, imgB], outputs=outputs)
```

The Euclidean distance is calculated by finding out the square root of the sum of the squares of the difference of both the embeddings. Lambda API is used from TensorFlow Layers for this purpose. The distance value is adjusted to a range of 0–1 using Sigmoid.

In [4]:
```
model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"]
```

Loss function binary cross entropy is used

For my Face Recognition system, I'm using the Olivetti dataset fetched from sklearn datasets API. It has a total of 400 face images for 40 people with 10 images per person.

Fig. Top 2 rows have a set of 10 faces of the same person. Bottom 2 rows have faces of 10 different people shown. Pictures taken from Olivetti Faces from sklearn.datasets API. Image by Girija Behera

It has ids as the labels for the images. The Olivetti face dataset has the following features.

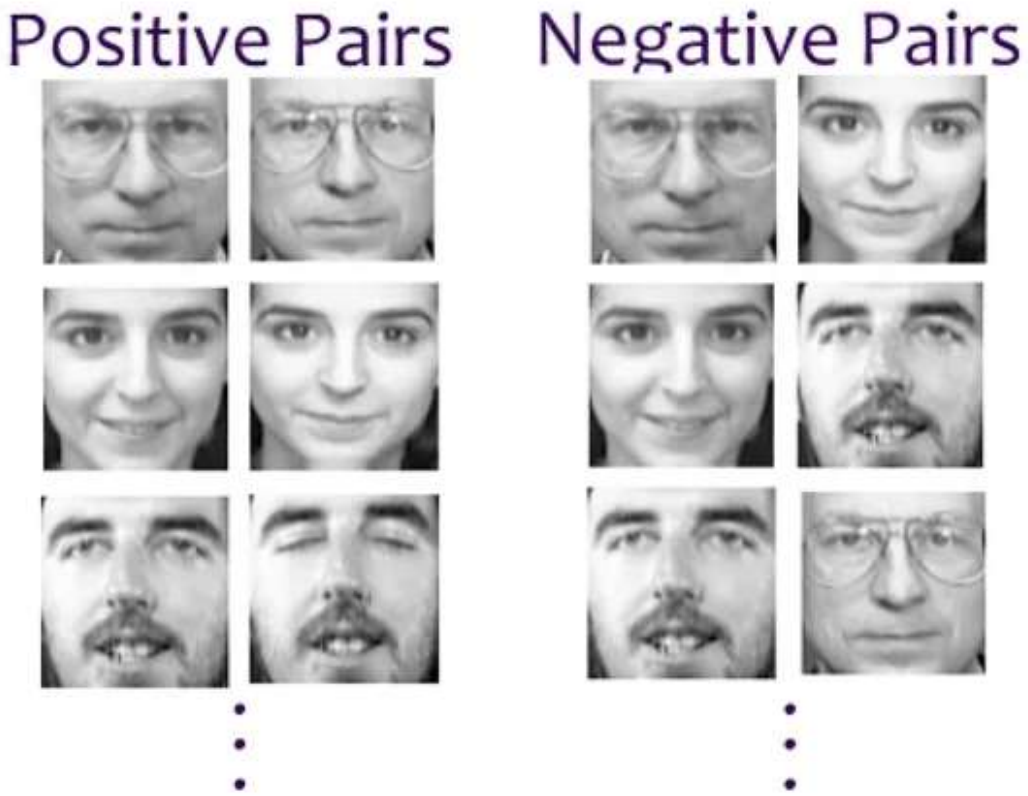
All the images have only the faces cropped in, even the ears have been cut out. The images are gray scaled. And it seems the contrast and the brightness are adjusted in them. A person has around 10 images, each one with possibly a different face expression.

Unlike a Regular CNN, here we don't generate one image at a time, rather we generate a pair of images from the dataset.

In [5]:
```
#siamese_train_image_pairs
def generate_train_image_pairs(images_dataset, labels_dataset):
    unique_labels = np.unique(labels_dataset)
    label_wise_indices = dict()
    for label in unique_labels:
        label_wise_indices.setdefault(label,
                                      [index for index, curr_label in enumerate(
                                       label == curr_label])

    pair_images = []
    pair_labels = []
    for index, image in enumerate(images_dataset):
```

```
        pos_indices = label_wise_indices.get(labels_dataset[index])
        pos_image = images_dataset[np.random.choice(pos_indices)]
        pair_images.append((image, pos_image))
        pair_labels.append(1)

        neg_indices = np.where(labels_dataset != labels_dataset[index])
        neg_image = images_dataset[np.random.choice(neg_indices[0])]
        pair_images.append((image, neg_image))
        pair_labels.append(0)
    return np.array(pair_images), np.array(pair_labels)
```

An image can be paired up with another image of the same label making a positive pair, or with another image of a different label making a negative pair. The code above starts with collecting the indices for each label. Then it iterates over the images dataset, and pairs up each image with a random image of the same label as a positive pair, and a random image of any other label as a negative pair.



In summary, the function takes two input datasets (images_dataset and labels_dataset) and generates pairs of images along with corresponding labels for training a Siamese network. Positive pairs consist of images with the same label, and negative pairs consist of images with different labels.

This is some sample image pairs generated from the method above. The method generates two image pairs for each image in the dataset. i.e. for each image it generates a positive pair and a negative pair. Hence, a total of 800 image pairs will be generated and used for Model training.

In [13]:
```
from sklearn.datasets import fetch_olivetti_faces
import numpy as np
```

```
# Load the Olivetti faces dataset
data = fetch_olivetti_faces()
images_dataset = data.images
labels_dataset = data.target
```

```
downloading Olivetti faces from https://ndownloader.figshare.com/files/5976027
to C:\Users\Hp\scikit_learn_data
```

In [14]:

```
#Model training
images_pair, labels_pair = generate_train_image_pairs(images_dataset, labels_dat
history = model.fit([images_pair[:, 0], images_pair[:, 1]], labels_pair[:],valid
```

```
Epoch 1/100
12/12 [==============================] - 29s 2s/step - loss: 0.6539 - accuracy:
0.5000 - val_loss: 0.5692 - val_accuracy: 0.5500
Epoch 2/100
12/12 [==============================] - 26s 2s/step - loss: 0.6587 - accuracy:
0.5000 - val_loss: 0.5799 - val_accuracy: 0.5750
Epoch 3/100
12/12 [==============================] - 25s 2s/step - loss: 0.6246 - accuracy:
0.5000 - val_loss: 0.5817 - val_accuracy: 0.5750
Epoch 4/100
12/12 [==============================] - 25s 2s/step - loss: 0.6190 - accuracy:
0.5097 - val_loss: 0.5808 - val_accuracy: 0.5875
Epoch 5/100
12/12 [==============================] - 25s 2s/step - loss: 0.6139 - accuracy:
0.5292 - val_loss: 0.5810 - val_accuracy: 0.5750
Epoch 6/100
12/12 [==============================] - 24s 2s/step - loss: 0.6560 - accuracy:
0.5542 - val_loss: 0.5724 - val_accuracy: 0.6500
Epoch 7/100
12/12 [==============================] - 24s 2s/step - loss: 0.6185 - accuracy:
0.5264 - val_loss: 0.6106 - val_accuracy: 0.6000
Epoch 8/100
12/12 [==============================] - 24s 2s/step - loss: 0.6056 - accuracy:
0.5556 - val_loss: 0.5644 - val_accuracy: 0.6875
Epoch 9/100
12/12 [==============================] - 24s 2s/step - loss: 0.6040 - accuracy:
0.5903 - val_loss: 0.5883 - val_accuracy: 0.6375
Epoch 10/100
12/12 [==============================] - 24s 2s/step - loss: 0.6014 - accuracy:
0.5875 - val_loss: 0.5863 - val_accuracy: 0.6625
Epoch 11/100
12/12 [==============================] - 25s 2s/step - loss: 0.5955 - accuracy:
0.6083 - val_loss: 0.6170 - val_accuracy: 0.6500
Epoch 12/100
12/12 [==============================] - 25s 2s/step - loss: 0.5989 - accuracy:
0.6042 - val_loss: 0.5888 - val_accuracy: 0.7250
Epoch 13/100
12/12 [==============================] - 25s 2s/step - loss: 0.6100 - accuracy:
0.6042 - val_loss: 0.6256 - val_accuracy: 0.6375
Epoch 14/100
12/12 [==============================] - 25s 2s/step - loss: 0.5909 - accuracy:
0.6264 - val_loss: 0.6174 - val_accuracy: 0.7000
Epoch 15/100
12/12 [==============================] - 25s 2s/step - loss: 0.6054 - accuracy:
0.6500 - val_loss: 0.5875 - val_accuracy: 0.6500
Epoch 16/100
12/12 [==============================] - 25s 2s/step - loss: 0.5750 - accuracy:
0.6361 - val_loss: 0.5863 - val_accuracy: 0.6875
Epoch 17/100
12/12 [==============================] - 25s 2s/step - loss: 0.5806 - accuracy:
0.6486 - val_loss: 0.5743 - val_accuracy: 0.6500
Epoch 18/100
12/12 [==============================] - 25s 2s/step - loss: 0.5804 - accuracy:
0.6667 - val_loss: 0.5598 - val_accuracy: 0.6500
Epoch 19/100
12/12 [==============================] - 25s 2s/step - loss: 0.5939 - accuracy:
0.6111 - val_loss: 0.5904 - val_accuracy: 0.7250
Epoch 20/100
12/12 [==============================] - 25s 2s/step - loss: 0.5768 - accuracy:
0.6819 - val_loss: 0.5833 - val_accuracy: 0.6750
```

```
Epoch 21/100
12/12 [==============================] - 25s 2s/step - loss: 0.5642 - accuracy:
0.6486 - val_loss: 0.6033 - val_accuracy: 0.6750
Epoch 22/100
12/12 [==============================] - 24s 2s/step - loss: 0.5710 - accuracy:
0.6653 - val_loss: 0.5820 - val_accuracy: 0.7250
Epoch 23/100
12/12 [==============================] - 24s 2s/step - loss: 0.5663 - accuracy:
0.6861 - val_loss: 0.6039 - val_accuracy: 0.7375
Epoch 24/100
12/12 [==============================] - 26s 2s/step - loss: 0.5645 - accuracy:
0.7056 - val_loss: 0.5627 - val_accuracy: 0.7375
Epoch 25/100
12/12 [==============================] - 25s 2s/step - loss: 0.5630 - accuracy:
0.7111 - val_loss: 0.5772 - val_accuracy: 0.7625
Epoch 26/100
12/12 [==============================] - 26s 2s/step - loss: 0.5556 - accuracy:
0.6833 - val_loss: 0.5867 - val_accuracy: 0.7375
Epoch 27/100
12/12 [==============================] - 26s 2s/step - loss: 0.5529 - accuracy:
0.7458 - val_loss: 0.5666 - val_accuracy: 0.7625
Epoch 28/100
12/12 [==============================] - 25s 2s/step - loss: 0.5490 - accuracy:
0.6861 - val_loss: 0.5858 - val_accuracy: 0.7375
Epoch 29/100
12/12 [==============================] - 25s 2s/step - loss: 0.5494 - accuracy:
0.6847 - val_loss: 0.5757 - val_accuracy: 0.7250
Epoch 30/100
12/12 [==============================] - 25s 2s/step - loss: 0.5493 - accuracy:
0.7333 - val_loss: 0.5863 - val_accuracy: 0.7750
Epoch 31/100
12/12 [==============================] - 25s 2s/step - loss: 0.5573 - accuracy:
0.7125 - val_loss: 0.5463 - val_accuracy: 0.7750
Epoch 32/100
12/12 [==============================] - 25s 2s/step - loss: 0.5463 - accuracy:
0.7361 - val_loss: 0.5703 - val_accuracy: 0.7625
Epoch 33/100
12/12 [==============================] - 25s 2s/step - loss: 0.5440 - accuracy:
0.7389 - val_loss: 0.5621 - val_accuracy: 0.7750
Epoch 34/100
12/12 [==============================] - 24s 2s/step - loss: 0.5407 - accuracy:
0.7556 - val_loss: 0.5707 - val_accuracy: 0.7875
Epoch 35/100
12/12 [==============================] - 24s 2s/step - loss: 0.5401 - accuracy:
0.7667 - val_loss: 0.5727 - val_accuracy: 0.7250
Epoch 36/100
12/12 [==============================] - 25s 2s/step - loss: 0.5393 - accuracy:
0.7722 - val_loss: 0.5451 - val_accuracy: 0.7375
Epoch 37/100
12/12 [==============================] - 25s 2s/step - loss: 0.5406 - accuracy:
0.7347 - val_loss: 0.5363 - val_accuracy: 0.7500
Epoch 38/100
12/12 [==============================] - 25s 2s/step - loss: 0.5573 - accuracy:
0.7083 - val_loss: 0.5686 - val_accuracy: 0.7500
Epoch 39/100
12/12 [==============================] - 26s 2s/step - loss: 0.5308 - accuracy:
0.7431 - val_loss: 0.5427 - val_accuracy: 0.7875
Epoch 40/100
12/12 [==============================] - 26s 2s/step - loss: 0.5325 - accuracy:
0.7597 - val_loss: 0.5403 - val_accuracy: 0.7375
```

```
Epoch 41/100
12/12 [==============================] - 25s 2s/step - loss: 0.5308 - accuracy:
0.7306 - val_loss: 0.5584 - val_accuracy: 0.7625
Epoch 42/100
12/12 [==============================] - 26s 2s/step - loss: 0.5260 - accuracy:
0.7667 - val_loss: 0.5497 - val_accuracy: 0.7750
Epoch 43/100
12/12 [==============================] - 26s 2s/step - loss: 0.5158 - accuracy:
0.7681 - val_loss: 0.5472 - val_accuracy: 0.8000
Epoch 44/100
12/12 [==============================] - 27s 2s/step - loss: 0.5208 - accuracy:
0.7722 - val_loss: 0.5488 - val_accuracy: 0.7625
Epoch 45/100
12/12 [==============================] - 26s 2s/step - loss: 0.5176 - accuracy:
0.7653 - val_loss: 0.5467 - val_accuracy: 0.7750
Epoch 46/100
12/12 [==============================] - 26s 2s/step - loss: 0.5221 - accuracy:
0.7722 - val_loss: 0.5652 - val_accuracy: 0.7500
Epoch 47/100
12/12 [==============================] - 26s 2s/step - loss: 0.5150 - accuracy:
0.7514 - val_loss: 0.5707 - val_accuracy: 0.7875
Epoch 48/100
12/12 [==============================] - 27s 2s/step - loss: 0.5400 - accuracy:
0.7583 - val_loss: 0.5791 - val_accuracy: 0.7625
Epoch 49/100
12/12 [==============================] - 27s 2s/step - loss: 0.5080 - accuracy:
0.7889 - val_loss: 0.6019 - val_accuracy: 0.6875
Epoch 50/100
12/12 [==============================] - 27s 2s/step - loss: 0.5085 - accuracy:
0.7819 - val_loss: 0.5514 - val_accuracy: 0.7375
Epoch 51/100
12/12 [==============================] - 28s 2s/step - loss: 0.5003 - accuracy:
0.7806 - val_loss: 0.5537 - val_accuracy: 0.7375
Epoch 52/100
12/12 [==============================] - 28s 2s/step - loss: 0.5060 - accuracy:
0.7500 - val_loss: 0.5541 - val_accuracy: 0.7250
Epoch 53/100
12/12 [==============================] - 26s 2s/step - loss: 0.4940 - accuracy:
0.7847 - val_loss: 0.5192 - val_accuracy: 0.8000
Epoch 54/100
12/12 [==============================] - 25s 2s/step - loss: 0.4929 - accuracy:
0.7833 - val_loss: 0.5355 - val_accuracy: 0.7875
Epoch 55/100
12/12 [==============================] - 26s 2s/step - loss: 0.4923 - accuracy:
0.7819 - val_loss: 0.5476 - val_accuracy: 0.7250
Epoch 56/100
12/12 [==============================] - 25s 2s/step - loss: 0.5009 - accuracy:
0.8042 - val_loss: 0.5460 - val_accuracy: 0.7625
Epoch 57/100
12/12 [==============================] - 24s 2s/step - loss: 0.5110 - accuracy:
0.7597 - val_loss: 0.5437 - val_accuracy: 0.7750
Epoch 58/100
12/12 [==============================] - 24s 2s/step - loss: 0.4998 - accuracy:
0.7736 - val_loss: 0.5503 - val_accuracy: 0.7625
Epoch 59/100
12/12 [==============================] - 23s 2s/step - loss: 0.4892 - accuracy:
0.7972 - val_loss: 0.5434 - val_accuracy: 0.7500
Epoch 60/100
12/12 [==============================] - 25s 2s/step - loss: 0.4823 - accuracy:
0.8069 - val_loss: 0.5273 - val_accuracy: 0.7750
```

```
Epoch 61/100
12/12 [==============================] - 25s 2s/step - loss: 0.4849 - accuracy:
0.8056 - val_loss: 0.5596 - val_accuracy: 0.7375
Epoch 62/100
12/12 [==============================] - 25s 2s/step - loss: 0.4818 - accuracy:
0.8042 - val_loss: 0.5298 - val_accuracy: 0.7375
Epoch 63/100
12/12 [==============================] - 25s 2s/step - loss: 0.4796 - accuracy:
0.7972 - val_loss: 0.5448 - val_accuracy: 0.7875
Epoch 64/100
12/12 [==============================] - 25s 2s/step - loss: 0.4803 - accuracy:
0.8000 - val_loss: 0.5597 - val_accuracy: 0.7000
Epoch 65/100
12/12 [==============================] - 25s 2s/step - loss: 0.4801 - accuracy:
0.8278 - val_loss: 0.5320 - val_accuracy: 0.7375
Epoch 66/100
12/12 [==============================] - 26s 2s/step - loss: 0.4655 - accuracy:
0.8431 - val_loss: 0.5297 - val_accuracy: 0.7500
Epoch 67/100
12/12 [==============================] - 26s 2s/step - loss: 0.4620 - accuracy:
0.7931 - val_loss: 0.5497 - val_accuracy: 0.7000
Epoch 68/100
12/12 [==============================] - 27s 2s/step - loss: 0.4804 - accuracy:
0.8153 - val_loss: 0.5361 - val_accuracy: 0.7000
Epoch 69/100
12/12 [==============================] - 25s 2s/step - loss: 0.4579 - accuracy:
0.8222 - val_loss: 0.5437 - val_accuracy: 0.7125
Epoch 70/100
12/12 [==============================] - 24s 2s/step - loss: 0.4467 - accuracy:
0.8597 - val_loss: 0.5243 - val_accuracy: 0.7625
Epoch 71/100
12/12 [==============================] - 24s 2s/step - loss: 0.4506 - accuracy:
0.8236 - val_loss: 0.5410 - val_accuracy: 0.7625
Epoch 72/100
12/12 [==============================] - 24s 2s/step - loss: 0.4406 - accuracy:
0.8236 - val_loss: 0.5208 - val_accuracy: 0.7750
Epoch 73/100
12/12 [==============================] - 24s 2s/step - loss: 0.4635 - accuracy:
0.8097 - val_loss: 0.5279 - val_accuracy: 0.6875
Epoch 74/100
12/12 [==============================] - 24s 2s/step - loss: 0.4489 - accuracy:
0.8278 - val_loss: 0.5099 - val_accuracy: 0.7625
Epoch 75/100
12/12 [==============================] - 24s 2s/step - loss: 0.4355 - accuracy:
0.8389 - val_loss: 0.4998 - val_accuracy: 0.7625
Epoch 76/100
12/12 [==============================] - 24s 2s/step - loss: 0.4346 - accuracy:
0.8417 - val_loss: 0.5173 - val_accuracy: 0.7250
Epoch 77/100
12/12 [==============================] - 24s 2s/step - loss: 0.4459 - accuracy:
0.8236 - val_loss: 0.5005 - val_accuracy: 0.7625
Epoch 78/100
12/12 [==============================] - 24s 2s/step - loss: 0.4395 - accuracy:
0.8347 - val_loss: 0.5376 - val_accuracy: 0.7000
Epoch 79/100
12/12 [==============================] - 23s 2s/step - loss: 0.4432 - accuracy:
0.8319 - val_loss: 0.5074 - val_accuracy: 0.7375
Epoch 80/100
12/12 [==============================] - 24s 2s/step - loss: 0.4408 - accuracy:
0.8431 - val_loss: 0.5104 - val_accuracy: 0.7250
```

```
Epoch 81/100
12/12 [==============================] - 25s 2s/step - loss: 0.4274 - accuracy:
0.8556 - val_loss: 0.4964 - val_accuracy: 0.7500
Epoch 82/100
12/12 [==============================] - 25s 2s/step - loss: 0.4244 - accuracy:
0.8389 - val_loss: 0.5117 - val_accuracy: 0.7125
Epoch 83/100
12/12 [==============================] - 25s 2s/step - loss: 0.4261 - accuracy:
0.8417 - val_loss: 0.5187 - val_accuracy: 0.7250
Epoch 84/100
12/12 [==============================] - 24s 2s/step - loss: 0.4218 - accuracy:
0.8528 - val_loss: 0.5239 - val_accuracy: 0.7500
Epoch 85/100
12/12 [==============================] - 24s 2s/step - loss: 0.4224 - accuracy:
0.8528 - val_loss: 0.5077 - val_accuracy: 0.7375
Epoch 86/100
12/12 [==============================] - 24s 2s/step - loss: 0.4173 - accuracy:
0.8556 - val_loss: 0.5084 - val_accuracy: 0.7250
Epoch 87/100
12/12 [==============================] - 24s 2s/step - loss: 0.4194 - accuracy:
0.8667 - val_loss: 0.5040 - val_accuracy: 0.7250
Epoch 88/100
12/12 [==============================] - 23s 2s/step - loss: 0.4057 - accuracy:
0.8569 - val_loss: 0.5291 - val_accuracy: 0.7500
Epoch 89/100
12/12 [==============================] - 23s 2s/step - loss: 0.4315 - accuracy:
0.8278 - val_loss: 0.5285 - val_accuracy: 0.7125
Epoch 90/100
12/12 [==============================] - 23s 2s/step - loss: 0.4199 - accuracy:
0.8556 - val_loss: 0.5170 - val_accuracy: 0.7875
Epoch 91/100
12/12 [==============================] - 23s 2s/step - loss: 0.4249 - accuracy:
0.8306 - val_loss: 0.5034 - val_accuracy: 0.7500
Epoch 92/100
12/12 [==============================] - 23s 2s/step - loss: 0.4079 - accuracy:
0.8500 - val_loss: 0.4901 - val_accuracy: 0.7875
Epoch 93/100
12/12 [==============================] - 24s 2s/step - loss: 0.4007 - accuracy:
0.8708 - val_loss: 0.4988 - val_accuracy: 0.8000
Epoch 94/100
12/12 [==============================] - 23s 2s/step - loss: 0.4148 - accuracy:
0.8556 - val_loss: 0.4691 - val_accuracy: 0.7750
Epoch 95/100
12/12 [==============================] - 24s 2s/step - loss: 0.4029 - accuracy:
0.8694 - val_loss: 0.5247 - val_accuracy: 0.7000
Epoch 96/100
12/12 [==============================] - 24s 2s/step - loss: 0.3943 - accuracy:
0.8736 - val_loss: 0.4914 - val_accuracy: 0.7375
Epoch 97/100
12/12 [==============================] - 24s 2s/step - loss: 0.3982 - accuracy:
0.8681 - val_loss: 0.4853 - val_accuracy: 0.7750
Epoch 98/100
12/12 [==============================] - 24s 2s/step - loss: 0.3830 - accuracy:
0.8875 - val_loss: 0.5237 - val_accuracy: 0.7375
Epoch 99/100
12/12 [==============================] - 24s 2s/step - loss: 0.4039 - accuracy:
0.8597 - val_loss: 0.5493 - val_accuracy: 0.7125
Epoch 100/100
12/12 [==============================] - 24s 2s/step - loss: 0.4066 - accuracy:
0.8542 - val_loss: 0.5159 - val_accuracy: 0.7375
```

Each time we provide 64 pairs of images in a batch and this runs for 100 iterations.

The validation loss decreases constantly up to 80 iterations, then it does not change much. Similarly, no noticeable increment in the validation accuracy after the first few iterations.

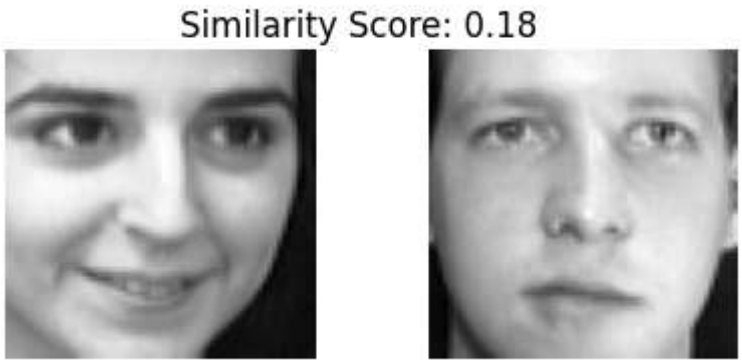
The test image will be paired up with one random image for each person in the dataset. To do this, I find all the indices for each of the labels again. (like earlier during training image pair generation). Then for each label a random image is fetched and paired up with the test image. The whole test image pairs now consist of a total of 40 image pairs, as the dataset has 400 images for 40 people.

In [17]:
```
#siamese_test
def generate_test_image_pairs(images_dataset, labels_dataset, image):
    unique_labels = np.unique(labels_dataset)
    label_wise_indices = dict()
    for label in unique_labels:
          label_wise_indices.setdefault(label,
                                        [index for index, curr_label in enumerat
                                         label == curr_label])

    pair_images = []
    pair_labels = []
    for label, indices_for_label in label_wise_indices.items():
        test_image = images_dataset[np.random.choice(indices_for_label)]
        pair_images.append((image, test_image))
        pair_labels.append(label)
    return np.array(pair_images), np.array(pair_labels)
```

In [20]:
```
image = images_dataset[92]  # a random image as a test image
test_image_pairs, test_label_pairs = generate_test_image_pairs(images_dataset, l

# for each pair in the test image pair, predict the similarity between the image
import matplotlib.pyplot as plt

def show_images_and_prediction(pair, prediction):
    plt.figure(figsize=(5, 2))

    # Display first image
    plt.subplot(1, 2, 1)
    plt.imshow(pair[0].reshape(64, 64), cmap='gray')
    plt.axis('off')

    # Display second image
    plt.subplot(1, 2, 2)
    plt.imshow(pair[1].reshape(64, 64), cmap='gray')
    plt.axis('off')

    # Print prediction
    plt.suptitle(f"Similarity Score: {prediction:.2f}")
    plt.show()

for index, pair in enumerate(test_image_pairs):
    pair_image1 = np.expand_dims(pair[0], axis=-1)
    pair_image1 = np.expand_dims(pair_image1, axis=0)
    pair_image2 = np.expand_dims(pair[1], axis=-1)
```

```
    pair_image2 = np.expand_dims(pair_image2, axis=0)
    prediction = model.predict([pair_image1, pair_image2])[0][0]

    show_images_and_prediction(pair, prediction)
```

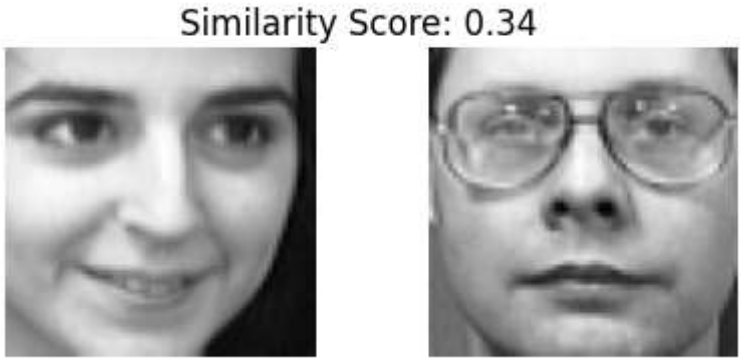
```
1/1 [==============================] - 0s 41ms/step
```




```
1/1 [==============================] - 0s 27ms/step
```


```
1/1 [==============================] - 0s 37ms/step
```


```
1/1 [==============================] - 0s 33ms/step
```




```
1/1 [==============================] - 0s 31ms/step
```

Similarity Score: 0.22

```
1/1 [==============================] - 0s 30ms/step
```

Similarity Score: 0.42

```
1/1 [==============================] - 0s 33ms/step
```

Similarity Score: 0.56

```
1/1 [==============================] - 0s 33ms/step
```

Similarity Score: 0.60

```
1/1 [==============================] - 0s 35ms/step
```

Similarity Score: 0.53

```
1/1 [==============================] - 0s 29ms/step
```

Similarity Score: 0.66

```
1/1 [==============================] - 0s 30ms/step
```

Similarity Score: 0.22

```
1/1 [==============================] - 0s 32ms/step
```

Similarity Score: 0.56

```
1/1 [==============================] - 0s 31ms/step
```

Similarity Score: 0.22

```
1/1 [==============================] - 0s 31ms/step
```

Similarity Score: 0.65

```
1/1 [==============================] - 0s 36ms/step
```

Similarity Score: 0.10

```
1/1 [==============================] - 0s 30ms/step
```

Similarity Score: 0.20

```
1/1 [==============================] - 0s 30ms/step
```

Similarity Score: 0.58

```
1/1 [==============================] - 0s 28ms/step
```

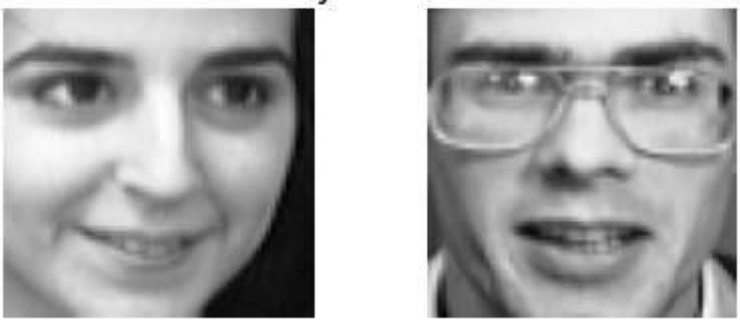
Similarity Score: 0.43

```
1/1 [==============================] - 0s 28ms/step
```

Similarity Score: 0.34

```
1/1 [==============================] - 0s 28ms/step
```

Similarity Score: 0.18



```
1/1 [==============================] - 0s 27ms/step
```

Similarity Score: 0.05

```
1/1 [==============================] - 0s 30ms/step
```

Similarity Score: 0.01

```
1/1 [==============================] - 0s 26ms/step
```

Similarity Score: 0.63

```
1/1 [==============================] - 0s 28ms/step
```

Similarity Score: 0.03

```
1/1 [==============================] - 0s 29ms/step
```

Similarity Score: 0.48

```
1/1 [==============================] - 0s 31ms/step
```

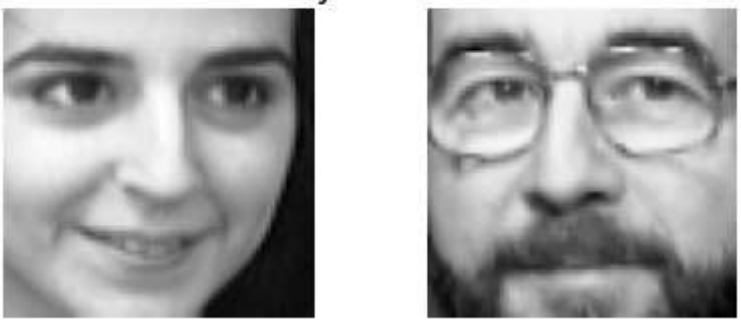
Similarity Score: 0.52

```
1/1 [==============================] - 0s 33ms/step
```

Similarity Score: 0.37

```
1/1 [==============================] - 0s 32ms/step
```

Similarity Score: 0.47

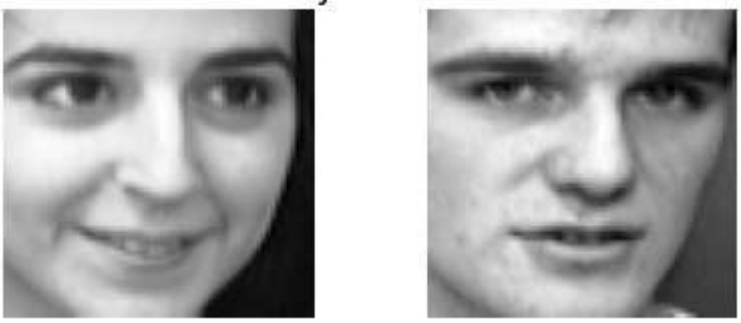


```
1/1 [==============================] - 0s 33ms/step
```

Similarity Score: 0.02

```
1/1 [==============================] - 0s 30ms/step
```

Similarity Score: 0.12



```
1/1 [==============================] - 0s 28ms/step
```

Similarity Score: 0.10

```
1/1 [==============================] - 0s 31ms/step
```

Similarity Score: 0.72

```
1/1 [==============================] - 0s 32ms/step
```

Similarity Score: 0.00

```
1/1 [==============================] - 0s 32ms/step
```

Similarity Score: 0.02

```
1/1 [==============================] - 0s 32ms/step
```

Similarity Score: 0.15

```
1/1 [==============================] - 0s 32ms/step
```

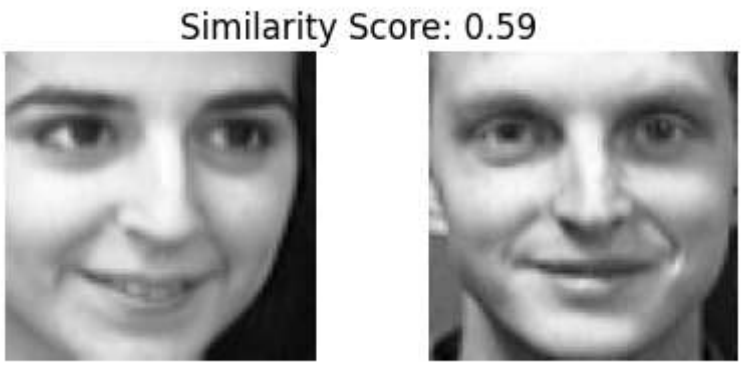
Similarity Score: 0.71

```
1/1 [==============================] - 0s 31ms/step
```


```
1/1 [==============================] - 0s 29ms/step
```


```
1/1 [==============================] - 0s 28ms/step
```


```
1/1 [==============================] - 0s 28ms/step
```

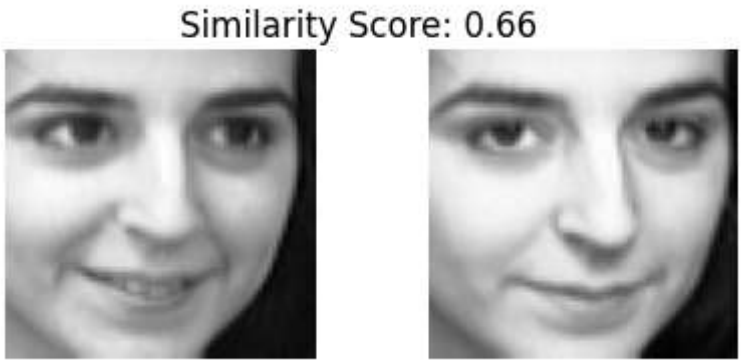


From above results we can observe that if similarity is >0.65 then both the images are likely of same person. So we can write the code as follows with threshold.

In [24]:
```
#It will show only those pairs in which both the images are of same person
# Define a threshold for similarity
similarity_threshold = 0.655
for index, pair in enumerate(test_image_pairs):
    pair_image1 = np.expand_dims(pair[0], axis=-1)
    pair_image1 = np.expand_dims(pair_image1, axis=0)
```

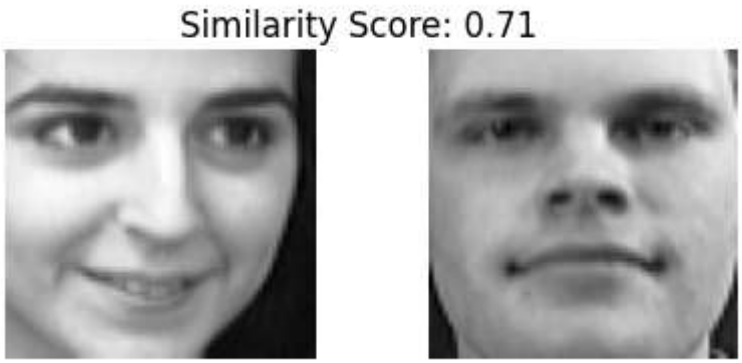
```
    pair_image2 = np.expand_dims(pair[1], axis=-1)
    pair_image2 = np.expand_dims(pair_image2, axis=0)
    prediction = model.predict([pair_image1, pair_image2])[0][0]

    if prediction > similarity_threshold:
        show_images_and_prediction(pair, prediction)
```

```
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 29ms/step
```




```
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 33ms/step
```


```
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 31ms/step
```




```
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 33ms/step
```

Here we can observe that model is giving some extra pairs also. So it's not very good. Following are some of the changes that can be tried to improve the model. But it takes a lot of time for training and due to time constraint I can't try those changes now. But I will try them in future.

Model Improvement Strategies

1. **Adjust Convolutional Layers:**
    - Experiment with different sizes and numbers of filters in your Conv2D layers.
    - Consider smaller filters (e.g., 3x3 or 5x5) to capture finer details.
2. **Add More Convolutional Layers:**
    - Deeper networks can capture more complex patterns.
    - Be cautious of overfitting when adding more layers.
3. **Batch Normalization:**
    - Include BatchNormalization layers to normalize activations and prevent internal covariate shift.
4. **Experiment with Dropout Rates:**
    - Adjust dropout rates to reduce overfitting.

- Experiment with different rates to find the optimal value.

5. **Use More Advanced Layers:**
   - Consider layers like Residual Connections, Depthwise Separable Convolutions, or Attention Mechanisms.
6. **Fine-tune Dense Layers:**
   - Adjust the number of neurons in Dense layers for improved model performance.
7. **Learning Rate Scheduling:**
   - Implement learning rate scheduling or use adaptive optimizers (e.g., Adam, RMSprop) for efficient convergence.
8. **Data Augmentation:**
   - Increase dataset size with modified versions of input images (flipped, rotated, scaled, etc.).
9. **Regularization Techniques:**
   - Use L1 or L2 regularization in layers to prevent overfitting.
10. **Hyperparameter Optimization:**
    - Use grid search or random search to experiment with different hyperparameter sets.
11. **Evaluate Different Activation Functions:**
    - Experiment with activations like LeakyReLU or ELU, in addition to the common ReLU.
12. **Increase Model Complexity Carefully:**
    - If the model is too simple, cautiously increase complexity to capture underlying patterns without overfitting.
13. **Use Pretrained Models:**
    - Consider transfer learning by starting with a model pretrained on a large dataset and fine-tuning for your specific task.

Siamese model is mostly used when we don't have a huge number of images for the training. A Deep Learning Neural Network for Classification performs better only when the number of images in the dataset is huge.

For ex. If we are building a Face Recognition System for an Office with only 100 employees. Their face repository has only a maximum of 10 face images for each of them. Here the number of images are really less, to create a Classification Neural Network. Siamese Network comes in as a handy replacement here. It does not learn the classes, rather it only finds the distance between the images. The number of images need not be high for a Siamese Network.

Main benefit of Siamese Network

When we want to add new image to the dataset then a Regular CNN would need a new model to be retrained with the new image data for classification. But for **Siamese Network, we don’t need to retrain the model**. It would be able to find the distance out between two unseen images as well. So if we want to add an image of a new person then we don't need to retrain the model. WE can simply add it in the dataset and the model can give us the predictions i.e. its similarity with other images.

In [ ]: