Instituto Superior de Engenharia de Lisboa

Visão Artificial e Realidade Mista

Deteção e Reconhecimento de Faces para Realidade Aumentada

Mestrado em Engenharia Informática de Multimédia

Pedro Gonçalves, 45890

Semestre de Verão, 2021/2022

Introdução

Pretende-se explorar técnicas de deteção e reconhecimento de faces. Para lidar com a deteção de faces, utilizar-se-ão o classificador **Haar Cascade** e o módulo **Deep Neural Network**. Para lidar com o reconhecimento de faces, utilizar-se-á o método EigenFaces. Antes da fase de reconhecimento, recorrer-se-á a um processo de normalização das imagens escolhidas para o ***dataset***, que tirará partido dos métodos de deteção de faces e olhos.

Todo o código associado a este projeto pode ser encontrado aqui.

Face Detection

A primeira parte do projeto consiste em tirar partido de algoritmos já capazes de detetar faces presentes numa imagem. Neste projeto, com auxílio da biblioteca **OpenCV** (disponibiliza ferramentas de visão artificial otimizada em tempo real), estes métodos serão utilizados exclusivamente para reconhecer faces e olhos.

Num primeiro experimento, desenharam-se retângulos a circundar as áreas em que faces foram detetadas numa determinada imagem.

O primeiro método, proposto por Paul Viola e Michael Jones, permite não só a deteção de faces, mas como de vários objetos, e denomina-se **Haar Cascade Classifier**. O segundo método em que se tira partido de uma **Convolution Neural Network** pré-treinada (**CNN**). Utilizar-se-á uma biblioteca chamada **MTCNN**, que tem como referência a implementação **MTCNN** de David Sandberg in Facenet.

Observe-se as figuras 1 e 2, em que se utiliza **Haar Cascade Classifier**, e as figuras 3 e 4, em que se utiliza **Convolution Neural Network**.

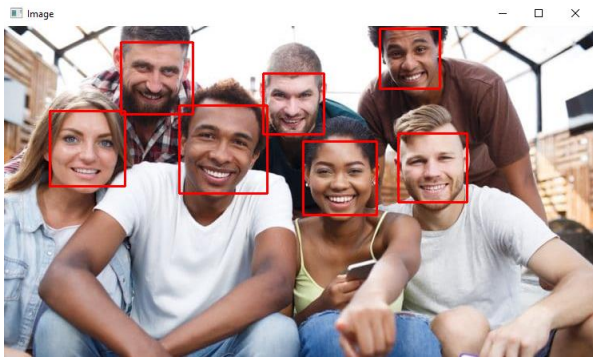

Figura 1 - friends.jpg (HCC)

Figura 2 – friends2.jpg (HCC)

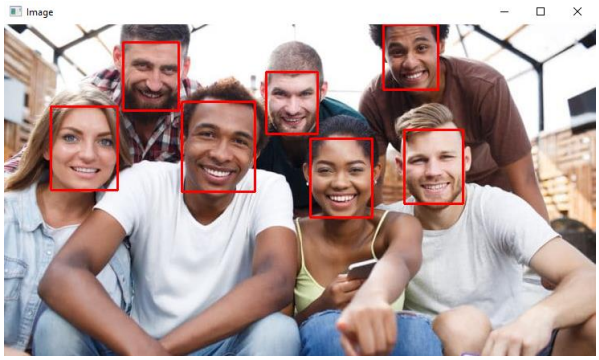

Figura 3 - friends.jpg (CNN)

Figura 4 - friends2.jpg (CNN)

Note-se que com ambos os algoritmos não existem falsos positivos, mas em cada um existe um falso negativo (uma face não identificada).

Utilizando estes algoritmos, torna-se relativamente trivial a implementação de uma aplicação que permita coletar imagens de uma câmera e proceder à deteção de faces para cada uma. A biblioteca **OpenCV** dispõe de um conjunto de funções que ajudam nesse sentido. Observe-se o exemplo da **figura 5**.

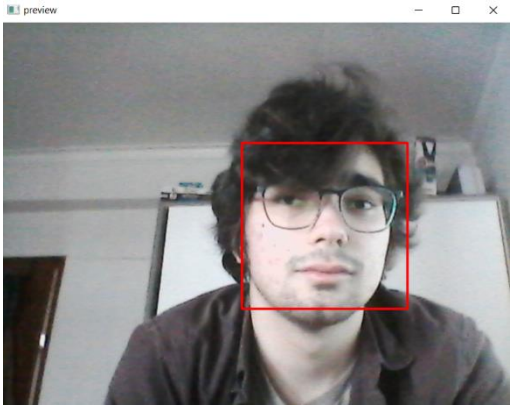

Figura 5 - Deteção de faces em tempo real

Dataset de Faces

Para que se consiga explorar o conceito de reconhecimento facial com algum itneresse, é necessário reunir um conjunto razoável de faces. Optou-se por juntar faces de **5** pessoas diferentes. O ***dataset*** resultante é a da **tabela 1**.

Tabela 1 – Dataset de faces

Pessoa	Número de imagens	Classe
Cillian Murphy	12	'cillian'
Benedict Cumberbatch	12	'cumberbatch'
Random Woman	12	'lady'
Elisabeth Olsen	12	'olsen'
Professor	9	'stor'

Normalização das Faces do Dataset

A necessidade de normalizar as faces no ***dataset***, surge do facto de os classificadores que serão utilizados para reconhecimento facial, terem melhor desempenho quando isso se verifica. O processo de normalização consiste em transformar todas as imagens da ***dataset*** numa imagem com determinadas dimensões e com os olhos em determinadas coordenadas da imagem.

As imagens da ***dataset***, nesta fase, apresentam dimensões muito variadas. Por exemplo, a imagem **cillian01.jpg** tem dimensões de **1200x1200** e a imagem **cillian04.jpg** tem dimensões de **1080x1349**. Além disso, os olhos estão e em coordenadas muito variadas. Note-se por exemplo a imagem da **figura 6**, em que os olhos estão aproximadamente nivelados, a imagem da **figura 7**, em que os olhos traçam um ângulo mais exagerado.

Figura 6 - olsen04.jpg

Figura 7 - olsen07.jpg

Ainda tomando como exemplo as **figuras 6** e **7**, note-se também que ambos os olhos estão em coordenadas totalmente diferentes.

Como resultado do processo de normalização, pretende-se que todas as imagens:

- Apresentem uma dimensão de **46x56** (**56** de altura);
- Sejam monocromáticas (**0** até **255**, para cada pixel);
- Contem com o centro do olho esquerdo na coluna **16** e linha **24**;
- Contem com o centro do olho direito na coluna **31** e linha **24**;

Para implementar o processo de normalização, foi gerada a classe da **figura 8**.

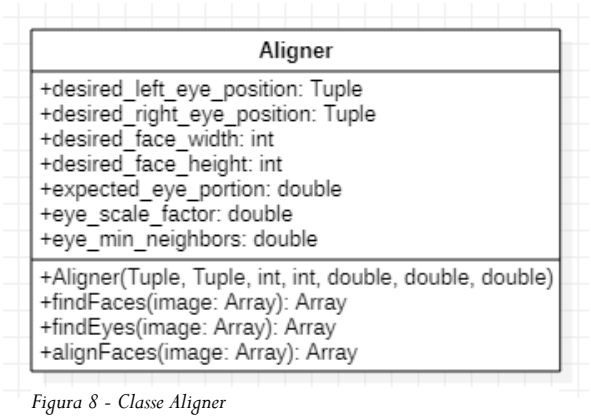

Figura 8 - Classe Aligner

A primeira fase da normalização, que tira partido das funções **findFaces()** e **findEyes()**, é encontrar as posições dos olhos da imagem que se quer normalizar. Ao encontrar todas as faces presentes na imagem (assumindo que é apenas uma, já que as imagens servem esse propósito), a pesquisa dos olhos é realizada apenas na área da face encontrada, para que não sejam encontrados falsos positivos fora da face.

Obtendo o centro dos olhos, procede-se ao cálculo do anglo traçado pelos mesmos (**alpha**) e também a distância entre os centros (**d**).

$$\Delta Y = olho_{direito}.y - olho_{esquerdo}.y$$

$$\Delta X = olho_{direito}.x - olho_{esquerdo}.x$$

$$\alpha = \arctan(\Delta Y - \Delta X)$$

$$d = \sqrt{\Delta X^2 + \Delta Y^2}$$

Com a distância entre os centros dos olhos, é possível calcular o scale factor, que irá indicar o quão a imagem terá de ser aproximada ou afastada, para que os olhos fiquem na posição desejada.

$$scale_{factor} = \frac{desired\ eye\ distance}{d}$$

Falta também calcular o centro entre os dois olhos, para que a rotação seja feita centrada nesse ponto:

$$eyes_{center}.x = \frac{olho_{esquerdo}.x + olho_{direito}.x}{2}$$

$$eyes_{center}.y = \frac{olho_{esquerdo}.y + olho_{direito}.y}{2}$$

Por fim, resta aplicar a translação, rotação e aproximação/afastamento, consoante os parâmetros calculados, com o auxílio das funções disponibilizadas pela biblioteca **OpenCV**.

Na **figura 9** verificam-se alguns exemplos de imagens do ***dataset*** normalizadas.

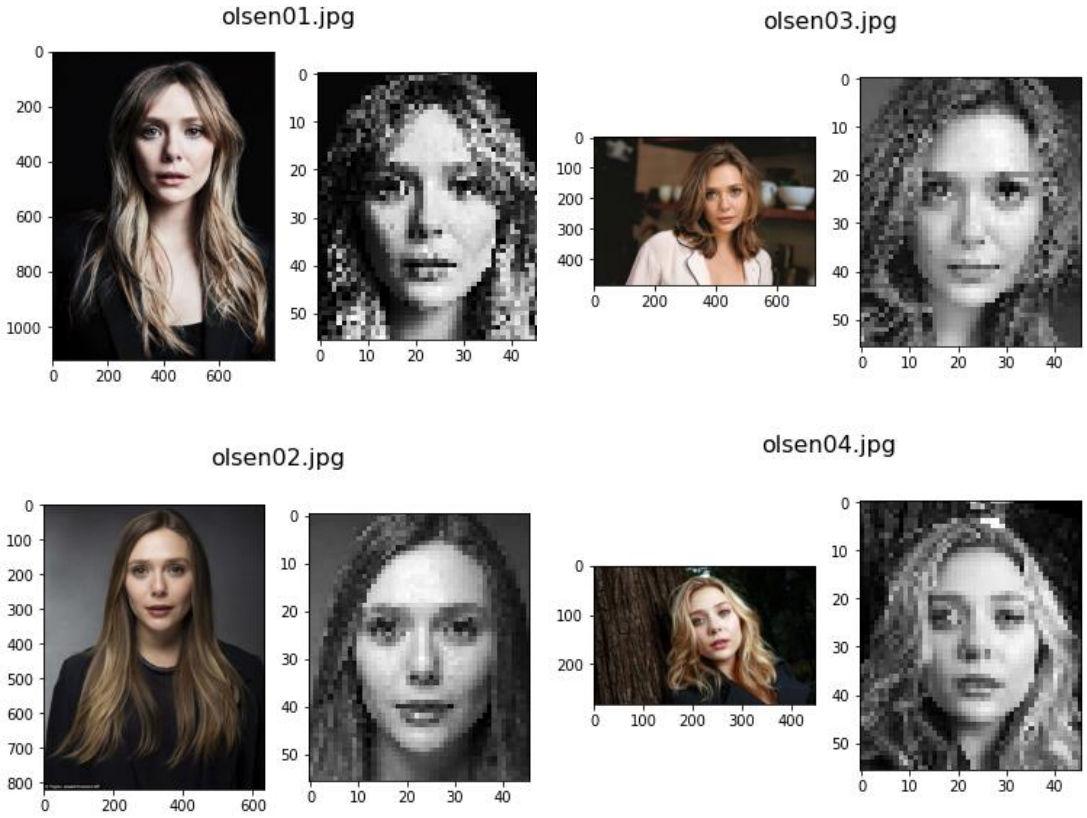

Figura 9 - Normalização

Preparação do Dataset

O próximo passo é carregar o ***dataset*** e separá-lo em treino e teste. O treino será composto por **47** exemplares, enquanto o teste será composto por **10** exemplares. Note-se que esta separação é feita com o método **stratify**, garantindo que o teste contém o mesmo número de exemplares correpondentes a cada classe.

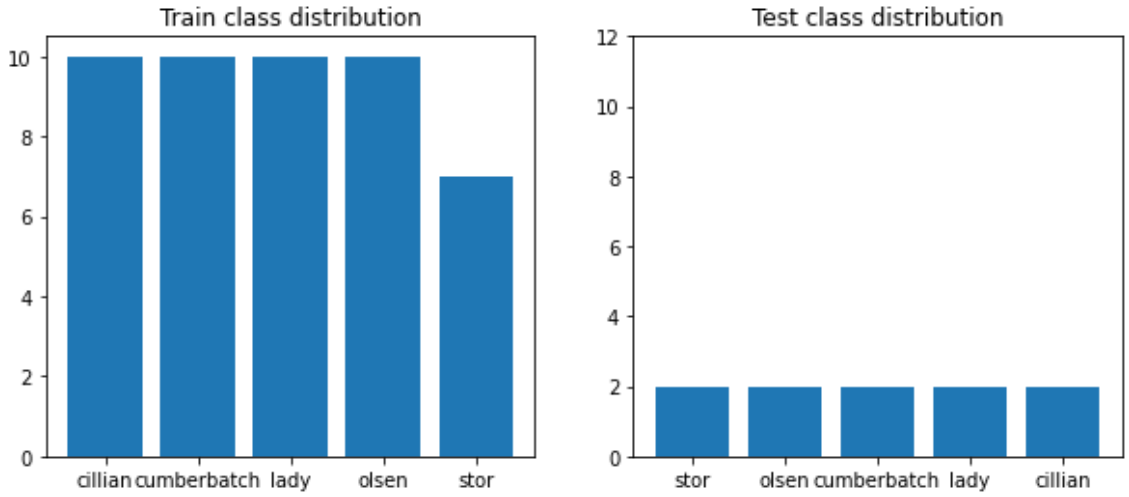

Figura 10 - Distribuição das classes do dataset

Classificação com SGD

Antes de passar às técnicas de reconhecimento de melhor desempenho, utilizar-se-á um classificador mais genérico (**Stochastic Gradient Descent Classifier**), que receberá como dados de treino um array contendo as informações de cada pixel (em ***grayscale***, **0** a **255**) e a classe correspondente (uma **string**).

Verifica-se que o facto de as imagens estarem normalizadas permite que mesmo um classificador genérico tenho um desempenho razoável na identificação de faces. Observem-se os resultados das **figuras 11** e **12**.

Figura 11 - Resultados da predição SGD

Na matriz de confusão, verifica-se uma exatidão de **70%**. Por outro lado, nunca é demais destacar que o dataset utilizado é muito limitado, pelo que este projeto tem um interesse maioritariamente implementacional.

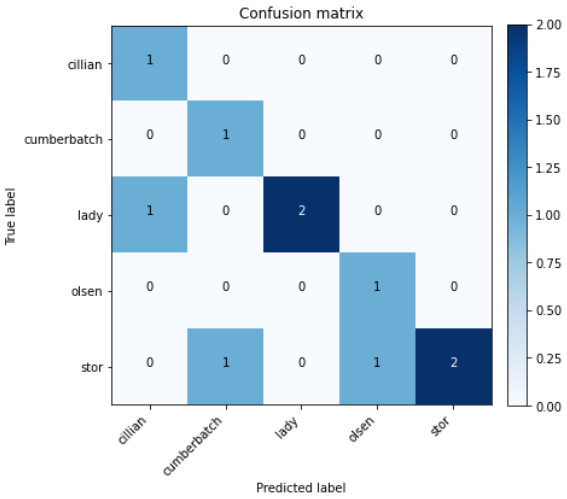

Figura 12 - Matriz de confusão da predição SGD

EigenFaces

O algoritmo EigenFaces pode ser visto, na sua essência, como um compressor de imagens, pelo que ajuda a guardar mais informação (ou melhor informação) em tamanhos mais pequenos, recorrendo apenas a métodos algébricos fundamentais.

Este processo tira partido do PCA, que é algoritmo muito utilziado em diversas circunstâncias. Neste caso será útil, principalmente, para redução dimensional. Permitirá projetar, de forma linear, dados originais num subespaço de dimensões inferiors, gerando os **eigenvectors** (***principal components***).

Tendo já o dataset preparado, e assumindo **N** como o número de faces no conjunto de treino, o primeiro passo consiste em determinar a face média (***mean face***), denominada **μ**.

Para determinar a **mean face**, computa-se a média entre os pixels de todas as imagens que estejam nas mesmas coordenadas, resultando numa imagem contendo esses valores médios. Depois, subtrai-se essa imagem a todas as imagens do conjunto de treino.

$$X_{train_{ac}} = X_{train} - mean_{face}$$

A matriz **A** consiste na matrix cujas colunas são as componentes **AC** das faces do conjunto de treino. A partir da matriz **A**, calcula-se também a matriz **R**, a partir da qual computar-se-ão os **eigenvectors** e **eigenvalues**.

$$A = X_{train_{ac}}^{T}$$

$$R = A^{T}A$$

Note-se que o cálculo do produto entre $\mathbf{A^T}$, com dimensões de **(N x 2576)** e **A** com dimensões de **(2576 x N)**, resulta numa matriz com dimensões de **(N x N)**. Decidiu-se proceder desta forma, porque o dataset é muito reduzido. Normalmente o algoritmo EigenFaces faria o produto ao contrário ($\mathbf{A\,A^T}$), resultando numa matriz de dimensões **(2576 x 2576)**.

Selecionando apenas **N-1** (**m**) **eigenvectors** (aqueles que corresponderem a um **eigenvalue** mais elevado), constrói-se a matriz V, de dimensões **(N x m)**, cujas colunas consistem precisamente nos **eigenvectors** selecionados. Resta calcular a matriz **W**.

$$W = AV$$

Para determinar se os **eigenvectors** formam uma base ortogonal, calcula-se a matriz **Z**:

$$Z = W^{T}W$$

Se **Z** for igual à matriz identidade de **W**, então os **eigenvectors** formam uma base ortogonal. Verifique-se na **figura 13** a matriz **Z**.

As projeções no subespaço das faces, que serão utilizadas para classificar, calculam-se da seguinte forma:

$$Y = W^{T}(X - \mu)$$

Sendo **X** qualquer conjunto de imagens.

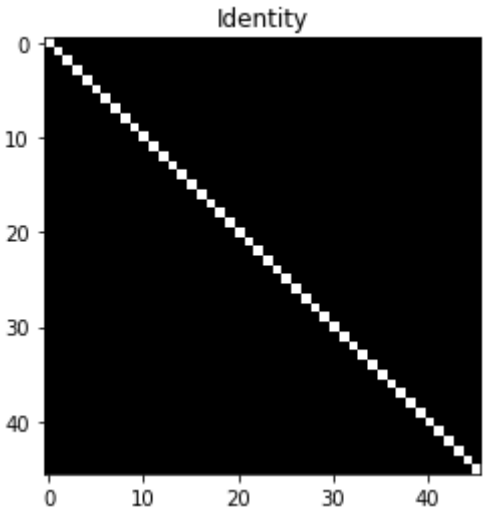

Figura 11 - Matriz Identidade

Reconstrução de Faces

Viu-se que o **EigenFaces** funciona como um compressor de imagens, pelo que projeta-as num subespaço de menores dimensões. Mas qual seria o procedimento para, a partir dessa projeção, recuperar a imagem original? E qual seria o erro associado a essa recuperação?

Para reconstruir, por exemplo, as imagens utilizadas para o treino, calcula-se a matriz **Xp** e sua transposta.

$$Xp = W\, Y_{train}$$
$$Xp = Xp^T$$

Depois, é só somar a **mean face** a cada uma das imagens da matriz **Xp**, obtendo a face original. Observe-se o exemplo da **figura 14**, em que se ilustra também a face de erro, calculada da seguinte forma:

$$error = original_{face} - reconstructed_{face}$$

Figura 12 - Reconstrução de uma face

Note-se que, apesar de a escala estar ausente, a intensidade dos pixels destas imagens não é proporcional entre si. Ou seja, o pixel mais intenso da imagem é considerado preto, enquanto o menos intenso é considerado branco, independentemente do valor ***grayscale***.

Para determinar se a face de erro é ortogonal ao subespaço de faces, calcula-se a sua projeção:

$$Y_{error} = W^T error$$

Se o resultado for uma matriz de zeros, então a face de erro é ortogonal ao subespaço de faces, como se verifica neste caso.

Classificação das Componentes de Projeção

Para comparar com o classifcador genérico **SGD** abordado anteriormente, desenvolver-se-á um classificador baseado em modelos treinados com observações das projeção no subespaço de imagens dado por:

$$Y_{treino} = W^T(x - \mu)$$

Em posse já da projeção do conjunto de treino, falta obter a projeção do conjunto de teste. Para isso, começa-se pro subtrair a mean face (**μ**) a cada um dos elementos (faces) do conjunto de teste. Depois efetuam-se os seguintes calculos (muito semelhantes ao que foi realizado para o conjunto de treino):

$$A = X_{test}^T$$
$$Y_{teste} = W^T A$$

Agora, para cada vector da projeção $\mathbf{Y}_{\mathbf{teste}}$ procurar-se-á o vector da projeção $\mathbf{Y}_{\mathbf{treino}}$ que mais se aproxima (distância euclideana). A imagem de teste respetiva será classificada como a classe associada ao vector da projeção $\mathbf{Y}_{\mathbf{treino}}$ encontrado. Este algoritmo denomina-se **Nearest Neighbour**.

Na **figura 15** verfica-se a classificação das diferentes imagens do conjunto de teste, e na **figura 16** a matriz de confusão associada às predições realizadas.

Figura 13 - Resultados da predição EigenFaces

Comparativamente com a abordagem de classificação anteriormente utilziada, em que se utilizavam os ***arrays*** de pixels (***grayscale***) como dados de entrada do classificar, verifica-se uma melhoria no desempenho deste rede, que tira partido dos vectores de projeção do algoritmo **EigenFaces**.

Desta vez, a exatidão atingiu os **90%**, obtendo apenas **1** erro em **10** imagens de teste.

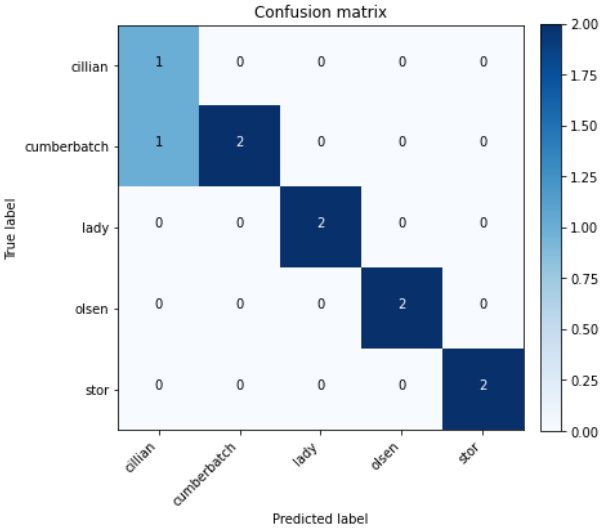

Figura 14 - Matriz de confusão da predição EigenFaces

Objetos Virtuais

Como parte final deste projeto, pretende-se aplicar, em tempo real, um objeto (como um chapéu ou óculos) a uma face presente numa imagem.

Para isso, tirar-se-á partido das estratégias de normalização abordadas anteriormente, mas, desta vez, não se pretende alinhar ou redimensionar a face, mas sim o objecto. Isto é, anteriormente pretendia-se levar os olhos da face detetada na imagem, para posições predefinidas, de forma a normalizar uma imagem. Desta vez pretende-se identificar as posições dos olhos relativamente a um objeto, e levá-los até aos olhos da face detetada na imagem (**figura 17**).

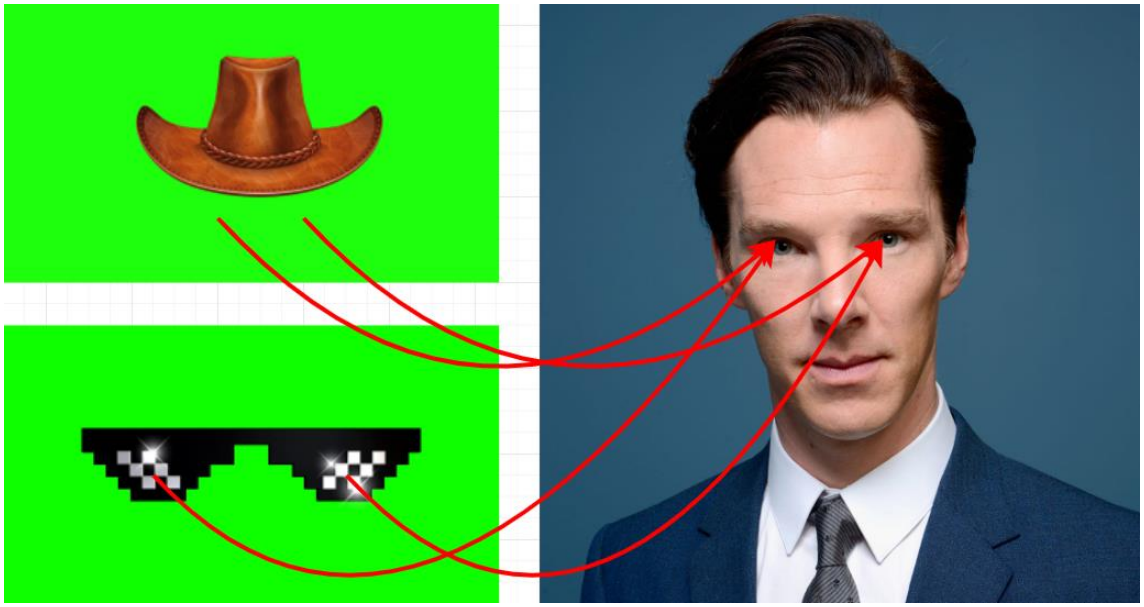

Figura 15 - Normalização de objetos

Primeiro, define-se, para cada objeto, as coordenadas de cada olho.

Obtendo também o centro dos olhos da imagem com a face, procede-se ao cálculo do anglo traçado pelos mesmos (**alpha**) e também a distância entre os centros (**distance**).

$$\Delta Y = olho_{direito}.y - olho_{esquerdo}.y$$

$$\Delta X = olho_{direito}.x - olho_{esquerdo}.x$$

$$\alpha_{\text{face}} = \arctan(\Delta Y - \Delta X)$$

$$distância_{face} = \sqrt{\Delta X^2 + \Delta Y^2}$$

Agora, repetem-se exatamente os mesmo cálculos para as coordenadas dos olhos nos objetos, obtendo $\boldsymbol{\alpha}_{\text{objeto}}$ e $\text{distância}_{\text{objeto}}$.

Com ambas as distâncias entre os centros dos olhos, é possível calcular o scale factor, que irá indicar o quão o objeto terá de ser aproximado ou afastado, para que fique na posição correta, relativamente aos olhos da imagem.

$$scale_{factor} = \frac{distância_{face}}{distância_{objeto}}$$

Falta também calcular o ângulo que irá rotacionar o objeto:

$$\alpha = \alpha_{\text{objeto}} - \alpha_{\text{face}}$$

Por fim, resta aplicar a translação, rotação e aproximação/afastamento, consoante os parâmetros calculados, mais uma vez com o auxílio das funções disponibilizadas pela biblioteca **OpenCV**.

Para que, ao aplicar a translação, deixar a verde (cor de fundo, para facilitar o processo de **chroma key**) as zonas fora da imagem, utiliza-se o parâmetro **borderValue** da função **warpAffine()**.

Finalmente, para transformar a imagem original (com a face) de forma a incluri os objetos virtuais, é necessário processar os objetos. Para criar uma máscara para cada objeto, utiliza-se a função **inRange()** para criar uma máscara em que os pixels do objeto são colocados a **0**, e os restantes a **1**.

A imagem resultante é calculada da seguinte forma:

$$output = (image * normalized_{object_{mask}}) +$$
$$(normalized_{object} * \ !normalized_{object_{mask}})$$

Na **figura 18** verifica-se um exemplo de uma imagem com objetos virtuais normalizados, que passaram por este processo.

Figura 16 - Resultados da virtualização de objetos

Finalmente, criando uma aplicação que adiciona objetos em tempo real, a uma imagem obtida através de uma câmera, obtém-se o resultado da **figura 19**.

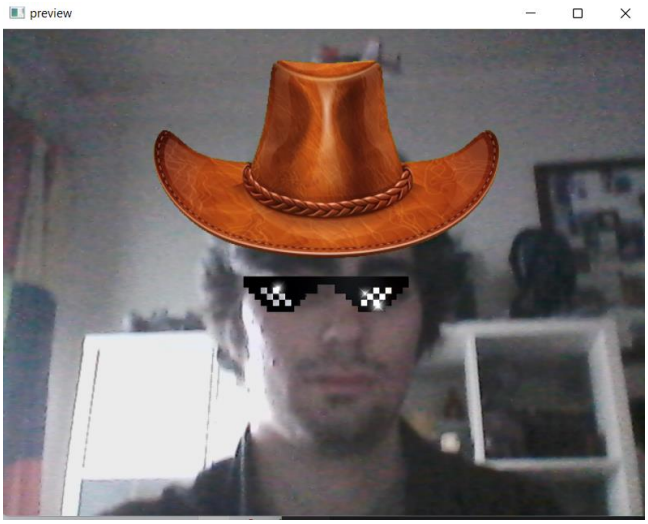

Figura 17 - Objetos virtuais em tempo real

Conclusão

A deteção e reconhecimento de faces é uma ferramenta muito útil em várias aplicações. Neste projeto, foi utilizada a deteção **Haar Cascade** como base, que é um dos algoritmos de deteção de faces mais antigos, mas mais poderosos. Permite não só detetar faces, mas também olhos, lábios e matriculas, por exemplo.

O reconhecimento através do algoritmo **EigenFaces**, tira partido do **PCA**, e oferece um desempenho razoável como classificador.