DM

# Interfaces e Métodos de Pesquisa Visual em Imagens

DISSERTAÇÃO DE MESTRADO

**Diogo Henrique da Silva Cruz**
MESTRADO EM ENGENHARIA INFORMÁTICA

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

dezembro | 2020

# Interfaces e Métodos de Pesquisa Visual em Imagens

DISSERTAÇÃO DE MESTRADO

**Diogo Henrique da Silva Cruz**

MESTRADO EM ENGENHARIA INFORMÁTICA

ORIENTAÇÃO

Pedro Filipe Pereira Campos

CO-ORIENTAÇÃO

Diogo Nuno Crespo Ribeiro Cabral

UNIVERSIDADE da MADEIRA

# Interfaces e Métodos de Pesquisa Visual em Imagens


## Diogo Henrique da Silva Cruz


**Constituição do júri de provas públicas:**

**Presidente:**

- Karolina Baras, (Professora Auxiliar da Universidade da Madeira)

**Arguente:**

- Amâncio Lucas de Sousa Pereira, (Investigador Júnior, ITI/LARSyS, Instituto Superior Técnico, Universidade de Lisboa)

**Vogais:**

- Pedro Filipe Pereira Campos, (Professor Associado com Agregação da Universidade da Madeira)


Novembro 2020

Funchal – Portugal

# Resumo

O objetivo principal deste trabalho foi desenvolver uma aplicação com uma interface gráfica que permita a utilização de métodos de processamento de imagem em pesquisas efetuadas por utilizadores comuns, assim como visualizar os resultados obtidos de forma rápida e eficiente. Isto porque a recente e rápida evolução de técnicas de processamento de imagem e do *hardware* suscitam interesse no estudo da sua aplicabilidade em pesquisa visual de imagens. A par desta evolução, atualmente podem também ser geradas com facilidade uma grande quantidade de imagens, tornando-se necessário desenvolver interfaces para visualizá-las, mas não tem havido muito progresso nos últimos anos. Assim sendo, e como os algoritmos implementados podem ser aplicados a coleções de imagens, foram também estudadas e desenvolvidas interfaces para visualizá-las.

Foi feita uma revisão de literatura extensa que serviu para determinar que métodos seriam implementados, e como inspiração para desenvolver quatro modos de visualização, mais concretamente uma grelha de *thumbnails*, uma grelha de *thumbnails* de tamanho variável, uma pilha de imagens e uma espiral.

Os métodos foram avaliados quanto ao desempenho e qualidade dos resultados. As visualizações foram avaliadas num teste com 9 participantes, em que foram realizadas tarefas de pesquisa geral/específica. Relativamente ao desempenho, todos os métodos foram testados com *CPU*, e os compatíveis com *GPU*. Foram testadas várias configurações de *hardware*. Constatou-se que o desempenho é satisfatório, especialmente com *GPU*. A qualidade dos resultados de alguns métodos ficou aquém dos valores anunciados nas suas publicações, mas foi suficiente para serem úteis e satisfazerem as necessidades da aplicação. Os testes com utilizadores indicaram que as visualizações da mais rápida para a menos são Grelha Normal > Grelha Variável > Espiral > Pilha, sem diferenças significativas entre as grelhas. Constatou-se que a Grelha Normal tem a melhor pontuação *SUS*, seguida da Grelha Variável, Espiral e Pilha. As visualizações da mais útil para a menos são Grelha Variável > Grelha Normal > Pilha > Espiral. Os aspetos mais importantes foram o tempo necessário para encontrar os objetos, a dificuldade de localizá-los e a intuição. Não foram encontradas diferenças significativas de precisão, revocação e *f-measure* em ambos os tipos de tarefas.

## Palavras-chave:

Pesquisa Visual, Interfaces de Pesquisa, Visualização de Imagens, *Feature Matching*, Deteção de Objetos, *Machine Learning*

# Abstract

The main goal of this work was to develop an application with a GUI that allows common users access to complex image processing algorithms, as well as quick and efficient result browsing. This is because the recent and quick evolution of visual search techniques, hardware and its increased accessibility sparked interest in studying what is possible today. Along with this evolution there has been a huge increase in the number of images that are generated, making it necessary to develop new interfaces to visualize them, but there has not been significant progress recently. Because of this, and because the algorithms implemented can be applied to image collections, different interfaces were studied and developed in this work.

The starting point was a literature review that served to determine which methods would be implemented and as inspiration for the development of four interfaces, including a grid of thumbnails, another with varying size thumbnails, a pile of images and a spiral.

The performance and the quality of the results of the methods were evaluated. The visualizations were evaluated in a user test with 9 participants, where they were asked to perform broad/specific search tasks. Regarding the performance, every method was tested with CPU and when supported with GPU, in four different hardware configurations. It was found that the performance satisfies the application's needs, especially when using a GPU. The quality of the results of some methods didn't match the values announced by their authors in their original publications, but it was enough to be fulfil their purpose.

The user tests indicated that the visualizations ordered from fastest to slowest are Regular Grid > Variable Size Grid > Spiral > Pile, with no significative difference between the grids. The Regular Grid got the best SUS score, followed by the Variable Size Grid, Spiral and Pile. The visualizations ordered from most to least useful are Variable Size Grid > Regular Grid > Pile > Spiral. The key aspects were the time required to locate the objects, the difficulty of spotting them and the intuitiveness. Regarding precision, recall and f-measure, no significative differences were found in both types of tasks.

## Keywords:

Visual Search, Search Interfaces, Image Visualization, Feature Matching, Object Detection, Machine Learning

# Agradecimentos

Este trabalho é o culminar de um longo percurso académico e não teria sido possível sem os meus pais e o meu irmão, que me acompanharam desde o início. Como tal, agradeço por terem estado sempre ao meu lado e por me terem ajudado a tornar-me quem sou.

Quero mencionar e agradecer aos amigos mais próximos que conheci ao longo do meu percurso académico, assim como a alguns dos professores que tive, que se destacaram dos restantes pela sua excelência e pelo apoio dado.

Quero agradecer aos meus orientadores, Doutor Diogo Cabral e Doutor Pedro Campos, por todo o suporte e orientação que me deram durante o desenvolvimento desta tese.

Por fim, quero agradecer a todos os participantes dos testes com utilizadores realizados neste trabalho, assim como às pessoas que disponibilizaram os seus computadores para realizar testes de desempenho.

# Índice

# Índice de figuras

# Índice de tabelas

# Lista de acrónimos e de siglas

| | |
|---|---|
| 2D | Duas dimensões |
| AKAZE | *Accelerated-KAZE* |
| ANOVA | *Analysis of Variance* |
| AP | *Average Precision* |
| BRIEF | *Binary Robust Independent Elementary Features* |
| BRISK | *Binary Robust Invariant Scalable Keypoints* |
| CBIR | *Content-Based Image Retrieval* |
| CNN | *Convolutional Neural Network* |
| COCO | *Common Objects in Context* |
| CPU | *Central Processing Unit* |
| ENet | *Efficient Neural Network* |
| FAST | *Features from Accelerated Segment Test* |
| FCN | *Fully Convolutional Network* |
| FPS | *Frames Per Second* |
| GPL | *General Public License* |
| GPU | *Graphics Processing Unit* |
| GUI | *Graphical User Interface* |
| IoU | *Intersection over Union* |
| JSON | *JavaScript Object Notation* |
| kNN | *k-Nearest Neighbors* |
| ORB | *Oriented FAST and Rotated BRIEF* |
| PIL | *Python Imaging Library* |
| RANSAC | *Random Sample Consensus* |
| RCNN | *Region-based Convolutional Neural Networks* |
| SIFT | *Scale-Invariant Feature Transform* |
| SSD | *Single-Shot Detector* |
| SURF | *Speeded-Up Robust Features* |
| VOC | *Visual Object Classes* |
| YOLO | *You Only Look Once* |
| mAP | *Mean Average Precision* |

# 1. Introdução

Este trabalho foca-se na procura de informação visual semelhante em imagens. Esta tarefa pode ser feita automaticamente por computadores, manualmente por humanos, ou combinada de modo em que o computador se torna uma ferramenta auxiliar de pesquisa ao ser humano. A automatização desta tarefa é um problema do domínio da área de Visão por Computador, que procura desenvolver técnicas que permitam que um computador possa analisar imagens e/ou vídeos, e que possa compreender o seu conteúdo (Marr, 2019). As técnicas de Visão por Computador inspiram-se na visão humana, baseando-se em princípios semelhantes para alcançarem os seus objetivos. Uma diferença entre estas são as ferramentas que os humanos e os computadores têm à sua disposição (retinas/sensores, …). Além disto, tipicamente pretende-se que os computadores desenvolvam estas capacidades em muito menos tempo do que os humanos ("IBM - Computer Vision," 2020), apesar de estarem dependentes da capacidade do seu *hardware* e dos algoritmos que executam.

A área de Visão por Computador tem as suas próprias subdivisões, abrangendo problemas como reconhecimento e seguimento de objetos, classificação de objetos e imagens, processamento de imagem, entre outras. Todas estas subdivisões têm diversos casos de aplicação nas indústrias da sociedade contemporânea, o que torna o seu estudo importante e crucial para o progresso da humanidade. Alguns casos de aplicação incluem sistemas de carros autónomos (indústria automóvel) (Eady, 2019), monitorização de plantas (indústria agrícola) (Shimizu and Heins, 1995), sistemas de videovigilância (indústria de defesa) ("SAS - Computer Vision," 2020), diagnósticos médicos (indústria médica) ("SAS - Computer Vision," 2020), classificação de espécies (investigação), entre muitos outros. Esta área surgiu no fim da década de 50, quando engenheiros informáticos decidiram programar computadores para serem capazes de analisar fotografias e identificar características típicas de uma cara humana, com o intuito de determinar se uma pessoa está presente ou não. No entanto, tal não foi possível devido a limitações da tecnologia disponível na altura. Na década de 80 já era possível identificar formas através de métodos matemáticos, possibilitando a identificação de padrões. Nos anos 90, começaram a ser utilizadas redes neurais para aprendizagem e classificação de imagens. Apesar disto, só nos anos 2000 é que o hardware disponível tinha capacidade de processamento suficiente para tornar viável a utilização destas redes (Leone, 2017). Até 2022, estima-se que o mercado de tecnologias de Visão por Computador atinja cerca de 48.6 mil milhões de dólares norte-americanos (Marr, 2019).

Um dos fatores principais por trás do rápido crescimento desta área é a evolução do *hardware* e a sua cada vez maior disponibilidade. Outro fator é a grande quantidade de dados gerados atualmente, particularmente imagens e vídeos. Isto porque, com o avanço dos telemóveis, televisões, carros, entre outros, começaram-se a incorporar câmaras nos mesmos. A utilidade das tecnologias de Visão por Computador para melhorar e modernizar diversas áreas da sociedade é outro fator importante, já que cria uma necessidade de novos desenvolvimentos e de investigação na área ("SAS - Computer Vision," 2020).

Esta rápida e recente evolução da área de Visão por Computador é o principal contribuinte para a motivação deste trabalho, suscitando o interesse no estudo de métodos de pesquisa visual em imagens desenvolvidos ao longo dos últimos anos e dos resultados que

permitem obter com *hardware* recente. No entanto, a pura automatização destes processos levanta questões ao nível da transparência, produção de erros e apresentação de resultados ao utilizador final. Por outro lado, a grande quantidade de imagens geradas atualmente faz com que também seja necessário desenvolver novas interfaces para visualizar grandes coleções de imagens.

Esta necessidade não é algo recente, sendo que a maioria dos motores de busca de imagens permitem pesquisar e visualizar imagens há já muitos anos. Por exemplo, o *Google Images* ("Google Images," n.d.) está disponível desde 2001, o *Flickr* ("Flickr," n.d.) desde 2004, o *Bing Images* ("Bing Images," n.d.) desde 2009, entre muitos outros. No entanto, a interface para visualização de imagens utilizada por estes motores de busca são as típicas grelhas de duas dimensões compostas por miniaturas das imagens (*thumbnails*). Apesar destas grelhas permitirem visualizar grandes quantidades de imagens, por vezes simplesmente o utilizador não consegue encontrar o que procura. Por esta razão, tipicamente os motores de busca permitem especificar filtros para reduzir o número de imagens apresentadas, além de ordenarem os resultados por relevância. Os algoritmos utilizados para classificar e ordenar os resultados são um tópico de interesse na área de *Content-Based Image Retrieval* (*CBIR*). A possibilidade de incluir o utilizador no processo de classificação tem sido muito estudada. Contudo, o que todas estas técnicas têm em comum é que assumem que o utilizador sabe o que está a procurar e/ou que sabe especificá-lo, o que nem sempre é o caso. Além disso, a maioria dos estudos e desenvolvimentos feitos atualmente não se focam nas interfaces, mas sim em melhorar os métodos de pesquisa, e os trabalhos que desenvolvem novos conceitos e interfaces para visualização de coleções de imagens tendem a não estudar a sua usabilidade. Por estas razões, e como os métodos de pesquisa visual que serão abordados ao longo deste trabalho são tipicamente aplicados a coleções de imagens, é importante estudar e desenvolver novas interfaces para visualizá-las.

## 1.1. Problema

No campo do processamento de imagens e vídeo, um problema que se destaca mesmo nos dias de hoje com o *hardware* e *software* que temos ao nosso dispor, pelas limitações que acarreta, é a quantidade de tempo e de recursos computacionais necessários para efetuar um processamento minimamente complexo. Além de se utilizar mais *hardware*, ou *hardware* com mais poder computacional, não é possível eliminar este problema facilmente (Yuille and Liu, 2019). Ainda que o *software* possa ser desenvolvido e otimizado de forma a minimizar os recursos necessários, a exigência de recursos e de tempo é algo inerente ao processamento de imagens e vídeos. Este problema irá ser considerado e estudado no decorrer deste trabalho, já que um dos seus objetivos é estudar a evolução do desempenho das tecnologias utilizadas para procura de informação visual.

Outro problema que assume um papel de destaque atualmente e que é foco de muitas investigações e estudos é regularmente referido por *Content-Based Image Retrieval* (*CBIR*). Este problema abrange o cálculo da relevância de imagens para seleção das que deverão ser apresentadas e/ou em que ordem. A apresentação de coleções de imagens para visualização é outro problema importante, mas que ao longo dos anos não tem sido tão desenvolvido. Um

exemplo disto são os motores de busca de imagens, como o *Google Images* (“Google Images,” n.d.), que após tantos anos continuam a usar as clássicas grelhas 2D com *thumbnails*. Estes problemas serão abordados ao longo desta dissertação.

## 1.2. Objetivo

Um dos objetivos deste trabalho passa por estudar a evolução ao longo dos anos das técnicas de procura de informação visual em imagens, mais especificamente o seu desempenho e a qualidade dos resultados obtidos. Este estudo deve abranger o que é possível fazer atualmente, com métodos de pesquisa visual e *hardware* relativamente recentes.

O estudo das limitações temporais e da possibilidade de processamento na hora é importante, dado que o objetivo principal é desenvolver uma aplicação que permita utilizar estas técnicas em tempo real, sem necessidade de efetuar qualquer pré-processamento. O funcionamento em tempo real permite que uma aplicação seja usada por utilizadores comuns de uma maneira fluida, sem necessidade de intervenções constantes de especialistas que quebram o ritmo de trabalho. De notar que por “tempo real” não se refere à capacidade de fazer todo o processamento sem tempos de espera percetíveis, como por exemplo processar os *frames* de um vídeo a uma *framerate* igual ou superior à do vídeo, mas sim a um processamento que é feito na hora e que não é limitado a exemplos predefinidos. Dito isto, a capacidade de fazer um processamento sem tempos de espera percetíveis também é algo que poderá ser mencionado durante esta dissertação, pelo que deverá ter-se em conta o contexto para diferenciar os dois conceitos. Além de fazer todo o seu processamento em tempo real, a aplicação deverá ser capaz de processar quaisquer imagens providenciadas pelo utilizador, não estando limitada a exemplos predefinidos.

Por último, toda a aplicação deverá ser acessível a utilizadores comuns, incluindo as partes de processamento de imagem e de visualização de resultados. Os métodos de pesquisa visual geralmente são bastante complexos e requerem que o utilizador tenha conhecimentos acerca do seu funcionamento, de processamento de imagem e até de programação, o que impossibilita a sua utilização por um grande grupo de utilizadores. Assim sendo, a aplicação deverá facilitar a utilização destes métodos e deverá permitir visualizar e explorar os seus resultados de forma rápida e eficaz, através de uma interface gráfica adequada.

## 1.3. Contribuições

No contexto desta dissertação foi desenvolvida uma aplicação para *desktop* com uma interface gráfica que permite a utilizadores comuns (sem conhecimentos técnicos específicos) usar diferentes métodos, cinco baseados em *features* visuais e cinco baseados em classificação de objetos, para pesquisa visual em imagens. Além disso, a interface desenvolvida permite cinco visualizações distintas dos resultados obtidos, com o objetivo de estudar a usabilidade, preferência e eficiência destas durante tarefas de pesquisa.

Para tal, começou-se por fazer uma revisão de literatura abrangente das áreas e conceitos relevantes, incluindo sistemas semelhantes que permitem procurar informação visual,

métodos de pesquisa visual e interfaces para pesquisa e visualização de coleções de imagens. Esta revisão permitiu:

- Selecionar os métodos de pesquisa visual a implementar na aplicação a desenvolver
- Obter inspiração para desenvolver a interface e os modos de visualização de resultados.

Após o desenvolvimento estar concluído, foi feita uma avaliação dos aspetos principais do protótipo desenvolvido, aspetos estes que decorrem dos problemas identificados inicialmente e dos objetivos do trabalho.

- O desempenho de cada método de pesquisa visual foi avaliado quanto à sua velocidade de processamento (com *CPU*/*GPU* e com diferentes configurações de *hardware*) e quanto à qualidade dos seus resultados.
- As interfaces para visualização de resultados foram avaliadas num teste com 9 participantes, onde foram estudadas as suas usabilidades e a sua rapidez de uso. Estudaram-se também os valores de precisão e de revocação que permitiram obter em tarefas de pesquisa geral e de pesquisa específica. Foi feita uma análise dos resultados obtidos e foram tiradas conclusões.

## 1.4. Estrutura do documento

Esta dissertação está dividida em cinco capítulos:

- No primeiro capítulo é feita uma introdução ao tema e aos problemas relacionados com este trabalho. Posteriormente são indicados os objetivos, as contribuições e a estrutura do mesmo.
- No segundo capítulo é feita uma revisão de trabalhos relacionados relevantes, abrangendo as áreas de estudo e os conceitos abordados ao longo do trabalho. Esta secção divide-se em três subsecções onde são analisados sistemas semelhantes, métodos de pesquisa visual disponíveis e interfaces para pesquisa e visualização de coleções de imagens.
- No terceiro capítulo são listados os requisitos funcionais e não funcionais levantados, assim como as ferramentas e bibliotecas externas utilizadas. Posteriormente é apresentada a arquitetura da aplicação desenvolvida e é detalhado o desenvolvimento que foi feito. Por fim, é apresentada a interface da aplicação e os diferentes modos de visualização desenvolvidos.
- No quarto capítulo são discutidas as avaliações feitas aos diferentes aspetos deste trabalho, incluindo as metodologias seguidas, os resultados obtidos e análises dos mesmos. Primeiro é apresentada a avaliação da velocidade de processamento de todos os métodos de pesquisa visual implementados. Posteriormente é avaliada a qualidade dos resultados obtidos pelos métodos. Por fim, as interfaces desenvolvidas para visualização de resultados são avaliadas quanto à sua usabilidade, eficácia e rapidez de uso.

- No quinto capítulo são apresentadas todas as conclusões tiradas ao longo do trabalho. Por fim, são indicadas as limitações encontradas e são mencionados alguns aspetos que poderiam ser o foco de trabalhos futuros.

# 2. Trabalho relacionado

Neste capítulo são analisados alguns sistemas que de uma forma ou de outra procuram informação visual em imagens, desde as suas implementações, métodos e técnicas utilizadas, desempenho e até os resultados que obtêm. Para completar a análise de sistemas é feita uma revisão de literatura de técnicas para processamento de imagens, incluindo algoritmos para deteção de objetos, segmentação de imagens, extração e *matching* de pontos de interesse, entre outras. Desta forma, pretende-se determinar as técnicas e métodos disponíveis atualmente mais adequados para a aplicação, estudando como funcionam e analisando o seu desempenho quanto à velocidade de processamento e qualidade de resultados. Por fim, é analisado um estudo sobre as características da pesquisa e *browsing* de imagens, e são analisados alguns trabalhos relacionados onde são apresentadas várias ideias e interfaces para visualização de coleções de imagens.

Com estas análises pretende-se obter conhecimento acerca das abordagens que podem ser seguidas, que limitações acarretam, que obstáculos precisam de ser ultrapassados e como, sendo que o objetivo principal é preparar o desenvolvimento da aplicação.

## 2.1. Sistemas de pesquisa de imagens e vídeo

O *Video Google* (Sivic and Zisserman, 2003) é um sistema semelhante ao que se pretende desenvolver. Este sistema permite selecionar uma região retangular, que idealmente delimita um objeto a ser procurado num *frame* de um filme. Posteriormente, o sistema identifica essa região/objeto em todos os *frames* em que aparece. O *Video Google* pode ser testado numa página *web online*, em (Sivic et al., 2003).

A implementação deste sistema baseia-se em métodos de *information retrieval*, mais especificamente *text retrieval* (Sivic and Zisserman, 2003). Genericamente, métodos de *text retrieval* consistem em utilizar palavras-chave ou uma descrição da informação que o utilizador pretende procurar – *query* - para identificar um conjunto de documentos de texto que são relevantes. Desta forma, para fazer uma pesquisa é necessário uma *query*, uma coleção de documentos e um critério para selecionar os que são relevantes ("University of Illinois - Computer Science," 2014). Um exemplo de um sistema que utiliza métodos de *text retrieval* é o motor de busca *Google* ("Google - How search works," n.d.).

Um dos objetivos dos autores (Sivic et al., 2003) era que o sistema desenvolvido fosse capaz de fazer o seu processamento com a facilidade, rapidez e precisão com que o *Google* encontra páginas *web* relevantes. Comparativamente à pesquisa do *Google*, podemos dizer que a região selecionada corresponde à *query* textual, o conjunto de *frames* do filme corresponde a todas as páginas *web* indexadas pelo *Google* e o processamento feito pelo *Video Google* corresponde aos algoritmos de seleção de resultados da *Google*.

Quanto ao processamento propriamente dito, os autores indicam que a identificação de um objeto numa base de imagens é algo que já tinha atingido uma certa maturidade e que existiam métodos para o fazer. No entanto, afirmam que ainda era um desafio pois a aparência visual de um objeto pode variar consideravelmente dependendo da iluminação, perspetiva e escala. Além disto, também é possível que um objeto se encontre parcialmente

obstruído por outros objetos. Estes problemas serão explorados neste capítulo, já que os métodos de pesquisa a incluir na aplicação terão de ser robustos o suficiente para solucioná-los ou até evitá-los.

Ao contrário do pretendido para este trabalho, o *Video Google* faz um pré-processamento de todos os *frames*, neste caso os *frames* dos filmes disponibilizados para demonstração, de forma a que em *runtime* seja possível encontrar a região procurada e apresentar os *frames* que a contêm sem qualquer tempo de espera significativo. De acordo com (Sivic and Zisserman, 2003), o pré-processamento leva a que qualquer objeto ou grupo de objetos possam ser identificados, ainda que não tenha existido qualquer interesse pelos mesmos quando o pré-processamento foi efetuado. Por alto, o pré-processamento consiste em analisar todos os *frames* e construir um vocabulário visual, que mais tarde é usado como referência para procurar os que contêm o objeto selecionado.

Os autores concluem que esta abordagem inspirada por *text retrieval* demonstrou o seu valor, pois permite identificar objetos mesmo com mudanças significativas de perspetiva. Afirmam também que ainda há margem para melhorias, como por exemplo definir o objeto em mais de um *frame* para permitir efetuar uma pesquisa considerando todos os seus aspetos visuais (Sivic and Zisserman, 2003). Apesar das várias vantagens que o pré-processamento traz, também tem desvantagens. Por exemplo, sempre que pretendemos analisar um conjunto de imagens diferente, os autores indicam que é necessário voltar a fazer um pré-processamento, e apesar de não especificarem o tempo necessário para tal, é provável que seja considerável. De acordo com testes realizados na análise a este sistema, o *Video Google* funciona bem com os exemplos providenciados pelos autores (filme, *frame* e região específica), e às vezes com outros objetos bem destacados do *background*. Segue-se um exemplo sugerido pelos autores.


*Figura 2.1 - Resultados da procura do quadro no frame 106725 do filme "Charade" (Sivic and Zisserman, 2003)*

Contudo, na maioria dos casos testados os resultados foram completamente irrelevantes (exemplo apresentado abaixo), o que sugere que além de assentar no pré-processamento, este método não é robusto o suficiente para lidar com os problemas levantados por diferenças de perspetiva, escala e iluminação. Por estas razões, a abordagem seguida pelo *Video Google* não é adequada para a aplicação a desenvolver.


*Figura 2.2 - Resultados da procura do chapéu no frame 11250 do filme "Dressed to Kill" (não é um exemplo dos autores)*

O *GeoVisual Search* (“GeoVisual Search,” 2017) (Keisler, 2017) é um sistema desenvolvido para demonstração de como se pode recorrer a visão por computador para procurar informação visual semelhante em imagens. A abordagem deste sistema é semelhante à do *Video Google*, na medida em que também constrói um vocabulário visual (Keisler, 2017).

Este sistema trabalha com imagens da Terra, obtidas via satélite. A ideia é dividir a superfície da Terra numa grelha, constituída por pequenas imagens com 128 pixéis de lado. O sistema permite selecionar qualquer uma destas imagens no mapa e retorna uma lista de outras semelhantes (“GeoVisual Search,” 2017) (Keisler, 2017). Por exemplo, ao selecionar uma célula onde está representado um campo de futebol, o sistema retorna uma lista de células que também contêm campos de futebol, assim como as suas localizações na superfície da Terra. Segue-se uma figura que demonstra o funcionamento do sistema.

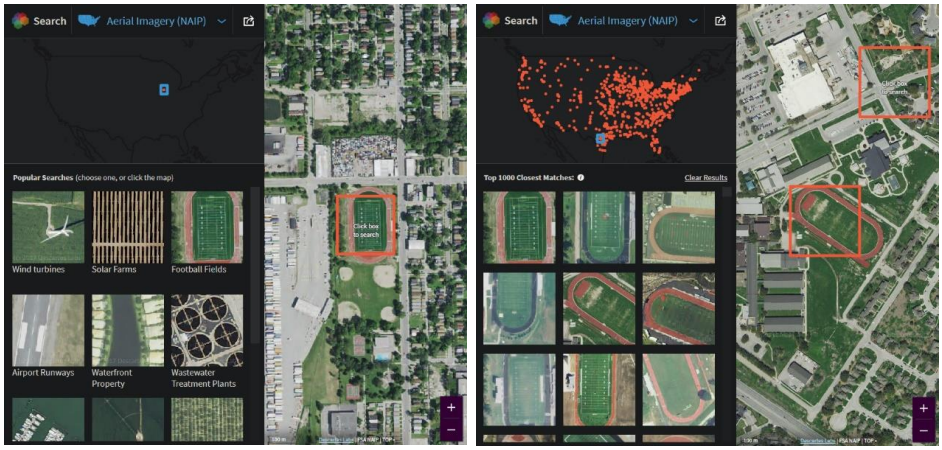


*Figura 2.3 - Exemplo de utilização do GeoVisual Search; Esquerda - Seleção; Direita - Resultados; (“GeoVisual Search,” 2017)*

De acordo com (“GeoVisual Search,” 2017), o *GeoVisual Search* permite procurar imagens do território norte-americano e do resto do mundo. É feita esta distinção entre territórios porque são utilizados *datasets* diferentes para cada um. Para todo o mundo são utilizadas 200

milhões de imagens captadas pelo satélite "Landsat 8", que pertence ao programa de observação terrestre da *NASA* ("NASA - Landsat 8," 2020). A resolução destas imagens é de 20 metros por píxel. Para o território norte-americano são utilizadas 1.9 mil milhões de imagens obtidas pelo programa *NAIP* (*National Agriculture Imagery Program*) ("USDA - NAIP," 2020), com uma resolução de 1 metro por píxel.

Segundo o autor (Keisler, 2017), o repositório visual foi construído recorrendo a dezenas de milhar de processadores da plataforma *Google Cloud*. Este pré-processamento consistiu em extrair *features* (formas, cores, texturas, arestas, …) de cada imagem, em várias regiões do espectro eletromagnético, utilizando uma rede neural.

O autor indica que para procurar uma imagem selecionada pelo utilizador, o sistema calcula a distância visual entre as *features* extraídas da imagem de *query* e as *features* extraídas de cada uma das imagens que compõem a superfície. As imagens apresentadas são as 1000 com menor distância visual relativamente à imagem de *query*. Para determinar que imagens são as mais próximas, são utilizados dois métodos de procura. O primeiro método é utilizado para o *dataset* do território mundial. Este consiste numa procura direta, com recurso a *brute-force*, e permite procurar entre os 200 milhões de imagens em cerca de 2 segundos. Como o *dataset* do território norte-americano é composto por 1.9 mil milhões de imagens, o primeiro método é demasiado lento para permitir uma interação fluida, pelo que é utilizado um segundo método. Este também recorre a *brute-force,* mas utiliza *bit sampling* e *hash functions* para reduzir a lista de procura, sendo capaz de encontrar os resultados em cerca de 0.1 segundos (Keisler, 2017).

De acordo com testes informais realizados no decorrer desta análise, o *GeoVisual Search* permitiu obter sempre resultados relevantes, muito melhores do que os do *Video Google*. Tendo em conta que ambos os sistemas seguem uma abordagem idêntica, e que o *Video Google* foi desenvolvido em 2003 e o *GeoVisual Search* em 2017, podemos assumir que as melhorias se devem, pelo menos em parte, a uma evolução significativa do *hardware* e dos algoritmos de pesquisa visual utilizados.

Apesar do *dataset* utilizado pelo *GeoVisual Search* ser muito interessante e até impressionante, é predefinido e pré-processado. Tal como o *Video Google*, a abordagem seguida por este sistema não é adequada para a aplicação a desenvolver, dado que deve permitir processar na hora quaisquer imagens providenciadas pelo utilizador.

## 2.2. Técnicas de pesquisa visual

A procura de informação visual em imagens pode ser feita de diversas formas e com diferentes níveis de detalhe. A deteção de objetos é uma dessas formas, tendo ganho grande relevo nos últimos anos. Este é um problema que abrange várias subtarefas, como a localização e segmentação de objetos em imagens de profundidade (três dimensões) ("Papers With Code - Computer Vision," n.d.), por exemplo. No entanto, o foco deste trabalho e desta análise vai estar na deteção de objetos em duas dimensões. De mencionar que é claro que existe muita informação visual em imagens que não representa objetos. Contudo, por serem familiares, interessantes, e por geralmente se diferenciarem do *background*, constituem um bom ponto de referência e de partida para o desenvolvimento da aplicação. De seguida, é

feita uma análise a diferentes abordagens que podem ser seguidas para detetar objetos em imagens.

Ao longo dos anos foram desenvolvidos algoritmos de classificação, que indicam que classes de objetos estão presentes numa imagem. Existem também algoritmos de localização, que localizam os objetos na imagem com *bounding boxes*. Para obter resultados mais específicos, estão também disponíveis algoritmos que permitem localizar objetos através de *segmentation/pixel masks*. Além disto, é possível combinar estas diferentes técnicas e obter tanto a classificação como a localização de um objeto, por exemplo. Os métodos de deteção de objetos mais relevantes atualmente produzem resultados bastante completos, incluindo no mínimo a classificação e uma forma de localização. Segue-se um resumo dos diferentes níveis de detalhe com que se pode detetar objetos em imagens de duas dimensões e uma figura que os ilustra (Abdulla, 2018) (Garcia-Garcia et al., 2017):

1. *Classification* – Determina que classes de objetos estão presentes;
2. *Object Localization* – Localiza os objetos na imagem através de *bounding boxes* e identifica as suas classes;
3. *Semantic Segmentation* – Associa cada píxel à classe mais provável que pertença, sendo que tipicamente o píxel é apresentado com a cor que representa essa mesma classe. O resultado final é a segmentação da imagem por classes, sem ser possível distinguir instâncias de objetos.
4. *Instance Segmentation* – Associa todos os pixéis dos objetos detetados às suas classes e é capaz de distinguir objetos da mesma classe.
5. *Panoptic Segmentation* – Segmenta a imagem tal como a técnica *Semantic Segmentation,* mas distingue os objetos como a técnica *Instance Segmentation*.

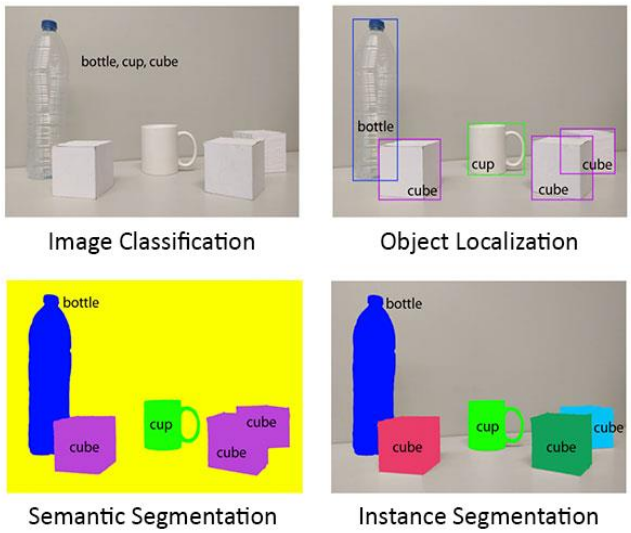


*Figura 2.4 - Diferentes técnicas de deteção de objetos (Garcia-Garcia et al., 2017)*

Tendo em conta o que se pretende da aplicação, as técnicas mais adequadas são *Object Localization*, *Semantic Segmentation*, *Instance Segmentation* e *Panoptic Segmentation*. Contudo, enquanto estas técnicas são muito relevantes atualmente, também há outras abordagens que podem ser seguidas para detetar informação visual semelhante. Por exemplo, a extração e *matching* de pontos de interesse, juntamente com outras técnicas

como a homografia, permite detetar qualquer região de uma imagem e não apenas objetos, pelo que também parece ser adequada.

Como é típico da área de Visão por Computador, há sempre um compromisso entre a qualidade dos resultados e os recursos (temporais e computacionais) necessários para obtê-los. Assim sendo, foram desenvolvidos métodos que fazem um processamento bastante rápido (desde <1 até 8 segundos por imagem), enquanto outros podem fazer um processamento mais demorado (até vários minutos por imagem) mas obter resultados mais precisos e de melhor qualidade. Nas secções seguintes são analisados vários métodos com diferentes compromissos para deteção de objetos e para extração e *matching* de pontos de interesse. De mencionar que os métodos analisados não são necessariamente os mais recentes nem os melhores. Na verdade, desde que esta dissertação foi escrita já foram publicados dezenas de outros modelos *state-of-the-art* para deteção de objetos ("Papers with Code - Object Detection," n.d.), sendo que a maioria deles não são disponibilizados publicamente até algum tempo mais tarde ou nunca chegam a ser, ou então apenas são dadas instruções para treiná-los de raiz. Este último caso está fora do âmbito deste trabalho. Por esta razão, não será possível colocar em prática modelos de deteção de objetos muito recentes. Como tal, os métodos abordados serão alguns dos mais conhecidos e/ou que são mais relevantes para a área, e os métodos implementados estarão limitados aos que tiverem modelos pré-treinados disponíveis publicamente e que sejam suportados pelas bibliotecas externas utilizadas. Na secção seguinte são estudadas algumas das primeiras arquiteturas desenvolvidas para deteção de objetos, que ao longo dos anos e que ainda hoje servem de base para novos modelos.

### 2.2.1. Localização de objetos com bounding boxes

Muitos dos métodos de deteção de objetos que são analisados de seguida recorrem a redes neurais para classificação (*VGGNet*, *ResNet*, *MobileNet*, *DenseNet*, ...), que apenas reconhecem e classificam os objetos numa imagem (Xu, 2017), e combinam-nos com outros componentes para localização de objetos. O resultado é a localização de cada objeto através de uma *bounding box* que o delimita e a sua classificação. De seguida é analisada uma família de modelos que fazem uma abordagem deste tipo.

#### 2.2.1.1. R-CNN, Fast R-CNN, Faster R-CNN, R-FCN

Os métodos *R-CNN* (*Region-Based Convolutional Neural Network*) (Girshick et al., 2014), *Fast R-CNN* (Girshick, 2015), *Faster R-CNN* (Girshick et al., 2016) e *R-FCN* (Dai et al., 2016) pertencem à mesma família e foram todos desenvolvidos para abordar o problema de deteção de objetos, mais especificamente localização de objetos com *bounding boxes*. Todos estes algoritmos recorrem a redes neurais convolucionais como parte das suas *pipelines* de processamento. As redes neurais são modelos computacionais inspirados no cérebro, compostos por conjuntos de nós/neurónios que podem ser organizados de diversas formas. De forma genérica, estes modelos são capazes de reconhecer e aprender padrões em grandes conjuntos de dados (fase de treino), e mais tarde fazer previsões com novos dados (fase de

inferência). Esta capacidade de aprendizagem independente é muito útil para a localização de objetos pois esta é uma tarefa muito complexa e difícil de abordar com programação tradicional. As redes neurais convolucionais são uma subclasse específica destas redes e são frequentemente utilizadas para fazer processamento de imagem.

O *R-CNN* foi um dos primeiros métodos que recorreu a redes neurais convolucionais para abordar o problema de localização de objetos com grande sucesso, obtendo resultados que nessa altura eram *state-of-the-art* (Brownlee, 2019). Segundo (Xu, 2017), este algoritmo é composto por três módulos. O primeiro módulo obtém 2000 propostas de regiões (*bounding boxes*) utilizando um algoritmo conhecido por *Selective Search*. O segundo módulo extrai *features* de cada região proposta utilizando uma rede neural convolucional. O terceiro módulo é responsável por classificar as *features*, determinando a que classe cada região proposta pertence, e por aproximar melhor as dimensões da *bounding box* de cada objeto. Por outras palavras, esta técnica seleciona um conjunto de regiões da imagem onde estima-se que existam objetos e aplica um algoritmo de classificação a cada. De certa forma, transforma a tarefa de deteção em várias subtarefas de classificação (Xu, 2017) (Brownlee, 2019). De seguida encontra-se um diagrama da arquitetura.

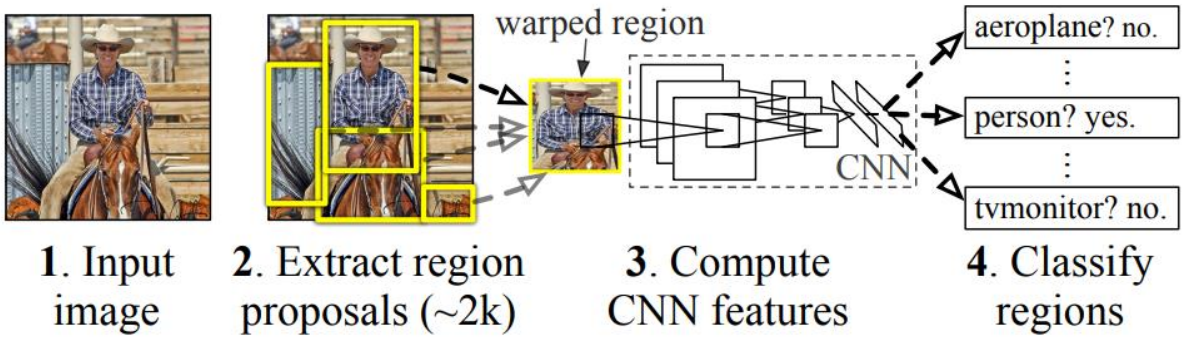


*Figura 2.5 - Resumo da arquitetura do modelo R-CNN (Girshick et al., 2014)*

Apesar desta técnica ser relativamente simples e intuitiva, o processamento é lento e pode rondar os 40-50 segundos por cada imagem a processar, de acordo com (Sharma, 2018). Por essa razão, foram desenvolvidas versões melhoradas e mais rápidas do *R-CNN*. A primeira foi o *Fast R-CNN* (Girshick, 2015), que consiste num único modelo para determinar as regiões e as classificações diretamente (Brownlee, 2019). Comparando com o *R-CNN*, a extração de *features* é feita na imagem original, antes de propor regiões, e o terceiro módulo é substituído por uma camada *softmax* (Xu, 2017). A imagem a analisar passa primeiro por uma *CNN*, que gera mapas convolucionais de *features* e utiliza-os para extração de propostas de regiões. Posteriormente, estas regiões passam por uma camada de *pooling* de regiões de interesse, que modifica as suas dimensões para umas fixas, garantindo que todas as regiões de interesse são do mesmo tamanho. De seguida, cada região de interesse passa por uma camada *fully connected*, assim como por uma camada *softmax* para classificação e uma camada de regressão linear para determinar as coordenadas da *bounding box* (Xu, 2017) (Ouaknine, 2018) (Sharma, 2018) (Brownlee, 2019). A arquitetura e o processamento do *Fast R-CNN* encontram-se resumidos nos diagramas abaixo.

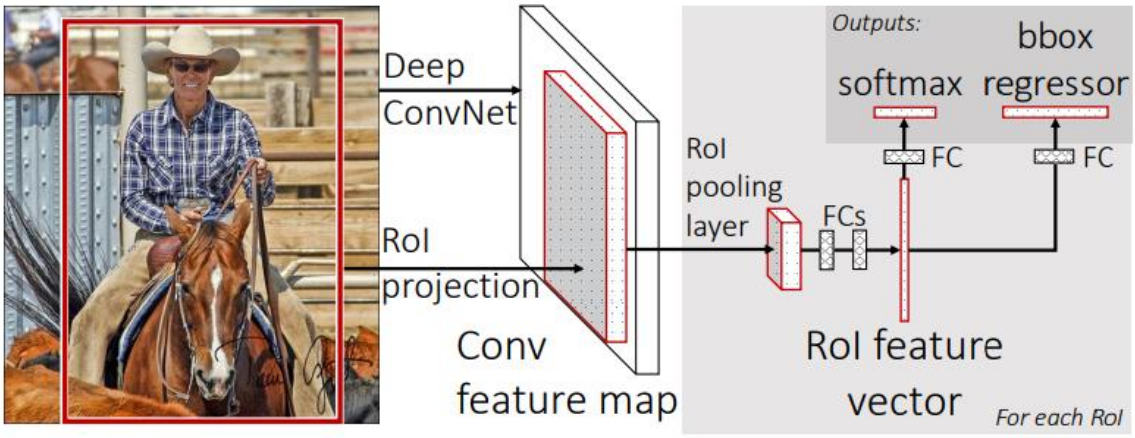


*Figura 2.6 - Arquitetura do Fast R-CNN (Girshick, 2015)*

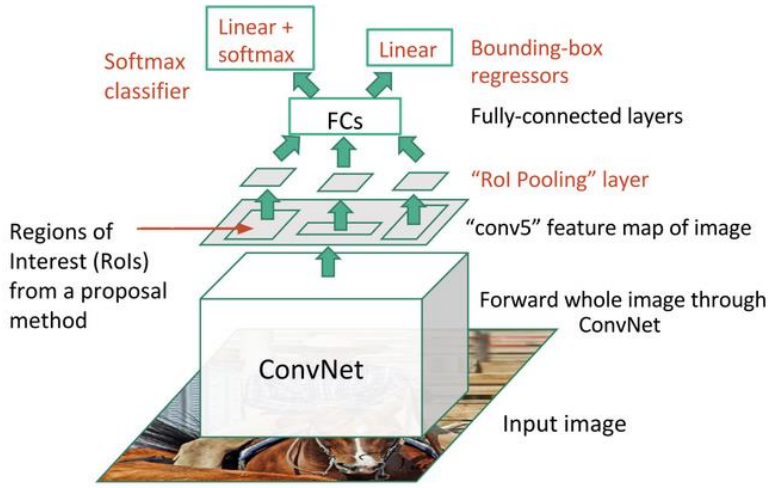


*Figura 2.7 - Processamento efetuado pelo Fast R-CNN (Xu, 2017)*

O desempenho do *Fast R-CNN* é muito melhor do que o do *R-CNN*, sendo que o tempo de processamento é cerca de 2 segundos por imagem, de acordo com (Sharma, 2018). Comparando com o tempo de processamento do *R-CNN*, também reportado pela mesma fonte, o *Fast R-CNN* é cerca de 20-25 vezes mais rápido. Ainda assim, 2 segundos por imagem não é propriamente rápido. Apesar do *Fast R-CNN* resolver muitas das limitações do *R-CNN*, continua a depender do algoritmo *Selective Search* para gerar propostas de regiões de interesse (Xu, 2017).

Em 2016 foi publicado mais um trabalho onde é descrita uma versão mais rápida do *Fast R-CNN*, nomeadamente o *Faster R-CNN* (Girshick et al., 2016). Este também consiste num único modelo unificado, ainda que a arquitetura seja composta por dois módulos. O primeiro módulo consiste numa *CNN*, denominada por *Region Proposal Network* (*RPN*), e veio substituir o algoritmo *Selective Search* na produção de propostas de regiões de interesse. O segundo módulo é essencialmente o *Fast R-CNN*, responsável por extrair *features* das regiões propostas, definindo as suas *bounding boxes* e classes (Brownlee, 2019). O processamento feito pelo *Faster R-CNN* começa por passar a imagem original por uma *CNN*, que retorna mapas de *features* dessa imagem. Posteriormente, a *RPN* é aplicada a estes mapas, passando uma janela deslizante sobre os mesmos. Em cada janela são geradas k *anchor boxes* de diferentes formas e dimensões. Para cada uma, o *RPN* estima a probabilidade de serem um

objeto, atribuindo pontuações, e prevê como ajustá-las para delimitarem melhor o objeto detetado (Xu, 2017) (Ouaknine, 2018) (Sharma, 2018). De seguida encontra-se uma representação do mecanismo de geração de *anchor boxes* para uma determinada janela 3x3, assim como exemplos de regiões propostas obtidas utilizando a *RPN*.

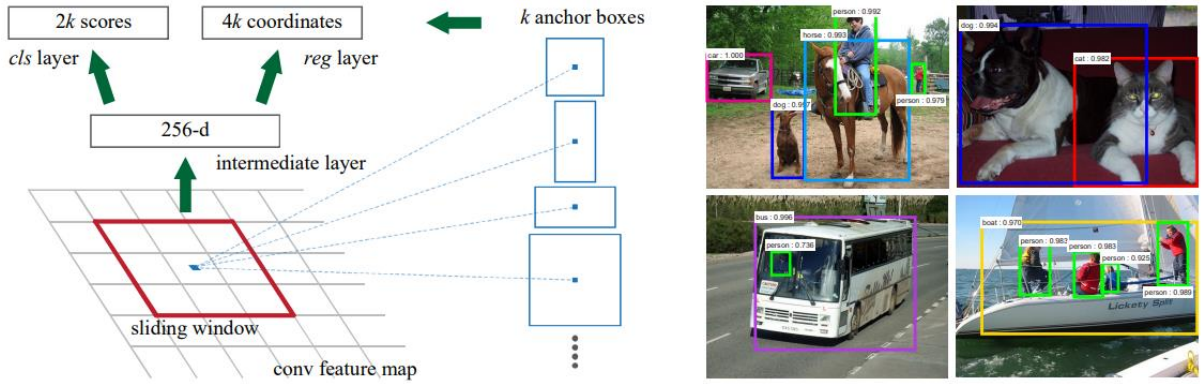


*Figura 2.8 - Anchor boxes para uma determinada janela e exemplos de propostas obtidas com a RPN (Girshick et al., 2016)*

Após serem obtidas, as propostas passam para o segundo módulo, onde seguem um procedimento idêntico ao *Fast R-CNN* (Xu, 2017). A arquitetura do *Faster R-CNN* encontra-se representada na figura seguinte.

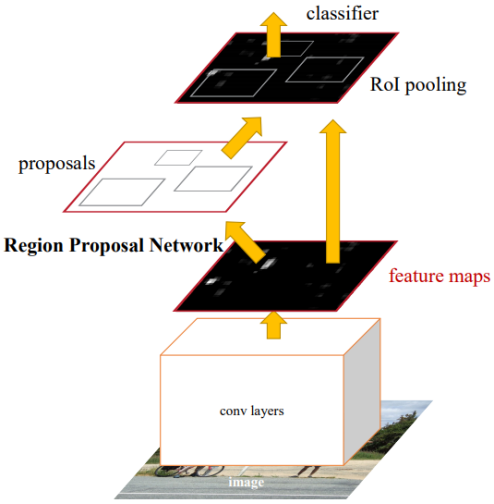


*Figura 2.9 - Arquitetura do Faster R-CNN (Girshick et al., 2016)*

O *Faster R-CNN* elevou a fasquia em termos de velocidade de processamento e precisão. Segundo a fonte (Sharma, 2018), que também reportou os tempos de processamento mencionados anteriormente dos modelos *R-CNN* e *Fast R-CNN*, o *Faster R-CNN* é capaz de processar uma imagem em 0.2 segundos, cerca de dez vezes mais rápido que o *Fast R-CNN* e 200-250 vezes mais rápido que o *R-CNN*. Desde então, vários modelos foram desenvolvidos com o objetivo de obter desempenhos ainda melhores. No entanto, de acordo com (Xu, 2017), muitos deles não eram capazes de superar o *Faster R-CNN* por margens significativas, razão pela qual o *Faster R-CNN* continuava a ser um dos modelos com melhor desempenho até à data (Brownlee, 2019). Ainda assim, o *Faster R-CNN* também tem problemas e limitações resultantes da sua arquitetura, de acordo com (Sharma, 2018). Tal como os outros modelos da família, a *CNN* não processa toda a imagem de uma só vez. Em vez disso, o

processamento é feito sequencialmente, analisando região a região. Como é composto por módulos sequenciais, o desempenho dos módulos seguintes está dependente do desempenho dos módulos anteriores.

Mais tarde, o *Faster R-CNN* serviu também de ponto de partida para o desenvolvimento do *R-FCN* (*Region-Based Fully Convolutional Net*) (Dai et al., 2016), que melhora ainda mais a velocidade de processamento ao maximizar a computação que é partilhada. Para tal, os autores juntaram os dois passos chave do *Fast-RCNN* e *Faster-RCNN* (obtenção de propostas e sucessiva análise para classificação e determinação das *bounding boxes*) num único modelo. No entanto, os autores depararam-se com um problema no desenho do modelo, resultante do facto de a rede utilizada ser *fully convolutional*. Isto porque, esta rede é responsável por classificar os objetos detetados, independentemente da posição onde se encontram na imagem (*location invariance*). Por outro lado, também é responsável por identificar a posição destes objetos na imagem através de *bounding boxes* (*location variance*). A solução para este problema passa por utilizar mapas de *features* (conhecidos por *position-sensitive score maps*), cada um especializado na deteção de uma categoria num determinado local. Para tal, a imagem original é passada como *input* a uma *CNN* (neste caso *ResNet*), que os produz. Uma *RPN* é utilizada em paralelo para gerar regiões de interesse, que posteriormente são divididas em sub-regiões. Cada uma destas sub-regiões é analisada para determinar se contêm alguma parte de um objeto de uma determinada classe. Quando muitas sub-regiões encontram uma correspondência válida então a região é classificada com essa mesma classe (Xu, 2017) (Ouaknine, 2018). Os autores do trabalho onde o *R-FCN* é descrito dão o exemplo seguinte para ilustrar como esta análise é feita.

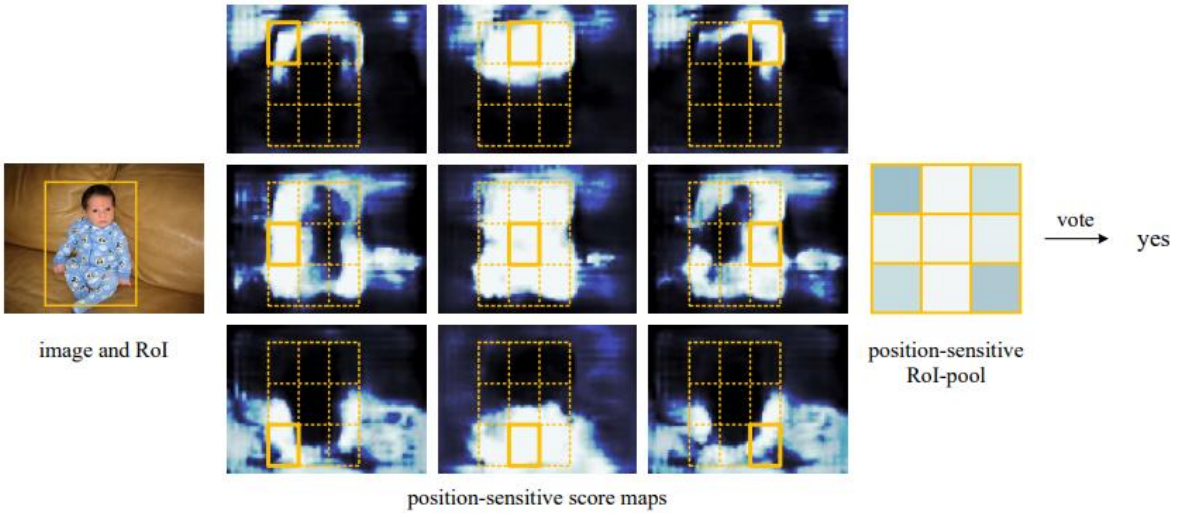


*Figura 2.10 - Análise de sub-regiões (Dai et al., 2016)*

Segundo os mesmos autores, o *R-FCN* é capaz de velocidades 2.5-20 vezes mais rápidas que o *Faster R-CNN*, com uma precisão comparável (Dai et al., 2016) (Xu, 2017). A arquitetura do *R-FCN* é ilustrada pela figura seguinte.

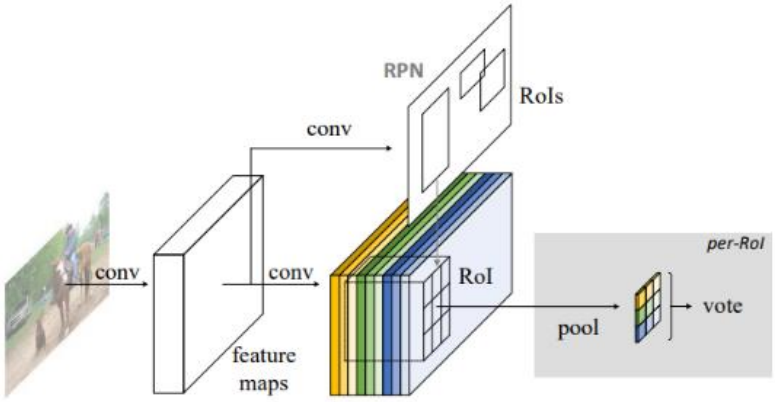


*Figura 2.11 - Arquitetura do R-FCN (Dai et al., 2016)*

Seguem-se alguns exemplos da aplicação destes modelos para deteção de objetos em imagens.

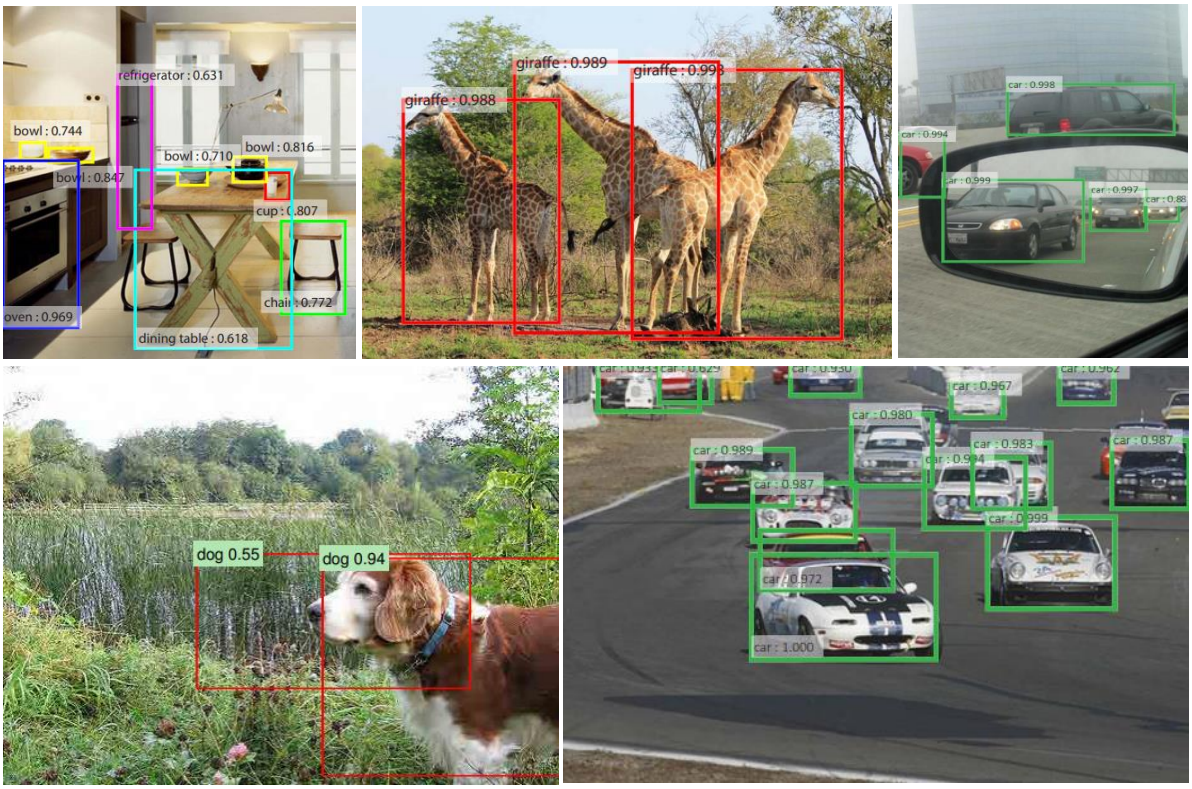


*Figura 2.12 - Exemplos de deteção de objetos utilizando os modelos R-CNN (Girshick et al., 2014) (Girshick et al., 2016) (Dai et al., 2016)*

Nenhum destes modelos foi incluído na aplicação porque não foi possível encontrar modelos pré-treinados disponíveis publicamente e suportados pelas ferramentas de *back-end* utilizadas. Uma alternativa seria treinar modelos de raiz. No entanto, esse é um processo bastante complexo e demorado, e requer muitos recursos computacionais. Como tal, esta alternativa estava fora do âmbito deste trabalho. Em vez disso, foi implementado um outro modelo mais recente desta família, nomeadamente o *Mask R-CNN*. Este é um modelo que deteta objetos através de *bounding boxes* e de *segmentation masks*, pelo que é abordado mais tarde na secção respetiva.

Apesar dos métodos da família *R-CNN* geralmente serem precisos, e apesar de todas as melhorias que os tornaram mais rápidos, o desempenho continua a não satisfazer para aplicações de tempo real, como por exemplo detetar pessoas na transmissão ao vivo de uma câmara de vigilância. Para tais casos de aplicação foram desenvolvidos métodos mais rápidos, que são analisados de seguida.

#### 2.2.1.2. YOLO (You Only Look Once)

Outra família de modelos para deteção de objetos é conhecida por *YOLO*, que atualmente conta com três versões principais (*YOLO*, *YOLOv2*, *YOLOv3*). A primeira versão foi introduzida em (Redmon et al., 2016), onde os autores descrevem como funciona. O acrónimo *YOLO* significa "*You Only Look Once*", precisamente porque todo o processamento é feito numa única fase, ao contrário dos métodos abordados anteriormente, que analisam a imagem original várias vezes, em fases de processamento diferentes. Mais especificamente, os modelos da família *YOLO* abordam a deteção de objetos como um problema de regressão, obtendo as *bounding boxes* e as classificações simultaneamente.

Segundo os autores, esta abordagem traz três grandes vantagens em relação aos métodos tradicionais de deteção de objetos. A primeira vantagem é a velocidade de processamento, que é muito mais alta do que os modelos da família *R-CNN*, chegando mesmo a atingir e a ultrapassar níveis de desempenho em tempo real. Os autores indicam que a primeira versão do *YOLO* é capaz de processar cerca de 45 imagens por segundo com recurso à placa gráfica GTX Titan X, da NVIDIA ("GTX TITAN X Specs," 2020). Mencionam também uma outra versão mais leve e otimizada para velocidade, conhecida como *Fast YOLO*. Apesar de não permitir obter resultados tão precisos, os autores indicam que é capaz de processar 155 imagens por segundo com o mesmo *hardware*.

A segunda vantagem está relacionada com a quantidade de deteções falsas que são feitas no *background* da imagem. Os modelos *YOLO* utilizam *features* de toda a imagem para prever cada *bounding box*, tendo em conta o contexto dos objetos na imagem. Por esta razão, detetam menos objetos no *background* incorretamente do que os modelos da família *R-CNN* (menos de metade quando comparados com o *Fast R-CNN*).

A terceira vantagem decorre do facto de que os modelos *YOLO* aprendem representações genéricas dos objetos. Quando treinado com imagens naturais e obras de arte, superam outros métodos como o *R-CNN* por uma margem significativa. Por serem altamente generalizáveis, os modelos *YOLO* são mais robustos quando aplicados a novos domínios e quando as imagens para análise são desconhecidas.

Contudo, estes modelos também têm desvantagens e limitações. Os autores indicam que os modelos *YOLO* são geralmente menos precisos do que outros métodos de deteção de objetos, especialmente no que toca a detetar objetos de pequenas dimensões.

O processamento feito pelas primeiras versões do *YOLO* é simples. Primeiro é recebida a imagem para análise, que é dividida numa grelha de células. Para cada célula são previstas um determinado número de *bounding boxes*, cada uma com uma confiança associada, e são determinadas as probabilidades de cada classe. Por fim, estas informações são combinadas para obter as *bounding boxes* e as classificações finais. A diferença entre a versão normal e a

rápida é a configuração das camadas da rede. Os autores resumem todo o processo através do diagrama da figura seguinte.

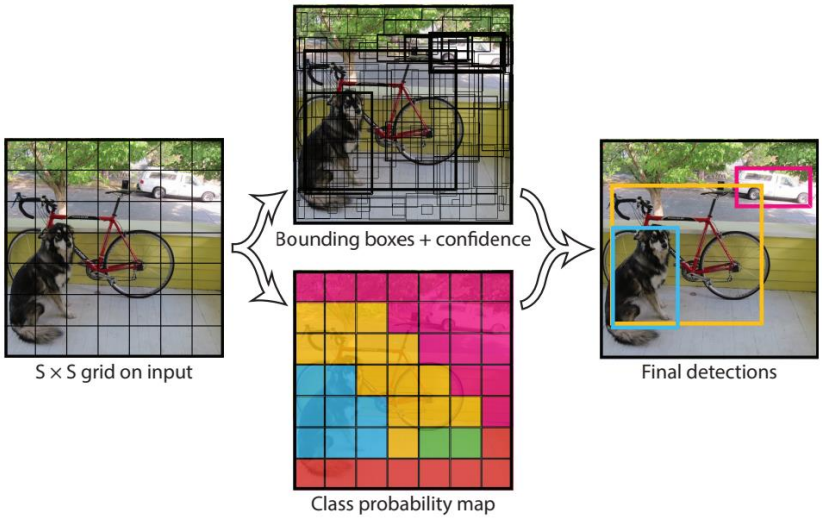


*Figura 2.13 - Processamento efetuado pelas primeiras versões do YOLO (Redmon et al., 2016)*

No fim de 2016 foi submetido um trabalho onde é descrita uma atualização ao modelo *YOLO* original, com o objetivo de melhorar o desempenho (Redmon and Farhadi, 2016). Este trabalho introduz o modelo *YOLOv2*, assim como uma versão conhecida por *YOLO9000*. Esta última foi treinada com dois *datasets* diferentes e é capaz de detetar objetos de 9000 classes. Os autores indicam que as melhorias vão de encontro às limitações da primeira versão do *YOLO*, como a falta de precisão ao localizar objetos em alguns casos e a baixa revocação comparativamente aos modelos baseados em regiões, mas que o processamento tem de continuar a ser rápido. Por essa razão, o *YOLOv2* foi desenhado com base numa rede neural mais simples e as representações foram melhoradas de forma a facilitar a aprendizagem (Redmon and Farhadi, 2016). Além disto, foram feitas algumas alterações à arquitetura do modelo, tais como o uso de *batch normalization*, imagens para análise de alta resolução, *anchor boxes* semelhantes às utilizadas pelo *Faster R-CNN*, entre outras. Estas alterações fazem com que o *YOLOv2* seja mais rápido que outros sistemas de deteção em vários *datasets*, além de poder ser aplicado a imagens de diferentes tamanhos (com um compromisso entre velocidade e precisão) (Redmon and Farhadi, 2016).

Desde então, os mesmos autores submeteram uma publicação onde descrevem o *YOLOv3* (Redmon and Farhadi, 2018). Este modelo conta com algumas pequenas melhorias, mais especificamente na configuração da rede e na representação. Ao contrário das versões anteriores, o *YOLOv3* não tem necessariamente dificuldades com objetos pequenos. Por outro lado, o seu desempenho relativamente a objetos de tamanho médio e grande é pior. Seguem-se alguns exemplos da aplicação dos modelos *YOLO*.

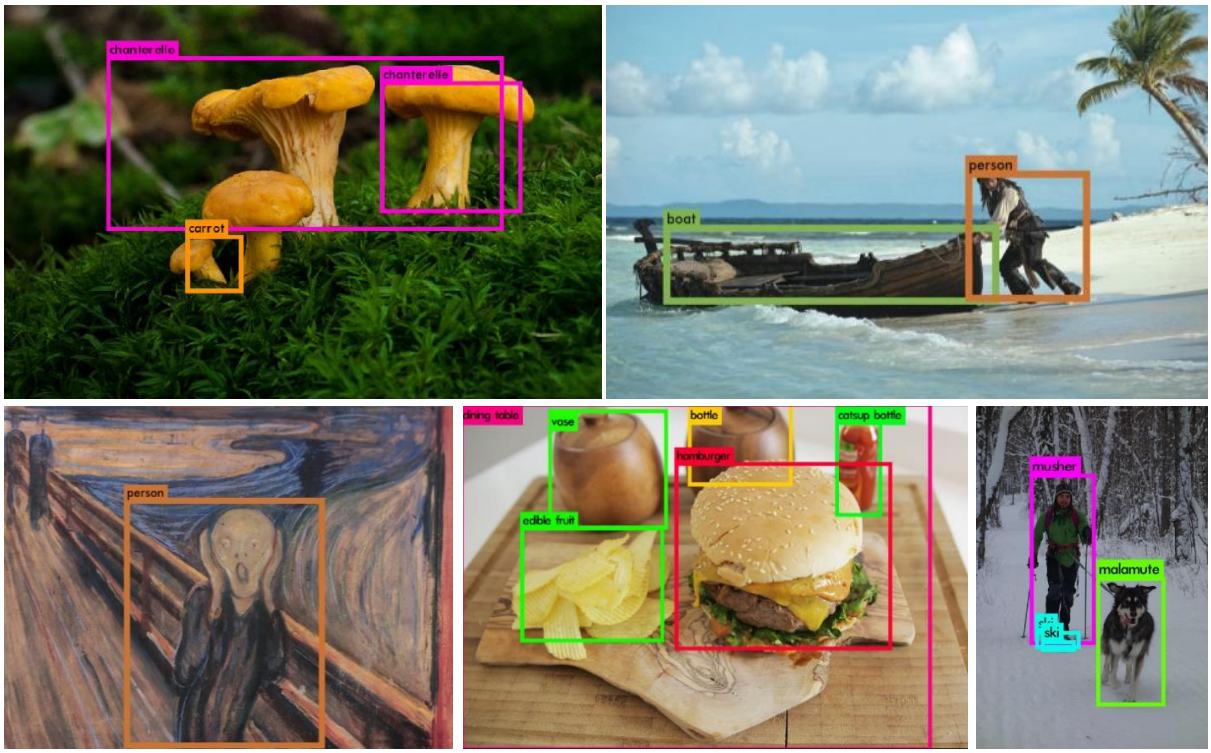


*Figura 2.14 - Exemplos de deteção de objetos com os modelos YOLO (Redmon et al., 2016) (Redmon and Farhadi, 2016)*

Os modelos *YOLO* são métodos de deteção de objetos muito rápidos, mas em contrapartida as *bounding boxes* não são sempre muito precisas. Observando as figuras acima verifica-se que é possível detetar objetos em obras de arte (filme e pintura), assim como objetos de muitas classes diferentes (vaso, fruto comestível, embalagem de *ketchup*, cão malamute, …) (*YOLO9000*). Por outro lado, é evidente que as *bounding boxes* não são muito precisas e que por vezes, quando dois ou mais objetos encontram-se próximos, a qualidade baixa bastante. De qualquer forma, o *YOLOv3* foi selecionado para ser implementado na aplicação por ser a versão mais recente e melhorada desta família de modelos. O *YOLO9000* também parecia ser uma boa escolha, devido à grande quantidade de objetos diferentes que deteta. Contudo, não foi possível encontrar um modelo pré-treinado disponível publicamente, e treinar um modelo de raiz para detetar tantas classes está fora do âmbito deste trabalho.

### 2.2.1.3. Single Shot Detector (SSD)

Tal como os modelos *YOLO*, o *SSD* é um método de deteção de objetos que analisa a imagem e prevê todas as *bounding boxes* e as *classes* de uma só vez (Liu et al., 2016). Os autores afirmam que o *SSD* é capaz de superar a primeira versão do *YOLO* tanto em velocidade como em precisão. O processamento é semelhante ao do *YOLO*, na medida em que recorre a uma rede neural convolucional. No entanto, a configuração da rede é diferente, sendo que o *SSD* utiliza mais camadas e mapas de *features* para obter mais *bounding boxes* relevantes (Liu et al., 2016). A figura seguinte ilustra as diferenças entre as arquiteturas do *SSD* e do *YOLO*.

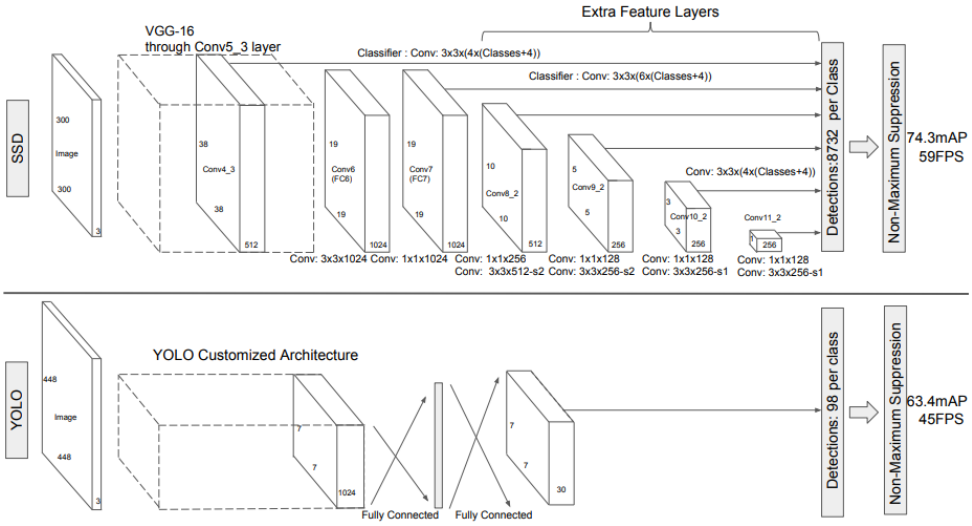


*Figura 2.15 - Comparação entres as arquiteturas do SSD e da primeira versão do YOLO (Liu et al., 2016)*

Os autores incluem vários exemplos da aplicação do *SSD* para deteção de objetos em imagens do *dataset COCO* ("COCO Dataset," 2020). Estes resultados parecem bastante bons, com revocação alta, classificações corretas e *bounding boxes* precisas. Seguem-se alguns desses mesmos exemplos para ilustrar o desempenho do *SSD*.

*Figura 2.16 - Exemplos de deteção de objetos com o SSD (Liu et al., 2016)*

O *SSD* foi selecionado para ser implementado na aplicação devido à sua velocidade de processamento. Além disto, poderá ser interessante compará-lo com o *YOLOv3*, que também foi selecionado, dado que ambos se baseiam no conceito de fazer todo o processamento de uma só vez.

Ao longo desta análise foram estudados vários modelos para localizar objetos através de *bounding boxes*. Contudo, o desempenho dos modelos em termos de velocidade e de qualidade das deteções não foi analisado em termos absolutos até agora. Isto porque o desempenho está dependente de muitas variáveis, tais como os *datasets* utilizados (treino e teste), o *hardware* utilizado, as métricas avaliadas, etc.

Os números anunciados pelos autores dos vários modelos normalmente estão associados aos *datasets* utilizados. Isto deve-se ao facto de que ao longo dos anos foram organizados diversos desafios competitivos para deteção de objetos, e tipicamente os *datasets* utilizados para avaliar os métodos concorrentes eram predefinidos. Desde então, estes *datasets* continuaram a ser utilizados para avaliar novos modelos desenvolvidos, incluindo por autores de alguns dos trabalhos analisados.

As métricas consideradas nestes desafios não são sempre as mesmas. Enquanto um desafio pode considerar uma deteção como correta quando a sua *bounding box* está 90% alinhada com a *bounding box* verdadeira, outro pode exigir 95%. Desta forma, é comum cada *dataset* ser associado a determinadas métricas. Apesar de toda esta variabilidade, algumas métricas são sempre relevantes e adequadas, tais como a precisão (percentagem de estimativas corretas), a revocação (quantas estimativas corretas obtemos de todas as possíveis) e a IoU (*Intersection over Union*) (Hui, 2019). A IoU consiste no quociente entre: a área de interseção da estimativa com a *bounding box* verdadeira; a área de união da estimativa com a *bounding box* verdadeira (Hui, 2019) (Rosebrock, 2016). Esta métrica é compreendida melhor com um exemplo, como o que se segue.

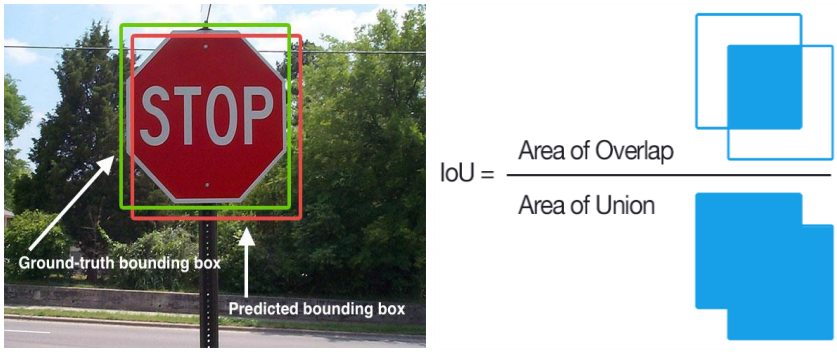


*Figura 2.17 - Representação visual da métrica IoU (Rosebrock, 2016)*

Uma outra métrica bastante utilizada é a *mAP* (*mean average precision*). Segundo (Hulstaert, 2018), o cálculo desta métrica é feito da seguinte forma: Uma pontuação é associada a cada *bounding box*. Baseado nas estimativas obtidas, uma curva precisão-revocação (*PR – precision-recall curve*) é calculada para cada classe. A área por baixo da curva é a precisão média (*AP – average precision*) dessa classe. Por fim, a *mAP* é obtida ao fazer a média de *AP* de todas as classes e é representada por um número de 0 a 100, sendo que tipicamente também é associada a uma *IoU*. Estas métricas são analisadas em mais detalhe na secção de avaliação, onde são utilizadas para avaliar os métodos implementados na aplicação.

De seguida, é apresentada uma tabela onde é listado o desempenho dos diferentes métodos juntamente com o *dataset* utilizado. A métrica utilizada é a *mAP*. Contudo, não é indicado o *hardware* utilizado para cada um dos testes. Esta tabela é adaptada das tabelas disponíveis no artigo (Ouaknine, 2018). Este artigo foi publicado em 2018, antes da publicação do *YOLOv3*, pelo que não inclui os seus resultados. No entanto, no trabalho onde o *YOLOv3* é descrito (Redmon and Farhadi, 2018), os autores afirmam que está ao nível das variantes *SSD* quanto à *mAP*, mas que é três vezes mais rápido.

| | PASCAL VOC 2007 | PASCAL VOC 2010 | PASCAL VOC 2012 | COCO 2015 (IoU=0.5) | COCO 2015 (IoU=0.75) | COCO 2015 (Official Metric) | Real Time Speed |
|---|---|---|---|---|---|---|---|
| R-CNN | - | 62.4% | - | - | - | - | Não |
| Fast R-CNN | 70.0% | 68.8% | 68.4% | - | - | - | Não |
| Faster R-CNN | 78.8% | - | 75.9% | - | - | - | Não |
| R-FCN | 82.0% | - | - | 53.2% | - | 31.5% | Não |
| YOLO | 63.4% | - | 57.9% | - | - | - | Sim |
| Fast YOLO | 52.7% | - | - | - | - | - | Sim |
| YOLOv2 | 78.6% | - | - | 44.0% | 19.2% | 21.6% | Sim |
| SSD | 83.2% | - | 82.2% | 48.5% | 30.3% | 31.5% | Não |

*Tabela 2.1 - Desempenho de alguns dos métodos abordados (mAP) (adaptado de (Ouaknine, 2018))*

Da análise feita a estes modelos pode-se concluir que é possível localizar objetos em imagens através de *bounding boxes* com qualidades e velocidades de processamento variadas, dependendo dos recursos computacionais e temporais disponíveis, e do modelo utilizado. Como tal, não há um único modelo que se possa considerar como o melhor absoluto. Pelo contrário, o melhor modelo a utilizar poderá depender do caso de aplicação em questão, pelo que devem ser avaliadas as necessidades temporais, de qualidade, rigor e precisão. Os modelos da família *R-CNN* permitem obter resultados mais precisos, enquanto os modelos da família *YOLO* trocam um pouco de qualidade por velocidades mais altas. O modelo *SSD* pode ser visto como um meio termo entre estas duas famílias, oferecendo geralmente velocidades mais altas que os modelos *R-CNN* e qualidade igual ou superior aos modelos *YOLO*. De mencionar que estes dados de desempenho dependem muito do treino que é feito para obter cada modelo, sendo que o desempenho poderá variar bastante mesmo entre dois modelos com a mesma arquitetura. É expectável que os métodos implementados na aplicação não sejam capazes de igualar o desempenho anunciado pelos seus autores.

### 2.2.1.4. Extração e correspondência de features visuais + Homografia

Uma possível forma de interação com a aplicação pode ser selecionar uma região de uma imagem de referência para procura num conjunto de imagens para análise. Dependendo do conteúdo da imagem, é possível que o utilizador selecione uma região que não contenha nenhum objeto, ou caso contenha não esteja bem delimitado pela seleção. No entanto, todas as técnicas analisadas até agora focam-se na deteção de objetos como um todo. Assim sendo, caso o utilizador pretenda procurar uma região que não representa um objeto ou que não o representa na sua totalidade, torna-se bastante complicado obter resultados relevantes. No pior caso a região selecionada não representa nenhum objeto, logo é impossível que qualquer método de deteção de objetos encontre algo relevante. Num caso menos mau, em que a região representa apenas parte de um objeto, pode ser possível determinar a classe desse objeto e procurar objetos da mesma classe nas imagens para análise. Além destas situações, podem existir complicações devido a diferenças de escala, luminosidade e rotação entre a imagem de referência e as imagens para análise, por exemplo. Mais ainda, os modelos analisados são capazes de detetar apenas um determinado número de classes de objetos,

pelo que poderia acontecer o utilizador pretender procurar um objeto que não é conhecido. Neste caso, seria necessário treinar um novo modelo que fosse capaz de detetar esse objeto. Este treino seria essencialmente uma fase de pré-processamento, e como pode ser um processo bastante demorado, inviabilizaria a utilização em tempo real.

Os métodos de extração e correspondência de *features* visuais (*feature matching*) permitem contornar todos estes problemas, dado que não são exclusivos para deteção de objetos. Em vez disso, fazem a extração e correspondência de pontos relevantes entre imagens. Por esta razão são capazes de detetar qualquer coisa sem treino prévio, mesmo que não seja um objeto (por exemplo texto ou um logótipo). Contudo, enquanto os modelos de deteção de objetos conseguem identificar que objetos foram detetados, os métodos de *feature matching* não obtêm esse conhecimento. Por esta razão, poderá ser difícil encontrar regiões que representam a mesma coisa, mas que tenham aspetos diferentes.
Segue-se um diagrama que apresenta um exemplo de como estes métodos podem ser aplicados para localizar regiões de interesse em imagens.

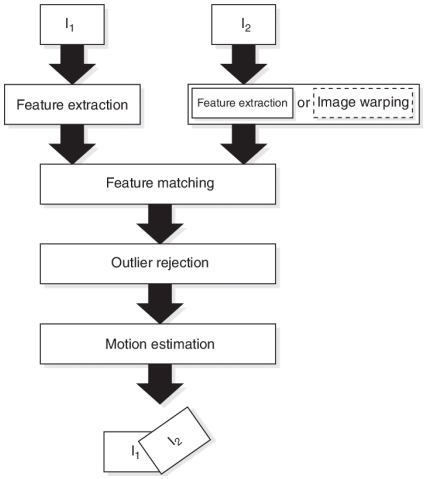


*Figura 2.18 - Exemplo de aplicação de feature matching (Gracias et al., 2017)*

No primeiro passo são extraídas *features* de uma imagem de referência (do objeto/região a detetar) e de uma imagem para análise (onde o objeto/região devem ser procurados). De notar que também é possível extrair *features* apenas de uma imagem, tornando-se necessário fazer um processamento adicional no segundo passo. Uma *feature* representa características que se destacam, por exemplo pelo contraste ou pela saliência. As *features* podem ser detetadas em localizações específicas da imagem, tais como pontos e arestas de objetos, e também podem representar outros pontos que são de interesse pela sua orientação ou aparência local (Tyagi, 2020). Idealmente uma *feature* deve ser robusta e identificável, de forma a que perdure mesmo na presença de ruído, *blur,* diferenças de perspetiva, iluminação, escala, obstruções (parciais), entre outros.

Este primeiro passo divide-se em duas tarefas, nomeadamente a deteção de pontos de interesse e a sua descrição. A deteção consiste em identificá-los na imagem (onde se localizam), enquanto que a descrição consiste em descrever a aparência local de cada ponto, de forma a que não seja suscetível a mudanças de iluminação, escala, rotação, etc. Os pontos de interesse são tipicamente representados por estruturas de dados próprias (*keypoints*), e as suas descrições por vetores (*descriptors*) (Tyagi, 2020) (Jakubovic and Velagic, 2018).

Ao longo dos anos, vários algoritmos para extração de *features* foram desenvolvidos. Entre estes, destacam-se o *SIFT* (Lowe, 2004) e o *SURF* (Bay et al., 2006) pela sua robustez relativamente a transformações (rotação, escala, …), e o *BRISK* (Leutenegger et al., 2011) e o *ORB* (Rublee et al., 2011) pela sua rapidez de execução. Esta diferença deve-se, em parte, à forma como estes algoritmos calculam as distâncias entre *descriptors*, uma vez que os dois primeiros utilizam distância euclidiana e os segundos utilizam distância de *Hamming*. Outra diferença significativa resulta da deteção de *features*. Os dois primeiros têm um tempo de execução mais demorado e tendem a detetar um menor número de *features,* mas estas encontram-se mais dispersas pela imagem. O facto de se obterem *features* de diversas regiões da imagem faz com que possivelmente, na fase de *matching,* seja mais provável ter-se informação acerca da região onde o objeto procurado se encontra. Por outro lado, os últimos dois têm um tempo de execução mais rápido e detetam um grande número de *features*. Contudo, muitas destas encontram-se concentradas nas mesmas regiões da imagem, logo poderão ser redundantes e menos úteis. Ainda assim, todos eles destacam-se pela sua capacidade tanto de detetar como de descrever *features* (Jakubovic and Velagic, 2018). De notar que existem outros algoritmos que se destacaram. Alguns de deteção são o *Harris Corner* (Harris and Stephens, 1988) e o *FAST* (Rosten and Drummond, 2006), e outro de descrição é o *BRIEF* (Calonder et al., 2010). Na verdade, estes três algoritmos são utilizados no *ORB*, ainda que com algumas modificações (Karami et al., n.d.).

O segundo passo depende se foram extraídas *features* de duas imagens ou de apenas uma. Caso tenham sido extraídas de apenas uma, deverá ser feita uma estimativa da transformação necessária e que deverá ser aplicada, de forma a ser possível identificar essas mesmas *features* na segunda imagem, através de métodos como correlação cruzada (*cross-correlation*) ou erro quadrático médio (*sum of squared differences*). Caso tenham sido extraídas de cada uma das imagens deverá ser feita a correspondência (*matching*) dos *descriptors* de ambas (Gracias et al., 2017). Este processo pode ser feito recorrendo a *brute-force*, por exemplo, e pode utilizar distâncias euclidianas ou de *Hamming*, dependendo de como os *descriptors* foram obtidos.

Posteriormente podem se eliminar os *outliers*, sendo que dois algoritmos frequentemente utilizados para tal são conhecidos por "*k-Nearest Neighbors*" (kNN) (Jakubovic and Velagic, 2018) e "*Random sample consensus*" (RANSAC). Segue-se um exemplo de utilização do *ORB*, em que à esquerda está a imagem de referência e à direita a imagem para análise. É possível observar as *features*/*keypoints* correspondidos, com cada par ligado por uma linha. As imagens utilizadas são exemplos providenciados pelo *OpenCV* e estão disponíveis *online* em ("OpenCV 3.0," 2015).

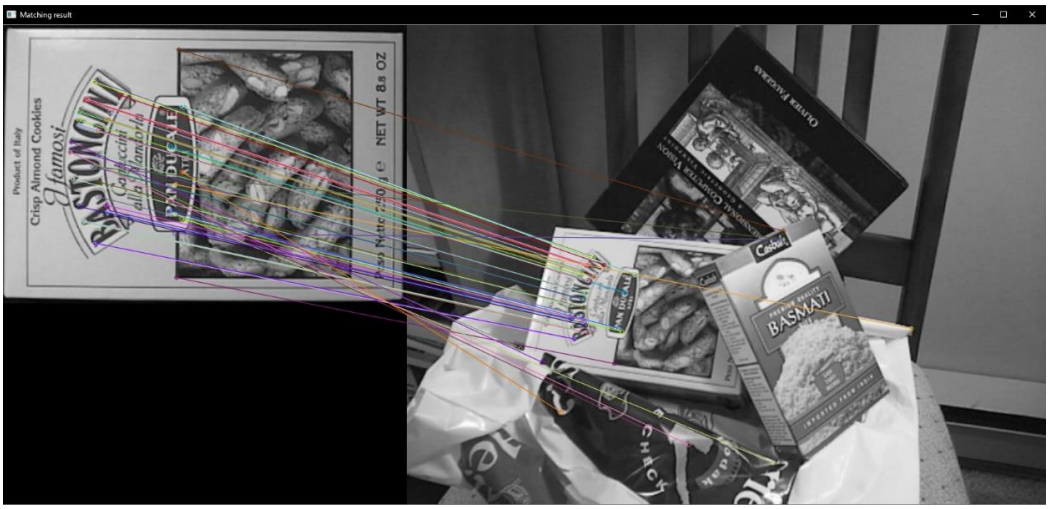


*Figura 2.19 - Feature matching utilizando ORB*

Para que se possa utilizar *feature matching* para localizar objetos e/ou regiões, é necessário que se possa obter uma estimativa de onde estes se encontram na imagem para análise. No entanto, à exceção do caso em que as imagens são exatamente iguais e em que as regiões correspondentes encontram-se no mesmo sítio e com a mesma disposição, existem diferenças de perspetiva entre as imagens resultantes de diferenças de posição, escala e rotação, por exemplo. Por essa razão, e por pretendermos obter uma estimativa (*bounding box*) na imagem de análise, é necessário calcularmos a transformação que mapeia os pontos correspondentes da imagem de referência na de análise. Esta transformação é conhecida por Homografia e consiste numa matriz de dimensão 3x3 (Mallick, 2016) ("OpenCV Python Docs," 2020). Posteriormente, recorre-se à aplicação de uma transformação de perspetiva para traduzir os pontos do espaço para o plano, e para que se obtenham os pontos que definem a *bounding box* na imagem de análise. Por último, resta desenhar a *bounding box*.

Esta técnica permite obter resultados bastante bons, como se pode observar na figura que se segue. O código para fazer esta estimativa encontra-se disponível na documentação do *OpenCV* para *Python*, em ("OpenCV Python Docs," 2020).


*Figura 2.20 - Feature matching utilizando ORB +* homografia *para destacar o objeto detetado*

Como podemos observar, a imagem de referência utilizada neste exemplo não possuía *background*. Por esta razão, todos os *keypoints* detetados encontram-se no objeto de interesse, o que ajudou a obter bons resultados. Podemos também verificar que apesar do objeto encontrar-se parcialmente obstruído, a estimativa foi correta e foi possível delineá-lo. Por esta razão, esta técnica para obter estimativas parece promissora e poderá ser considerada para a aplicação a desenvolver. Contudo, a possibilidade de processar quaisquer imagens na hora é um aspeto fundamental deste trabalho, logo é necessário analisar o desempenho e a robustez destas técnicas.

Nos trabalhos (Jakubovic and Velagic, 2018), (Karami et al., n.d.) e (Tareen and Saleem, 2018) é feita uma análise dos algoritmos *SIFT*, *SURF*, *ORB*, *BRISK*, *KAZE* e *AKAZE*, comparando os seus desempenhos (tempo de execução, número de *keypoints*, número de *matches*, taxa de sucesso, …). No primeiro trabalho (Jakubovic and Velagic, 2018) foram utilizadas as mesmas imagens de referência e de análise dos exemplos acima, com resoluções 324x223 e 512x384, respetivamente. Os resultados obtidos pelos autores encontram-se de seguida.

| | *SIFT* | *SURF* | *ORB* | *BRISK* |
|---|---|---|---|---|
| Número de *matches* | 94 | 119 | 168 | 70 |
| Tempo de execução (s) | 0.68 | 0.413 | 0.568 | 1.128 |
| Erro médio de *matching* | 166.206 | 0.211 | 42.601 | 84.157 |

*Tabela 2.2 - Resultados da aplicação de feature matching à versão original da imagem da figura 2.20 (Jakubovic and Velagic, 2018)*

Neste exemplo, o *SURF* foi o algoritmo que teve o menor erro médio de *matching*. Além disto, o *SURF* e o *ORB* foram os mais rápidos, tendo em conta os seus números de *matches* e os seus tempos de execução. O *BRISK* foi o que obteve menos *matches* e o que demorou mais tempo. Todos os algoritmos foram capazes de detetar o objeto de forma satisfatória. Num segundo exemplo, a imagem de referência representava a palavra "*see*" e a imagem para análise representava uma nuvem de palavras, de resoluções 101x41 e 1662x786, respetivamente. Segundo os autores, este exemplo é mais exigente devido à abundância de letras 's' e 'e' na imagem para análise. Os resultados encontram-se na tabela seguinte.

| | *SIFT* | *SURF* | *ORB* | *BRISK* |
|---|---|---|---|---|
| Número de *matches* | 16 | 30 | 2 | 4 |
| Tempo de execução (s) | 2.674 | 2.51 | 2.113 | 1.489 |
| Erro médio de *matching* | 68.364 | 0.052 | 0 | 50.75 |

*Tabela 2.3 - Resultados da aplicação de feature matching à imagem de uma word cloud (Jakubovic and Velagic, 2018)*

O *ORB* obteve apenas 2 *matches* relevantes, e tendo em conta que para calcular a homografia são necessários pelo menos 4, não foi possível fazer uma estimativa da localização. O *BRISK* obteve 4 *matches* (1 relevante), mas não foi possível obter uma estimativa correta. Por outro lado, os algoritmos *SIFT* e *SURF* obtiveram 16 *matches* (6 relevantes) e 30 *matches* (13 relevantes), respetivamente, e foi possível detetar e delimitar a

palavra corretamente. Os autores concluem que nos exemplos testados o *SURF* foi o mais eficiente e o mais preciso.

No segundo trabalho (Karami et al., n.d.), os autores investigam a sensibilidade do *SIFT*, *SURF* e *ORB* a diferentes intensidades, rotações, escalas, distorções, *shearing* e ruído. O *ORB* foi o algoritmo com o menor tempo de execução em todos os testes, exceto no teste de rotação em que igualou o *SURF*. Quanto à taxa de sucesso, o *SIFT* foi o melhor na maioria dos testes, à exceção do de escala, em que foi o pior, do de rotação (ângulos múltiplos de 90), e do de ruído, em que foi idêntico ao *ORB*. Enquanto o *SURF* não se destacou como o mais rápido ou com melhor taxa de sucesso, foi sempre capaz de obter resultados satisfatórios com tempos de processamento aceitáveis, exceto no teste de ruído. Concluindo, pode-se dizer que o *SURF* é um meio-termo entre o *SIFT* e o *ORB*, com taxas de sucesso melhores que o *ORB,* mas com tempos de execução menores que o *SIFT*.

Os autores do terceiro trabalho (Tareen and Saleem, 2018) fazem uma análise comparativa do *SIFT*, *SURF*, *KAZE*, *AKAZE*, *ORB* e *BRISK*, com o intuito de determinar qual deles é mais robusto a diferenças de escala (5% até 500%) e de rotação (0° até 360°). Todos os testes foram executados 100 vezes, de forma a poder calcular valores médios e minimizar os erros resultantes da variabilidade do processamento. A robustez foi determinada em função da repetibilidade, ou seja, da percentagem de *features* detetadas que resistem a transformações. Entre as conclusões principais deste estudo, os autores constataram que o *ORB* deteta o maior número de *features*, seguido pelo *BRISK*. Por essa razão, estes também foram analisados com um limite imposto sobre o número máximo de *features* detetadas, dado que o tempo de processamento aumenta rapidamente com o número de *features*. O *SURF* deteta mais *features* que o *SIFT,* e o *AKAZE* mais que o *KAZE*, que foi o que detetou menos. O *ORB* foi o mais eficiente e com menor custo computacional, enquanto o *KAZE* foi o que teve o maior custo (mais do dobro do custo do *SIFT*). O *SIFT*, *SURF* e o *BRISK* foram os algoritmos mais robustos, de acordo com a repetibilidade. Destes três, o *SIFT* foi dado como o mais preciso, seguido do *BRISK*. Os autores concluem o estudo ordenando os algoritmos do melhor para o pior com base em diferentes aspetos, como se segue:

- Número de *features* detetadas: *ORB* > *BRISK* > *SURF* > *SIFT* > *AKAZE* > *KAZE*.
- Eficiência de extração de *features* por cada *feature-point*:
  *ORB* > *ORB* (limitado) > *BRISK* > *BRISK* (limitado) > *SURF* (64D) > *SURF* (128D) > *AKAZE* > *SIFT* > *KAZE*.
- Eficiência de *matching* por cada *feature-point*: *ORB* (limitado) > *BRISK* (limitado) > *AKAZE* > *KAZE* > *SURF* (64D) > *ORB* > *BRISK* > *SIFT* > *SURF* (128D).
- Velocidade de todo o processamento: *ORB* (limitado) > *BRISK* (limitado) > *AKAZE* > *KAZE* > *SURF* (64D) > *SIFT* > *ORB* > *BRISK* > *SURF* (128D).

Nestes últimos trabalhos relacionados abordados são feitas análises e comparações exaustivas entre diversos algoritmos de *feature matching*, incluindo a sua robustez, precisão, eficiência e tempo de processamento. Contudo, apesar de se constatar que determinados algoritmos obtêm melhores resultados em determinados casos (diferenças de escala, rotação, …) ou são mais rápidos que outros, existem muitas variáveis que determinam qual deles será o mais indicado. Estas variáveis dependem não só das imagens utilizadas como também das necessidades do utilizador. Dessa forma, para cobrir os casos de aplicação e as

necessidades que os utilizadores possam ter, a aplicação deverá disponibilizar vários algoritmos. Neste caso, serão implementados os algoritmos *SIFT*, *SURF*, *ORB*, *BRISK* e *AKAZE*.

## 2.2.2. Localização de objetos com *segmentation masks*

A segmentação de imagens pode ser bastante útil para focar nas regiões de interesse e ignorar tudo o que está no *background*, por exemplo, efetivamente reduzindo a quantidade de informação a ser processada nas fases seguintes da *pipeline* (Xiong et al., 2018). Nesta secção serão analisados métodos de segmentação de imagens, incluindo uns mais simples que permitem separar o *foreground* do *background*, e uns mais avançados que permitem segmentar os próprios objetos de interesse com *segmentation masks*.

### 2.2.2.1. Threshold Segmentation

Uma técnica simples para segmentar uma imagem é fazer uma análise píxel a píxel e agrupá-los pelos seus valores (Sharma, 2019). Esta é uma técnica conhecida como *Threshold Segmentation*, onde são usados *thresholds* (limites) para separar os pixéis em ***n+1*** intervalos, e onde ***n*** é o número de *thresholds*. Mais especificamente, esta técnica começa por remover a cor da imagem (converte para *grayscale*). De seguida, compara o valor de cada píxel com os *thresholds* pré-definidos, determinando a que intervalo pertence, e altera o seu valor para o correspondente a esse intervalo.

Uma vantagem desta técnica é o tempo de processamento, que por ser simples não é muito longo. Quando um objeto e o *background* têm um contraste alto, esta técnica permite obter bons resultados. Por outro lado, também tem limitações, como por exemplo no caso em que não existe grande diferença de valores entre pixéis ou quando há ruído.

Seguem-se os resultados de alguns testes realizados com esta técnica. A imagem processada tem uma resolução de 719 pixéis de largura por 480 pixéis de altura. Foram realizados três testes, todos com esta imagem e com o mesmo algoritmo, mas com *thresholds* diferentes. O código para realizar estes testes está em anexo. O primeiro teste foi realizado com um *threshold* cujo valor é a média dos valores dos pixéis da imagem, com um tempo de processamento de 0.817 segundos. O segundo teste foi realizado com dois *thresholds*, que dividem o intervalo de valores possíveis de cada píxel (0-255) em três intervalos de tamanhos aproximados ([0,85], ]85, 170], ]170, 255]). O processamento demorou 0.843 segundos, um aumento de cerca de 3.2% em relação ao primeiro teste. O terceiro teste foi realizado com três *thresholds*. Tal como no segundo teste, o intervalo de valores foi dividido em intervalos de dimensões semelhantes ([0,63], ]63, 126], ]126, 189], ]189, 255]), e o processamento demorou 1.145 segundos, um aumento de cerca de 35.8% em relação ao segundo teste e de 40.1% em relação ao primeiro teste.

*Figura 2.21 - Em cima: Original à esquerda, 1 threshold à direita; Baixo: 2 thresholds à esquerda, 3 thresholds à direita;*

Esta é uma técnica relativamente simples, mas que pode ser eficaz em casos de aplicação bem definidos, com imagens conhecidas. Algumas áreas onde esta técnica é aplicada incluem a área da medicina, geologia, biologia, entre outras (Zanaty, 2016) (Senthilkumaran and Vaithegi, 2016) (Malarvizhi, 2017) (Xie et al., 2010). Naturalmente estes casos de aplicação e o processamento que fazem são mais complexos do que é aqui abordado, recorrendo por exemplo a algoritmos próprios para determinar os valores de *threshold* a utilizar (Senthilkumaran and Vaithegi, 2016). Contudo, tipicamente as imagens analisadas são bem conhecidas e adequadas para aplicar esta técnica, como é o caso do exemplo seguinte (Niessen et al., 1999).

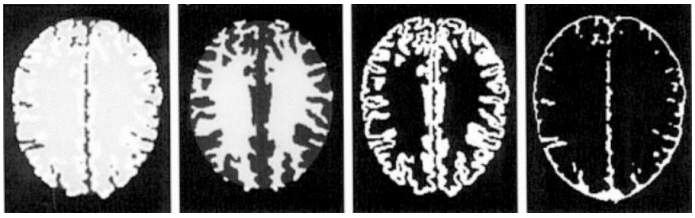

*Figura 2.22 - Exemplo de segmentação de uma imagem adequada (Niessen et al., 1999)*

### 2.2.2.2. Edge Detection Segmentation

Esta técnica baseia-se na deteção de arestas, que podem ser descritas como descontinuidades nas *features* locais de uma imagem. Como cada objeto pode ser diferenciado do *background* pelas suas arestas (quando visíveis), podemos obter o seu contorno e localizá-lo ao defini-las, com recurso a *weight matrices*, por exemplo (Sharma, 2019). Ao longo dos anos foram desenvolvidos vários métodos para deteção de arestas (Oliveira et al., 2014), tais como o *Sobel operator* (Sobel, 2014), *Laplacian filter* e *Canny Edge Detector* (Canny, 1986). Abaixo seguem exemplos da aplicação destas duas últimas. O código utilizado para realizar estes testes encontra-se em anexo.

*Figura 2.23 - Gaussian Blur + Laplacian Filter; 3ms*

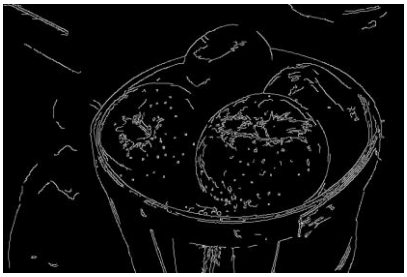

*Figura 2.24 - Canny Edge Detector; 2ms*

Esta é uma técnica relativamente simples que pode ser usada como um passo auxiliar num processamento mais complexo, como por exemplo no trabalho (Xie et al., 2010) onde foi utilizada para determinar valores de *threshold*. Tal como a técnica anterior, obtém resultados mais úteis com imagens bem definidas, obtidas em ambientes controlados.

### 2.2.2.3. ENet

Segundo os autores do *ENet* (*Efficient Neural Network*) (Paszke et al., 2016), a capacidade de fazer a segmentação semântica de imagens em tempo real é de extrema importância em aplicações móveis, como por exemplo em carros autónomos. Os autores indicam que as outras arquiteturas disponíveis na altura baseavam-se em modelos bastante grandes (*VGG16*, …), com um grande número de parâmetros e com tempos de inferência longos, o que os tornava inadequados para muitas aplicações móveis. Como tal, desenvolveram o *ENet* com o intuito de ser um modelo pequeno e com poucos parâmetros, mas que permita obter bons resultados com tempos de inferência baixos. Os autores indicam que o *ENet* é até dezoito vezes mais rápido, requer 79 vezes menos parâmetros e obtém exatidão semelhante ou superior aos outros modelos. Indicam ainda que o tamanho do modelo em disco é de cerca de 0.7 MB, em contraste com os 56.2 MB do modelo *SegNet* com o qual fazem comparações ao longo da publicação. Por ser tão pequeno, os autores sugerem que o modelo pode ser guardado na memória interna de processadores embutidos, o que certamente permitiria obter tempos de inferência ainda mais baixos. De seguida encontram-se exemplos de resultados obtidos pelos autores ao aplicarem o *ENet* aos *datasets Cityscapes* (“Cityscapes Dataset,” n.d.), *CamVid* (“CamVid Dataset,” n.d.) e *SUN* (“SUN Database,” n.d.). O *ENet* foi selecionado para ser incluído na aplicação para fazer segmentação semântica.

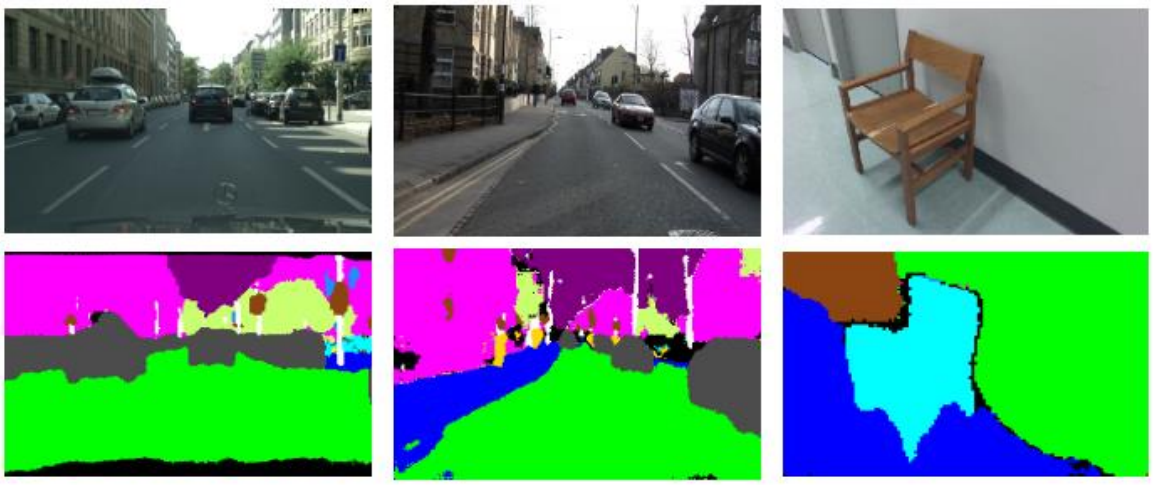

*Figura 2.25 - Resultados obtidos com o ENet (da esquerda para a direita - Cityscapes, CamVid e SUN) (Paszke et al., 2016)*

### 2.2.2.4. Mask R-CNN

O *Mask R-CNN* (Girshick et al., 2018) pertence à família de modelos *R-CNN* abordados anteriormente e foi desenvolvido a partir do *Faster R-CNN*. Enquanto o *Faster R-CNN* recorre apenas a *bounding boxes* para localizar os objetos detetados, o *Mask R-CNN* utiliza tanto

*bounding boxes* como *segmentation masks*. De acordo com os autores em (Girshick et al., 2018), a arquitetura do *Mask R-CNN* baseia-se na do *Faster R-CNN*, e foi adicionado um ramo responsável pela produção de *segmentation masks* em paralelo com os ramos responsáveis pela classificação e pelas *bounding boxes*. É aplicada uma *FCN* (*Fully Convolutional Network*) a cada região de interesse, que passam a estar associadas a três *outputs* cada: a classificação, *bounding box* e *segmentation mask*. Além destas alterações, os autores indicam que o *Faster R-CNN* não foi desenhado para que os *inputs* e *outputs* da rede neural estejam alinhados píxel a píxel, pelo que foi necessário recorrer a uma camada a que chamam *RoIAlign* para corrigir o desalinhamento. Segue-se uma figura onde está representada a *framework* do *Mask R-CNN*.

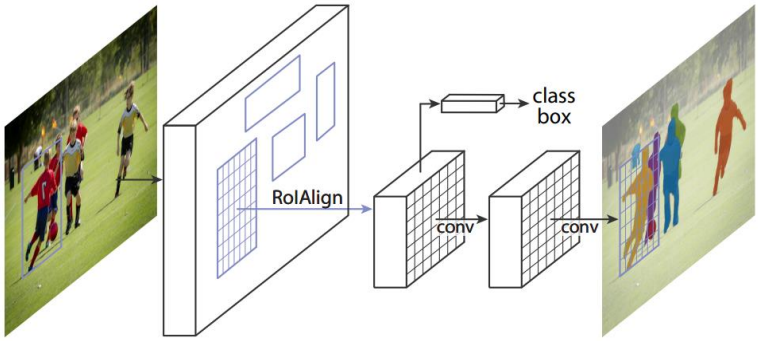


*Figura 2.26 - Framework do Mask R-CNN (Girshick et al., 2018)*

Os autores afirmam que o *Mask R-CNN* é capaz de processar 5 imagens por segundo (*COCO dataset*) e que obtém bons resultados, incluindo em situações difíceis como quando há sobreposição de objetos. Seguem-se alguns dos exemplos apresentados pelos autores. As imagens pertencem ao *COCO dataset*, o mesmo em que o *Mask R-CNN* atinge 5 *fps*.

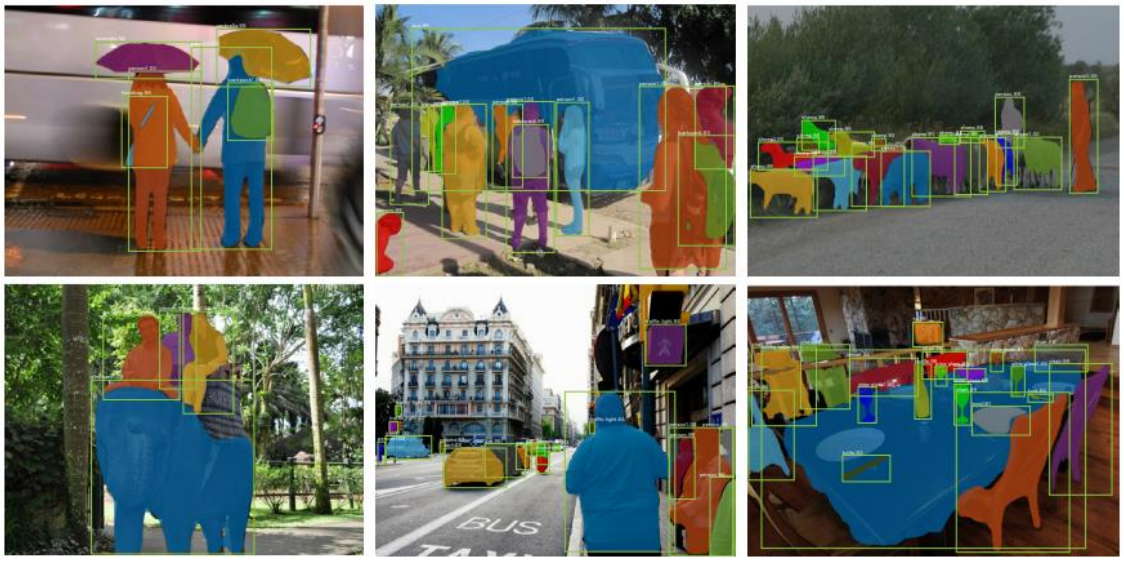

*Figura 2.27 - Exemplos de deteções utilizando o Mask R-CNN (Girshick et al., 2018)*

Segue-se uma tabela onde é listado o desempenho do *Mask R-CNN* em três testes diferentes. Esta tabela é adaptada da mesma fonte que reportou o desempenho dos outros modelos da família *R-CNN*, do *YOLO* e do *SSD* (Ouaknine, 2018), referenciada anteriormente na secção onde são analisados. O *hardware* utilizado para fazer estes testes não é

especificado. A métrica utilizada é a *mAP*. O *Mask R-CNN* foi selecionado para ser implementado na aplicação por ser o único modelo abordado capaz de produzir *instance segmentation masks*.

| | COCO 2016 (IoU=0.5) | COCO 2016 (IoU=0.75) | COCO 2016 (Official Metric) | Real Time Speed |
|---|---|---|---|---|
| Mask R-CNN | 62.3% | 43.3% | 39.8% | Não |

*Tabela 2.4 - Desempenho do Mask R-CNN no dataset COCO 2016 (adaptado de (Ouaknine, 2018))*

Os métodos de *feature matching* selecionados para serem implementados na aplicação foram o *SIFT*, *SURF*, *ORB*, *BRISK* e o *AKAZE*. O *YOLOv3* e o *SSD* foram selecionados para deteção de objetos com *bounding boxes*. O *ENet* foi selecionado para fazer *semantic segmentation*. O *Mask R-CNN* foi selecionado para detetar objetos com *bounding boxes* e para fazer *instance segmentation*. A combinação do *ENet* e do *Mask R-CNN* também permitirá fazer *panoptic segmentation*.

## 2.3. Interfaces para pesquisa de imagens

Um dos objetivos deste trabalho é permitir que métodos de pesquisa de informação visual possam ser usados por utilizadores comuns, permitindo assim uma combinação entre automatização e análise humana. A configuração e utilização destes métodos tipicamente requer um conhecimento técnico que a grande maioria das pessoas não possui. Assim sendo, a interface a desenvolver para a aplicação deverá ser atraente, clara, fácil de compreender e de aprender a utilizar. Deverá também ser possível utilizar os diferentes métodos de pesquisa com formas de interação adequadas para cada um. Além disso, deverá ser possível visualizar os resultados obtidos através de modos de visualização/visualizações próprias.

Como estes métodos de pesquisa poderão ser aplicados a coleções de imagens relativamente grandes, as visualizações deverão conseguir apresentar muitos resultados simultaneamente, de forma a que seja possível analisá-los e/ou encontrar o que se pretende rapidamente. Este é um problema que atualmente é foco de interesse e de estudo no campo da Interação Humano-Computador. Com o aumento da quantidade de imagens captadas e disponíveis online surge uma necessidade de desenvolver modos de visualização que permitam encontrar o que se procura no meio de tantas imagens. Os motores de busca de imagens, como o *Google Images* ("Google Images," n.d.), são alguns dos sistemas mais conhecidos para procura e visualização de imagens.

Continuando com o exemplo destes sistemas, ao longo dos anos alguns conceitos foram e continuam a ser utilizados com mais frequência, tais como as *thumbnails* e as grelhas de imagens (Rodden, 2002) (Zhang et al., 2006). A combinação destes dois conceitos permite visualizar grandes quantidades de imagens, mas é necessário algo mais para melhorar as chances de os utilizadores encontrarem o que procuram. Atualmente a utilização de filtros é muito comum, sendo que estes permitem filtrar as imagens apresentadas por tamanho, cor, tipo, entre outros. A utilização de algoritmos para classificação e ordenação dos resultados por relevância também é muito comum, e é alvo de muito interesse e investigação na área de *CBIR*.

Apesar das *thumbnails* e grelhas continuarem a ser predominantes atualmente, foram desenvolvidas várias outras técnicas que são interessantes e que em determinados casos podem ser bastante úteis. Um exemplo é a procura de imagens por referência (*query by example*), em que o utilizador fornece uma imagem semelhante à que procura. Por sua vez, o sistema analisa as características dessa imagem tais como a cor, textura e forma, e retorna um conjunto de imagens com características semelhantes (Ruger, 2005). Um exemplo de um sistema que oferece a possibilidade de pesquisar imagens por referencia é também o *Google Images* ("Google Images," n.d.), mais concretamente a função *reverse image search* ("Google Search," 2020). Outra técnica que assumiu um papel de destaque é conhecida por *relevance feedback*. Esta técnica consiste no aprimoramento dos resultados obtidos através de *feedback* do utilizador. Para tal, após fazer uma pesquisa e obter os resultados, o utilizador pode indicar se um ou mais resultados são relevantes ou não e é feita uma nova pesquisa tendo essas indicações em consideração. Desta forma, espera-se que ao envolver o utilizador no processo de classificação dos resultados seja possível melhorar gradualmente os resultados obtidos até o satisfazerem. Estas duas técnicas são demonstradas em (Ruger, 2005), ilustradas pelas duas figuras que se seguem. Na primeira figura é procurada uma porta através de uma imagem de referência. De seguida, o utilizador seleciona três imagens de portas azuis de entre os resultados obtidos e os resultados são aprimorados.

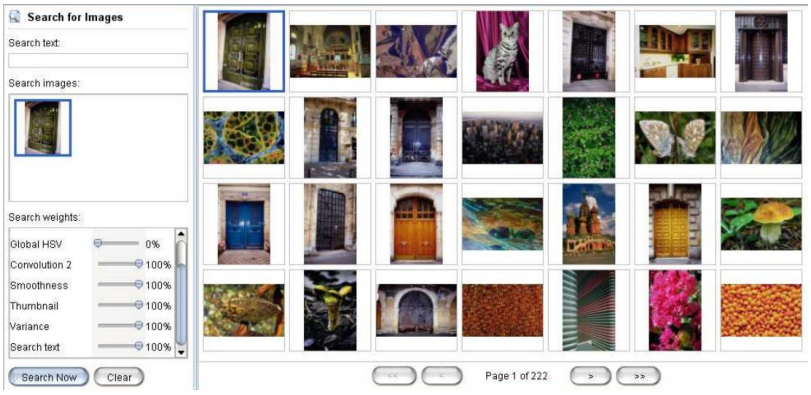


*Figura 2.28 - Exemplo de procura por referência (Ruger, 2005)*


*Figura 2.29 - Exemplo de aplicação de relevance feedback (Ruger, 2005)*

O que todas as técnicas abordadas têm em comum é que assumem que o utilizador sabe o que pretende, o que muito frequentemente não é o caso. Por esta razão, torna-se ainda mais importante ser possível percorrer e analisar grandes coleções de imagens de forma

eficiente. Além de nem sempre saberem o que procuram, não se pode assumir que todos os utilizadores são iguais e que pesquisam e analisam os resultados da mesma forma. Assim sendo, parece ser boa ideia analisar os hábitos de pesquisa de imagens em motores de busca para se obter uma ideia de cuidados a ter e aspetos a explorar.

Em (André et al., 2009), os autores começam por comparar os hábitos de pesquisa de documentos *web* com os de imagens, indicando que estes últimos normalmente são mais exploratórios do que os primeiros. Posteriormente, na sequência de oito entrevistas com investigadores e *designers* associados a motores de busca de imagens, os autores derivam quatro características da pesquisa de imagens que podem ser relevantes para o desenho de uma interface. A primeira característica é a capacidade dos próprios resultados satisfazerem as necessidades do utilizador. Por exemplo, ao procurar "como é o aspeto de uma tulipa?" é possível obter a resposta olhando para a maioria dos resultados, sem ser necessário clicar em nenhum específico. A segunda é a natureza exploratória da pesquisa de imagens. Segundo os autores, os utilizadores poderão procurar por imagens com um visual específico ou com determinadas características, mas não serem capazes de expressar esses requisitos até encontrarem o que procuram. A terceira característica também assenta na natureza exploratória da pesquisa de imagens, pois pode ser uma atividade sem objetivo e apenas para entretenimento ou por curiosidade. A quarta e última característica indicada pelos autores decorre das duas anteriores. Segundo os autores, a pesquisa de imagens possibilita e causa tangentes, na medida em que os utilizadores podem mudar de objetivo ao verem algo que lhes interesse. Sugerem também que quando um utilizador pesquisa por entretenimento, estas tangentes são desejáveis. Ainda que os motores de busca e os seus propósitos e aspetos nem sempre se alinhem com as necessidades da aplicação a desenvolver, todas estas características são interessantes e poderão ser relevantes para o desenvolvimento, dado que a aplicação não se destina apenas a um grupo de utilizadores e a um caso de utilização específico.

Os autores analisaram um conjunto de mais de 55 milhões de pesquisas feitas num dia e apresentam estatísticas de pesquisas *web* e pesquisas de imagens. As pesquisas de imagens tiveram em média mais cliques por pesquisa, assim como mais do dobro de páginas de resultados vistas por pesquisa e mais pesquisas efetuadas por sessão. No entanto, o número de pesquisas de imagens com cliques foi menor do que o de pesquisas *web* e o tempo passado a consultar os resultados de uma página foi maior. Estes pontos parecem comprovar que a pesquisa de imagens é mais exploratória e/ou menos eficiente do que a pesquisa *web*.

Outra estatística interessante é a relação entre as pesquisas feitas numa sessão. Em média, 70% de uma amostra de 1000 pesquisas de imagens estavam relacionadas com pesquisas anteriores (aranhas → aranhas venenosas → aranhas voadoras), 10% eram tangenciais (lagarto → lagarto do deserto → cor do deserto) e 20% não tinham qualquer relação. Destes dados podemos inferir que grande parte das pesquisas de imagens assentam na exploração visual dos resultados para se conseguir precisar o que se procura, e/ou são uma atividade exploratória que nem sempre tem um objetivo concreto.

Os autores colocam a hipótese do maior número de cliques e da maior quantidade de tempo gasto nas pesquisas de imagens comparado às pesquisas *web* resultarem de um dos seguintes casos: Ou a relevância não é tão importante para pesquisas de imagens como para *web* ou então é simplesmente muito pior. Além desta observação, os autores consideram a

possibilidade do maior número de cliques pode dever-se a dois fatores: O primeiro é que a qualidade dos resultados de uma pesquisa de imagens é subjetiva e não tem sempre uma resposta definitiva. Por outras palavras, o utilizador pode sempre pensar que haverá outra imagem melhor e continuar a procurar, aumentando o número de cliques. O segundo é o aspeto visual das *thumbnails*, que podem atrair o clique para visualizar a imagem em tamanho maior, por exemplo.

Segundo os autores, o menor número de pesquisas de imagens com cliques parece suportar a primeira característica mencionada anteriormente, nomeadamente a capacidade de os resultados satisfazerem a pesquisa só ao observá-los. Por fim, os autores dão um conjunto de sugestões para *design* de interfaces e/ou modos de visualização que suportam as conclusões retiradas, e desenham uma interface com base nisso. As sugestões estão resumidas de seguida e a interface que desenharam está representada pela figura seguinte.

1. Suporte à exploração (apresentar termos alternativos, pesquisas por referência semelhantes, imagens pesquisadas com frequência, …)
2. Estética e entretenimento (tirar partido da natureza visual dos resultados para tornar a experiência mais apelativa)
3. Suporte ao aprimoramento da pesquisa (permitir especificar características como cores, texturas, arestas, …, permitir alargar ou restringir o campo de procura com recurso a *tags* e permitir procurar mais imagens semelhantes).
4. Apresentação do histórico de pesquisas e aprimoramentos
5. Permitir guardar imagens para ver mais tarde

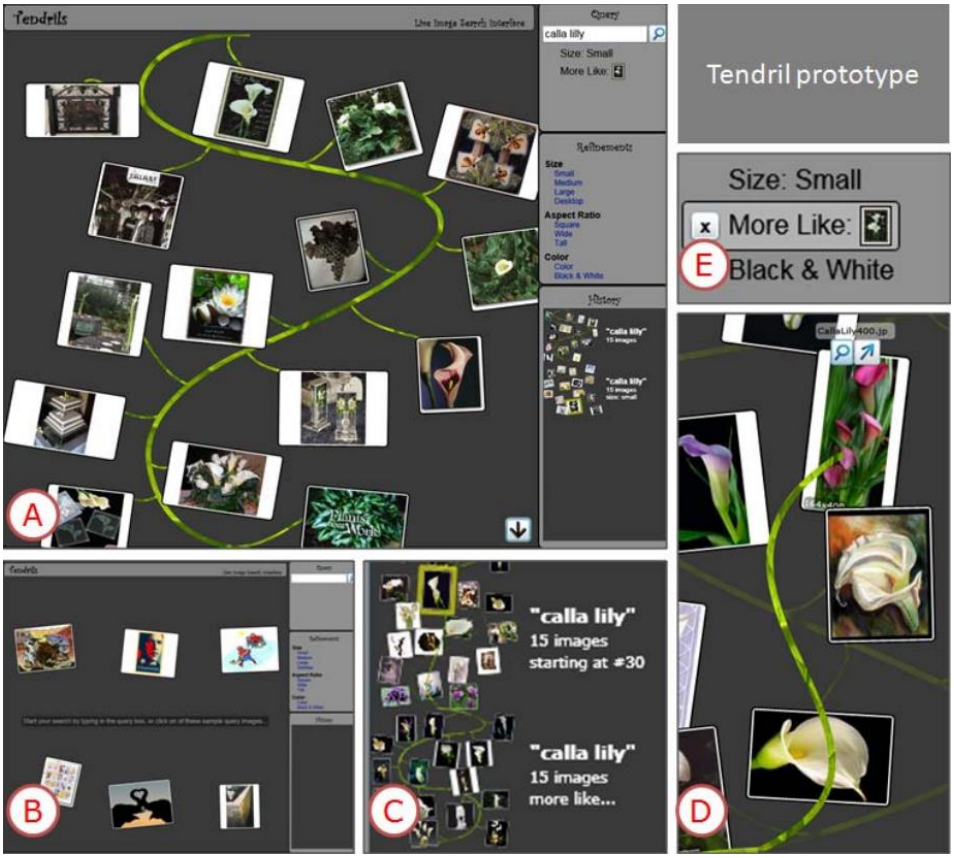


*Figura 2.30 - (A). Página completa; (B). Página inicial com sugestões de imagens para pesquisa; (C). Ampliação do histórico de pesquisas; (D). Exemplo de pesquisa de mais imagens semelhantes; (E). Ampliação da área de pesquisa (André et al., 2009)*

No entanto, a implementação dos autores e as visualizações que utilizaram é apenas uma de muitas. No trabalho (Thomee and Lew, 2012) são analisadas tendências e ideias de mais de 170 publicações diferentes, incluindo a interface desenhada pelos autores mencionados acima. Uma das técnicas utilizadas frequentemente nessas publicações é a distribuição das imagens de acordo com o nível de parecença. Esta técnica é explorada em (Rodden, 2002), onde são ilustrados vários exemplos criados com diferentes algoritmos para distribuição das imagens. Seguem-se alguns dos exemplos apresentados.

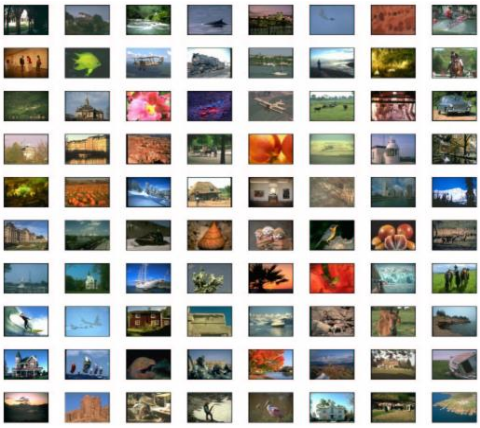

*Figura 2.31 - Grelha de imagens com distribuição aleatória (Rodden, 2002)*

*Figura 2.32 - Distribuição de imagens baseada em parecença visual (Rodden, 2002)*

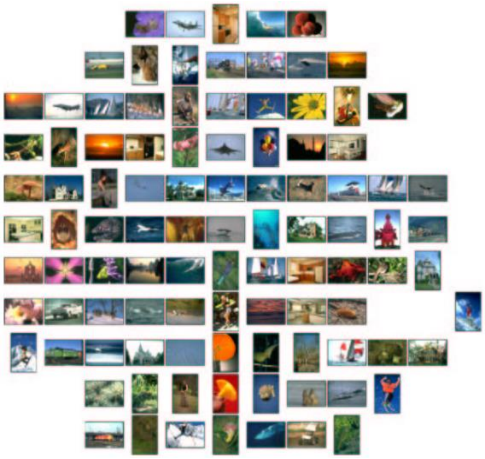

*Figura 2.33 - Grelha de imagens alternativa com distribuição aleatória (Rodden, 2002)*

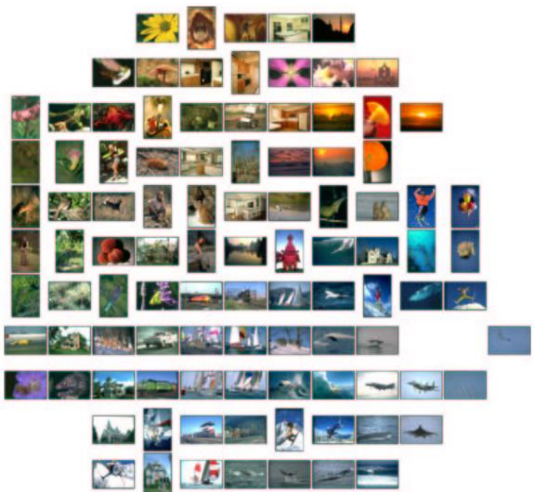

*Figura 2.34 - Grelha de imagens alternativa com distribuição baseada em parecença visual (Rodden, 2002)*

Para determinar os níveis de parecença entre imagens existem várias técnicas que podem ser utilizadas, tais como a comparação de *features*, cores, texturas, entre muitas outras. Em (Nguyen and Worring, 2008), os autores analisam várias funções conhecidas como *Similarity/Distance functions*, que podem ser utilizadas para determinar onde é que cada imagem deve ser colocada. Em (Basalaj, 2001), o autor estuda e desenvolve vários métodos para organizar as imagens no plano, sendo que alguns destes foram utilizados para criar os exemplos ilustrados pelas figuras acima (Rodden, 2002). Uma organização por semelhança visual pode trazer vantagens à exploração de um conjunto de imagens, tais como reduzir o espaço de procura e apelar à exploração com um visual atraente. Por outro lado, estas

distribuições demoram algum tempo a gerar e poderão rapidamente sobrecarregar os utilizadores com informação.

Além das distribuições dos exemplos acima, nesse trabalho (Rodden, 2002) são estudadas diversas visualizações baseadas em grelhas de duas dimensões que podem ser interessantes alternativas à grelha clássica. Algumas das características analisadas são o alinhamento das imagens e se há sobreposição ou não, por exemplo. Os autores em (Heesch and Rüger, 2004) apresentam várias visualizações interessantes. Uma delas é semelhante a uma grelha, em que as imagens mais relevantes são maiores e ficam mais perto do centro, e as outras são gradualmente mais pequenas e colocadas à volta sem sobreposição. A figura seguinte ilustra dois exemplos.

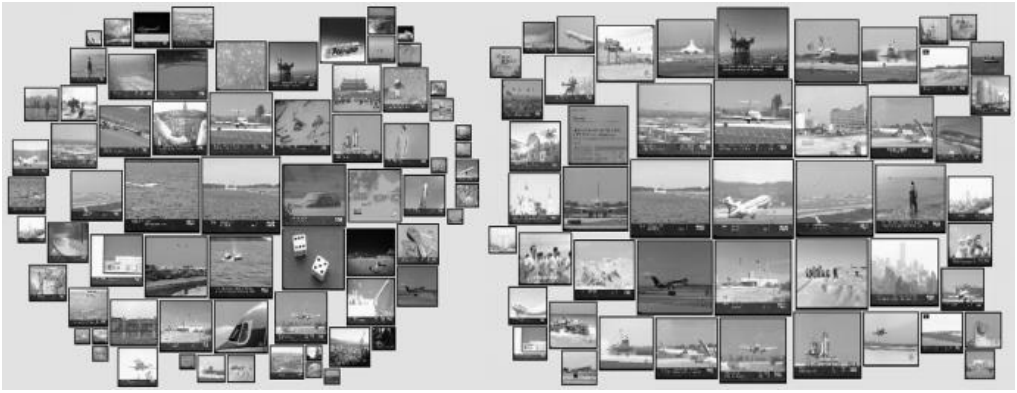

*Figura 2.35 - Exemplos de grelhas com thumbnails de tamanhos diferentes (Heesch and Rüger, 2004)*

Além destas grelhas, os autores mencionam uma outra forma de visualização em que as imagens são dispostas à volta de um ponto central, com a distância ao centro proporcional à sua semelhança visual. Quando as imagens são ordenadas por semelhança e distribuídas desta forma o resultado é uma espiral. Esta visualização é desenvolvida em mais detalhe e implementada num protótipo pelos autores da publicação (Torres et al., n.d.). A distribuição das imagens é feita de acordo com o caminho traçado por uma espiral de Arquimedes, sendo que as imagens nos anéis mais interiores são mais semelhantes/relevantes do que as imagens nos anéis mais exteriores. Seguem-se imagens que ilustram como as imagens podem ser dispostas neste modo de visualização.

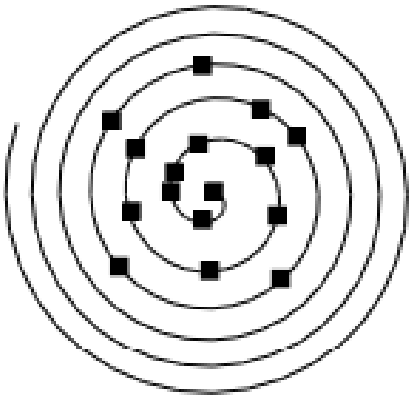

*Figura 2.36 - Disposição das imagens em espiral com intervalos proporcionais às diferenças de semelhança entre imagens consecutivas (Torres et al., n.d.)*

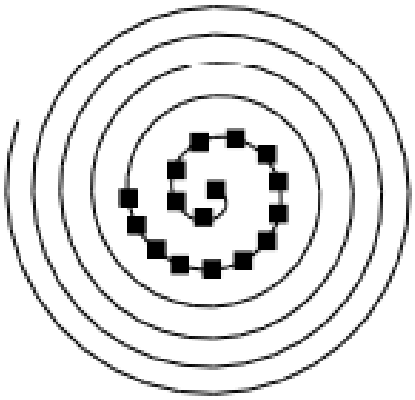

*Figura 2.37 - Disposição das imagens em espiral com intervalos equidistantes (Torres et al., n.d.)*

As interfaces desenvolvidas em alguns destes trabalhos serviram como inspiração para o desenvolvimento dos modos de visualização. As grelhas de *thumbnails* utilizadas frequentemente em motores de busca de imagens serviram como inspiração para o desenvolvimento de duas grelhas semelhantes, referidas neste trabalho por Grelha Normal e Grelha Variável. Estas grelhas foram desenvolvidas porque, apesar de serem antigas, funcionam bem e permitem visualizar coleções de imagens de forma rápida e eficaz. Além disso, podem servir como base para comparar com os outros modos de visualização.

Alguns dos exemplos apresentados em (Rodden, 2002) serviram como inspiração para o desenvolvimento de uma visualização com sobreposição de *thumbnails*, referida ao longo deste trabalho por Pilha. Estes exemplos suscitaram interesse pela possibilidade de se desenvolverem formas de interação diferentes, resultante do facto de poder haver sobreposição.

Por último, foi desenvolvida uma visualização em que as imagens são dispostas em espiral, inspirada pelos dois trabalhos estudados (Heesch and Rüger, 2004) (Torres et al., n.d.). Esta visualização é referida por Espiral ao longo deste trabalho. Este conceito despertou interesse pela sua estética e pela possibilidade de apresentar os resultados de uma forma que encoraje a exploração dos mesmos.

# 3. Protótipo

No contexto deste trabalho, foi desenvolvida uma aplicação para pesquisa visual em imagens que inclui diferentes métodos de pesquisa, cinco baseados em *features* visuais e cinco em reconhecimento e classificação de objetos. Além disso, a interface desenvolvida permite cincos visualizações distintas dos resultados obtidos.
Este capítulo documenta o desenvolvimento de um protótipo da aplicação desenvolvida. Primeiro são indicados os requisitos levantados e as bibliotecas externas utilizadas. Posteriormente é detalhada a arquitetura do sistema e são apresentadas as interfaces e os modos de visualização desenvolvidos.

## 3.1. Requisitos

Os requisitos da aplicação estão ligados a alguns dos objetivos deste trabalho, mais concretamente à facilidade de uso, acessibilidade e processamento na hora. A aplicação deve permitir a utilização dos métodos que se verificaram mais adequados, na sequência das análises feitas na secção de trabalhos relacionados. De seguida, os requisitos encontram-se listados e divididos em funcionais e não funcionais. Os requisitos funcionais foram definidos de forma a que a aplicação cumpra com os requisitos não funcionais, que decorrem dos objetivos deste trabalho.

### 3.1.1. Funcionais

- **RF 1 –** O utilizador deverá poder especificar as imagens que quer processar;

- **RF 2 –** O utilizador deverá poder escolher o método de processamento a utilizar;

- **RF 3 –** Os resultados deverão ser apresentados visualmente;
  - **RF 3.1 –** Para cada imagem analisada, deverá ser possível visualizar diferentes passos do processamento efetuado;
  - **RF 3.2 –** Para cada imagem analisada, deverá ser possível visualizar o resultado final (estimativa da localização da região procurada);
  - **RF 3.3 –** Para cada imagem analisada, deverá ser apresentada informação relevante;

### 3.1.2. Não funcionais

- **RNF 1 –** A aplicação deverá ser capaz de processar quaisquer imagens na hora, na medida em que não deverá depender de imagens predefinidas e/ou pré-processadas;
- **RNF 2 –** A aplicação deverá ter uma interface gráfica que facilite a pesquisa de informação visual a utilizadores sem conhecimentos técnicos próprios;

- **RNF 3 –** A aplicação deverá implementar métodos de procura de informação visual que demonstrem o que é possível fazer atualmente;

## 3.2. Ferramentas e bibliotecas externas utilizadas

A linguagem de programação escolhida para desenvolver o protótipo foi *Python*, dado que é uma das mais relevantes e das mais utilizadas atualmente, na área de Visão por Computador. Isto porque é fácil de escrever e de ler, o que permite ter uma rápida prototipagem. Além disso, existem muitas bibliotecas externas que permitem fazer todo o tipo de processamento de imagem, assim como uma grande comunidade de programadores e de suporte *online*.

### 3.2.1. Front-end

Para desenvolver a interface optou-se por utilizar o *PyQt5* (versão 5.14.2), que é um conjunto de *bindings* para *Python* (licença GPL) da *framework Qt*, desenvolvida em *C++*. O *PyQt5* permite desenvolver interfaces gráficas para aplicações e suporta várias plataformas (*Windows*, *iOS*, *Android*, …) (Riverbank Computing Limited, 2020). Optou-se por utilizar esta ferramenta pois é bastante flexível, disponibilizando uma enorme variedade de módulos e funcionalidades que satisfazem qualquer necessidade que se possa ter.

Recorreu-se também à biblioteca *Pillow* (versão 7.1.1), que é um *fork* da biblioteca *PIL* (*Python Imaging Library*) (“Pillow Documentation,” 2020), para fazer a ligação entre as representações das imagens no *back-end* e no *front-end*.

Por último, recorreu-se ao *OpenCV* para implementar o *front-end* da função em que o utilizador seleciona uma região de interesse da imagem de referência. Contudo, o *OpenCV* foi mais utilizado na parte de processamento e é abordado em maior detalhe de seguida.

### 3.2.2. Back-end

Para a parte de processamento da aplicação optou-se por utilizar o *OpenCV* (*Open Source Computer Vision Library*), que é uma biblioteca *open source* para o desenvolvimento de aplicações baseadas em visão por computador e *machine learning*. *O OpenCV* conta com mais de 2500 algoritmos otimizados, incluindo algoritmos para deteção de rostos, identificação de objetos, classificação de ações, manipulação e processamento de imagem, entre outros. É uma biblioteca escrita originalmente em *C++*, mas também está disponível para *Python*, *Java* e *MATLAB* (“About OpenCV,” 2020).

A versão do *OpenCV* para *Python* está inter-relacionada com a biblioteca *numpy*, na medida em que as suas estruturas de dados (neste caso das imagens) podem ser trocadas e combinadas livremente. Esta é uma biblioteca bastante otimizada para operações numéricas, o que é bastante útil para desenvolvimento em conjunto com o *OpenCV*, e por essa razão também foi utilizada (versão 1.18.2) (“OpenCV: Introduction,” 2020).

No decorrer das análises feitas na secção de trabalhos relacionados constatou-se que os métodos de *feature matching* eram adequados para esta aplicação. Por se ter estudado o

*SIFT*, *SURF*, *ORB*, *BRISK* e o *AKAZE* sabia-se que eram boas opções, dado que se destacam dos demais por permitirem extrair *features* (deteção e descrição) com resultados e tempos de processamento bons. Como se concluiu que o algoritmo adequado depende do caso de aplicação e das suas necessidades, parece apropriado disponibilizar todos estes algoritmos e remeter a escolha para o utilizador. Todos estes estão incluídos no *OpenCV*, além de outros algoritmos e ferramentas também necessárias, como *brute-force matchers*, algoritmos para remoção de *outliers*, algoritmos para cálculo de homografia e transformações de perspetiva.

No entanto, a partir da versão 3.4.3 do *OpenCV* disponibilizada em 2018 ("OpenCV – 3.4.3," n.d.), alguns algoritmos patenteados deixaram de estar incluídos por norma, incluindo o *SIFT* e o *SURF*. Por outro lado, apesar de não poderem ser utilizados para fins comerciais, estão disponíveis para fins académicos num *package* de módulos experimentais/extra (*opencv-contrib*). Para tal, foi necessário ativar os módulos respetivos e compilar o *OpenCV* a partir do código fonte ("Where did SIFT and SURF go in OpenCV 3?," 2015).

Além disto, também foi necessário instalar o *CUDA Toolkit* (versão 10.2) ("CUDA Toolkit - NVIDIA," 2020) e a biblioteca *cuDNN* (versão 7.6.5) ("cuDNN - NVIDIA," 2020) para compilar o *OpenCV* com suporte para utilização de placas gráficas (*CUDA*). Assim sendo, foi compilada a versão 4.3.0 do *OpenCV* a partir do código fonte, incluindo todos os módulos opcionais extra, os algoritmos patenteados e todas as *features* disponibilizadas pelo *CUDA Toolkit* e pela biblioteca *cuDNN*.

De referir que é possível que num futuro próximo alguns algoritmos patenteados voltem a ser incluídos por norma, depois das suas patentes expirarem, como é o caso da patente do *SIFT* que expirou recentemente, em março de 2020 (Lowe, 2004b).

## 3.3. Desenvolvimento e implementação

A arquitetura da aplicação desenvolvida está detalhada no diagrama que se segue. A aplicação foi desenvolvida de forma modular, para que seja relativamente simples fazer alterações, adicionar novas funcionalidades ou até novos métodos de pesquisa visual. De notar que os módulos mais básicos (métodos de pesquisa, …) estão ocultos da arquitetura. Os módulos principais são os seguintes:

- *GUI*: Responsável pela interface e pelos *inputs* do utilizador.
- Módulo de processamento: Recebe os pedidos de processamento da *GUI* e delega-os aos módulos dos métodos de pesquisa, juntamente com as imagens para análise que recebe do módulo de referência/resultados. Após o processamento, passa os resultados ao módulo de resultados.
- Módulo de resultados: Responsável por guardar os resultados e gerir o acesso aos mesmos.
- Módulo de referência: Responsável por guardar e gerir a referência (métodos baseados em *feature matching*).
- Módulo de ficheiros: Responsável pelo acesso ao sistema de ficheiros para procura e seleção de imagens/vídeo.

O *CPU* e a *GPU* estão incluídos na arquitetura pois são as duas unidades de processamento que podem ser utilizadas pelos métodos de pesquisa visual.

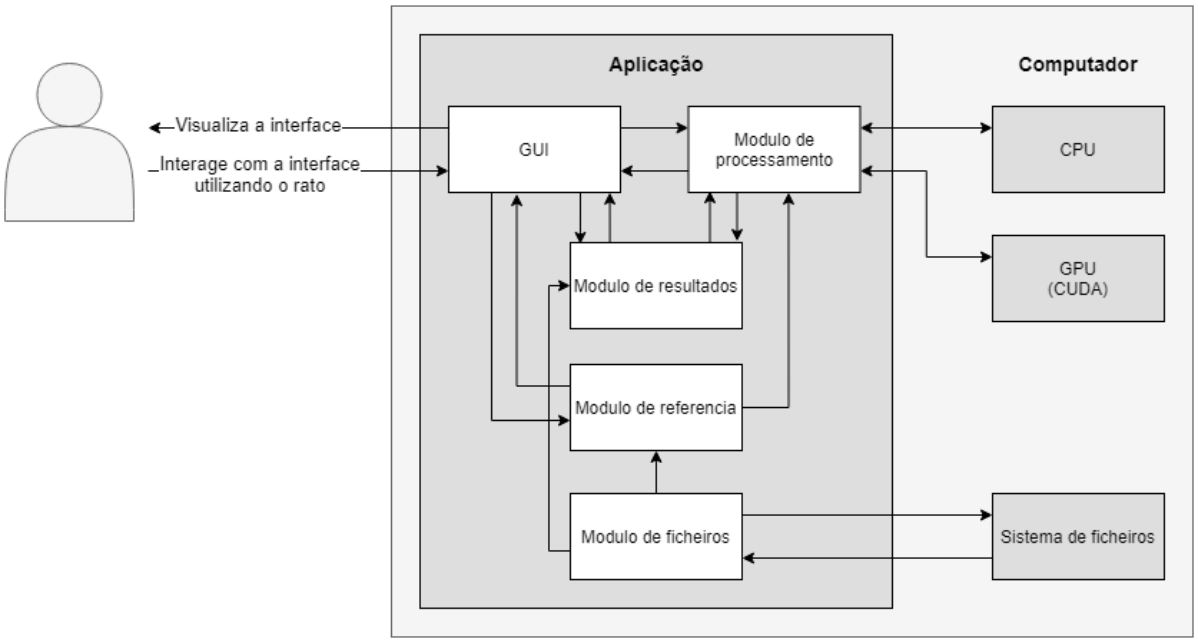


*Figura 3.1 - Diagrama da arquitetura da aplicação*

### 3.3.1. Desenvolvimento do protótipo

O protótipo foi desenvolvido simultaneamente com a pesquisa feita no decorrer deste trabalho, evoluindo de forma a acompanhar o que estava a ser estudado e o que se pretendia fazer. Inicialmente foram estudados alguns métodos de pesquisa baseados em *feature matching*, na secção de trabalhos relacionados. Após determinar que métodos eram adequados para satisfazer as necessidades da aplicação, foram realizados testes de desempenho com o intuito de obter alguma experiência e conhecimento prático, assim como comprovar as análises feitas em trabalhos relacionados. Para realizar estes testes foi primeiro escrito o código dos módulos para cada um dos métodos, que foram posteriormente disponibilizados numa interface desenvolvida paralelamente. Esta interface apresentava um *layout* adequado para utilizar métodos de *feature matching* e para visualizar os resultados obtidos, ainda que fosse rudimentar. Desde então, foram feitas inúmeras alterações graduais para torná-la mais completa, assim como alterações significativas para acomodar os métodos baseados em deteção de objetos, que foram adicionados mais tarde. No entanto este foi um processo simples, dado que todo o código da aplicação foi desenvolvido para ser modular.

Os métodos de pesquisa visual implementados podem dividir-se em dois grupos de acordo com a abordagem que seguem, nomeadamente *feature matching* ou deteção de objetos, como se seguem:

I. *Feature Matching*
   - *SIFT*
   - *SURF* (versões *CPU* e *GPU*)
   - *ORB* (versões *CPU* e *GPU*)
   - *BRISK*

- *AKAZE*

II. Deteção de objetos
   - *YOLOv3* (*bounding boxes*)
   - *SSD* (*bounding boxes*)
   - *Mask R-CNN* (*bounding boxes, instance segmentation masks*)
   - *ENet* (*semantic segmentation masks*)
   - *ENet + Mask R-CNN* (*panoptic segmentation masks*)

A visualização de resultados e as formas de interação foram sempre aspetos bastante importantes ao longo do desenvolvimento, ditando a grande maioria das decisões relacionadas com a interface. Após a implementação dos métodos de pesquisa visual, o foco passou para o estudo de trabalhos acerca de interfaces e para o desenvolvimento de modos de visualização. Foram desenvolvidos cinco modos (discutidos em detalhe na secção seguinte), incluindo um em que é apresentado um único resultado, dois baseados em grelhas de duas dimensões, uma pilha e uma espiral. Além disto, foi desenvolvido um sistema de *overlays*, que permite sobrepor às imagens dos resultados quaisquer passos e resultados secundários do processamento (discutidos em detalhe na secção seguinte). Foi também adicionada a possibilidade de processar vídeo (os *frames* são extraídos e processados). Posteriormente, o vídeo pode ser reproduzido com ou sem *overlays* sobrepostos, e é possível escolher a velocidade de reprodução (a velocidade real pode ser mais baixa, dependendo do número de *overlays* sobrepostos e da resolução das imagens).

Dependendo do número de imagens a analisar, o processamento pode ser bastante demorado e consumir muita memória. Como os vídeos tipicamente possuem muitos *frames*, tornou-se necessário permitir guardar os dados do processamento em disco, sendo que o utilizador pode decidir quando o fazer (antes/depois do processamento, com imagens/vídeo e com qualquer método de pesquisa). Os dados guardados são as imagens originais (selecionadas para análise) e todos os *overlays* (imagens com transparência), dado que são guardadas em memória sem compressão e são o que ocupa mais espaço. A leitura/escrita de/para disco pode ser demorada, dependendo da quantidade de dados. Por esta razão, estas operações são realizadas numa *thread* dedicada, separada da *thread* principal que é responsável pela *GUI*, evitando-se bloqueios da interface. Para diminuir os tempos de processamento, foi adicionada a possibilidade de utilizar *GPU* com os métodos de pesquisa que o suportam (*CUDA*), dependendo da presença de uma *GPU* compatível no computador. O processamento de imagens é feito numa *thread* dedicada, para evitar bloqueios da interface.

### 3.3.2. Interface

A interface final da aplicação desenvolvida é constituída por duas interfaces principais, uma para os métodos baseados em *feature matching* e outra para os métodos de deteção de objetos. Cada uma destas interfaces divide-se em várias secções, cada uma responsável por um determinado aspeto do funcionamento da aplicação. Algumas secções são comuns às

duas interfaces, mas com algumas diferenças. Outras secções são exclusivas de uma ou de outra interface.

De seguida as duas interfaces são apresentadas secção a secção, indicando os seus propósitos e responsabilidades. São incluídas várias figuras que ilustram o que é dito, mas poderão escapar alguns detalhes como títulos de janelas (apresentam informação acerca dos resultados), *tooltips*, comportamentos de *widgets* (quando são ativados/desativados), entre outros.

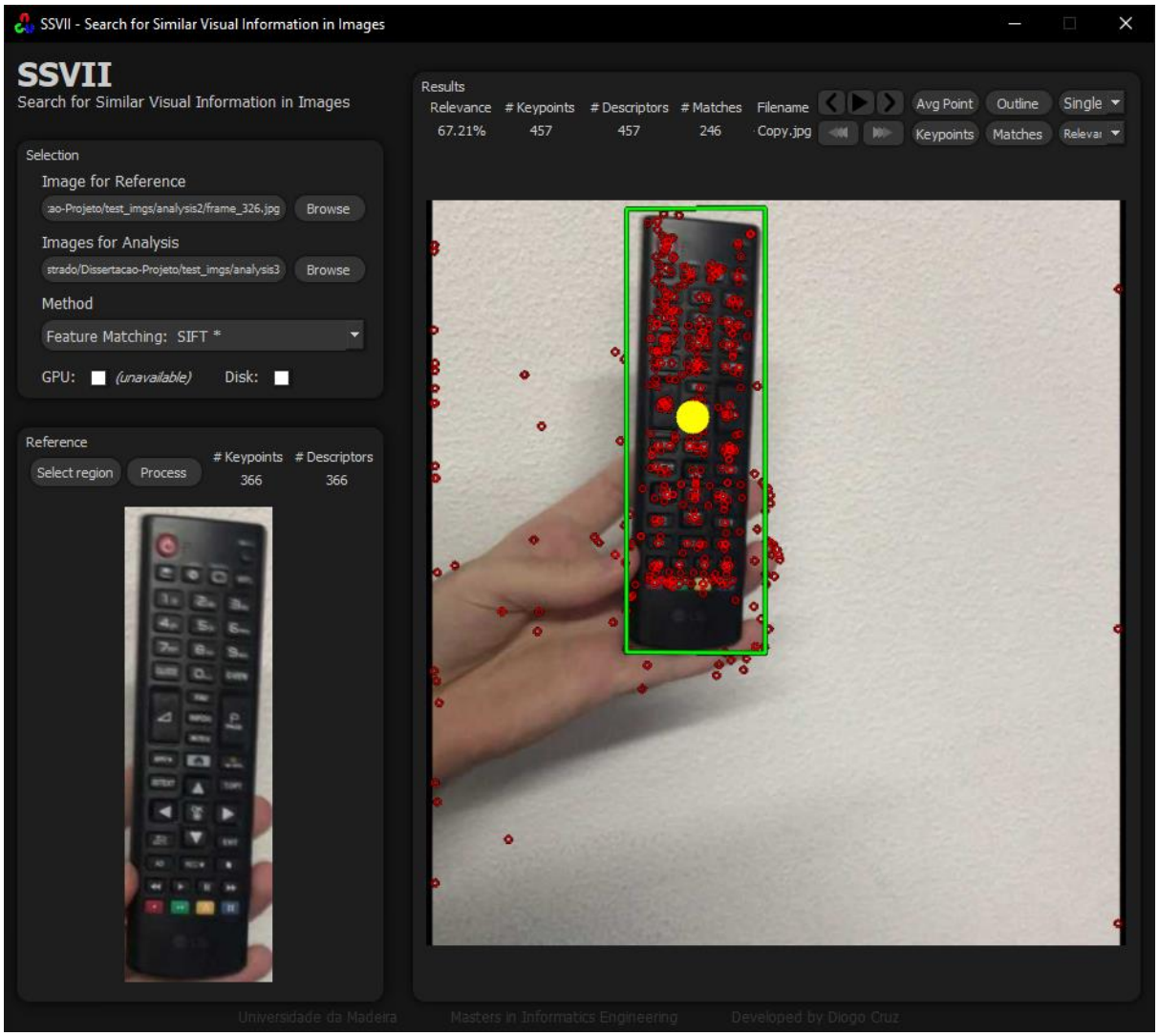


*Figura 3.2 - Interface de feature matching*

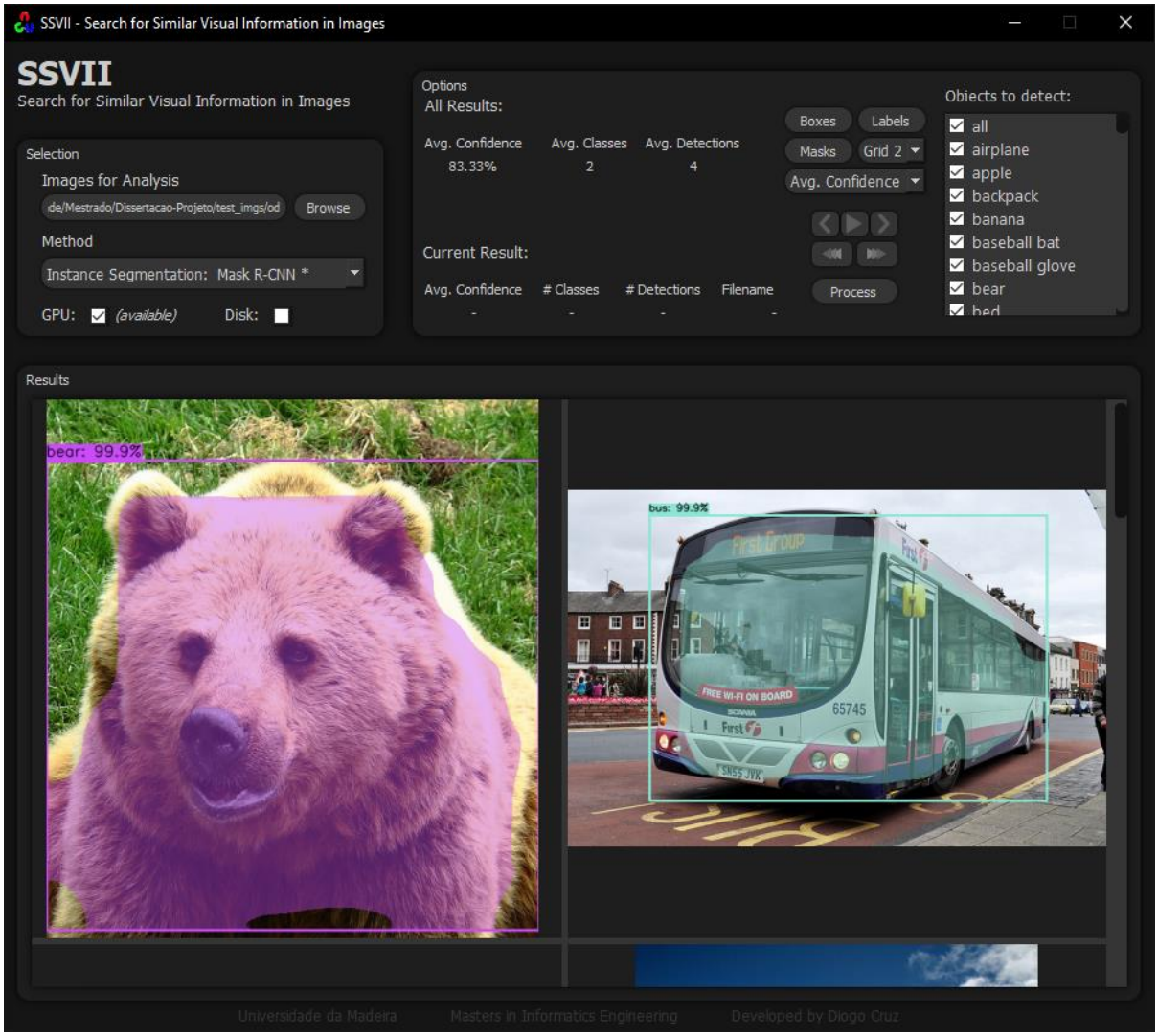


*Figura 3.3 - Interface de deteção de objetos*

A primeira secção é responsável pela seleção das imagens/vídeos a processar e do método de pesquisa a utilizar. Além disto permite especificar algumas preferências de processamento, tais como utilizar *GPU* (ou não) e utilizar disco (ou não). Esta secção é comum tanto à interface de *feature matching* como à interface de deteção de objetos, ainda que seja ligeiramente diferente pois na interface de *feature matching* também permite escolher uma imagem de referência. Estão disponíveis *tooltips* em muitos *widgets* de toda a interface da aplicação para clarificar as suas funções e propósitos, incluindo nesta secção. Seguem-se algumas figuras que ilustram ambas as versões, as opções *GPU*/Disco e os *tooltips* associados às imagens selecionadas e às opções:

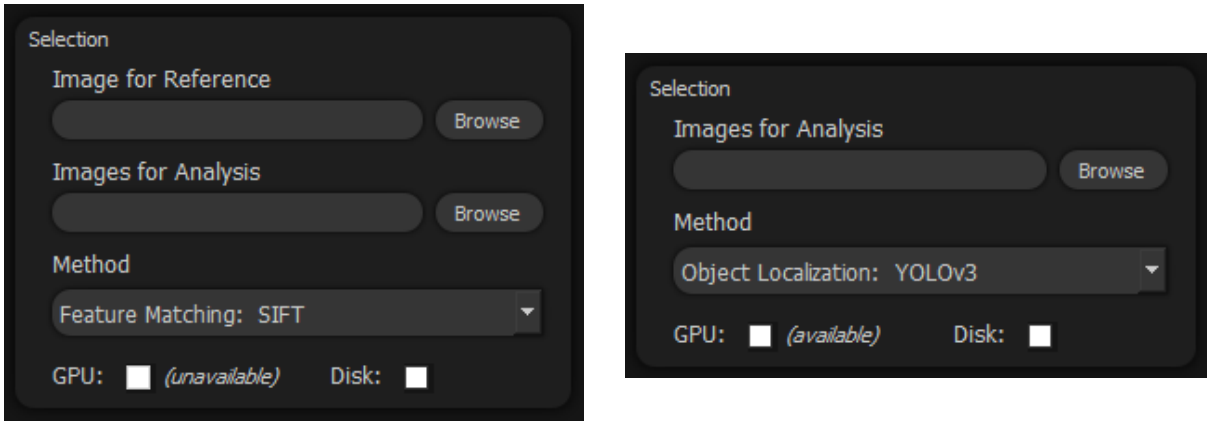


*Figura 3.4 - Secção de seleção de imagens e método (Esquerda - feature matching; Direita – deteção de objetos)*

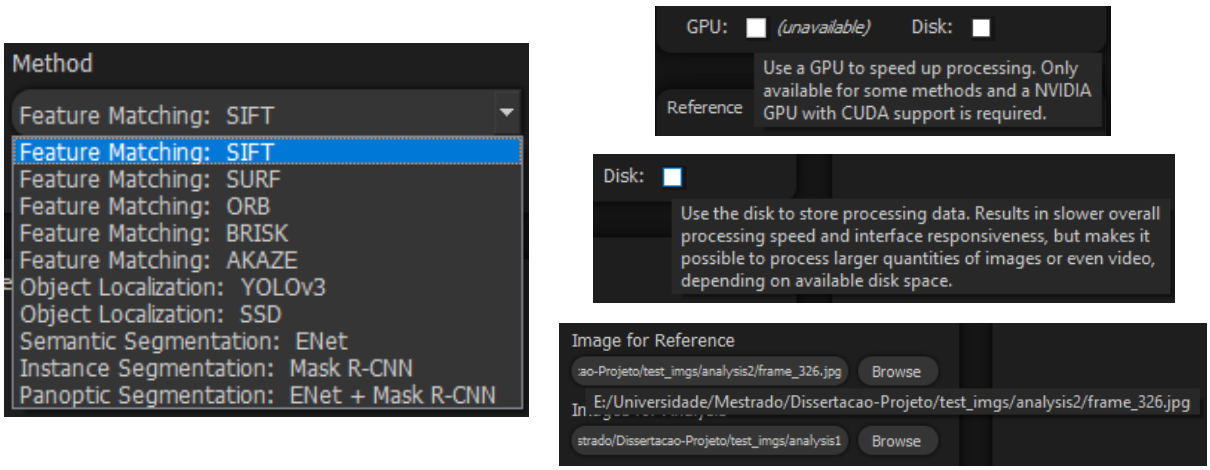


*Figura 3.5 - Esquerda - Seleção do método de pesquisa; Direita - Opções e tooltips*

A segunda secção pertence apenas à interface de *feature matching*. Esta secção permite selecionar uma região de interesse da imagem de referência, de forma semelhante à vista em (Sivic et al., 2003). Para tal, tem um botão intitulado "*Select region*" que ao ser clicado abre uma janela com a imagem em maior escala. Nesta janela é possível fazer uma seleção ao clicar um ponto inicial e ao arrastar o rato até o ponto final. Esta região é a que irá ser procurada no conjunto de imagens para análise. Por fim, o processamento pode ser iniciado ao clicar num botão intitulado "*Process*". Os *widgets* desta secção ficam desativados até o processamento concluir. No fim é apresentada alguma informação (número de *keypoints*, …).

As figuras seguintes ilustram a seleção de uma região de interesse, e a secção antes, durante e após o processamento.

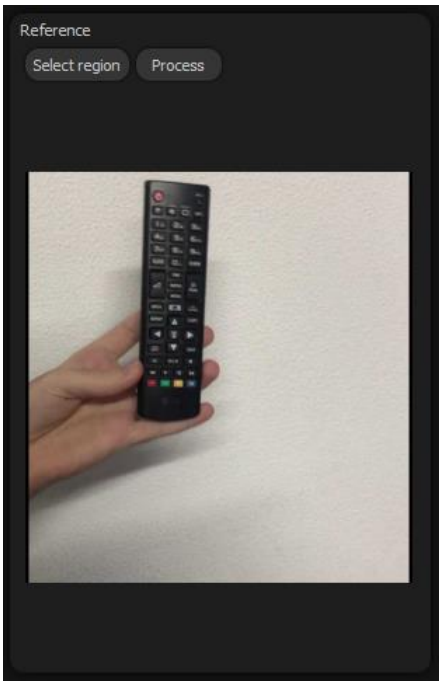


*Figura 3.6 - Secção da referência*

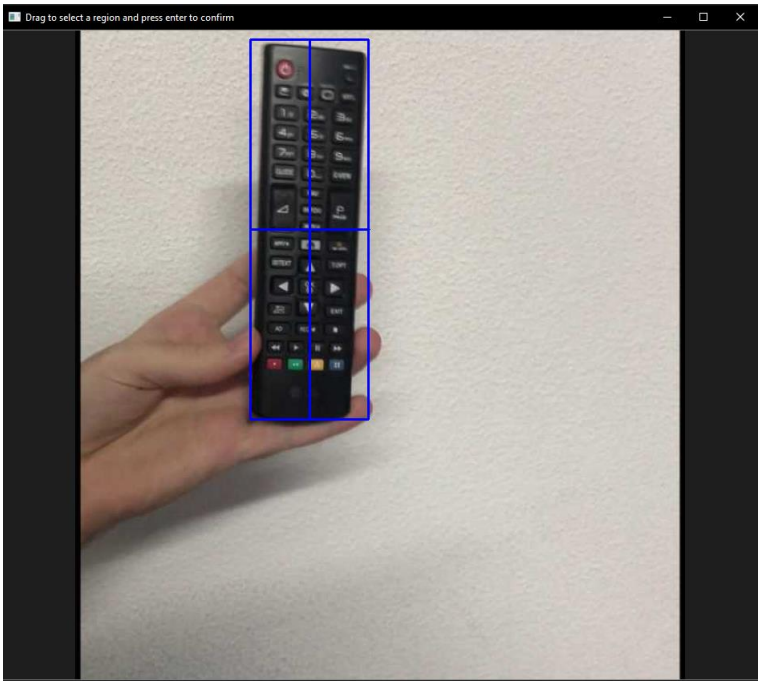


*Figura 3.7 - Seleção da região de interesse*


*Figura 3.8 - Região selecionada*


*Figura 3.9 - Processamento iniciado*

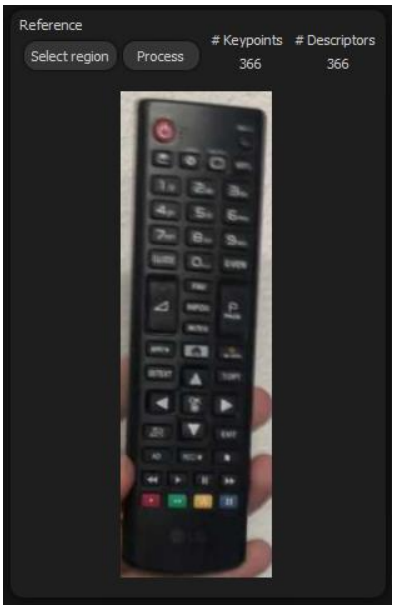


*Figura 3.10 - Processamento concluído*

A terceira secção é a dos resultados. Esta secção existe nas duas interfaces, mas existem diferenças significativas, tais como as dimensões, o formato, a informação apresentada e os controlos disponíveis. Além disto, esta secção varia bastante conforme o modo de visualização selecionado. Os modos de visualização disponíveis são comuns a todos os métodos de pesquisa e, portanto, às duas interfaces principais. Nesta dissertação é feita referência a estes métodos por:

- Resultado Único - apresenta apenas um resultado de cada vez
- Grelha Normal - apresenta os resultados numa grelha de duas dimensões
- Grelha Variável - apresenta os resultados numa grelha de duas dimensões com células de tamanhos diferentes
- Pilha – apresenta os resultados empilhados com sobreposição parcial ou completa
- Espiral – apresenta os resultados dispostos em espiral.

Começando pela secção da interface de *feature matching* e pelo modo Resultado Único, a secção divide-se verticalmente em duas partes. No topo é apresentada informação relativa ao processamento efetuado, tal como a relevância, o número de *keypoints*, *descriptors* e *matches*, e o nome do ficheiro. Ao lado encontram-se botões que permitem mudar o resultado apresentado para o anterior ou para o próximo. A ordem dos resultados apresentados pode ser escolhida através de um *drop-down*, e os critérios disponíveis são a relevância, o número de *keypoints*/*descriptors*, o número de *matches* e o nome do ficheiro. Além destes botões também existe um botão para reproduzir/pausar, e botões para diminuir e aumentar a velocidade de reprodução (mínimo 1 *fps*, máximo 60 *fps*, incremento 10 *fps*). Esta funcionalidade é útil para animar os resultados obtidos ao processar os frames de um vídeo ou para visualizar os resultados estilo *slideshow*. Este modo de visualização é o único que apresenta estas informações e estes botões. Ao lado destes botões encontram-se quatro botões tipo *toggle* para seleção dos *overlays* a apresentar. Um *overlay* é uma imagem maioritariamente transparente com algum aspeto ou resultado de uma fase de processamento desenhado. Cada método de pesquisa produz *overlays* conforme o processamento efetuado, pelo que nem todos disponibilizam todos os *overlays*. Os *overlays* que não estão disponíveis simplesmente não são sobrepostos ao clicar no botão correspondente. Os *overlays* que podem ser gerados e visualizados são "*Avg Point*" (média da localização dos *matches*), "*Outline*" (linha que delimita a previsão), "*Keypoints*" (pontos de interesse detetados) e "*Matches*". Por último, os menus de *drop-down* para seleção do modo de visualização e critério de ordenação encontram-se ao lado dos botões dos *overlays*.

Por baixo desta parte é apresentada a imagem do resultado com os *overlays* ativados sobrepostos. A imagem é apresentada no maior tamanho que não ultrapasse as dimensões da *image label*, mantendo o *aspect ratio* original. As possíveis bordas resultantes são ocultadas. Também é possível visualizar a imagem apresentada em maior escala numa janela separada, ao clicar na *image label*. Esta funcionalidade está disponível também na secção da referência. Seguem-se várias figuras que ilustram o topo desta secção em detalhe, a secção na íntegra e a janela com a imagem do resultado em maior escala:

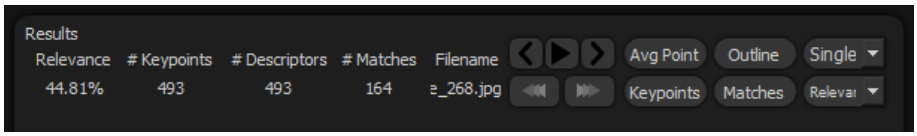


*Figura 3.11 - Topo da secção dos resultados de feature matching no modo de visualização Resultado Único*

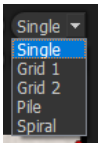


*Figura 3.12 - Visualizações*

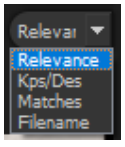


*Figura 3.13 - Critérios*

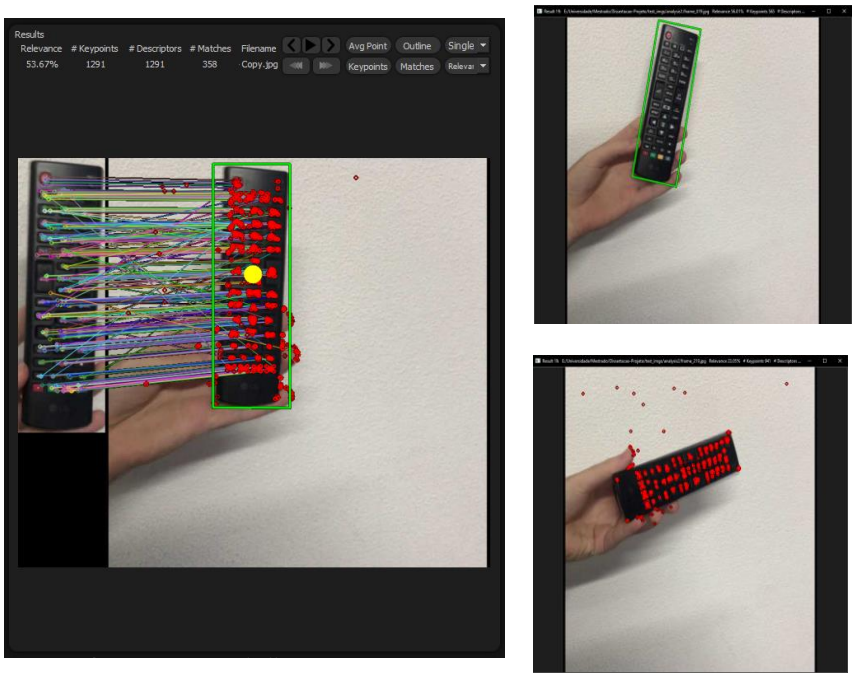


*Figura 3.14 - Esquerda: Todos os overlays sobrepostos; Direita: Janelas com resultado em maior escala*

A Grelha Normal apresenta os resultados numa grelha de duas dimensões, ordenados de acordo com o critério selecionado. O primeiro é apresentado em cima à esquerda. Quando existem mais resultados do que a grelha consegue apresentar aparece uma *scrollbar*. Além disto, é possível visualizar cada resultado em detalhe através do seu *tooltip*, ao passar o rato por cima. Este modo foi desenvolvido porque este tipo de grelha funciona muito bem para visualizar grandes quantidades de imagens e porque complementa a aplicação. Além disso, ao avaliá-lo nos testes com utilizadores será possível usá-lo como base para comparações. Este foi inspirado por (“Google Images,” n.d.) e está representado na figura seguinte.

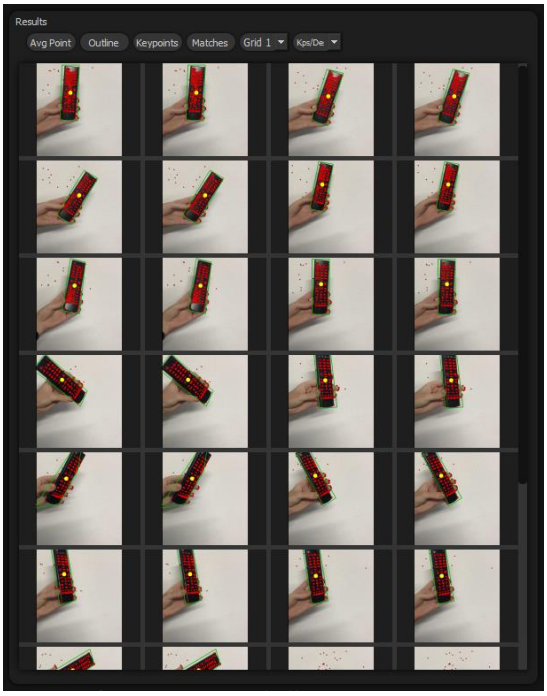


*Figura 3.15 - Modo de visualização Grelha Normal*

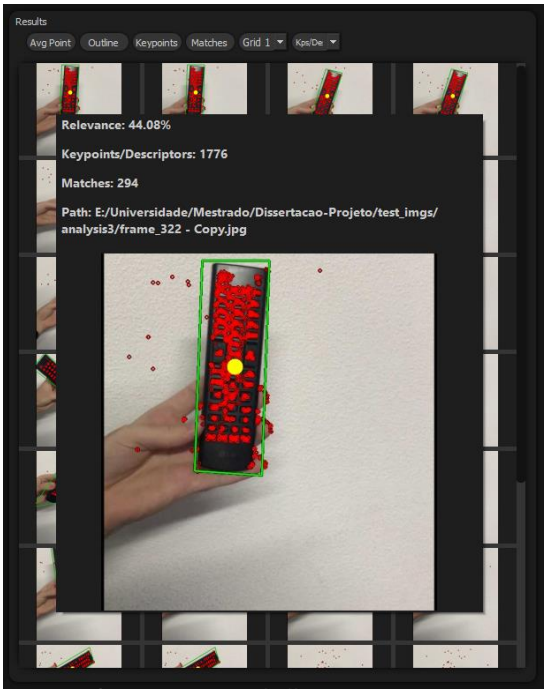


*Figura 3.16 - Visualização de um resultado com tooltip*

A Grelha Variável é idêntica à Grelha Normal. A diferença entre estes dois modos é que a Grelha Variável apresenta *thumbnails* de diferentes tamanhos. Esta ideia não é nova. Por exemplo, é muito comum motores de busca de imagens apresentarem *thumbnails* com tamanhos diferentes para manter os seus *aspect ratios* originais. No entanto, a Grelha Variável explora esta ideia de outra forma, com outro propósito. O número de imagens aumenta gradualmente linha a linha, ao longo do eixo das ordenadas, mas as imagens são cada vez mais pequenas. Assim sendo, as linhas mais perto do início da grelha têm menos imagens mas estas são maiores, e as linhas mais perto do fim têm mais mas são menores. A razão por trás desta escolha é que pretendia-se tentar melhorar a Grelha Normal ao permitir observar os resultados mais relevantes em maiores dimensões. No entanto, os resultados também podem ser ordenados por qualquer outro critério de ordenação disponível. De mencionar que o número de imagens e os seus tamanhos em cada linha são predefinidos. O que pode mudar é o número de linhas de cada tipo, dependendo do número de resultados a apresentar, mas geralmente há aproximadamente o mesmo número. Os modos de interação são os mesmos da Grelha Normal. Este modo de visualização foi inspirado em parte por (Heesch and Rüger, 2004), como visto na secção de trabalhos relacionados, e está representado na figura seguinte.

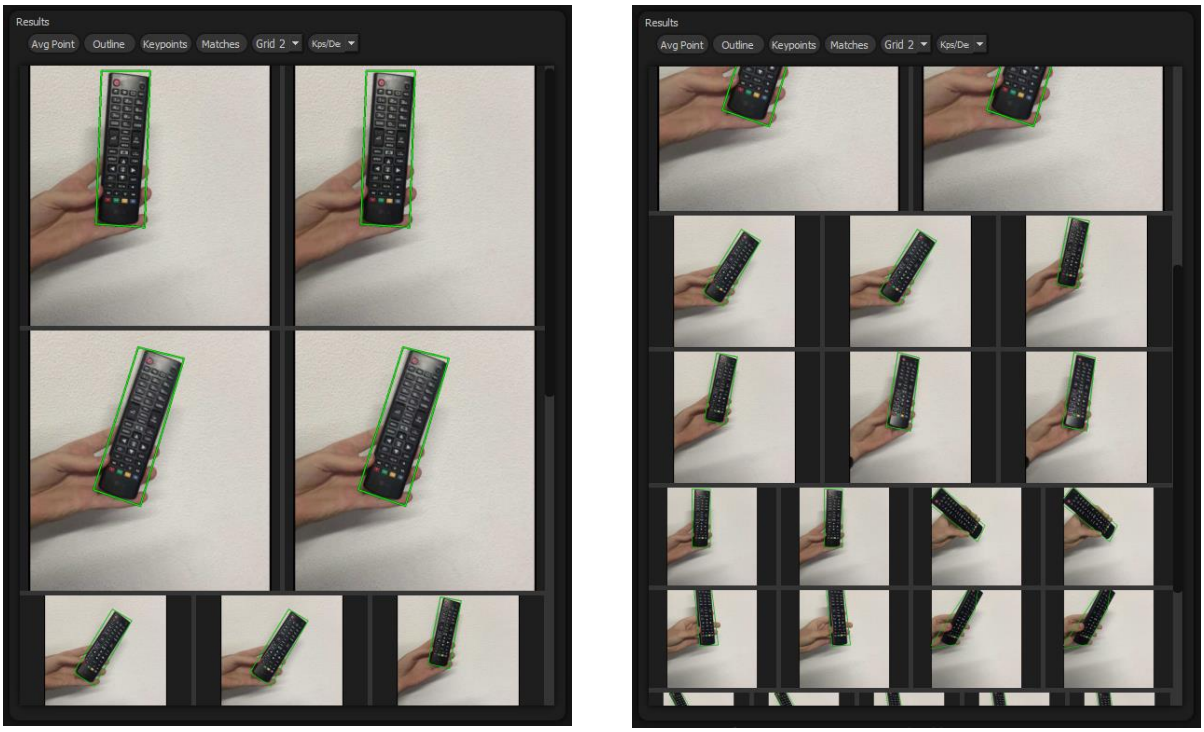


*Figura 3.17 - Modo de visualização Grelha Variável*

A Pilha foi desenvolvida para possibilitar uma interação mais direta, de forma semelhante ao manuseio de cartas em cima de uma mesa. É possível clicar e arrastar as imagens dos resultados, assim como aumentar e diminuir o seu tamanho com a roda do rato.

A posição (coordenadas horizontal e vertical) é aleatória para cada imagem, logo pode haver sobreposição parcial ou completa dos resultados. A ordem de sobreposição é determinada conforme o critério de ordenação, sendo que o primeiro resultado é colocado no *foreground*, o segundo é colocado no nível por trás do primeiro e assim por diante, até ao

último. Optou-se por colocar os resultados mais relevantes (segundo o critério de ordenação selecionado) à frente dos menos, pois provavelmente são os que mais valem a pena visualizar. Quando um resultado é clicado passa para o *foreground*, sobrepondo-se a todos os outros.

Estas formas de interação podem ser usadas para ganhar algum espaço, por exemplo ao arrastar *thumbnails* para fora da área visível ou ao diminuir bastante as suas dimensões, mas também para trazer qualquer resultado para o *foreground* e visualizá-lo em tamanho maior. É expectável que os pontos fortes deste modo sejam a interação mais direta e a capacidade de exploração, e que os pontos fracos sejam a dificuldade em visualizar os resultados rapidamente e a escalabilidade. Este modo está ilustrado nas figuras seguintes e foi inspirado em parte por (Rodden, 2002), como visto na secção de trabalhos relacionados.

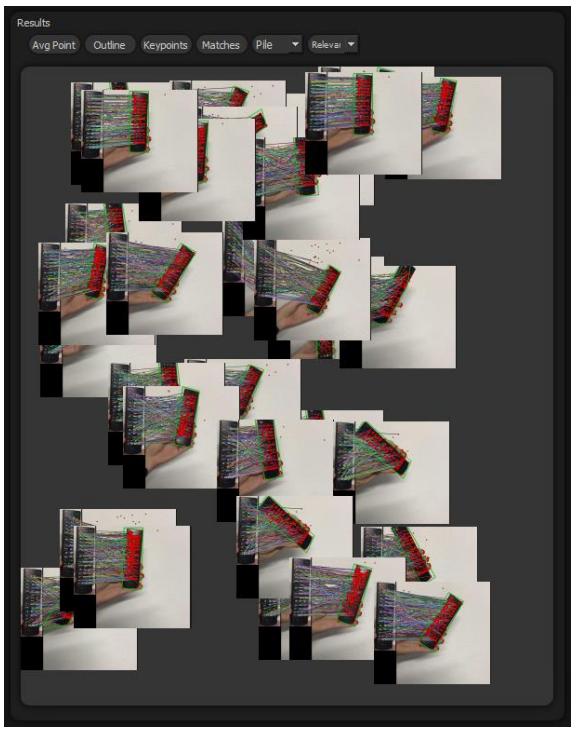


*Figura 3.18 - Modo de visualização Pilha*

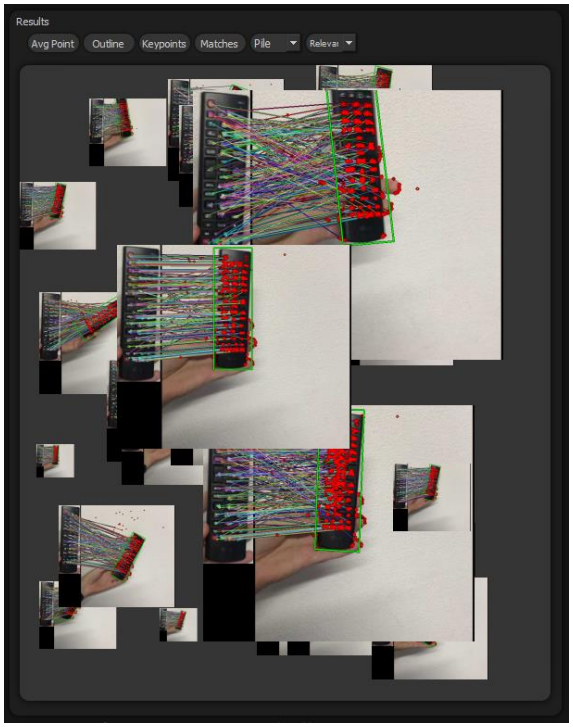


*Figura 3.19 - Imagens com tamanhos e ordem de sobreposição diferentes*

O modo de visualização Espiral apresenta os resultados em espiral. Este modo é inspirado nalgumas visualizações vistas em trabalhos relacionados (Heesch and Rüger, 2004) (Torres et al., n.d.), sendo que as imagens vão sendo dispostas à volta de um ponto central, com o raio definido de acordo com o traçado de uma espiral baseada numa espiral de Arquimedes. Esta espiral foi criada e modificada cuidadosamente para que possa acomodar muitos resultados sem convergir ou divergir demasiado, e para que seja capaz de apresentar imagens sequenciais com uma quantidade adequada de sobreposição.

A Espiral foi desenvolvida para tentar explorar a estética das *thumbnails*, como sugerido pelos autores no trabalho relacionado (André et al., 2009), analisado anteriormente. Tal como com a Pilha, é expectável que os resultados não possam ser visualizados rapidamente e que a escalabilidade não seja boa, mas espera-se que o utilizador seja encorajado a explorar mais. É possível aproximar e afastar o centro da espiral com a roda do rato, assim como visualizar qualquer resultado em detalhe no seu *tooltip* e arrastar qualquer imagem para qualquer sítio.

Esta última funcionalidade pode ser útil para ver determinada imagem em maior escala, quando esta é colocada no centro e a espiral é aproximada.

A Espiral apresenta as 85 imagens mais relevantes, de acordo com o critério de ordenação selecionado. Inicialmente a espiral encontra-se com o centro bastante próximo, e o resultado apresentado no centro é o menos relevante, de acordo com o critério selecionado. Optou-se por ordenar desta forma pois é mais fácil visualizar mais resultados com o centro da espiral afastado. Seguem-se figuras que ilustram este modo de visualização com diferentes aproximações ao centro da espiral.

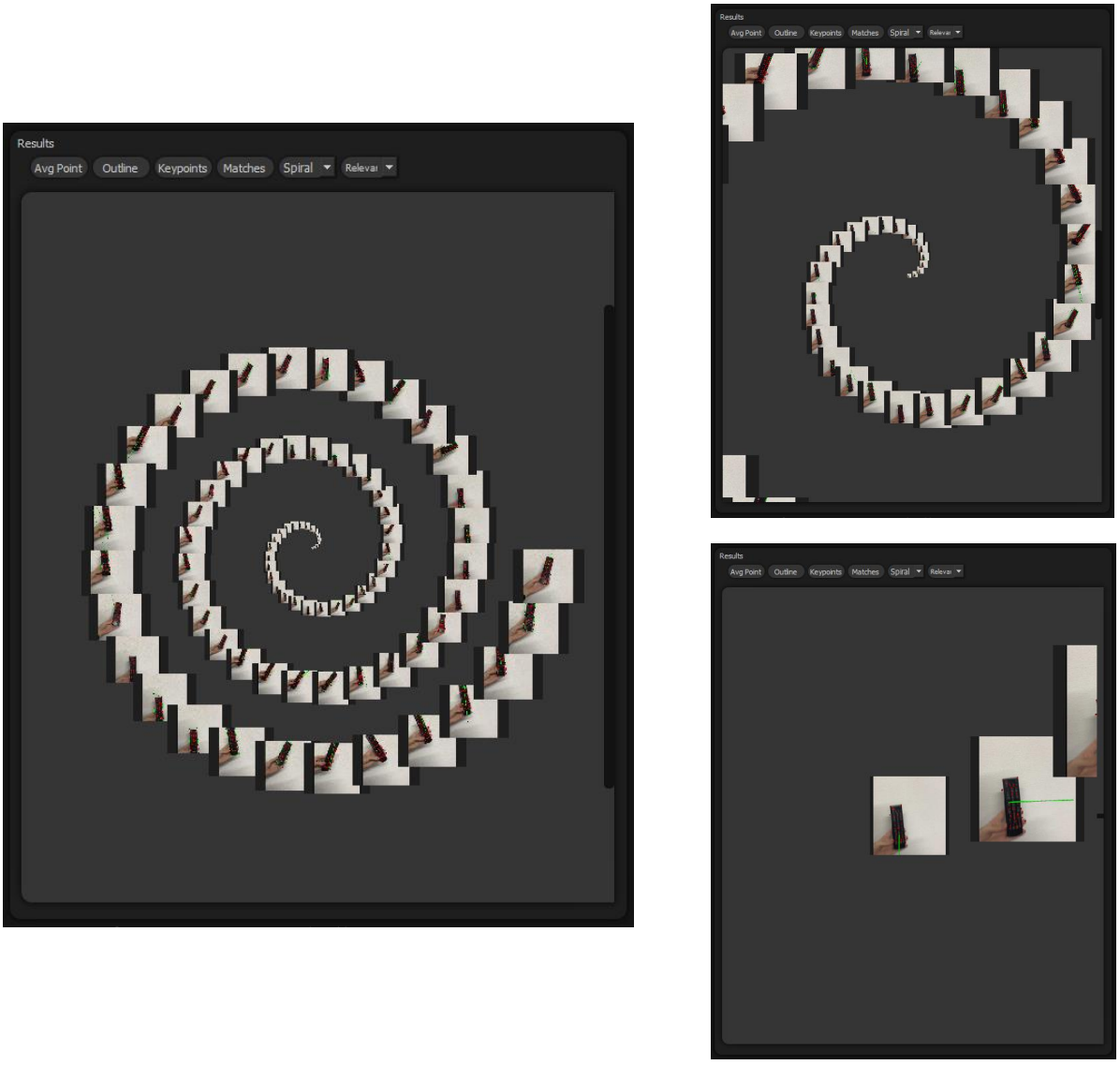


*Figura 3.20 - Esquerda - Modo de visualização Espiral; Direita – Diferentes aproximações ao centro da espiral*

Passando à interface de deteção de objetos, a secção de seleção de imagens é parecida à de *feature matching*, como exposto anteriormente. A secção da referência não é necessária para os métodos de pesquisa baseados em deteção de objetos, pelo que não existe. Além disso, a secção de resultados foi dividida em duas para aproveitar melhor o espaço deixado pela secção da referência. Assim sendo, há uma secção de opções onde é apresentada informação relativa ao processamento efetuado, assim como os botões para escolha de

*overlays*, reprodução/animação dos resultados, e *drop-downs* para seleção do modo de visualização e critério de ordenação. As informações apresentadas são a confiança média de todos os resultados, número médio de classes detetadas de todos os resultados, número médio de deteções de todos os resultados, confiança média do resultado atual, número de classes detetadas do resultado atual, número de deteções do resultado atual e nome do ficheiro do resultado atual. Os *overlays* que podem ser gerados e visualizados são "*Boxes*" (*bounding boxes*), "*Labels*" (classes) e "*Masks*" (*semantic/instance/panoptic segmentation masks*). Por último, há uma lista das classes que podem ser detetadas pelo método de pesquisa selecionado, sendo que cada classe nesta lista tem uma caixa de seleção para que seja possível indicar quais classes deverão ser detetadas. Quando o número de classes excede as dimensões do *widget* da lista aparece uma *scrollbar*. Esta secção é ilustrada na figura seguinte.


*Figura 3.21 - Secção de opções*

Como mencionado anteriormente, os modos de visualização são comuns às duas interfaces, ainda que as diferentes dimensões das secções influenciem o aspeto de alguns modos. Seguem-se algumas figuras que ilustram os modos de visualização no *layout* da interface de deteção de objetos.


*Figura 3.22 - Modo de visualização Resultado Único na interface de deteção de objetos*

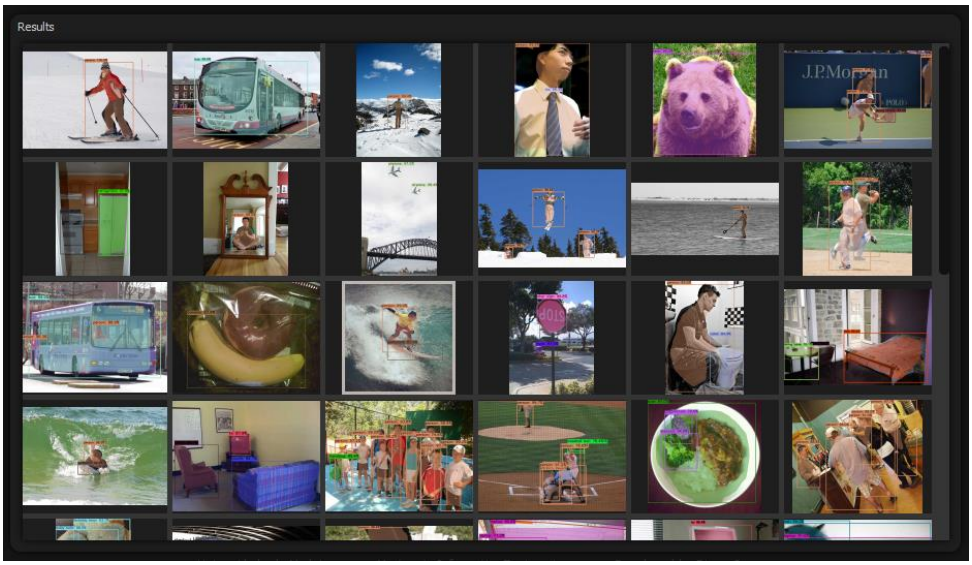


*Figura 3.23 - Modo de visualização Grelha Normal na interface de deteção de objetos*

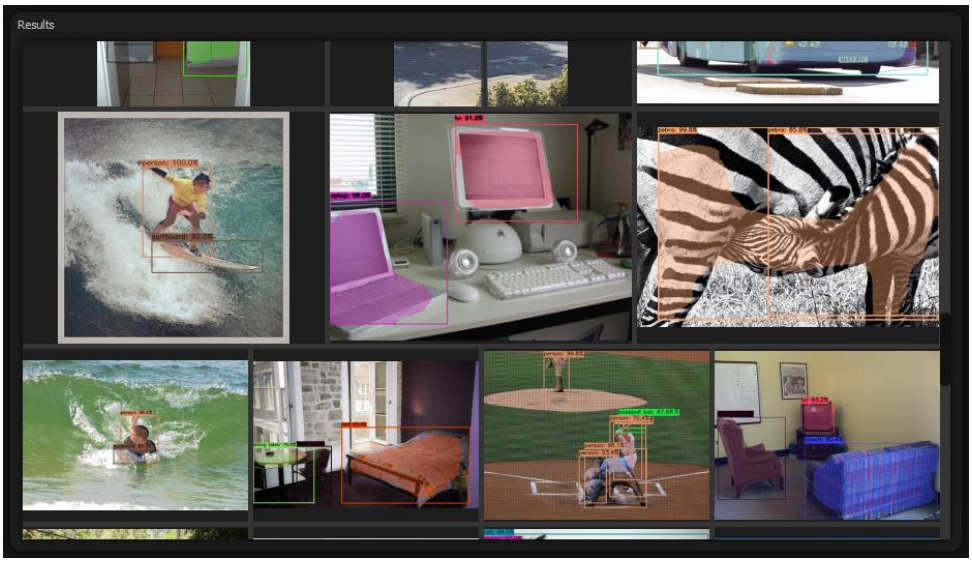


*Figura 3.24 - Modo de visualização Grelha Variável na interface de deteção de objetos*

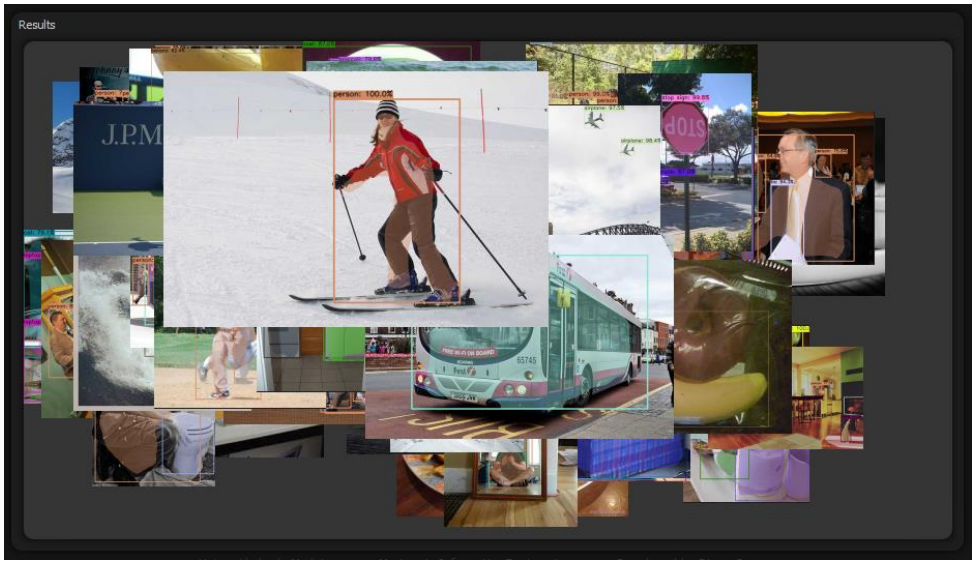


*Figura 3.25 - Modo de visualização Pilha na interface de deteção de objetos*

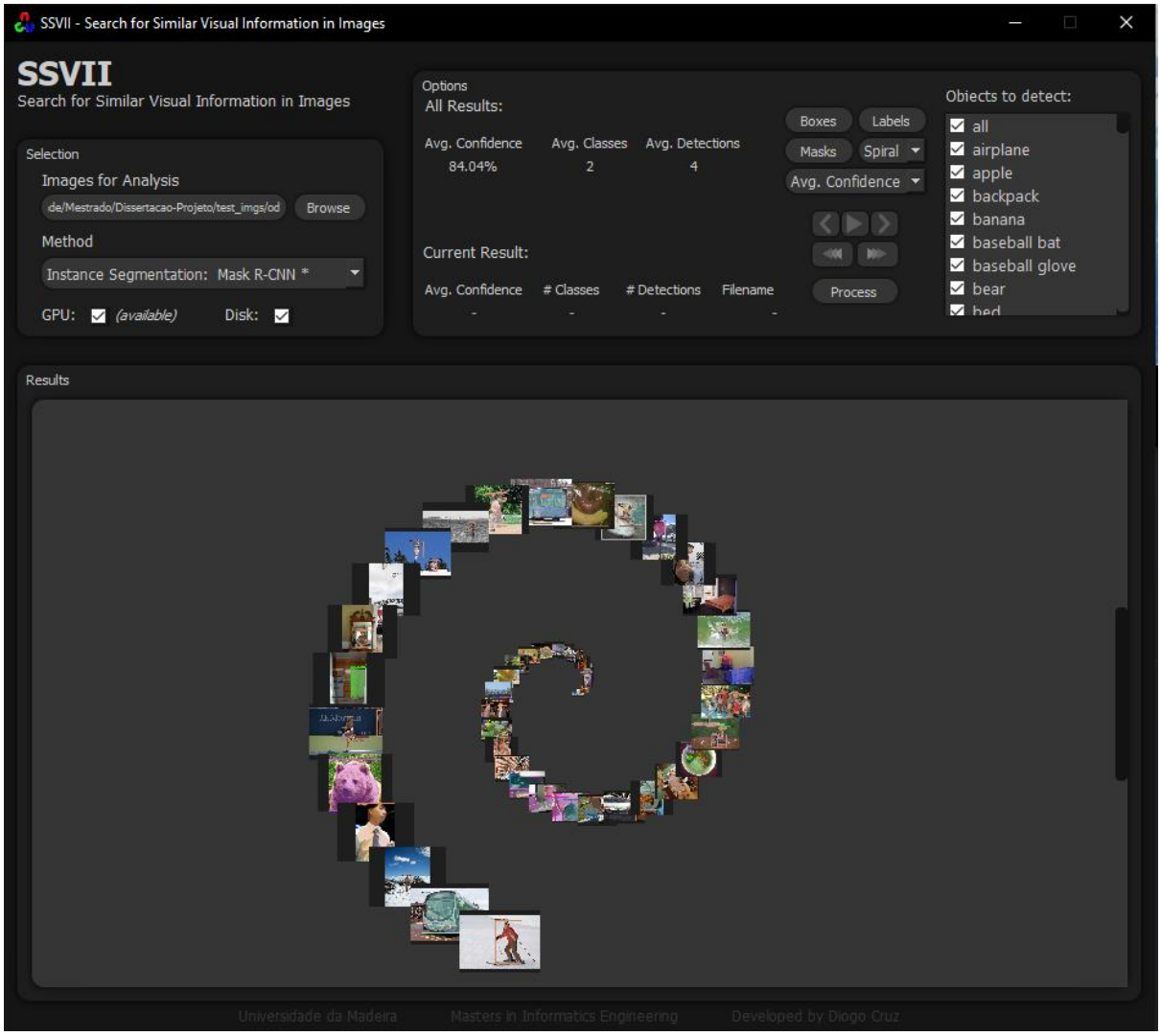


*Figura 3.26 - Modo de visualização Espiral na interface de deteção de objetos (sem cortes)*

# 4. Avaliação

Neste capítulo são avaliados vários aspetos da aplicação, incluindo o desempenho e a qualidade dos resultados dos métodos de pesquisa, e os modos de visualização de resultados. Cada um destes aspetos será testado com uma metodologia própria e os resultados serão analisados no contexto deste trabalho.

## 4.1. Desempenho

O objetivo destes testes passa por comparar os métodos uns com os outros e obter uma ideia do seu desempenho. Assim sendo, cada um destes testes foi realizado com um único conjunto de imagens que pode representar um caso típico de utilização. Todos os métodos foram testados com o processador *Intel® Core™ i7-6700K* @4.2GHz ("Intel Product Specifications," 2020). Os métodos que suportam *GPU* (*CUDA*) foram testados com a *GPU NVIDIA GeForce GTX 1080* @1.9GHz ("GTX 1080 Specs," 2020). Todos os testes foram repetidos com diferentes configurações de *hardware* para avaliar o seu impacto no desempenho. O código utilizado para fazer todos os testes está em anexo.

### 4.1.1. Testes de algoritmos de *feature matching*

Foi escolhido um objeto para ser usado como referência, neste caso um comando de um televisor, e foi gravado um vídeo de alta resolução e *framerate* onde o objeto em questão pode ser observado. As especificações da imagem e do vídeo foram escolhidas para que o teste fosse representativo de um caso de utilização muito exigente em termos de recursos computacionais, e encontram-se em detalhe de seguida.

| | |
|---|---|
| **Format:** | PNG |
| **Width:** | 884 pixels |
| **Height:** | 2963 pixels |
| **Bit depth:** | 32 bits |

*Figura 4.1 - Especificações da imagem de referência utilizada*

| | |
|---|---|
| **Format:** | MPEG-4 |
| **Bit rate:** | 20.7 Mb/s |
| **Width:** | 2 160 pixels |
| **Height:** | 2 304 pixels |
| **Frame rate:** | 59.940 FPS |
| **Bit depth:** | 8 bits |
| **Duration:** | 12.879s |

*Figura 4.2 - Especificações do vídeo utilizado*

Para fazer a extração de *features* foram utilizados o *SIFT*, *SURF*, *ORB*, *BRISK* e o *AKAZE*. Para fazer o *matching* foram utilizados dois *brute-force matchers* disponibilizados pelo *OpenCV*, dependendo do algoritmo usado no primeiro passo (*SIFT* e *SURF – L2 Norm*; *ORB*, *BRISK* e *AKAZE – Hamming Norm*). O *OpenCV* também foi utilizado para fazer o cálculo da homografia e transformação de perspetiva, e para desenhar os resultados.

Foram calculados os tempos de processamento de cada passo, mais especificamente o tempo de extração de *features*, o tempo de *matching*, o tempo de cálculo para estimar a localização do objeto detetado e o tempo para desenhar os *matches* e a *bounding box*. A

informação recolhida foi usada para calcular estatísticas. Todos os métodos foram testados utilizando *CPU*. O *SURF* e o *ORB* também foram testados utilizando *GPU*, sendo que são os únicos que têm implementações com suporte para tal no *OpenCV*. Todos os testes foram repetidos com quatro configurações de *hardware* distintas. Seguem-se algumas figuras que ilustram o processamento feito e os resultados obtidos.

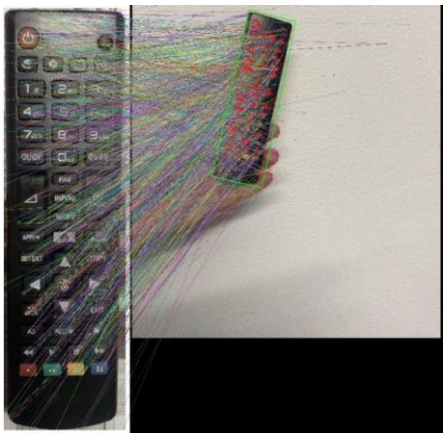

*Figura 4.3 - Representação do processamento feito e dos resultados obtidos utilizando o SIFT*

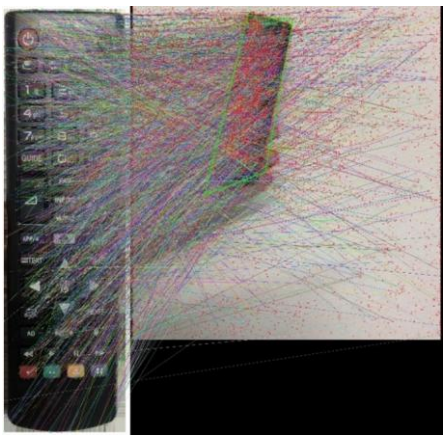

*Figura 4.4 - Representação do processamento feito e dos resultados obtidos utilizando o SURF*

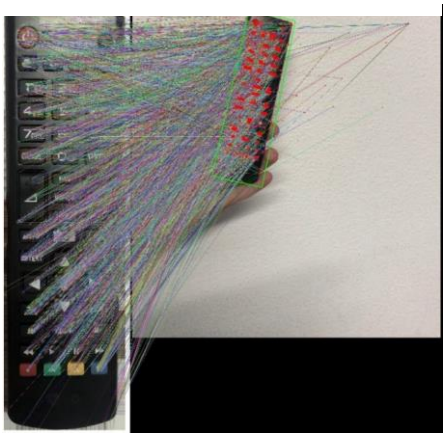

*Figura 4.5 - Representação do processamento feito e dos resultados obtidos utilizando o ORB*

*Figura 4.6 - Representação do processamento feito e dos resultados obtidos utilizando o BRISK*

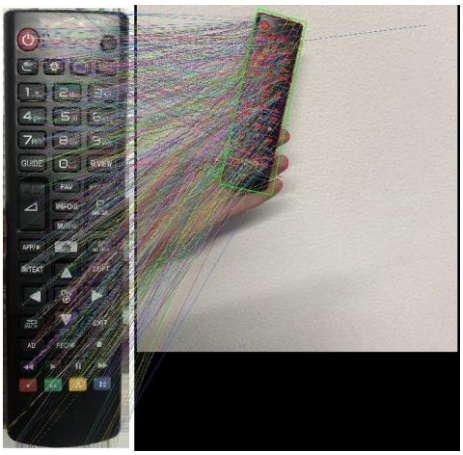

*Figura 4.7 - Representação do processamento feito e dos resultados obtidos utilizando o AKAZE*

### 4.1.1.1. Resultados e estatísticas

| | *SIFT* | *SURF* | *SURF* (*CUDA*) | *ORB* | *ORB* (*CUDA*) | *BRISK* | *AKAZE* |
|---|---|---|---|---|---|---|---|
| Média de *frames*/s | 0.974 | 0.809 | 2.369 | 1.943 | 0.861 | 4.787 | 1.332 |
| Tempo médio de extração de *features* (s) | 0.778 | 0.496 | 0.045 | 0.053 | 0.016 | 0.059 | 0.598 |
| Tempo médio de *matching* (s) | 0.111 | 0.53 | 0.001 | 0.256 | 0.012 | 0.025 | 0.024 |
| Tempo médio de cálculo da estimativa (s) | 0.05 | 0.059 | 0.07 | 0.055 | 0.178 | 0.048 | 0.049 |
| Tempo médio de desenho (s) | 0.073 | 0.138 | 0.285 | 0.137 | 0.934 | 0.064 | 0.066 |
| Média de *features* detetadas | 1186 | 5909 | 1000 | 4027 | 4058 | 1116 | 1092 |
| Média de *matches* | 846 | 1791 | 5203 | 2278 | 19848 | 698 | 706 |
| Média de *matches* obtidos por *feature* detetada | 0.713 | 0.303 | 5.203 | 0.565 | 4.891 | 0.625 | 0.647 |
| Média de *features* detetadas/s de extração | 1524 | 11913 | 22222 | 75981 | 253625 | 18915 | 1826 |
| Tempo total (s) | 792.613 | 954.826 | 325.868 | 397.242 | 896.514 | 161.257 | 579.496 |

*Tabela 4.1 - Desempenho dos métodos de feature matching com o CPU Intel® Core™ i7-6700K @4.2GHz ("Intel Product Specifications," 2020) e a GPU NVIDIA GeForce GTX 1080 @1.9GHz ("GTX 1080 Specs," 2020)*

Os testes com *GPU* foram repetidos com três outras configurações de *hardware* para que se pudesse analisar o seu impacto nos tempos obtidos. De seguida encontram-se listadas todas as quatro configurações comparadas e os resultados obtidos.

A. *CPU*: Intel 4770 @3.9GHz ("Intel Product Specifications," n.d.); *GPU*: NVIDIA GTX 1060 @1.7GHz ("GTX 1060 Specs," n.d.)
B. *CPU*: Intel i7 8750H @4.1GHz ("Intel Product Specifications," n.d.); *GPU*: NVIDIA GTX 1070 Max-Q @1.379GHz ("GTX 1070 Max-Q Specs," n.d.)
C. *CPU*: Intel i7 6700k @4.2GHz ("Intel Product Specifications," 2020); *GPU*: NVIDIA GTX 1080 @1.9GHz ("GTX 1080 Specs," 2020)
D. *CPU*: Intel i7 9700k @4.9GHz ("Intel Product Specifications," n.d.); *GPU*: NVIDIA RTX 2080 Ti @1.65GHz ("RTX 2080 Ti Specs," n.d., p. 208)

| | *SURF* (*CUDA*) | | | | *ORB* (*CUDA*) | | | |
|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | A | B | C | D |
| Média de *frames*/s | 1.997 | 1.324 | 2.369 | 2.317 | 0.746 | 0.462 | 0.861 | 0.828 |
| Tempo médio de extração de *features* (s) | 0.073 | 0.072 | 0.045 | 0.039 | 0.021 | 0.027 | 0.016 | 0.016 |
| Tempo médio de *matching* (s) | 0.002 | 0.002 | 0.001 | 0.001 | 0.025 | 0.02 | 0.012 | 0.01 |
| Tempo médio de cálculo da estimativa (s) | 0.089 | 0.141 | 0.07 | 0.077 | 0.221 | 0.358 | 0.178 | 0.196 |
| Tempo médio de desenho (s) | 0.317 | 0.508 | 0.285 | 0.296 | 1.051 | 1.728 | 0.934 | 0.968 |
| Tempo total (s) | 386.541 | 582.904 | 325.868 | 333.15 | 1035.35 | 1672.215 | 896.514 | 932.413 |

*Tabela 4.2 - Desempenho dos métodos de feature matching com diferentes configurações de hardware*

#### 4.1.1.2. Análise dos resultados

Nesta secção são analisados os resultados obtidos, começando pelos algoritmos que não utilizam *GPU*, e é feita uma comparação com conclusões obtidas em trabalhos relacionados. Esta análise é baseada nos resultados obtidos com o processador *Intel® Core™ i7-6700K* @4.2GHz ("Intel Product Specifications," 2020) e a *GPU NVIDIA GeForce GTX 1080* @1.9GHz ("GTX 1080 Specs," 2020), mas é semelhante para as outras configurações de *hardware*. Posteriormente é feita uma breve análise das diferenças de desempenho entre diferentes configurações de *hardware*.

Como era expectável, de acordo com as análises feitas em alguns trabalhos relacionados, observou-se que o *SIFT* teve um tempo médio de processamento relativamente longo, especialmente tendo em conta o número médio de *features* detetadas e de *matches* obtidos, que ficam longe de algoritmos como o *SURF* e o *ORB*. De qualquer forma, este algoritmo não se destaca pelos seus tempos de processamento, número de *features*/*matches* ou até pelo número de *features* detetadas por segundo, mas sim pela sua taxa de sucesso e precisão. Nesse aspeto, o *SIFT* obteve a melhor média de *matches* obtidos por cada *feature* detetada, tendo sido capaz de identificar o objeto corretamente, ainda que todos os outros algoritmos também o tenham conseguido, dado este ser um exemplo relativamente simples (o objeto não se encontra obstruído, o fundo destaca-se facilmente, …).

O *SURF* obteve o maior número médio de *features* detetadas, *features* estas que se encontram mais espalhadas por toda a imagem do que as detetadas por todos os outros algoritmos. Apesar do tempo de processamento ter sido o mais longo, o número de *features* detetadas por segundo é muito maior do que o do *SIFT*, o que era expectável visto que o *SURF* foi desenvolvido com esse propósito.

Relativamente ao *ORB*, observaram-se tempos de processamento melhores e um número de *matches* mais elevado que o do *SURF*, apesar de em média terem sido detetadas menos *features*. Ainda assim, o *ORB* foi o algoritmo mais rápido na fase de extração de *features* e foi o que detetou mais *features* por segundo. Outro ponto que se destacou foi o longo tempo médio de *matching*. Se tivermos em conta o número de *features* a serem processadas na fase de *matching*, constata-se que o *ORB* tem um melhor rácio número de *features*/tempo de *matching* do que o *SIFT* e o *SURF*. Esta diferença de tempos de *matching* provavelmente deve-se aos diferentes *brute-force matchers* utilizados. Com base nestes resultados podemos constatar que o *ORB* apresenta um dos melhores equilíbrios entre os diferentes fatores analisados.

O *BRISK* reduziu o tempo médio de processamento ainda mais, registando o menor tempo total de processamento de todos os algoritmos, ainda que também tenha registado o menor número médio de *matches*. Apesar de em média ter detetado menos *features* por segundo do que o *ORB*, obteve mais *matches* por cada *feature* detetada. Em média, foi capaz de processar cerca de 5 *frames* por segundo. Tendo em conta a resolução das imagens analisadas, o desempenho deste algoritmo parece promissor para aplicações em tempo real.

Segundo a análise feita em trabalhos relacionados, era expectável que o *AKAZE* obtivesse relativamente poucas *features* e *matches*. No entanto, o tempo de extração observado foi pior do que o esperado. Isto porque, não só nessa mesma análise foi dado como um dos mais rápidos, mas também porque obteve em média o menor número de *features* detetadas. Por

outro lado, obteve o segundo melhor registo relativamente ao número médio de *matches* obtidos por cada *feature* detetada.

Observou-se que o número de *features* detetadas está diretamente relacionado com o tempo de cálculo de *keypoints*/*descriptors*, tempo de *matching*, tempo de cálculo da estimativa e tempo de desenho nos cinco algoritmos.

Relativamente a possíveis melhorias, observou-se que o uso de *GPUs* tem potencial para diminuir drasticamente o tempo de processamento, como é o caso do *SURF* (*CUDA*) que foi cerca de três vezes mais rápido do que o *SURF*. Contudo, no seguimento do teste do *ORB* (*CUDA*) observou-se que a utilização de *GPU* foi prejudicial, não tendo se verificado o mesmo equilíbrio entre os diferentes fatores que se verificou no teste do *ORB* com *CPU*. Tipicamente isto pode acontecer quando o processamento a ser feito é reduzido ou quando já é bastante rápido, pelo que o tempo de transferência de dados de e para a *GPU* não compensa a maior velocidade de processamento. Isto é apoiado pelo facto de que se verificaram melhorias com a utilização da *GPU* no tempo de processamento do *SURF*, que era um dos algoritmos mais lentos, assim como tempos de processamento piores para o *ORB*, que era um dos mais rápidos. No entanto, se tivermos em atenção o grande aumento do número de *matches* e do tempo médio de desenho, e a correlação entre eles, torna-se claro que os maiores tempos de processamento se devem, pelo menos em grande parte, à maior quantidade de *features*/*matches* processados. Tendo isto em conta, parece menos provável que os tempos de transferência de dados de e para a *GPU* tenham sido o problema.

Ao contrário de todos os outros algoritmos, o *SURF* (*CUDA*) e o *ORB* (*CUDA*) obtiveram mais do que um *match* por cada *feature* detetada, resultando em números médios de *matches* muito mais altos do que as versões sem *GPU*. O *SURF* (*CUDA*) obteve em média menos *features* do que o *SURF*, enquanto o *ORB* (*CUDA*) detetou aproximadamente o mesmo número. No entanto ficam outras questões, como porque é que os métodos que utilizam *GPU* obtêm resultados tão diferentes dos métodos que utilizam *CPU*. O mais provável é que as próprias implementações no *OpenCV* sejam diferentes umas das outras.

Além de recorrer a *GPUs*, é possível controlar diferentes aspetos para diminuir o tempo de processamento. Por exemplo, é possível controlar o tempo de extração de *features* ao limitar o número de *features* detetadas, o que tem um impacto nos tempos de processamento das fases seguintes. Esta técnica foi analisada e comprovada no trabalho relacionado (Tareen and Saleem, 2018). É também possível controlar o tempo de cálculo da estimativa e o tempo de desenho. Estes são determinados em parte pelo número de matches considerados para o cálculo da homografia, pelo número de *matches* desenhados e pela quantidade de operações de desenho realizadas em geral (produção de *overlays*, ...). Contudo, observou-se que o tempo de cálculo da estimativa nem sempre varia consideravelmente com o aumento do número de *matches*, pelo que quaisquer possíveis ganhos seriam mínimos. Mais importante que isso é o impacto do número de *matches* na qualidade da estimativa, que através dos exemplos do *SIFT* e do *SURF* (figuras 4.3 e 4.4) se pode deduzir que nem sempre mais *matches* resultam numa melhor estimativa.

Concluindo, apesar de não ser rápido o suficiente para aplicações de tempo real, o desempenho dos métodos é aceitável, chegando a atingir cerca de 5 *frames* por segundo com *CPU*. Utilizando *GPU* não foi possível atingir uma *framerate* mais alta do que a *framerate* mais alta obtida com o *CPU* (o método mais rápido com *CPU* não tinha implementação para *GPU*).

Ainda assim, o *SURF* (*CUDA*) atingiu mais de 2 *frames* por segundo, cerca de três vezes mais rápido do que com *CPU*. De referir que as implementações que utilizam *GPU* no *OpenCV* são relativamente recentes e pouco documentadas, e é provável que o processamento feito não seja exatamente igual ao das implementações para *CPU*. Com base nas médias de *fps* observadas, o *BRISK* é o algoritmo mais indicado para aplicações em tempo real, seguido do *SURF* (*CUDA*) e do *ORB*. Por outro lado, o *SURF*, *ORB* (*CUDA*) e o *SIFT* são os menos adequados. Tendo em conta que a resolução das imagens utilizadas é bastante alta, é possível que o desempenho do *BRISK*, *SURF* (*CUDA*) e do *ORB* seja suficiente para aplicações em tempo real que utilizem imagens de resoluções mais razoáveis. De qualquer forma, o desempenho de todos os métodos de pesquisa avaliados satisfaz o que se pretendia para a aplicação.

Relativamente aos testes com diferentes configurações de *hardware*, observaram-se diferenças de desempenho em algumas fases do processamento. As *GPUs* mais capazes tiveram melhor desempenho nas fases de processamento que as utilizam, nomeadamente os processos de extração e *matching* de *features*. Curiosamente, o *CPU* utilizado teve um impacto maior do que o esperado no desempenho, nas fases de cálculo da estimativa e desenho. No entanto, tendo em conta que os tempos de processamento já são relativamente curtos, as diferenças de desempenho não são muito grandes. À exceção da configuração de *hardware* B, que é de um computador portátil, a diferença de desempenho num caso de utilização real não seria muito percetível, dado que em média é apenas cerca de 0.1 a 0.3 *fps*. Estes ganhos de desempenho obtidos com *hardware* mais capaz tornam-se ainda menos significativos quando são consideradas as diferenças de custos monetários. Ainda assim, as implementações dos métodos testados que suportam *GPU* (incluídas no *OpenCV*) são relativamente recentes, pelo que é perfeitamente possível que não estejam muito otimizadas. É expectável que as diferenças de desempenho entre *CPU* e *GPU* sejam mais significativas nos testes de modelos de deteção de objetos, discutidos na secção seguinte.

Como mencionado nesta secção e como se concluiu na análise de trabalhos relacionados, o desempenho dos algoritmos de *feature matching* depende de muitos fatores, como limites impostos no número de *features* a detetar, diferenças de escala, rotação e perspetiva, imagens utilizadas, entre outros. Isto é facilmente verificável se compararmos os diferentes resultados obtidos em diferentes trabalhos relacionados, ou se os compararmos com os resultados obtidos nestes testes, apresentados de seguida:

- Os métodos ordenados em ordem decrescente pelas *framerates* obtidas são: *BRISK* > *SURF* (*CUDA*) > *ORB* > *AKAZE* > *SIFT* > *ORB* (*CUDA*) > *SURF*
- Os métodos ordenados do mais rápido para o mais lento em média a detetar/extrair *keypoints* são: *ORB* (*CUDA*) > *SURF* (*CUDA*) > *ORB* > *BRISK* > *SURF* > *AKAZE* > *SIFT*
- Os métodos ordenados do mais rápido para o mais lento em média na fase de *matching* são: *SURF* (*CUDA*) > *ORB* (*CUDA*) > *AKAZE* > *BRISK* > *SIFT* > *ORB* > *SURF*
- Os métodos ordenados do mais rápido para o mais lento em média a calcular a estimativa são: *BRISK* > *AKAZE* > *SIFT* > *ORB* > *SURF* > *SURF* (*CUDA*) > *ORB* (*CUDA*)
- Os métodos ordenados do mais rápido para o mais lento em média na fase de desenho são: *BRISK* > *AKAZE* > *SIFT* > *ORB* > *SURF* > *SURF* (*CUDA*) > *ORB* (*CUDA*)

- Os métodos ordenados dos que tiveram em média mais *features* detetadas para os que tiveram menos são: *SURF* > *ORB* (*CUDA*) > *ORB* > *SIFT* > *BRISK* > *AKAZE* > *SURF* (*CUDA*)
- Os métodos ordenados dos que tiveram em média mais *matches* para os que tiveram menos são: *ORB* (*CUDA*) > *SURF* (*CUDA*) > *ORB* > *SURF* > *SIFT* > *AKAZE* > *BRISK*
- Os métodos ordenados dos que tiveram em média mais *matches* por cada *feature* detetada para os que tiveram menos são: *SURF* (*CUDA*) > *ORB* (*CUDA*) > *SIFT* > *AKAZE* > *BRISK* > *ORB* > *SURF*
- Os métodos ordenados dos que detetaram mais *features* por segundo para os que detetaram menos são: *ORB* (*CUDA*) > *ORB* > *SURF* (*CUDA*) > *BRISK* > *SURF* > *AKAZE* > *SIFT*

Entre estes resultados destaca-se o *BRISK* pela sua velocidade de processamento, sendo o mais indicado para aplicações em tempo real. O *ORB* destaca-se como tendo o melhor equilíbrio entre os diferentes fatores. O *SURF* (*CUDA*) e o *ORB* (*CUDA*) destacam-se nas fases de processamento que utilizam *GPU*, nomeadamente na fase de extração de *features* e na fase de *matching*, onde são os melhores.

### 4.1.2. Testes de modelos de deteção de objetos

Para estes testes foram analisados os *frames* de um excerto de um vídeo de monitorização de transito, disponível em (*YouTube*, 2020). As especificações do vídeo (figura seguinte) foram escolhidas para que os testes fossem bastante exigentes em termos de recursos computacionais. Ao analisar um exemplo muito exigente pretende-se observar o desempenho que é possível obter e que limites poderão existir. Se o desempenho nestes testes for satisfatório então será seguro assumir que também o será em qualquer caso de aplicação mais típico e menos exigente.

| | |
|---|---|
| **Format:** | Matroska (AVC) |
| **Bit rate:** | 17.0 Mb/s |
| **Width:** | 3 840 pixels |
| **Height:** | 2 160 pixels |
| **Frame rate:** | 29.970 FPS |
| **Bit depth:** | 8 bits |
| **Duration:** | 30.063s |

*Figura 4.8 - Especificações do vídeo utilizado*

Os modelos testados foram o *YOLOv3*, *SSD* (*MobileNets*) e *Mask R-CNN*, disponibilizados em (Rosebrock, 2018a), (Rosebrock, 2017) e (Rosebrock, 2018b), respetivamente. Os modelos *YOLOv3* e *Mask R-CNN* foram treinados com oitenta e noventa classes do *dataset COCO* ("COCO Dataset," 2020), respetivamente. O *SSD* foi treinado com vinte classes do *dataset COCO* mais o *background* e posteriormente foi afinado com o *dataset PASCAL VOC* ("PASCAL VOC," 2020). O código para realizar os testes foi escrito com base nos exemplos apresentados nas fontes mencionadas acima, onde os modelos foram disponibilizados, assim como em (Rosebrock, 2020). O *OpenCV* e o *numpy* foram utilizados para fazer todo o processamento

de imagem, assim como o tratamento de dados e produção das imagens de *output*. Todos estes modelos suportam a utilização de *GPU* (*CUDA*) através do *OpenCV*. Como tal, cada modelo foi testado utilizando *CPU* e *GPU*. Além disto, todos os modelos foram testados com diferentes configurações de *hardware* para analisar o seu impacto no desempenho. Seguem-se algumas figuras que ilustram o processamento feito e os resultados obtidos.

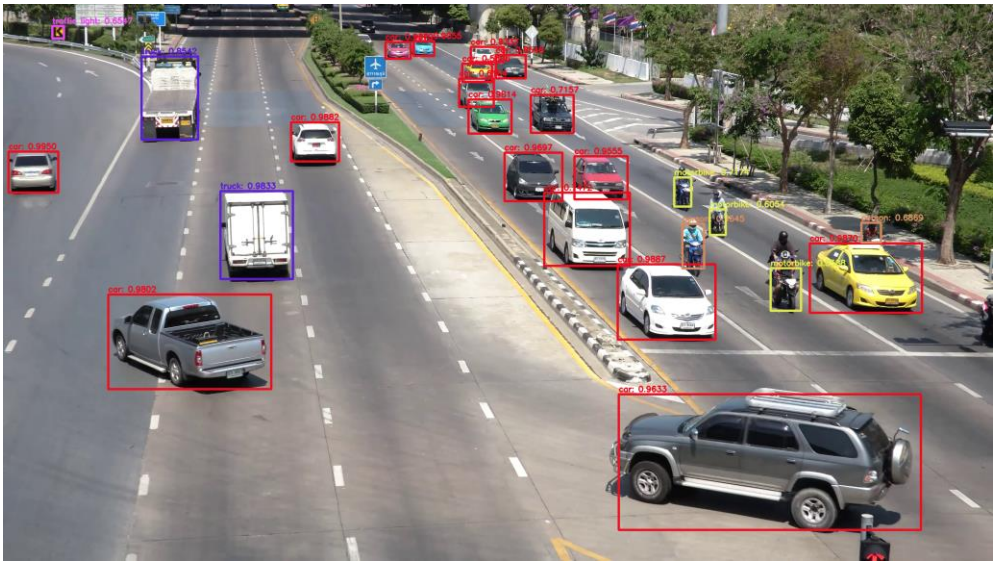

*Figura 4.9 - Exemplo de deteção com o YOLOv3*

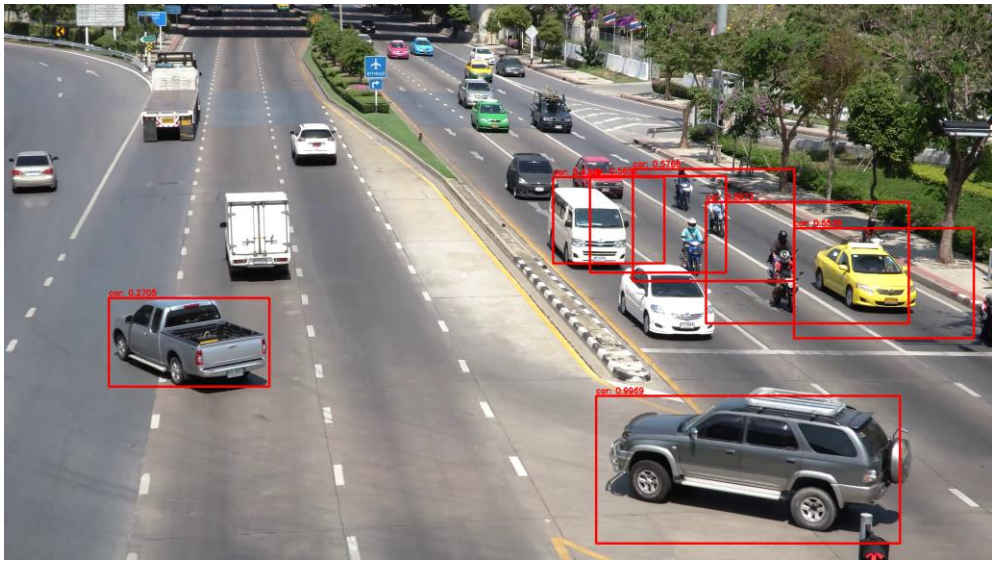

*Figura 4.10 - Exemplo de deteção com o SSD*

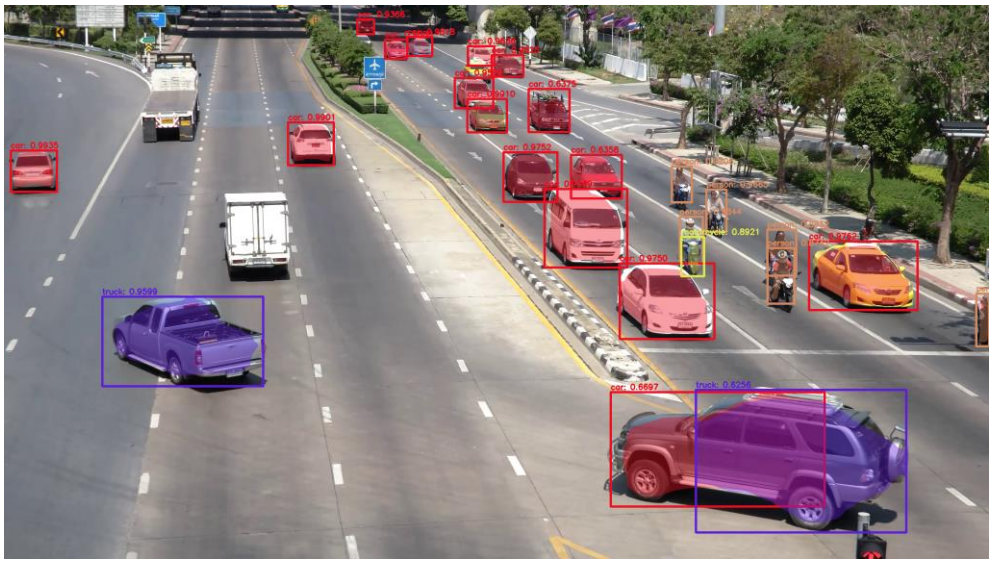

*Figura 4.11 - Exemplo de deteção com o Mask R-CNN*

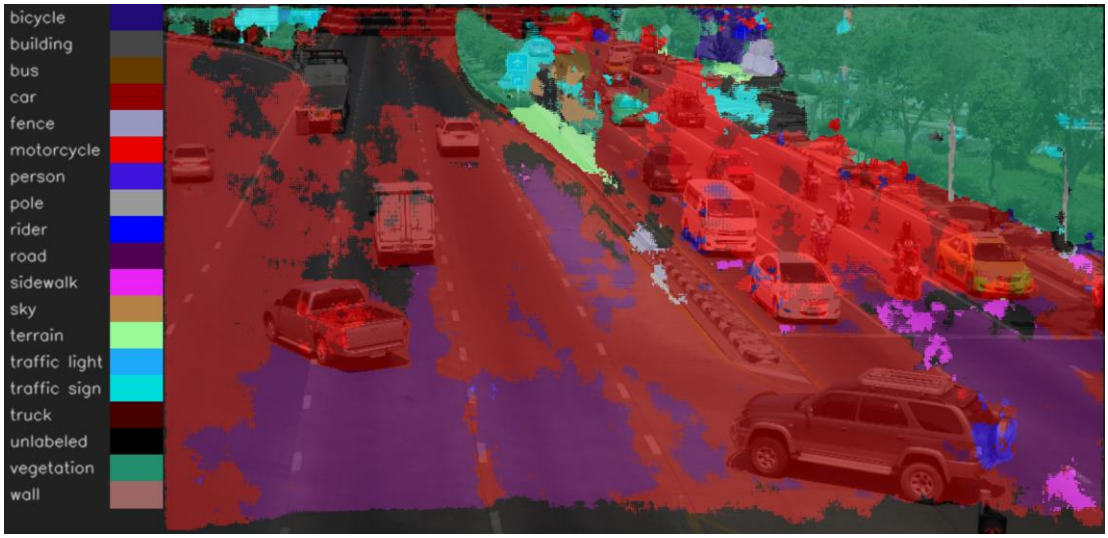


*Figura 4.12 - Exemplo de deteção com o ENet*

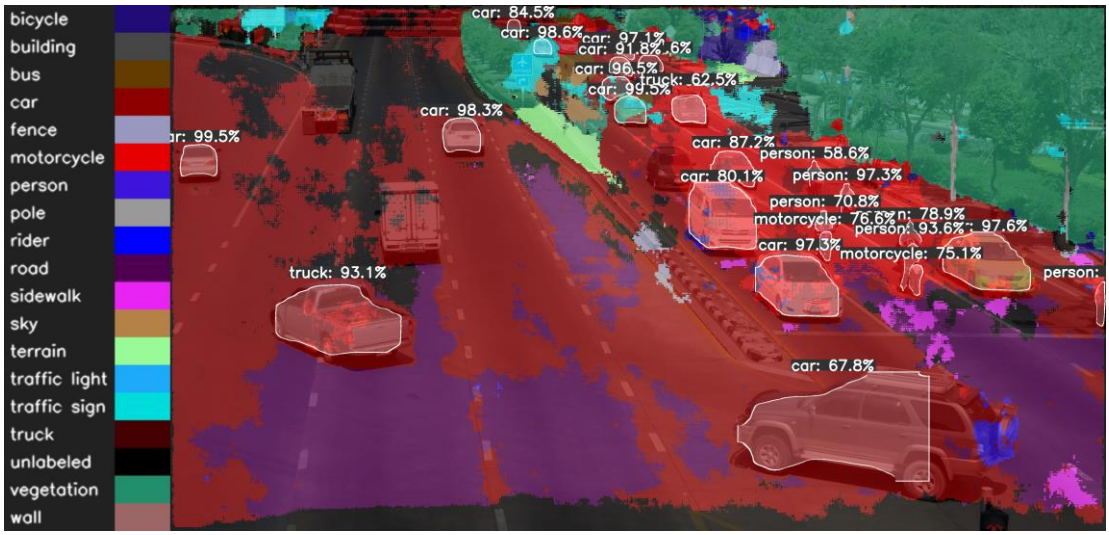


*Figura 4.13 - Exemplo de deteção com o ENet + Mask R-CNN*

### 4.1.2.1. Resultados e estatísticas

| | *YOLOv3* | *YOLOv3 (CUDA)* | *SSD* | *SSD (CUDA)* | *Mask R-CNN* | *Mask R-CNN (CUDA)* | *ENet* | *ENet (CUDA)* | *ENet + Mask R-CNN* | *ENet + Mask R-CNN (CUDA)* |
|---|---|---|---|---|---|---|---|---|---|---|
| Média de *frames*/s | 3.266 | 9.763 | 16.606 | 8.593 | 0.261 | 1.698 | 1.832 | 2.916 | 0.208 | 0.699 |
| Tempo médio de deteção (s) | 0.228 | 0.026 | 0.026 | 0.082 | 3.738 | 0.501 | 0.257 | 0.054 | 3.963 | 0.578 |
| Tempo médio de processamento de resultados e desenho (s) | 0.044 | 0.042 | 0.001 | 0.001 | 0.055 | 0.053 | 0.253 | 0.253 | 0.811 | 0.815 |
| Média de objetos relevantes detetados | 24 | 24 | 4 | 4 | 26 | 26 | - | - | - | - |
| Total de objetos relevantes detetados | 21766 | 21766 | 3371 | 3371 | 23697 | 23697 | - | - | - | - |
| Tempo total (s) | 275.554 | 92.183 | 54.198 | 104.741 | 3448.046 | 530.051 | 491.139 | 308.677 | 4331.738 | 1286.967 |

*Tabela 4.3 - Resultados dos testes dos modelos de deteção de objetos*

Os testes com *GPU* foram repetidos com três outras configurações de *hardware* para analisar o seu impacto nos tempos obtidos. De seguida encontram-se listadas todas as quatro configurações comparadas e os resultados obtidos.

A. *CPU*: Intel 4770 @3.9GHz ("Intel Product Specifications," n.d.); *GPU*: NVIDIA GTX 1060 @1.7GHz ("GTX 1060 Specs," n.d.)
B. *CPU*: Intel i7 8750H @4.1GHz ("Intel Product Specifications," n.d.); *GPU*: NVIDIA GTX 1070 Max-Q @1.379GHz ("GTX 1070 Max-Q Specs," n.d.)
C. *CPU*: Intel i7 6700k @4.2GHz ("Intel Product Specifications," 2020); *GPU*: NVIDIA GTX 1080 @1.9GHz ("GTX 1080 Specs," 2020)
D. *CPU*: Intel i7 9700k @4.9GHz ("Intel Product Specifications," n.d.); *GPU*: NVIDIA RTX 2080 Ti @1.65GHz ("RTX 2080 Ti Specs," n.d., p. 208)

| | *YOLOv3 (CUDA)* | | | | *SSD (CUDA)* | | | | *Mask R-CNN (CUDA)* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | A | B | C | D | A | B | C | D |
| Média de *frames*/s | 8.393 | 6.01 | 9.763 | 10.336 | 8.052 | 4.707 | 8.593 | 17.075 | 1.332 | 1.017 | 1.698 | 1.819 |
| Tempo médio de deteção (s) | 0.04 | 0.036 | 0.026 | 0.022 | 0.093 | 0.157 | 0.082 | 0.028 | 0.656 | 0.832 | 0.501 | 0.464 |
| Tempo médio de processamento de resultados e desenho (s) | 0.046 | 0.076 | 0.042 | 0.045 | 0.001 | 0.001 | 0.001 | 0.001 | 0.061 | 0.095 | 0.053 | 0.054 |
| Tempo total (s) | 107.231 | 149.742 | 92.183 | 87.075 | 111.778 | 191.193 | 104.741 | 52.708 | 675.638 | 885.038 | 530.051 | 494.833 |

*Tabela 4.4 - Comparação do desempenho de métodos (CUDA) com diferentes configurações de hardware*

| | *ENet (CUDA)* | | | | *ENet + Mask R-CNN (CUDA)* | | | |
|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | A | B | C | D |
| Média de *frames*/s | 2.895 | 2.032 | 2.916 | 3.527 | 0.626 | 0.465 | 0.699 | 0.808 |
| Tempo médio de deteção (s) | 0.058 | 0.07 | 0.054 | 0.04 | 0.71 | 0.915 | 0.578 | 0.5 |
| Tempo médio de processamento de resultados e desenho (s) | 0.255 | 0.369 | 0.253 | 0.213 | 0.856 | 1.18 | 0.815 | 0.707 |
| Tempo total (s) | 310.866 | 442.824 | 308.677 | 255.139 | 1438.795 | 1935.327 | 1286.967 | 1114.47 |

*Tabela 4.5 - Comparação do desempenho de métodos (CUDA) com diferentes configurações de hardware (continuação)*

### 4.1.2.2. Análise dos resultados

Esta análise é baseada nos resultados obtidos com o processador *Intel® Core™ i7-6700K* @4.2GHz ("Intel Product Specifications," 2020) e a *GPU NVIDIA GeForce GTX 1080* @1.9GHz ("GTX 1080 Specs," 2020), mas é semelhante para as outras configurações de *hardware*. Posteriormente é feita uma breve análise dos resultados obtidos com as diferentes configurações de *hardware*.

Os resultados destes testes são muito satisfatórios. Observou-se que a utilização de *GPU* pode reduzir drasticamente os tempos de processamento, como são os casos do *YOLOv3* (*CUDA*) (aproximadamente três vezes mais rápido que o *YOLOv3*), do *Mask R-CNN* (*CUDA*) (aproximadamente seis vezes e meia mais rápido que o *Mask R-CNN*), do *ENet* (*CUDA*) (aproximadamente uma vez e meia mais rápido que o *ENet*) e do *ENet + Mask R-CNN* (*CUDA*) (aproximadamente três vezes e meia mais rápido que o *ENet + Mask R-CNN*). Ainda assim, a utilização de *GPU* nem sempre foi benéfica, com o *SSD* (*CUDA*) levando aproximadamente o dobro do tempo do *SSD*. Tal como mencionado nos testes da secção anterior, isto provavelmente deve-se ao processamento do *SSD* já ser muito rápido, pelo que a maior velocidade de processamento da *GPU* não compensa os tempos de transferência de dados associados. De notar que se verificou uma exceção com a configuração de *hardware* D, com a qual o *SSD* (*CUDA*) obteve em média aproximadamente 17 *fps*.

Apesar de ter sido o mais rápido com uma média de mais de 16 *fps*, o *SSD* nem sempre permitiu obter resultados de grande qualidade, como se pode observar na representação visual dos resultados obtidos (figura 4.10). Foram detetados em média apenas quatro objetos e as *bounding boxes* são pouco precisas. Se tivermos em conta que a maioria dos objetos visíveis nos *frames* do vídeo eram carros, que o *SSD* foi treinado para detetar carros e que os outros modelos detetaram em média mais de vinte objetos, torna-se claro que os resultados não são tão bons. Por outro lado, o *YOLOv3* e o *Mask R-CNN* foram capazes de detetar a maioria dos objetos visíveis para os quais foram treinados, e as *bounding boxes*/*segmentation masks* foram bastante precisas. A qualidade dos resultados é avaliada em mais detalhe na secção seguinte.

Quanto ao *ENet*, os tempos de inferência foram bastante baixos, particularmente com *GPU*, tal como era expectável no decorrer da análise feita na secção de trabalhos relacionados. O tempo médio de processamento e desenho dos resultados é significativamente maior do que o dos outros modelos, o que é normal tendo em conta que abrange todos os pixéis e que é necessário criar uma legenda.

Os tempos de inferência do *ENet + Mask R-CNN* são semelhantes à soma dos tempos de inferência dos modelos *ENet* e *Mask R-CNN*, ainda que o tempo médio de processamento e desenho de resultados seja significativamente maior do que qualquer um dos modelos testados. Isto deve-se em parte ao processamento adicional que tem de ser feito para obter o contorno de cada objeto detetado pelo *Mask R-CNN* e para combinar os resultados obtidos por ambos os modelos.

A maior parte dos modelos foi capaz de atingir uma velocidade de processamento boa (com e/ou sem *GPU*), tendo em conta a qualidade dos resultados de cada um. Entre os métodos testados, os que utilizam o modelo *Mask R-CNN* foram os que levaram mais tempo, particularmente o *ENet + Mask R-CNN*. Uma parte significativa do tempo total de processamento de muitos dos modelos analisados deve-se ao processamento de resultados e desenho das visualizações, sendo que supera o tempo médio de inferência de alguns deles. Optou-se por incluir estes tempos nos resultados dos testes porque são uma etapa importante e porque cada método faz um processamento próprio.

Relativamente aos testes com diferentes configurações de *hardware*, observaram-se diferenças de desempenho bastante significativas com alguns métodos de pesquisa. Como era expectável, as *GPUs* mais capazes tiveram melhor desempenho na fase de processamento que as utiliza, neste caso no processamento efetuado pelas redes neurais, e o seu impacto foi maior do que nos testes de métodos de *feature matching*. Tal como nesses testes, o *CPU* utilizado teve um impacto maior do que o esperado no desempenho, nas fases de processamento e desenho de resultados. As diferenças de desempenho entre configurações de *hardware* não são muito grandes, à exceção do *YOLOv3* e do *SSD*, onde se observaram diferenças entre a configuração menos capaz (B) e a mais capaz (D) de cerca de 4 e 12 *fps* em média (72% e 263%), respetivamente. Quanto aos outros métodos, a diferença de desempenho num caso de utilização real não seria muito percetível, dado que em média é inferior a 1 *fps*. A diferença de desempenho entre a configuração de *hardware* mais cara (D) e a mais barata (A) é cerca de 2.5 *fps* em média.

## 4.2. Qualidade dos resultados

Os métodos de pesquisa disponíveis na aplicação podem ser divididos em dois grupos que fazem abordagens diferentes. Um grupo faz a extração e *matching* de *features* para localizar uma região de interesse, enquanto o outro deteta, localiza e identifica objetos.

Os métodos de deteção de objetos podem ser avaliados formalmente com recurso a métricas como "*Intersection over Union*" (*IoU*), "*Average Precision*" (*AP*) e "*Mean Average Precision*" (*mAP*). Contudo, os métodos de *feature matching* tipicamente não são avaliados com estas métricas pois não são adequadas. Isto porque, apesar destes métodos idealmente retornarem *bounding boxes* que delimitam a região de interesse, a verdade é que frequentemente a previsão obtida com as técnicas de homografia não constitui uma *bounding box* retangular, o que dificulta imenso o cálculo destas métricas. De qualquer forma, foi feita uma análise pormenorizada dos resultados obtidos pelos diferentes métodos de *feature matching* implementados (*SIFT, SURF*, *ORB*, …), em trabalhos relacionados estudados em secções anteriores.

Os métodos que detetam objetos através de *segmentation masks* também não são muito adequados para avaliação com estas métricas, dado que não produzem *bounding boxes*. Ainda assim, é possível calcular interseções e uniões de *segmentation masks*, por exemplo, e depois calcular a precisão, revocação, *AP* e *mAP* com base nesses valores. Como a deteção por segmentação é relativamente mais recente do que a deteção com *bounding boxes*, ainda não existem métricas adotadas pela maioria da comunidade. Por essa razão optou-se por não avaliar os métodos em questão (*Mask-RCNN* e o *ENet*). Em vez disso, refere-se aos valores anunciados pelos seus autores nas publicações originais. De notar que o *Mask-RCNN* também produz *bounding boxes*, pelo que será avaliado nesse aspeto. Na subsecção seguinte é indicada a metodologia seguida para calcular as diferentes métricas. Posteriormente é feita uma breve análise dos resultados obtidos.

### 4.2.1. Metodologia e métricas

A metodologia seguida é a apresentada em (Padilla et al., 2020) e (Padilla, 2020). Esta metodologia e as métricas calculadas são as usadas mais frequentemente em competições de deteção de objetos, tais como algumas mencionadas anteriormente na secção de trabalhos relacionados (*PASCAL VOC* e *COCO*). Os autores desenvolveram funções fáceis de aplicar a qualquer modelo de deteção de objetos, sem haver necessidade de conformar com formatos específicos que são muitas vezes adotados em competições. Os autores indicam que compararam cuidadosamente os resultados obtidos com a sua implementação aos resultados obtidos com implementações oficiais e verificaram que são exatamente iguais.

O processo começa por escolher um *dataset* onde aplicar o modelo que queremos avaliar. Para realizar estes testes foi utilizado parte do *dataset COCO* (*Common Objects in Context*) (“COCO Dataset,” 2020), mais especificamente o *subdataset* para validação de 2017 que conta com 5000 imagens, para calcular as métricas dos métodos de pesquisa cujos modelos foram treinados também com um *subdataset COCO*. De mencionar que todos os modelos disponíveis na aplicação exceto o *ENet* foram treinados com o *dataset COCO*. Para calcular as métricas do *ENet* foi utilizado um *subdataset* do *dataset Cityscapes* (“Cityscapes Dataset,” n.d.) com o qual o modelo foi treinado.

Posteriormente é necessário obter as coordenadas e tamanho da *ground-truth bounding box* de cada deteção de cada imagem do *dataset*. Esta informação é normalmente disponibilizada juntamente com o *dataset*. No caso do *dataset COCO*, esta informação é disponibilizada sob a forma de um ficheiro *JSON*. A ferramenta disponibilizada em (Padilla, 2020) necessita que esta informação seja fornecida através de ficheiros de texto, sendo que cada imagem deverá ter um ficheiro de texto associado com o seu id como nome. Nestes ficheiros, cada linha deverá representar uma deteção na imagem respetiva, no formato <nome da classe> <x1> <y1> <largura> <altura>. Para tal, foi escrito e utilizado um *script* para percorrer todas as imagens no *dataset* utilizado e guardar a sua informação em ficheiros de texto com o formato necessário.

Além disto, é necessário preparar a mesma informação para as deteções obtidas pelo modelo a avaliar. Assim sendo, todas as imagens do *dataset* utilizado foram processadas com cada modelo e foram criados ficheiros de texto tal como anteriormente. Desta forma, cada

imagem está associada a dois ficheiros, um com a informação correta e outro com as previsões do modelo. Por fim, estes dois conjuntos de ficheiros são colocados em pastas separadas que são passadas como parâmetros ao *script* disponibilizado pelos autores.

O primeiro passo passa por calcular a métrica "*Intersection Over Union*" (*IoU*) para cada deteção. Esta métrica já foi abordada anteriormente na secção de trabalhos relacionados. Recapitulando, a *IoU* consiste no quociente entre a área de interseção e a área de união das *bounding boxes* (*ground-truth* e previsão). A *IoU* é depois usada para classificar cada deteção. As classificações possíveis são as seguintes:

- *True Positive* (*TP*) – deteção correta. Considera-se que uma deteção é *TP* quando a sua *IoU* é igual ou superior a um determinado *threshold* (tipicamente entre 50% e 95%).
- *False Positive* (*FP*) – deteção incorreta. Considera-se que uma deteção é *FP* quando a sua *IoU* é inferior ao *threshold*.
- *False Negative* (*FN*) – deteção em falta. Há um *FN* quando o modelo falha em detetar um objeto presente na informação *ground-truth*.
- *True Negative* (*TN*) – Não se aplica à avaliação de modelos de deteção de objetos. Uma *TN* representa uma deteção que não ocorreu de um objeto que não existe. No contexto da deteção de objetos poderão existir inúmeros casos destes, pelo que não são usadas pelas métricas.

Após classificarmos as deteções e obtermos o número de *TP*, *FP* e *FN*, podemos calcular a precisão e revocação. A precisão representa a habilidade do modelo identificar apenas os objetos relevantes e é tida como a percentagem de deteções corretas. Para obtê-la é feito o quociente entre o número de *TP* e a soma dos números de *TP* com *FP*. A revocação representa a habilidade do modelo detetar todos os objetos relevantes (de acordo com a *ground-truth*) e é tida como a percentagem de *TP* detetados entre todas as deteções *ground-truth*. Para obtê-la é feito o quociente entre o número de *TP* e a soma dos números de *TP* com *FN* (número de deteções *ground-truth*). A partir destes valores é possível identificar se um modelo é bom ou não. Quanto mais alta a precisão e a revocação, melhor é o modelo. De notar que o valor de *threshold* utilizado pode afetar os valores de precisão e revocação, pelo que tipicamente é indicado juntamente com os resultados.

De seguida é desenhado um gráfico precisão x revocação para cada classe. Estes gráficos são utilizados muito frequentemente e permitem calcular a *AP*. O eixo das abscissas representa a revocação e o eixo das ordenadas representa a precisão. Tipicamente as curvas traçadas começam com altos níveis de precisão quando a revocação é baixa (poucas deteções, mas muito precisas) e vão diminuindo conforme a revocação aumenta (mais deteções, mas menos precisas). Além disto costumam subir e descer ocasionalmente, criando um traçado em ziguezague. A ferramenta disponibilizada em (Padilla, 2020) desenha estes gráficos e disponibiliza-os juntamente com as métricas calculadas. Seguem-se alguns exemplos de gráficos obtidos durante os testes realizados.

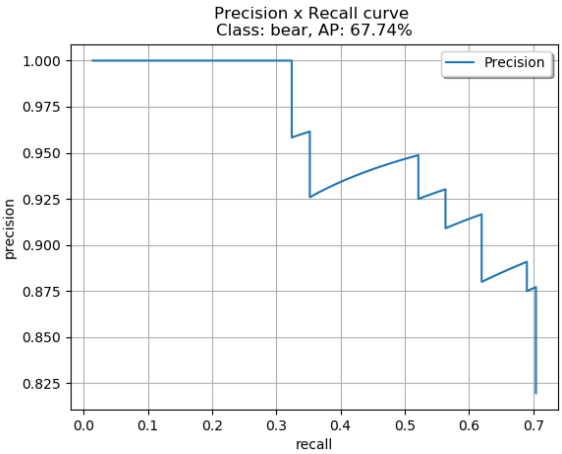


*Figura 4.14 - Gráfico precisão x revocação (YOLOv3, IoU 0.75, classe urso)*

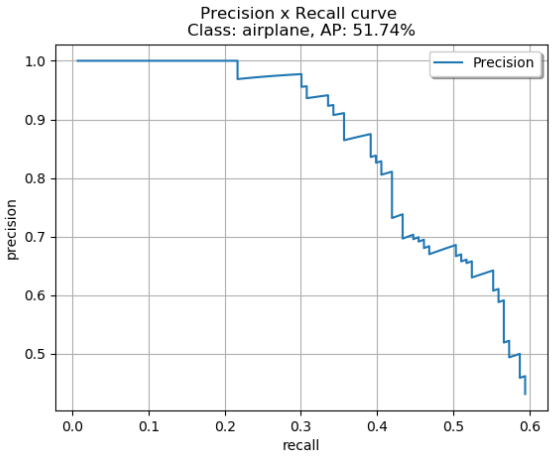


*Figura 4.15 - Gráfico precisão x revocação (YOLOv3, IoU 0.75, classe avião)*

Segundo (Padilla, 2020), estes gráficos são construídos ao calcular a precisão e revocação dos valores acumulados de *TP* e *FP*. O autor ilustra estre processo com o seguinte exemplo. Consideremos que o *dataset* utilizado nesta avaliação é composto pelas imagens seguintes:

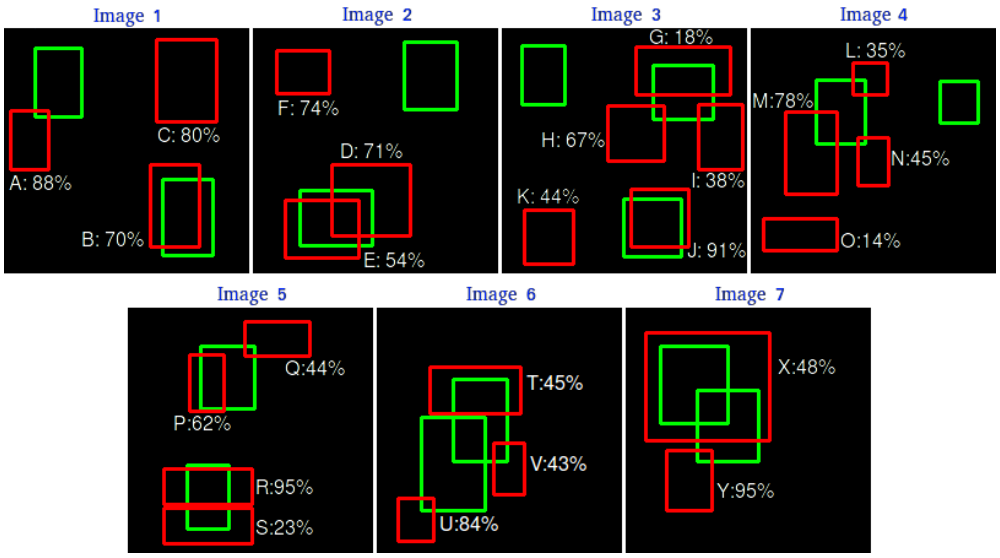


*Figura 4.16 - Imagens utilizadas no teste de exemplo (Padilla, 2020)*

Nestas imagens estão representadas quinze deteções *ground-truth* com retângulos verdes e vinte e quatro deteções previstas por um modelo com retângulos vermelhos. Cada deteção tem uma confiança associada e é identificada por uma letra. Neste exemplo o *threshold* utilizado é 30%. Em algumas imagens como a 2, 3 e 4, as deteções *ground-truth* são detetadas com mais do que uma deteção *TP*. Nestes casos, normalmente a primeira é considerada *TP* e as outras *FP*.

De seguida, as deteções são ordenadas por confiança e é calculada a precisão e a revocação com os valores acumulados de *TP* e *FP*, como é evidenciado no excerto seguinte de uma tabela apresentada pelo autor.

| Images | Detections | Confidences | TP | FP | Acc TP | Acc FP | Precision | Recall |
|---|---|---|---|---|---|---|---|---|
| Image 5 | R | 95% | 1 | 0 | 1 | 0 | 1 | 0.0666 |
| Image 7 | Y | 95% | 0 | 1 | 1 | 1 | 0.5 | 0.0666 |
| Image 3 | J | 91% | 1 | 0 | 2 | 1 | 0.6666 | 0.1333 |
| Image 1 | A | 88% | 0 | 1 | 2 | 2 | 0.5 | 0.1333 |
| Image 6 | U | 84% | 0 | 1 | 2 | 3 | 0.4 | 0.1333 |
| Image 1 | C | 80% | 0 | 1 | 2 | 4 | 0.3333 | 0.1333 |
| Image 4 | M | 78% | 0 | 1 | 2 | 5 | 0.2857 | 0.1333 |
| Image 2 | F | 74% | 0 | 1 | 2 | 6 | 0.25 | 0.1333 |
| Image 2 | D | 71% | 0 | 1 | 2 | 7 | 0.2222 | 0.1333 |
| Image 1 | B | 70% | 1 | 0 | 3 | 7 | 0.3 | 0.2 |
| Image 3 | H | 67% | 0 | 1 | 3 | 8 | 0.2727 | 0.2 |

*Figura 4.17 - Excerto de uma tabela onde são apresentados os cálculos efetuados (Padilla, 2020)*

Começando pelo topo da tabela, a deteção R é *TP* e o *TP* acumulado é 1. Como tal, a $Precisão = \frac{\text{Acc } TP}{\text{Acc } TP + \text{Acc } FP} = \frac{1}{1+0} = 1$ e a $Revocação = \frac{\text{Acc } TP}{\text{\# deteções } ground{-}truth} = \frac{1}{15}$ . Os valores das linhas seguintes são calculados de forma idêntica. Após obter os valores de precisão e revocação podemos construir o gráfico ao ligar os pontos pela ordem que aparecem na tabela, tal como demonstrado na figura seguinte.

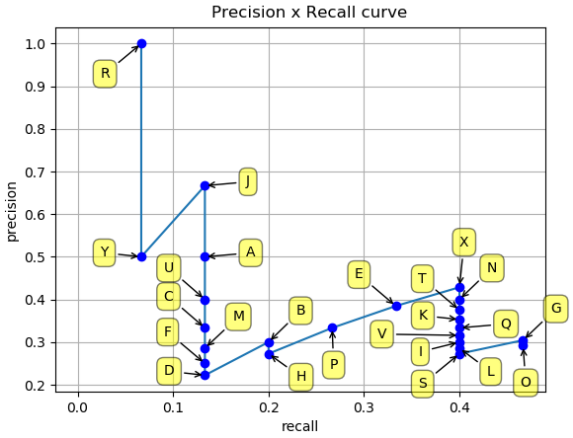


*Figura 4.18 - Construção do gráfico precisão x revocação (Padilla, 2020)*

A métrica *AP* é dada como a precisão média ao longo de todos os valores de revocação entre 0 e 1 e pode ser determinada ao calcular a área debaixo da curva traçada no gráfico precisão x revocação. Como a curva traçada é frequentemente em ziguezague, o cálculo da área é complicado. Por essa razão, tipicamente é calculada uma aproximação da área debaixo da curva. Ao longo dos anos foram sendo usadas diferentes técnicas para fazer esta aproximação. Inicialmente a técnica mais utilizada (nos desafios *PASCAL VOC* ("PASCAL VOC," 2020), …) era a técnica conhecida por "*11-point interpolation*", que consiste em calcular a média das precisões em onze diferentes valores de revocação, mais concretamente 0, 0.1, …, 1. Mais recentemente, a técnica conhecida como "*Interpolating all points*" tem sido usada mais frequentemente. Esta foi a técnica utilizada nestes testes e pode ser visualizada nas figuras seguintes, apresentadas pelo autor.

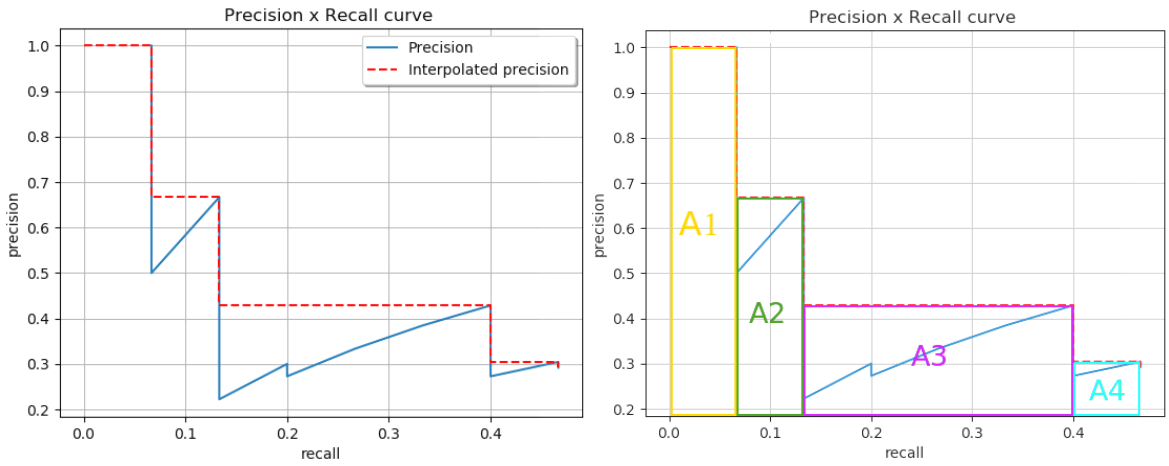


*Figura 4.19 - Aplicação da técnica "Interpolating all points" para aproximar a área debaixo da curva (Padilla, 2020)*

Por fim, para calcular a *mAP* é feita a média da *AP* calculada para todas as classes detetadas. De notar que na literatura e nesta área a *mAP* é ocasionalmente referida por *AP*, o que pode suscitar confusão.

## 4.2.2. Resultados e análise

Como mencionado anteriormente, era expectável que os modelos de deteção de objetos usados na aplicação não fossem capazes de igualar os resultados anunciados pelos seus autores nos trabalhos onde são introduzidos, dado que são modelos treinados por terceiros disponibilizados online e é provável que o treino não tenha sido tão minucioso. No seguimento destes testes verificou-se que esse é o caso.

Constatou-se que o modelo *YOLOv3* disponível na aplicação obteve a *mAP* mais alta com *threshold* (*IoU*) a 50% e 75%. O *SSD* teve o pior desempenho a 50% e 75%, mas teve o melhor desempenho a 95%. O *Mask R-CNN* (*bounding boxes*) teve o segundo melhor desempenho com todos os *thresholds*, ficando perto do melhor registo a 50% e 75%. Como era expectável, os valores de *mAP* diminuíram com o aumento do threshold, dado que a sobreposição das *bounding boxes* tem de ser mais significativa para a deteção ser considerada *True Positive*. Os resultados obtidos nestes testes estão resumidos na tabela seguinte.

| | *mAP@0.5* | *mAP@0.75* | *mAP@0.95* |
|---|---|---|---|
| *YOLOv3* | 36.56% | 26.55% | 0.12% |
| *SSD* | 26.15% | 17.74% | 0.65% |
| *Mask R-CNN (bounding boxes)* | 35.17% | 23.76% | 0.34% |

*Tabela 4.6 - Resultados dos testes. mAP@0.5 significa mAP obtida com threshold a 50%*

Um dos objetivos deste trabalho era explorar que métodos de pesquisa visual em imagens existem atualmente e o que é possível fazer com eles. Contudo, este objetivo esteve sempre um pouco fora de alcance no que diz respeito a colocar em prática modelos de deteção de objetos muito recentes, pois estão constantemente a ser desenvolvidos novos

métodos e treinados modelos melhores. Desde que esta dissertação começou a ser escrita e a aplicação desenvolvida já foram feitas dezenas de publicações com novos métodos que obtêm resultados ainda melhores. Um exemplo concreto é o do *YOLOv4* (Bochkovskiy et al., 2020, p. 4), que segundo os autores tem uma *AP* 10% melhor do que o *YOLOv3*, assim como 12% mais *frames* por segundo. Estas publicações podem ser acompanhadas em ("Papers with Code - Object Detection," n.d.).

## 4.3. Testes com utilizadores

Esta secção documenta os testes com utilizadores realizados após o desenvolvimento da aplicação. Estes testes foram elaborados de forma a ser possível avaliar e comparar os diferentes modos de visualização de resultados desenvolvidos quanto à sua usabilidade, utilidade, rapidez de uso e eficácia. Os testes foram feitos com 9 participantes, 7 homens e 2 mulheres. A média das idades foi de 27.78 anos e o desvio padrão foi 13.28.

### 4.3.1. Metodologia

Foi pedido aos participantes que realizassem quatro grupos de duas tarefas cada. Cada grupo de tarefas é composto por uma tarefa de pesquisa geral e por outra de pesquisa específica. Uma tarefa de pesquisa geral consiste em procurar por objetos que pertencem a um determinado grupo. Uma tarefa de pesquisa específica consiste em procurar por objetos com características específicas. Foram usados quatro *datasets* de 100 imagens cada, retiradas do *COCO dataset* ("COCO Dataset," 2020). Cada *dataset* continha imagens de um tema específico. Os temas escolhidos foram meios de transporte (aviões, bicicletas, etc.), desportos (raquete de ténis, bola de basebol, etc.), animais (cavalos, cães, etc.) e alimentos (maçãs, laranjas, etc.). Estes foram os temas escolhidos porque os objetos que abrangem são os que o método de pesquisa utilizado - *YOLOv3* (Redmon and Farhadi, 2018, p. 3) – foi treinado para detetar. As tarefas de pesquisa geral consistiam em procurar por objetos pertencentes aos grupos autocarros, bolas, cães e maçãs, e as tarefas de pesquisa específica consistiam em procurar por autocarros vermelhos, bolas de futebol, cães bege e maçãs verdes.

As quatro interfaces de visualização de imagens e os quatro *datasets* foram utilizados para realizar um grupo de tarefas cada. Os *datasets* continham vinte imagens corretas para as tarefas de pesquisa geral e cinco imagens corretas para as tarefas de pesquisa específica. A sequência de visualizações utilizadas foi diferente para cada participante para minimizar quaisquer possíveis efeitos de aprendizagem e diferenças entre *datasets*.

Antes da realização de cada grupo de tarefas, todas as imagens do *dataset* a utilizar foram processadas com o *YOLOv3*, e foram sobrepostas as *bounding boxes* de todos os objetos detetados pertencentes ao tema do grupo de tarefas em questão. Após o processamento estar concluído, os participantes começavam a tarefa ao clicar num botão que iniciava um temporizador. Para realizar cada tarefa, os participantes deviam selecionar todas as imagens que considerassem ter um ou mais objetos procurados com um duplo clique. Após estarem satisfeitos com as suas seleções, os participantes voltavam a clicar no mesmo botão para parar o temporizador e concluir a tarefa. As imagens selecionadas em cada tarefa e por cada

utilizador foram registadas, sendo posteriormente comparadas com a *ground-truth*. Desta forma, foi possível calcular métricas como precisão e revocação. Após completarem cada grupo de tarefas, foi pedido que os participantes respondessem a um questionário *SUS* (*System Usability Scale*) (Brooke, n.d.) sobre a visualização utilizada, que consiste em dez afirmações simples que devem ser atribuídas um valor do intervalo (1- *strongly disagree ; 5- strongly agree*). Após completarem todas as tarefas, os participantes responderam a um questionário final onde era pedido que indicassem a utilidade de cada visualização, ordenassem as visualizações de acordo com a sua preferência pessoal e indicassem que aspetos foram mais importantes para tomarem essa decisão.

### 4.3.2. Resultados e análise

Para comparar os modos de visualização foram avaliadas as suas usabilidades através do *SUS* e foi registado o tempo necessitado por cada participante para concluir cada tarefa. Além disto, a qualidade dos resultados dos testes foi determinada através de métricas como precisão, revocação, *f-measure* e preferência dos participantes. Os valores de precisão, revocação e *f-measure* foram calculados como se segue:

$$Precisão = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

$$Revocação = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

$$F\ Measure = 2 \times \frac{Precisão \times Revocação}{Precisão + Revocação}$$

Os resultados dos questionários *SUS* (tabela que se segue) indicam que as grelhas têm melhor usabilidade do que a Pilha e a Espiral.

| | M (SD) | Mdn |
|---|---|---|
| Grelha Normal | 94.69 (9.40) | 98.75 |
| Grelha Variável | 87.50 (11.50) | 91.25 |
| Pilha | 48.55 (13.69) | 62.50 |
| Espiral | 53.75 (26.22) | 48.70 |

*Tabela 4.7 - Pontuações SUS para cada modo de visualização*

O teste *Shapiro-Wilk* indicou que os dados dos questionários *SUS* não apresentavam uma distribuição normal, logo foi realizado o teste *Friedman* e constatou-se existem diferenças significativas $\chi2(3) = 17.169$, $p = 0.01$. Posteriormente, foram feitos testes *Wilcoxon Signed Rank*, que corresponderam as pontuações dos questionários a cada modo de visualização. Os testes mostraram uma diferença significativa entre a Grelha Normal e a Pilha ($Z = -2.547$, $p = 0.011$), assim como entre a Grelha Normal e a Espiral ($Z = -2.524$, $p = 0.012$). Foi obtido um resultado semelhante entre a Grelha Variável e a Pilha ($Z= -2.533$, $p = 0.011$), e entre a Grelha Variável e a Espiral ($Z = -2.552$, $p = 0.011$). No entanto, não houve diferença significativa entre as duas grelhas ($Z = -1.802$, $p = 0.072$), e entre a Pilha e a Espiral ($Z = -0.847$, $p = 0.397$).

Os tempos registados encontram-se na tabela seguinte.

| | Tarefas de pesquisa geral | Tarefas de pesquisa específica | |
|---|---|---|---|
| | M (SD) | M (SD) | M (SD) |
| Grelha Normal | 87.93 (33.86) | 30.78 (8.58) | 31.14 |
| Grelha Variável | 92.07 (31.31) | 49.41 (37.59) | 36.16 |
| Pilha | 251.17 (61.90) | 134.46 (37.15) | 141.87 |
| Espiral | 185.29 (60.09) | 96.48 (46.02) | 106.96 |

*Tabela 4.8 - Tempo em segundos para cada visualização/tipo de tarefa*

Relativamente ao tempo necessitado pelos participantes para concluir as tarefas com cada visualização, foi feito um teste *Shapiro-Wilk* que indicou que os tempos das tarefas de pesquisa geral apresentavam uma distribuição normal. Portanto, recorreu-se ao teste *Repeated Measures ANOVA* para procurar por diferenças significativas. O teste de Esfericidade de *Mauchly* indicou que a assunção de esfericidade não tinha sido violada $\chi2(5) = 6.295$, $p = 0.283$. O teste *ANOVA* mostrou que os tempos para pesquisas gerais apresentam diferenças estatisticamente significativas entre visualizações ($F(3, 24) = 22.523$, $p = 0.0000003687$). Testes *post hoc* usando a correção de *Bonferroni* revelaram que a Grelha Normal foi significativamente mais rápida do que a Pilha ($p = 0.001$) e a Espiral ($p = 0.044$). Foi obtido um resultado semelhante para a Grelha Variável em comparação com a Pilha ($p = 0.01$) e com a Espiral ($p = 0.022$). No entanto, não houve diferença significativa entre as duas grelhas ($p = 1$) e entre a Pilha e a Espiral ($p = 0.256$).

Relativamente aos tempos registados para as tarefas de pesquisa específica, o teste *Shapiro-Wilk* indicou que os dados não apresentavam uma distribuição normal. Portanto, foi feito um teste de *Friedman* para procurar por diferenças significativas. Este teste mostrou que os tempos registados eram significativamente diferentes $\chi2(3) = 21.00$, $p = 0.000105$. Posteriormente, foram feitos testes *Wilcoxon Signed Rank* que associaram os tempos registados nas tarefas específicas a cada visualização. Os testes indicaram que a Grelha Normal foi significativamente mais rápida que a Pilha ($Z = -2.666$, $p = 0.008$) e que a Espiral ($Z = -2.666$, $p = 0.008$). Um resultado semelhante foi obtido para a Grelha Variável em comparação com a Pilha ($Z = -2.666$, $p = 0.008$) e com a Espiral ($Z = -2.547$, $p = 0.011$). No contexto desta tarefa, A Espiral foi significativamente mais rápida que a Pilha ($Z = -2.073$, $p = 0.038$). Não foram encontradas diferenças significativas entre as duas grelhas ($Z = -1.362$, $p = 0.173$). Os valores de precisão, revocação e *f-measure* encontram-se nas tabelas seguintes.

| Tarefas de pesquisa geral | | | | | | |
|---|---|---|---|---|---|---|
| | Precisão | | Revocação | | *F-Measure* | |
| | M (SD) | Mdn | M (SD) | Mdn | M (SD) | Mdn |
| Grelha Normal | 0.99 (0.4) | 1.00 | 0.88 (0.20) | 1.00 | 0.92 (0.15) | 1.00 |
| Grelha Variável | 1.00 (0.00) | 1.00 | 0.93 (0.07) | 0.95 | 0.96 (0.04) | 0.97 |
| Pilha | 0.98 (0.03) | 1.00 | 0.97 (0.04) | 1.00 | 0.97 (0.02) | 0.98 |
| Espiral | 0.98 (0.05) | 1.00 | 0.89 (0.12) | 0.95 | 0.93 (0.07) | 0.94 |

*Tabela 4.9 - Tarefas de pesquisa geral: Precisão, revocação e f-measure*

| Tarefas de pesquisa específica | | | | | | |
|---|---|---|---|---|---|---|
| | Precisão | | Revocação | | *F-Measure* | |
| | M (SD) | Mdn | M (SD) | Mdn | M (SD) | Mdn |
| Grelha Normal | 1.00 (0.00) | 1.00 | 0.87 (0.14) | 0.80 | 0.93 (0.09) | 0.89 |
| Grelha Variável | 1.00 (0.00) | 1.00 | 0.93 (0.10) | 1.00 | 0.96 (0.06) | 1.00 |
| Pilha | 0.98 (0.06) | 1.00 | 1.00 (0.00) | 1.00 | 0.99 (0.03) | 1.00 |
| Espiral | 1.00 (0.00) | 1.00 | 0.89 (0.18) | 1.00 | 0.93 (0.11) | 1.00 |

*Tabela 4.10 - Tarefas de pesquisa específica: Precisão, revocação e f-measure*

Os testes *Shapiro-Wilk* indicaram que a precisão, revocação e *f-measure* não apresentavam uma distribuição normal em ambas as tarefas. Portanto, foi feito um teste *Friedman* para procurar por diferenças significativas em ambas as tarefas. Em relação às tarefas de pesquisa geral, os testes não mostraram diferenças significativas para precisão ($\chi2(3) = 5.545$, $p = 0.136$), revocação ($\chi2(3) = 3.958$, $p = 0.266$) e *f-measure* ($\chi2(3) = 2.920$, $p = 0.404$). As tarefas de pesquisa específica tiveram resultados semelhantes, dado que não se observaram diferenças significativas para precisão ($\chi2(3) = 3.00$, $p = 0.292$), revocação ($\chi2(3) = 5.830$, $p = 0.120$) e *f-measure* ($\chi2(3) = 4.393$, $p = 0.222$).

Em relação às preferências dos participantes, a Grelha Variável foi a visualização preferida (Mdn = 1), seguida pela Grelha Normal (Mdn = 2), Pilha (Mdn = 3) e pela Espiral (Mdn = 4). Os aspetos chave para determinar as preferências pessoais dos participantes indicados mais frequentemente foram: tempo necessário para identificar os objetos procurados, a dificuldade de localizá-los e a intuição da visualização.

Em suma, observou-se que a sequência de visualizações ordenada da mais rápida para a mais lenta é Grelha Normal > Grelha Variável > Espiral > Pilha, sendo que a diferença entre as grelhas não é significativa. Em relação à precisão, revocação e *f-measure*, não foram encontradas diferenças significativas em ambos os tipos de tarefas. As visualizações ordenadas por pontuação de usabilidade (*SUS*) são Grelha Normal > Grelha Variável > Espiral > Pilha. As visualizações ordenadas da mais útil para a menos são Grelha Variável > Grelha Normal > Pilha > Espiral, e os aspetos chave mais comuns foram o tempo necessário para encontrar os objetos, a dificuldade de localizá-los e a intuição das visualizações.

## 4.4. Discussão

Constatou-se que o desempenho dos algoritmos de *feature matching* e de deteção de objetos é suficiente para satisfazer as necessidades da aplicação desenvolvida, ainda que não seja ideal para aplicações de tempo real. Com *CPU* foi possível atingir cerca de 5 *fps* com um método de *feature matching*, e cerca de 17 *fps* com um modelo de deteção de objetos. Apesar de as *framerates* mais altas terem sido conseguidas com *CPU*, constatou-se que a utilização de *GPU* pode permitir obter velocidades de processamento muito superiores, particularmente com os modelos de deteção de objetos.

Relativamente aos testes com diferentes configurações de *hardware*, observou-se que as *GPUs* mais capazes tiveram melhor desempenho nas fases de processamento que as utilizam, e que o *CPU* utilizado teve um impacto maior do que o esperado. Observou-se também que

a utilização de *GPUs* mais capazes teve maior impacto nos modelos de deteção de objetos do que nos métodos de *feature matching*. Contudo, à exceção de alguns métodos onde observaram-se melhorias mais significativas, as diferenças de desempenho não seriam muito percetíveis num caso de utilização real.

Relativamente à avaliação da qualidade dos resultados obtidos com os modelos de deteção de objetos (*bounding boxes*) incluídos na aplicação, os resultados obtidos ficaram aquém dos anunciados pelos autores nos trabalhos ondes os modelos são introduzidos, como era esperado, dado que são modelos treinados por terceiros disponibilizados online e o treino pode não ter sido tao minucioso. O *YOLOv3* foi o que teve melhores resultados na maior parte dos testes e o *Mask R-CNN* (*bounding boxes*) obteve resultados satisfatórios consistentemente.

Os testes realizados com utilizadores indicaram que as grelhas são mais rápidas do que a Espiral e a Pilha, sem diferenças significativas entre as grelhas. Além disso, verificou-se que a Espiral é mais rápida do que a Pilha. Quanto à usabilidade, a Grelha Normal obteve a pontuação *SUS* mais alta, seguida pela Grelha Variável, Espiral e Pilha. Os participantes indicaram que a Grelha Variável foi mais útil do que a Grelha Normal, seguidas pela Pilha e pela Espiral, e os aspetos indicados mais frequentemente como critérios usados foram o tempo necessário para encontrar os objetos, a dificuldade de localizá-los e a intuição dos modos de visualização. Em relação à precisão, revocação e *f-measure*, não foram encontradas diferenças significativas em ambos os tipos de tarefas. No entanto, a Grelha Variável apresenta valores de revocação e *f-measure* mais elevados do que a Grelha Normal.

Estes resultados indicam que as visualizações habituais no formato de Grelha tendem a ser melhores do que as mais fora do comum Pilha e Espiral, e tornam claro que visualizações do tipo Pilha podem ser interessantes para apresentação de resultados de algoritmos, mas pouco úteis para tarefas de pesquisa.

A Grelha Variável foi considerada mais útil para os participantes do estudo e apresenta melhores valores de precisão nas tarefas de pesquisa específica do que a Grelha Normal. Este é um dado bastante interessante, pois apesar da Grelha Variável ser baseada na Grelha Normal, o conceito que a define (*thumbnails* de diferentes tamanhos) é pouco explorado atualmente, e quando é explorado é com outros propósitos, como manter os *aspect ratios* originais das imagens, por exemplo. Uma possível combinação de dois tipos de Grelhas para pesquisas distintas seria uma solução interessante a explorar.

# 5. Conclusões

Nesta dissertação foi desenvolvida uma aplicação para *desktop* com uma interface gráfica que permite a utilizadores sem conhecimentos técnicos específicos usar diferentes métodos para pesquisa visual em imagens, cinco baseados em *features* visuais e cinco baseados em classificação de objetos. A interface desenvolvida inclui cinco modos distintos para visualização dos resultados obtidos pelos diversos métodos.

Foi feita uma revisão de literatura abrangente que serviu para selecionar os métodos de pesquisa visual a implementar e para obter inspiração para o desenvolvimento dos modos de visualização de resultados. Após o desenvolvimento estar concluído, foi feita uma avaliação do desempenho de cada método (velocidade de processamento com *CPU*/*GPU* e com diferentes configurações de *hardware*, e qualidade dos resultados) e dos modos de visualização (usabilidade, preferência e eficiência).

Constatou-se que o desempenho dos algoritmos de *feature matching* e de deteção de objetos é suficiente para satisfazer as necessidades da aplicação, ainda que não seja ideal para aplicações de tempo real. Apesar de as *framerates* mais altas terem sido conseguidas com *CPU*, constatou-se que a utilização de *GPU* pode permitir obter velocidades de processamento muito superiores, particularmente com os modelos de deteção de objetos. Ao realizar testes com diferentes configurações de *hardware* observou-se que as diferenças de desempenho não seriam muito percetíveis num caso de utilização real, à exceção de alguns métodos onde observaram-se melhorias mais significativas.

A qualidade dos resultados obtidos com os modelos de deteção de objetos (*bounding boxes*) incluídos na aplicação ficou aquém dos valores anunciados pelos autores nos trabalhos onde os modelos são introduzidos, como era esperado. Um dos objetivos deste trabalho era explorar que métodos de pesquisa visual existem atualmente e o que é possível fazer com eles e com *hardware* recente. No que diz respeito a colocar em prática os modelos de deteção de objetos *state-of-art* mais recentes, isto foi algo que esteve sempre um pouco fora de alcance dado que estão sempre a ser desenvolvidos modelos melhores. De qualquer forma, foram estudados e incluídos na aplicação alguns dos modelos de deteção de objetos mais conhecidos e relevantes. No que diz respeito a *feature matching*, foram estudados e implementados na aplicação vários métodos incluindo alguns dos melhores atualmente, apesar destes não serem necessariamente recentes. Os resultados obtidos com os diferentes métodos são bons o suficiente para serem úteis e para satisfazerem as necessidades da aplicação desenvolvida. O *ORB* destaca-se entre os métodos de *feature matching* por ser aquele que apresentou resultados mais equilibrados, tendo em conta diferentes fatores, e o *BRISK* por ser o mais apto para aplicações em tempo real. Entre os modelos de deteção de objetos, o *Mask R-CNN* destaca-se pela sua versatilidade e o *SSD* pela sua velocidade de processamento.

Quanto às interfaces para visualização de resultados de pesquisas de imagens, os testes indicam que as Grelhas tendem a ser melhores do que a Pilha e a Espiral nos diferentes aspetos avaliados (usabilidade, tempo e precisão). Apesar de haver poucas diferenças entre as Grelhas, observou-se que a Grelha Variável foi considerada mais útil e com maior precisão em tarefas de pesquisa específica do que a Grelha Normal. É um resultado bastante

interessante, pois apesar de ser baseada na Grelha Normal, a ideia de usar *thumbnails* de diferentes tamanhos da Grelha Variável é pouco explorada atualmente, e quando é explorada é com outros propósitos.

## 5.1. Limitações

As limitações que foram encontradas durante o desenvolvimento deste trabalho estão relacionadas com o desempenho e com a qualidade dos resultados. As limitações de desempenho resultam da própria natureza do processamento de imagem, que pode ser muito exigente, tanto a nível de tempo como de memória.

Um dos objetivos deste trabalho era estudar a evolução de métodos de pesquisa visual e explorar o que é possível atualmente. Constatou-se que o desempenho temporal que é possível obter com os métodos testados e com *hardware* moderno ainda está longe do nível de desempenho ideal para aplicações de tempo real, por exemplo. Para acelerar o processamento foi adicionada a possibilidade de utilizar *GPU*, quando suportado. Constatou-se que o processamento com *GPU* é geralmente bastante mais rápido do que com *CPU*. Contudo, continua a não ser suficiente para, por exemplo, aplicações de tempo real.

Foram também encontradas limitações de memória, especialmente quando o número de imagens para análise era muito grande. Esta limitação pode ser solucionada ao permitir guardar os dados do processamento em disco. No entanto, as transferências de dados de e para disco podem levar bastante tempo, dependendo da quantidade de dados e da capacidade do *hardware*, levando a um aumento do tempo total de processamento.

A qualidade dos resultados obtidos pelos métodos de pesquisa visual geralmente não é muito boa. A qualidade dos resultados obtidos pelos métodos de *feature matching* está dependente dos próprios métodos, pelo que não pode ser melhorada. A qualidade dos resultados obtidos pelos métodos de deteção de objetos está dependente dos modelos utilizados. Teoricamente seria possível treinar modelos personalizados para detetar um conjunto arbitrário de classes. No entanto, este processo é muito complexo e demorado, e varia bastante de arquitetura para arquitetura, pelo que esta opção não foi explorada neste trabalho. Em vez disso foram utilizados modelos pré-treinados, disponíveis publicamente. Apesar de estarem constantemente a ser desenvolvidos novos métodos e treinados modelos melhores, tipicamente estes não são disponibilizados imediatamente para utilização e podem nunca chegar a ser. Por estas razões, a maioria dos métodos publicados recentemente que permitem obter resultados de melhor qualidade nunca puderam ser sequer considerados para implementação na aplicação desenvolvida.

## 5.2. Trabalho futuro

Este trabalho aborda diversos conceitos, abrangendo métodos de processamento de imagem e até interfaces para visualização de imagens, pelo que há muitos aspetos que podem ser mais trabalhados e desenvolvidos. Como um dos objetivos principais era estudar a evolução de métodos de pesquisa visual e como esta área está em constante evolução, poderia ser interessante continuar a acompanhá-la. Em relação aos métodos baseados em

*machine learning*, poderia ser interessante combinar os resultados obtidos por vários na sequência da análise de uma mesma imagem. Poderia também ser explorada a possibilidade de utilizar *machine learning ensembles* e sistemas de voto para obter resultados mais relevantes.

O estudo e desenvolvimento de interfaces para visualização de resultados parece ser mais interessante e mais relevante, dado que esta é uma área menos desenvolvida. O conceito de *thumbnails* com tamanho variável parece ter potencial. Poderá ser boa ideia explorá-lo em mais detalhe, em trabalhos futuros, dado que pode ser possível tirar partido deste conceito de formas que não foram exploradas neste trabalho. Uma ideia que não foi explorada e que podia ser interessante era mudar as dimensões das *thumbnails* dinamicamente e automaticamente, em vez de depender da interação do utilizador ou de ser estático. Seria também interessante explorar este conceito combinando-o com grelhas estáticas de modo a adaptar a visualização de resultados consoante o tipo de pesquisa, i.e., grelha estáticas para pesquisas mais gerais e *thumbnails* variáveis para pesquisas mais específicas.

Outro aspeto que não foi explorado em detalhe neste trabalho foi o cálculo da relevância dos resultados, sendo que para cada grupo de métodos de pesquisa visual foi utilizado uma fórmula simples, baseada em características da própria pesquisa, como número de *matches* e confiança média. Caso este projeto continuasse a ser desenvolvido, poderia ser interessante explorar em detalhe técnicas de *Content-Based Image Retrieval*, incluindo algumas que foram abordadas na secção de trabalhos relacionados, como a inclusão do utilizador no processo de cálculo de relevância (*relevance feedback*).

# 6. Bibliografia


Abdulla, W., 2018. Splash of Color: Instance Segmentation with Mask R-CNN and TensorFlow [WWW Document]. Medium. URL https://engineering.matterport.com/splash-of-color-instance-segmentation-with-mask-r-cnn-and-tensorflow-7c761e238b46

About OpenCV [WWW Document], 2020. . OpenCV. URL https://opencv.org/about/

André, P., Cutrell, E., Tan, D.S., Smith, G., 2009. Designing Novel Image Search Interfaces by Understanding Unique Characteristics and Usage, in: Gross, T., Gulliksen, J., Kotzé, P., Oestreicher, L., Palanque, P., Prates, R.O., Winckler, M. (Eds.), Human-Computer Interaction – INTERACT 2009, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 340–353. https://doi.org/10.1007/978-3-642-03658-3_40

Basalaj, W., 2001. Proximity visualisation of abstract data (No. UCAM-CL-TR-509). University of Cambridge, Computer Laboratory.

Bay, H., Tuytelaars, T., Van Gool, L., 2006. SURF: Speeded Up Robust Features, in: Leonardis, A., Bischof, H., Pinz, A. (Eds.), Computer Vision – ECCV 2006, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 404–417. https://doi.org/10.1007/11744023_32

Bing Images [WWW Document], n.d. URL https://www.bing.com/?scope=images&nr=1&FORM=NOFORM

Bochkovskiy, A., Wang, C.-Y., Liao, H.-Y.M., 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. ArXiv200410934 Cs Eess.

Brooke, J., n.d. SUS - A quick and dirty usability scale 7.

Brownlee, J., 2019. A Gentle Introduction to Object Recognition With Deep Learning. Mach. Learn. Mastery. URL https://machinelearningmastery.com/object-recognition-with-deep-learning/

Calonder, M., Lepetit, V., Strecha, C., Fua, P., 2010. BRIEF: Binary Robust Independent Elementary Features, in: ECCV. https://doi.org/10.1007/978-3-642-15561-1_56

Canny, J., 1986. A Computational Approach to Edge Detection. IEEE Trans. Pattern Anal. Mach. Intell. PAMI-8, 679–698. https://doi.org/10.1109/TPAMI.1986.4767851

Cityscapes Dataset – Semantic Understanding of Urban Street Scenes, n.d. URL https://www.cityscapes-dataset.com/

COCO - Common Objects in Context [WWW Document], 2020. URL http://cocodataset.org/#home

Computer Vision | Papers With Code [WWW Document], n.d. URL https://paperswithcode.com/area/computer-vision

CUDA Toolkit - NVIDIA Developer [WWW Document], 2020. . NVIDIA Dev. URL https://developer.nvidia.com/cuda-toolkit

cuDNN - NVIDIA Developer [WWW Document], 2020. . NVIDIA Dev. URL https://developer.nvidia.com/cudnn

Dai, J., Li, Y., He, K., Sun, J., 2016. R-FCN: Object Detection via Region-based Fully Convolutional Networks. ArXiv160506409 Cs.

Eady, T., 2019. Tesla's Computer Vision Master Plan [WWW Document]. Medium. URL https://medium.com/protopiablog/teslas-computer-vision-master-plan-512b36d8acbf

Find related images with reverse image search - Computer - Google Search Help [WWW Document], 2020. URL https://support.google.com/websearch/answer/1325808?p=ws_images_searchbyimagetooltip&visit_id=637287974347531774-1093985872&rd=1

Flickr [WWW Document], n.d. URL https://www.flickr.com/

Garcia-Garcia, A., Orts-Escolano, S., Oprea, S., Villena-Martinez, V., Garcia-Rodriguez, J., 2017. A Review on Deep Learning Techniques Applied to Semantic Segmentation. ArXiv170406857 Cs.

GeoVisual Search, 2017. . Descartes Labs. URL https://www.descarteslabs.com/company/geovisual/

Girshick, R., 2015. Fast R-CNN [WWW Document]. URL http://arxiv.org/abs/1504.08083

Girshick, R., Donahue, J., Darrell, T., Malik, J., 2014. Rich feature hierarchies for accurate object detection and semantic segmentation [WWW Document]. URL http://arxiv.org/abs/1311.2524

Girshick, R., He, K., Gkioxari, G., Dollár, P., 2018. Mask R-CNN. ArXiv170306870 Cs.

Girshick, R., Ren, S., He, K., Sun, J., 2016. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks [WWW Document]. URL http://arxiv.org/abs/1506.01497

Google Images [WWW Document], n.d. URL https://www.google.pt/imghp?hl=en&tab=wi&ogbl

Google Search - Discover How Google Search Works [WWW Document], n.d. URL https://www.google.com/search/howsearchworks/

Gracias, N., Garcia, R., Campos, R., Hurtos, N., Prados, R., Shihavuddin, A., Nicosevici, T., Elibol, A., Escartin, J., 2017. Application Challenges of Underwater Vision: Land, Sea & Air. pp. 133–160. https://doi.org/10.1002/9781118868065.ch7

Harris, C.G., Stephens, M., 1988. A Combined Corner and Edge Detector, in: Alvey Vision Conference. https://doi.org/10.5244/C.2.23

Heesch, D., Rüger, S., 2004. Three Interfaces for Content-Based Access to Image Collections, in: Enser, P., Kompatsiaris, Y., O'Connor, N.E., Smeaton, A.F., Smeulders, A.W.M. (Eds.), Image and Video Retrieval, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 491–499. https://doi.org/10.1007/978-3-540-27814-6_58

Hui, J., 2019. mAP (mean Average Precision) for Object Detection [WWW Document]. Medium. URL https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173

Hulstaert, L., 2018. A Beginner's Guide to Object Detection [WWW Document]. DataCamp Community. URL https://www.datacamp.com/community/tutorials/object-detection-guide

IBM - Computer Vision [WWW Document], 2020. URL https://www.ibm.com/topics/computer-vision

Intel® Core™ i7-4770 Processor (8M Cache, up to 3.90 GHz) Product Specifications [WWW Document], n.d. URL https://ark.intel.com/content/www/us/en/ark/products/75122/intel-core-i7-4770-processor-8m-cache-up-to-3-90-ghz.html

Intel® Core™ i7-6700K Processor (8M Cache, up to 4.20 GHz) Product Specifications [WWW Document], 2020. URL https://ark.intel.com/content/www/us/en/ark/products/88195/intel-core-i7-6700k-processor-8m-cache-up-to-4-20-ghz.html

Intel® Core™ i7-8750H Processor [WWW Document], n.d. . Intel. URL https://www.intel.com/content/www/us/en/products/processors/core/i7-processors/i7-8750h.html

Intel® Core™ i7-9700K Processor (12M Cache, up to 4.90 GHz) Product Specifications [WWW Document], n.d. . Intel. URL https://www.intel.com/content/www/us/en/products/processors/core/i7-processors/i7-9700k.html

Jakubovic, A., Velagic, J., 2018. Image Feature Matching and Object Detection Using Brute-Force Matchers. pp. 83–86. https://doi.org/10.23919/ELMAR.2018.8534641

Karami, E., Prasad, S., Shehata, M., n.d. Image Matching Using SIFT, SURF, BRIEF and ORB: Performance Comparison for Distorted Images 5.

Keisler, R., 2017. GeoVisual Search: Using Computer Vision to Explore the Earth [WWW Document]. Medium. URL https://medium.com/descarteslabs-team/geovisual-search-using-computer-vision-to-explore-the-earth-275d970c60cf

Leone, G., 2017. What is Computer Vision and Why Is It Important? [WWW Document]. URL https://www.linkedin.com/pulse/what-computer-vision-why-important-gabriella-leone

Leutenegger, S., Chli, M., Siegwart, R.Y., 2011. BRISK: Binary Robust invariant scalable keypoints, in: 2011 International Conference on Computer Vision. Presented at the 2011 International Conference on Computer Vision, pp. 2548–2555. https://doi.org/10.1109/ICCV.2011.6126542

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., Berg, A.C., 2016. SSD: Single Shot MultiBox Detector. ArXiv151202325 Cs 9905, 21–37. https://doi.org/10.1007/978-3-319-46448-0_2

Lowe, D.G., 2004. Distinctive Image Features from Scale-Invariant Keypoints. Int. J. Comput. Vis. 60, 91–110. https://doi.org/10.1023/B:VISI.0000029664.99615.94

Malarvizhi, C., 2017. Segmentation by Thresholding on Medical Imaging – A Survey [WWW Document]. ResearchGate. URL https://www.researchgate.net/publication/320835315_Segmentation_by_Thresholding_on_Medical_Imaging_-_A_Survey

Mallick, S., 2016. Homography Examples using OpenCV ( Python / C ++ ). URL https://www.learnopencv.com/homography-examples-using-opencv-python-c/

Marr, B., 2019. 7 Amazing Examples Of Computer And Machine Vision In Practice [WWW Document]. Forbes. URL https://www.forbes.com/sites/bernardmarr/2019/04/08/7-amazing-examples-of-computer-and-machine-vision-in-practice/

NASA - Landsat 8, 2020. URL https://landsat.gsfc.nasa.gov/landsat-8/

Nguyen, G.P., Worring, M., 2008. Optimization of interactive visual-similarity-based search. ACM Trans. Multimed. Comput. Commun. Appl. 4, 1–23. https://doi.org/10.1145/1324287.1324294

Niessen, W.J., Vincken, K.L., Weickert, J., Romeny, B.M.T.H., Viergever, M.A., 1999. Multiscale Segmentation of Three-Dimensional MR Brain Images. Int. J. Comput. Vis. 31, 185–202. https://doi.org/10.1023/A:1008070000018

NVIDIA GeForce GTX 1060 Graphics Card Specifications [WWW Document], n.d. . NVIDIA. URL https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1060/

NVIDIA GeForce GTX 1070 Max-Q GPU - Benchmarks and Specs [WWW Document], n.d. . Notebookcheck. URL https://www.notebookcheck.net/NVIDIA-GeForce-GTX-1070-Max-Q-GPU-Benchmarks-and-Specs.224732.0.html

NVIDIA GeForce GTX 1080 Graphics Card Specifications [WWW Document], 2020. . NVIDIA. URL https://www.nvidia.com/en-sg/geforce/products/10series/geforce-gtx-1080/

NVIDIA GeForce GTX TITAN X Graphics Card Specifications [WWW Document], 2020. URL https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications

NVIDIA GeForce RTX 2080 Ti Graphics Card Specifications [WWW Document], n.d. . NVIDIA. URL https://www.nvidia.com/en-eu/geforce/graphics-cards/rtx-2080-ti/

Object Recognition in Video Dataset [WWW Document], n.d. URL http://mi.eng.cam.ac.uk/research/projects/VideoRec/CamVid/

Oliveira, H., Rodrigues, C.M., Piteri, M., 2014. Detecção de Arestas em Imagens Digitais [WWW Document]. URL https://www.researchgate.net/publication/266316903_Deteccao_de_Arestas_em_Imagens_Digitais

OpenCV 3.0 [WWW Document], 2015. URL https://opencv.org/opencv-3-0/

OpenCV: Introduction [WWW Document], 2020. URL https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_setup/py_intro/py_intro.html

OpenCV Python Docs: Feature Matching + Homography to find Objects [WWW Document], 2020. URL https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html

Ouaknine, A., 2018. Review of Deep Learning Algorithms for Object Detection [WWW Document]. Medium. URL https://medium.com/zylapp/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852

Padilla, R., 2020. rafaelpadilla/Object-Detection-Metrics.

Padilla, R., Lima Netto, S., A. B. da Silva, E., 2020. Survey on Performance Metrics for Object-Detection Algorithms.

Papers with Code - Object Detection [WWW Document], n.d. URL https://paperswithcode.com/task/object-detection

Paszke, A., Chaurasia, A., Kim, S., Culurciello, E., 2016. ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation. ArXiv160602147 Cs.

Pillow Documentation [WWW Document], 2020. URL https://pillow.readthedocs.io/en/stable/

Redmon, J., Divvala, S., Girshick, R., Farhadi, A., 2016. You Only Look Once: Unified, Real-Time Object Detection. ArXiv150602640 Cs.

Redmon, J., Farhadi, A., 2018. YOLOv3: An Incremental Improvement. ArXiv180402767 Cs.

Redmon, J., Farhadi, A., 2016. YOLO9000: Better, Faster, Stronger. ArXiv161208242 Cs.

Riverbank Computing Limited, 2020. PyQt5: Python bindings for the Qt cross platform application toolkit [WWW Document]. URL https://www.riverbankcomputing.com/software/pyqt/

Rodden, K., 2002. Evaluating Similarity-Based Visualisations as Interfaces for Image Browsing [WWW Document]. ResearchGate. URL https://www.researchgate.net/publication/2477860_Evaluating_Similarity-Based_Visualisations_as_Interfaces_for_Image_Browsing

Rosebrock, A., 2020. OpenCV "dnn" with NVIDIA GPUs: 1549% faster YOLO, SSD, and Mask R-CNN. PyImageSearch. URL https://www.pyimagesearch.com/2020/02/10/opencv-dnn-with-nvidia-gpus-1549-faster-yolo-ssd-and-mask-r-cnn/

Rosebrock, A., 2018a. YOLO object detection with OpenCV. PyImageSearch. URL https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/

Rosebrock, A., 2018b. Mask R-CNN object detection with OpenCV [WWW Document]. URL https://www.pyimagesearch.com/2018/11/19/mask-r-cnn-with-opencv/

Rosebrock, A., 2017. SSD object detection with OpenCV. PyImageSearch. URL https://www.pyimagesearch.com/2017/09/11/object-detection-with-deep-learning-and-opencv/

Rosebrock, A., 2016. Intersection over Union (IoU) for object detection. PyImageSearch. URL https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

Rosten, E., Drummond, T., 2006. Machine Learning for High-Speed Corner Detection, in: ECCV. https://doi.org/10.1007/11744023_34

Rublee, E., Rabaud, V., Konolige, K., Bradski, G., 2011. ORB: An efficient alternative to SIFT or SURF, in: 2011 International Conference on Computer Vision. Presented at the 2011 International Conference on Computer Vision, pp. 2564–2571. https://doi.org/10.1109/ICCV.2011.6126544

Ruger, S., 2005. Putting the User in the Loop: Visual Resource Discovery | Request PDF [WWW Document]. ResearchGate. URL https://www.researchgate.net/publication/221367634_Putting_the_User_in_the_Loop_Visual_Resource_Discovery

SAS - Computer Vision: What it is and why it matters [WWW Document], 2020. URL https://www.sas.com/en_us/insights/analytics/computer-vision.html

Senthilkumaran, N., Vaithegi, S., 2016. Image Segmentation By Using Thresholding Techniques For Medical Images [WWW Document]. ResearchGate. URL https://www.researchgate.net/publication/297728148_Image_Segmentation_By_Using_Thresholding_Techniques_For_Medical_Images

Sharma, P., 2019. Step-by-Step Tutorial on Image Segmentation Techniques in Python. Anal. Vidhya. URL https://www.analyticsvidhya.com/blog/2019/04/introduction-image-segmentation-techniques-python/

Sharma, P., 2018. A Step-by-Step Introduction to the Basic Object Detection Algorithms. Anal. Vidhya. URL https://www.analyticsvidhya.com/blog/2018/10/a-step-by-step-introduction-to-the-basic-object-detection-algorithms-part-1/

Shimizu, H., Heins, R., 1995. Computer-vision-based System for Plant Growth Analysis. Trans. ASAE 38, 959–964. https://doi.org/10.13031/2013.27913

Sivic, J., Schaffalitzky, F., Coto, E., Zisserman, A., 2003. Video Google Demo - Visual Geometry Group - University of Oxford [WWW Document]. URL http://www.robots.ox.ac.uk/~vgg/research/vgoogle/

Sivic, Zisserman, 2003. Video Google: a text retrieval approach to object matching in videos, in: Proceedings Ninth IEEE International Conference on Computer Vision. Presented at the ICCV 2003: 9th International Conference on Computer Vision, IEEE, Nice, France, pp. 1470–1477 vol.2. https://doi.org/10.1109/ICCV.2003.1238663

Sobel, I., 2014. An Isotropic 3x3 Image Gradient Operator [WWW Document]. URL https://www.researchgate.net/publication/239398674_An_Isotropic_3x3_Image_Gradient_Operator

SUN Database [WWW Document], n.d. URL https://groups.csail.mit.edu/vision/SUN/

Tareen, S.A.K., Saleem, Z., 2018. A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK. https://doi.org/10.1109/ICOMET.2018.8346440

Text Retrieval Overview - University of Illinois - Computer Science [WWW Document], 2014. URL http://times.cs.uiuc.edu/course/410/note/tr.html

The PASCAL Visual Object Classes Homepage [WWW Document], 2020. URL http://host.robots.ox.ac.uk/pascal/VOC/

Thomee, B., Lew, M.S., 2012. Interactive search in image retrieval: a survey. Int. J. Multimed. Inf. Retr. 1, 71–86. https://doi.org/10.1007/s13735-012-0014-4

Torres, R.S., Silva, C.G., Medeiros, C.B., Rocha, H.V., n.d. Visual Structures for Image Browsing 7.

Tyagi, D., 2020. Introduction To Feature Detection And Matching [WWW Document]. Medium. URL https://medium.com/analytics-vidhya/introduction-to-feature-detection-and-matching-65e27179885d

USDA - NAIP Imagery [WWW Document], 2020. . temp_FSA_02_Landing_InteriorPages. URL https://www.fsa.usda.gov/programs-and-services/aerial-photography/imagery-programs/naip-imagery/

Xie, Y., Li, L.L., Wang, H., Zhao, X., 2010. The application of threshold methods for image segmentation in oasis vegetation extraction [WWW Document]. ResearchGate. URL https://www.researchgate.net/publication/220697282_The_application_of_threshold_methods_for_image_segmentation_in_oasis_vegetation_extraction

Xiong, B., Jain, S.D., Grauman, K., 2018. Pixel Objectness: Learning to Segment Generic Objects Automatically in Images and Videos. ArXiv180804702 Cs.

Xu, J., 2017. Deep Learning for Object Detection: A Comprehensive Review [WWW Document]. Medium. URL https://towardsdatascience.com/deep-learning-for-object-detection-a-comprehensive-review-73930816d8d9

YouTube - Traffic monitoring video, 2020.

Yuille, A.L., Liu, C., 2019. Limitations of Deep Learning for Vision, and How We Might Fix Them [WWW Document]. The Gradient. URL https://thegradient.pub/the-limitations-of-visual-deep-learning-and-how-we-might-fix-them/

Zanaty, E.A., 2016. Medical Image Segmentation Techniques: An Overview [WWW Document]. ResearchGate. URL https://www.researchgate.net/publication/294682473_Medical_Image_Segmentation_Techniques_An_Overview

Zhang, L., Chen, L., Jing, F., Deng, K., Ma, W.-Y., 2006. EnjoyPhoto: a vertical image search engine for enjoying high-quality photos, in: Proceedings of the 14th ACM International Conference on Multimedia, MM '06. Association for Computing Machinery, New York, NY, USA, pp. 367–376. https://doi.org/10.1145/1180639.1180719

# Anexo A: Código de testes realizados

## I. Testes da técnica *threshold segmentation*

```
1.  # Import necessary libraries
2.  import cv2, time
3.
4.  # Open image to process
5.  img = cv2.imread("apples.jpg", cv2.IMREAD_COLOR)
6.
7.  # region Test 1 - 1 threshold
8.  # Start timer
9.  start_time = int(time.time() * 1000)
10.
11. # Process image
12. # Convert image to grayscale
13. img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
14.
15. # Calculate average pixel value to use as threshold
16. sum = 0
17. for row in range(img_gray.shape[0]):
18.     for column in range(img_gray.shape[1]):
19.         sum += img_gray[row, column]
20.
21. threshold_1 = sum / (img_gray.shape[0] * img_gray.shape[1])
22.
23. # Compare each pixel with the thresholds and assign new values
24. for row in range(img_gray.shape[0]):
25.     for column in range(img_gray.shape[1]):
26.         if img_gray[row, column] <= threshold_1:
27.             img_gray[row, column] = 0
28.         else:
29.             img_gray[row, column] = 255
30.
31. # Stop timer
32. end_time = int(time.time() * 1000)
33.
34. # Calculate processing time
35. processing_time = end_time - start_time
36.
37. # Save image to disk
38. cv2.imwrite("TS_1_{}.jpg".format(processing_time), img_gray)
39. # endregion
40.
41. # region Test 2 - 2 threshold
42. # Start timer
43. start_time = int(time.time() * 1000)
44.
45. # Process image
46. # Convert image to grayscale
47. img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
48. threshold_1 = 85
49. threshold_2 = 170
50.
51. # Compare each pixel with the thresholds and assign new values
52. for row in range(img_gray.shape[0]):
53.     for column in range(img_gray.shape[1]):
54.         if img_gray[row, column] <= threshold_1:
55.             img_gray[row, column] = 0
56.         elif img_gray[row, column] <= threshold_2:
57.             img_gray[row, column] = 127
58.         else:
```

```
            img_gray[row, column] = 255

# Stop timer
end_time = int(time.time() * 1000)

# Calculate processing time
processing_time = end_time - start_time

# Save image to disk
cv2.imwrite("TS_2_{}.jpg".format(processing_time), img_gray)
# endregion

# region Test 3 - 3 threshold
# Start timer
start_time = int(time.time() * 1000)

# Process image
# Convert image to grayscale
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

threshold_1 = 63
threshold_2 = 126
threshold_3 = 189

# Compare each pixel with the thresholds and assign new values
for row in range(img_gray.shape[0]):
    for column in range(img_gray.shape[1]):
        if img_gray[row, column] <= threshold_1:
            img_gray[row, column] = 0
        elif img_gray[row, column] <= threshold_2:
            img_gray[row, column] = 85
        elif img_gray[row, column] <= threshold_3:
            img_gray[row, column] = 170
        else:
            img_gray[row, column] = 255

# Stop timer
end_time = int(time.time() * 1000)

# Calculate processing time
processing_time = end_time - start_time

# Save image to disk
cv2.imwrite("TS_3_{}.jpg".format(processing_time), img_gray)
# endregion
```

## II. Testes de algoritmos de *edge detection*

```
# Import necessary libraries
import cv2, time

# Open image to process and define thresholds
img = cv2.imread("apples.jpg", cv2.IMREAD_COLOR)

# region Test 1 - Laplacian Filter
# Process image
# Start timer
start_time = int(time.time() * 1000)

# Remove noise
img_less_noise = cv2.GaussianBlur(img, (3, 3), 0)

# Convert image to grayscale
```

```
16. img_gray = cv2.cvtColor(img_less_noise, cv2.COLOR_BGR2GRAY)
17.
18. # Laplacian Filter
19. output = cv2.Laplacian(img_gray, cv2.CV_64F)
20.
21. # Stop timer
22. end_time = int(time.time() * 1000)
23.
24. # Calculate processing time
25. processing_time = end_time - start_time
26.
27. # Save image to disk
28. cv2.imwrite("EDS_Laplacian_{}.jpg".format(processing_time), output)
29. # endregion
30.
31. # region Test 2 - Canny Edge Detector
32. # Process image
33. # Start timer
34. start_time = int(time.time() * 1000)
35.
36. # Convert image to grayscale
37. img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
38.
39. # Canny Edge Detection
40. output = cv2.Canny(img_gray, 100, 200)
41.
42. # Stop timer
43. end_time = int(time.time() * 1000)
44.
45. # Calculate processing time
46. processing_time = end_time - start_time
47.
48. # Save image to disk
49. cv2.imwrite("EDS_Canny_{}.jpg".format(processing_time), output)
50. # endregion
```

## III. Testes de métodos de *feature matching* (*SIFT*, *SURF*, *ORB*, *BRISK*, *AKAZE*) (*CPU*)

```
1.  # Import necessary libraries
2.  import time
3.  import cv2
4.  import numpy as np
5.
6.  # Reference image
7.  template = cv2.imread("./resources/remote_template1.png")
8.  template_gray = cv2.cvtColor(template, cv2.COLOR_BGR2GRAY)
9.
10. # Feature matching methods
11. sift = cv2.xfeatures2d.SIFT_create()
12. surf = cv2.xfeatures2d.SURF_create()
13. orb = cv2.ORB_create(nfeatures=100000)
14. brisk = cv2.BRISK_create()
15. akaze = cv2.AKAZE_create()
16.
17. # Brute-force matchers
18. bf_l2 = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
19. bf_hamming = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
20.
21. def process(method, matcher):
22.     """
23.     Processes each frame of a video file through feature matching
```

```
24.     and analyses the performance of the method used
25.     :param method: Algorithm to be used (SIFT/SURF/ORB/BRISK/AKAZE)
26.     :param matcher: Matcher to be used (L2 Norm or Hamming Norm)
27.     """
28.     # Video for analysis
29.     cap = cv2.VideoCapture("./resources/video_sample.mp4")
30.
31.     # Auxiliary variables
32.     num_frames = 0
33.     sum_num_features = 0
34.     sum_matches = 0
35.     loop_setup_times = []
36.     loop_kpdes_times = []
37.     loop_match_times = []
38.     loop_guess_times = []
39.     loop_draw_times = []
40.
41.     # Get time before starting to process
42.     start_time = time.time() * 1000
43.
44.     # Detect reference image keypoints/descriptors
45.     kp1, des1 = method.detectAndCompute(template_gray, None)
46.
47.     while cap.isOpened():
48.         # Get next frame
49.         ret, frame = cap.read()
50.         if not ret:
51.             break
52.
53.         # Get time before keypoints/descriptors detection
54.         loop_setup_times.append(time.time() * 1000)
55.         # Detect keypoints/descriptors
56.         gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
57.         kp2, des2 = method.detectAndCompute(gray, None)
58.         # Get time after keypoints detection and before matching
59.         loop_kpdes_times.append(time.time() * 1000)
60.
61.         # Brute Force Matching
62.         matches = matcher.match(des1, des2)
63.         # Get time after matching and before homography
64.         loop_match_times.append(time.time() * 1000)
65.
66.         # Homography and perspective transform
67.         matches = sorted(matches, key=lambda x: x.distance)
68.         src_pts = np.float32(
69.             [kp1[m.queryIdx].pt for m in matches]).reshape(-1, 1, 2)
70.         dst_pts = np.float32(
71.             [kp2[m.trainIdx].pt for m in matches]).reshape(-1, 1, 2)
72.         M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
73.         h, w = template.shape[:2]
74.         pts = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]])\
75.             .reshape(-1, 1, 2)
76.         dst = cv2.perspectiveTransform(pts, M)
77.         dst += (w, 0)
78.         # Get time after homography and before drawing
79.         loop_guess_times.append(time.time() * 1000)
80.
81.         # Draw result
82.         offset_kp2 = []
83.         for kp in kp2:  # Offset kps by the ref. image width
84.             offset_kp2.append(cv2.KeyPoint(kp.pt[0] + w, kp.pt[1],
85.                                            kp.size, kp.response,
86.                                            kp.octave, kp.class_id))
87.         # Draw matches
88.         matching_result = cv2.drawMatches(template, kp1, frame, kp2,
    matches,
89.                                           None, flags=2)
90.         # Draw keypoints
```

```
91.         matching_result = cv2.drawKeypoints(matching_result, offset_kp2,
92.                                             None, color=(0, 0, 255))
93.         # Draw bounding box
94.         matching_result = cv2.polylines(matching_result, [np.int32(dst)],
95.                                         True, (0, 255, 0), 3, cv2.LINE_AA)
96.         # Get time after drawing
97.         loop_draw_times.append(time.time() * 1000)
98.
99.         # Update auxiliary variables
100.          num_frames += 1
101.          sum_num_features += len(kp2)
102.          sum_matches += len(matches)
103.
104.     # Get time after finishing processing
105.     end_time = time.time() * 1000
106.
107.     # Process data and output results
108.     avg_kpdes_time = 0
109.     avg_match_time = 0
110.     avg_guess_time = 0
111.     avg_draw_time = 0
112.
113.     for i in range(num_frames):
114.         avg_kpdes_time += loop_kpdes_times[i] - loop_setup_times[i]
115.         avg_match_time += loop_match_times[i] - loop_kpdes_times[i]
116.         avg_guess_time += loop_guess_times[i] - loop_match_times[i]
117.         avg_draw_time += loop_draw_times[i] - loop_guess_times[i]
118.
119.     total_time = round((end_time - start_time) / 1000, 3)
120.     avg_fps = round(num_frames / total_time, 3)
121.     avg_kpdes_time = round((avg_kpdes_time / num_frames) / 1000, 3)
122.     avg_match_time = round((avg_match_time / num_frames) / 1000, 3)
123.     avg_guess_time = round((avg_guess_time / num_frames) / 1000, 3)
124.     avg_draw_time = round((avg_draw_time / num_frames) / 1000, 3)
125.     avg_num_features = round(sum_num_features / num_frames)
126.     avg_num_matches = round(sum_matches / num_frames)
127.     avg_ratio_matches_features = round(avg_num_matches / avg_num_features,
    3)
128.     avg_ratio_features_time = round(avg_num_features / avg_kpdes_time)
129.
130.     print("Method: {}\n"
131.           "Average frames/s: {}\n"
132.           "Average keypoints/descriptors calculation time: {}s\n"
133.           "Average matching time: {}s\n"
134.           "Average guess processing time: {}s\n"
135.           "Average drawing time: {}s\n"
136.           "Average number of features: {}\n"
137.           "Average number of matches: {}\n"
138.           "Average ratio matches/features: {}\n"
139.           "Average ratio features/feature detection time: {}\n"
140.           "Total time: {}s\n\n"
141.           .format(method, avg_fps, avg_kpdes_time, avg_match_time,
142.                   avg_guess_time, avg_draw_time, avg_num_features,
143.                   avg_num_matches, avg_ratio_matches_features,
144.                   avg_ratio_features_time, total_time))
145.
146.     cap.release()
147.
148.
149. process(sift, bf_l2)
150. process(surf, bf_l2)
151. process(orb, bf_hamming)
152. process(brisk, bf_hamming)
153. process(akaze, bf_hamming)
```

## IV. Teste do *SURF* (*GPU*)

```
1.  import time
2.  import cv2
3.  import numpy as np
4.
5.  # Reference image
6.  ref = cv2.imread("./resources/remote_template1.png")
7.  ref_g = cv2.cvtColor(ref, cv2.COLOR_RGB2GRAY)
8.  ref_g_cuda = cv2.cuda_GpuMat()
9.  ref_g_cuda.upload(ref_g)
10.
11. # Feature matching method
12. surf_cuda = cv2.cuda.SURF_CUDA_create(400)
13.
14. # Brute-force matcher
15. bf_l2 = cv2.cuda_DescriptorMatcher.createBFMatcher(cv2.NORM_L2)
16.
17. # Video for analysis
18. cap = cv2.VideoCapture("./resources/video_sample.mp4")
19.
20. # Auxiliary variables
21. num_frames = 0
22. sum_num_features = 0
23. sum_matches = 0
24. loop_setup_times = []
25. loop_kpdes_times = []
26. loop_match_times = []
27. loop_guess_times = []
28. loop_draw_times = []
29. warm = False
30.
31. # Get time before starting to process
32. start_time = time.time() * 1000
33.
34. # Detect reference image keypoints/descriptors
35. kp1_cuda, des1_cuda = \
36.     surf_cuda.detectWithDescriptors(img=ref_g_cuda, mask=None)
37.
38. # Convert Keypoints to CPU
39. kp1_cpu = surf_cuda.downloadKeypoints(kp1_cuda)
40.
41. while cap.isOpened():
42.     # Get next frame
43.     ret, frame = cap.read()
44.
45.     # If the video is over we stop
46.     if not ret:
47.         break
48.
49.     # The first time we process using the gpu takes a lot longer than after
50.     # it's been running for a while. As such we do a warmup before starting
51.     # the actual test
52.     if not warm:
53.         # Detect keypoints/descriptors
54.         analysis_g = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
55.         analysis_g_cuda = cv2.cuda_GpuMat()
56.         analysis_g_cuda.upload(analysis_g)
57.         kp2_cuda, des2_cuda = \
58.             surf_cuda.detectWithDescriptors(img=analysis_g_cuda, mask=None)
59.         kp2_cpu = surf_cuda.downloadKeypoints(kp2_cuda)
60.         # Only warmup once, before processing the first frame
61.         warm = True
62.
63.     # Get time before keypoints/descriptors detection
64.     loop_setup_times.append(time.time() * 1000)
```

```
65.     # Detect keypoints/descriptors
66.     analysis_g = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
67.     analysis_g_cuda = cv2.cuda_GpuMat()
68.     analysis_g_cuda.upload(analysis_g)
69.     kp2_cuda, des2_cuda = \
70.         surf_cuda.detectWithDescriptors(img=analysis_g_cuda, mask=None)
71.     kp2_cpu = surf_cuda.downloadKeypoints(kp2_cuda)
72.     # Get time after keypoints detection and before matching
73.     loop_kpdes_times.append(time.time() * 1000)
74.
75.     # Brute Force Matching
76.     matches = bf_l2.match(des1_cuda, des2_cuda)
77.     # Get time after matching and before homography
78.     loop_match_times.append(time.time() * 1000)
79.
80.     # Homography and perspective transform
81.     matches = sorted(matches, key=lambda x: x.distance)
82.     src_pts = np.float32(
83.         [kp1_cpu[m.queryIdx].pt for m in matches]) \
84.         .reshape(-1, 1, 2)
85.     dst_pts = np.float32(
86.         [kp2_cpu[m.trainIdx].pt for m in matches]) \
87.         .reshape(-1, 1, 2)
88.     M, mask = cv2.findHomography(src_pts,
89.                                  dst_pts,
90.                                  cv2.RANSAC,
91.                                  5.0)
92.     h, w = ref.shape[:2]
93.     pts = np.float32([[0, 0],
94.                       [0, h - 1],
95.                       [w - 1, h - 1],
96.                       [w - 1, 0]]).reshape(-1, 1, 2)
97.     dst = cv2.perspectiveTransform(pts, M)
98.     dst += (w, 0)
99.     # Get time after homography and before drawing
100.      loop_guess_times.append(time.time() * 1000)
101.
102.      # Draw result
103.      offset_kp2 = []
104.      for kp in kp2_cpu:  # Offset kps by the ref. image width
105.          offset_kp2.append(cv2.KeyPoint(kp.pt[0] + w, kp.pt[1],
106.                                         kp.size, kp.response,
107.                                         kp.octave, kp.class_id))
108.      # Draw matches
109.      matching_result = cv2.drawMatches(ref, kp1_cpu, frame,
110.                                        kp2_cpu, matches, None, flags=2)
111.      # Draw keypoints
112.      matching_result = cv2.drawKeypoints(matching_result,
113.                                          offset_kp2,
114.                                          None,
115.                                          color=(0, 0, 255))
116.      # Draw bounding box
117.      matching_result = cv2.polylines(matching_result,
118.                                      [np.int32(dst)],
119.                                      True, (0, 255, 0),
120.                                      3, cv2.LINE_AA)
121.      # Get time after drawing
122.      loop_draw_times.append(time.time() * 1000)
123.
124.      # Update auxiliary variables
125.      num_frames += 1
126.      sum_num_features += len(kp2_cpu)
127.      sum_matches += len(matches)
128.
129.  # Get time after finishing processing
130.  end_time = time.time() * 1000
131.
132.  # Process data and output results
```

```
avg_kpdes_time = 0
avg_match_time = 0
avg_guess_time = 0
avg_draw_time = 0

for i in range(num_frames):
    avg_kpdes_time += loop_kpdes_times[i] - loop_setup_times[i]
    avg_match_time += loop_match_times[i] - loop_kpdes_times[i]
    avg_guess_time += loop_guess_times[i] - loop_match_times[i]
    avg_draw_time += loop_draw_times[i] - loop_guess_times[i]

total_time = round((end_time - start_time) / 1000, 3)
avg_fps = round(num_frames / total_time, 3)
avg_kpdes_time = round((avg_kpdes_time / num_frames) / 1000, 3)
avg_match_time = round((avg_match_time / num_frames) / 1000, 3)
avg_guess_time = round((avg_guess_time / num_frames) / 1000, 3)
avg_draw_time = round((avg_draw_time / num_frames) / 1000, 3)
avg_num_features = round(sum_num_features / num_frames)
avg_num_matches = round(sum_matches / num_frames)
avg_ratio_matches_features = round(avg_num_matches / avg_num_features, 3)
avg_ratio_features_time = round(avg_num_features / avg_kpdes_time)

print("Average frames/s: {}\n"
      "Average keypoints/descriptors calculation time: {}s\n"
      "Average matching time: {}s\n"
      "Average guess processing time: {}s\n"
      "Average drawing time: {}s\n"
      "Average number of features: {}\n"
      "Average number of matches: {}\n"
      "Average ratio matches/features: {}\n"
      "Average ratio features/feature detection time: {}\n"
      "Total time: {}s\n\n"
      .format(avg_fps, avg_kpdes_time, avg_match_time,
              avg_guess_time, avg_draw_time, avg_num_features,
              avg_num_matches, avg_ratio_matches_features,
              avg_ratio_features_time, total_time))

cap.release()
```

## V. Teste do *ORB* (*GPU*)

```
import time
import cv2
import numpy as np

# Reference image
ref = cv2.imread("./resources/remote_template1.png")
ref_cuda = cv2.cuda_GpuMat()
ref_cuda.upload(ref)
ref_g_cuda = cv2.cuda.cvtColor(ref_cuda, cv2.COLOR_RGB2GRAY)

# Feature matching method
orb_cuda = cv2.cuda.ORB_create(nfeatures=100000)

# Brute-force matcher
bf_hamming = cv2.cuda_DescriptorMatcher.createBFMatcher(cv2.NORM_HAMMING)

# Video for analysis
cap = cv2.VideoCapture("./resources/video_sample.mp4")

# Auxiliary variables
num_frames = 0
sum_num_features = 0
```

```
sum_matches = 0
loop_setup_times = []
loop_kpdes_times = []
loop_match_times = []
loop_guess_times = []
loop_draw_times = []
warm = False

# Get time before starting to process
start_time = time.time() * 1000

# Detect reference image keypoints/descriptors
kp1_cuda, des1_cuda = \
    orb_cuda.detectAndComputeAsync(image=ref_g_cuda, mask=None)

# Convert Keypoints to CPU
kp1_cpu = orb_cuda.convert(kp1_cuda)

while cap.isOpened():
    # Get next frame
    ret, frame = cap.read()

    # If the video is over we stop
    if not ret:
        break

    # The first time we process using the gpu takes a lot longer than after
    # it's been running for a while. As such we do a warmup before starting
    # the actual test
    if not warm:
        # Detect keypoints/descriptors
        analysis_cuda = cv2.cuda_GpuMat()
        analysis_cuda.upload(frame)
        analysis_g_cuda = \
            cv2.cuda.cvtColor(analysis_cuda, cv2.COLOR_RGB2GRAY)
        kp2_cuda, des2_cuda = \
            orb_cuda.detectAndComputeAsync(image=analysis_g_cuda, mask=None)
        kp2_cpu = orb_cuda.convert(kp2_cuda)
        # Only warmup once, before processing the first frame
        warm = True

    # Get time before keypoints/descriptors detection
    loop_setup_times.append(time.time() * 1000)
    # Detect keypoints/descriptors
    analysis_cuda = cv2.cuda_GpuMat()
    analysis_cuda.upload(frame)
    analysis_g_cuda = cv2.cuda.cvtColor(analysis_cuda, cv2.COLOR_RGB2GRAY)
    kp2_cuda, des2_cuda = \
        orb_cuda.detectAndComputeAsync(image=analysis_g_cuda, mask=None)
    kp2_cpu = orb_cuda.convert(kp2_cuda)
    # Get time after keypoints detection and before matching
    loop_kpdes_times.append(time.time() * 1000)

    # Brute Force Matching
    matches = bf_hamming.match(des1_cuda, des2_cuda)
    # Get time after matching and before homography
    loop_match_times.append(time.time() * 1000)

    # Homography and perspective transform
    matches = sorted(matches, key=lambda x: x.distance)
    src_pts = np.float32(
        [kp1_cpu[m.queryIdx].pt for m in matches]) \
        .reshape(-1, 1, 2)
    dst_pts = np.float32(
        [kp2_cpu[m.trainIdx].pt for m in matches]) \
        .reshape(-1, 1, 2)
    M, mask = cv2.findHomography(src_pts,
                                 dst_pts,
```

```
                                    cv2.RANSAC,
                                    5.0)
    h, w = ref.shape[:2]
    pts = np.float32([[0, 0],
                      [0, h - 1],
                      [w - 1, h - 1],
                      [w - 1, 0]]).reshape(-1, 1, 2)
    dst = cv2.perspectiveTransform(pts, M)
    dst += (w, 0)
      # Get time after homography and before drawing
      loop_guess_times.append(time.time() * 1000)

      # Draw result
      offset_kp2 = []
      for kp in kp2_cpu:  # Offset kps by the ref. image width
          offset_kp2.append(cv2.KeyPoint(kp.pt[0] + w, kp.pt[1],
                                         kp.size, kp.response,
                                         kp.octave, kp.class_id))
      # Draw matches
      matching_result = cv2.drawMatches(ref, kp1_cpu, frame,
                                        kp2_cpu, matches, None, flags=2)
      # Draw keypoints
      matching_result = cv2.drawKeypoints(matching_result,
                                          offset_kp2,
                                          None,
                                          color=(0, 0, 255))
      # Draw bounding box
      matching_result = cv2.polylines(matching_result,
                                      [np.int32(dst)],
                                      True, (0, 255, 0),
                                      3, cv2.LINE_AA)
      # Get time after drawing
      loop_draw_times.append(time.time() * 1000)

      # Update auxiliary variables
      num_frames += 1
      sum_num_features += len(kp2_cpu)
      sum_matches += len(matches)

# Get time after finishing processing
end_time = time.time() * 1000

# Process data and output results
avg_kpdes_time = 0
avg_match_time = 0
avg_guess_time = 0
avg_draw_time = 0

for i in range(num_frames):
    avg_kpdes_time += loop_kpdes_times[i] - loop_setup_times[i]
    avg_match_time += loop_match_times[i] - loop_kpdes_times[i]
    avg_guess_time += loop_guess_times[i] - loop_match_times[i]
    avg_draw_time += loop_draw_times[i] - loop_guess_times[i]

total_time = round((end_time - start_time) / 1000, 3)
avg_fps = round(num_frames / total_time, 3)
avg_kpdes_time = round((avg_kpdes_time / num_frames) / 1000, 3)
avg_match_time = round((avg_match_time / num_frames) / 1000, 3)
avg_guess_time = round((avg_guess_time / num_frames) / 1000, 3)
avg_draw_time = round((avg_draw_time / num_frames) / 1000, 3)
avg_num_features = round(sum_num_features / num_frames)
avg_num_matches = round(sum_matches / num_frames)
avg_ratio_matches_features = round(avg_num_matches / avg_num_features, 3)
avg_ratio_features_time = round(avg_num_features / avg_kpdes_time)

print("Average frames/s: {}\n"
      "Average keypoints/descriptors calculation time: {}s\n"
      "Average matching time: {}s\n"
```

```
159.        "Average guess processing time: {}s\n"
160.        "Average drawing time: {}s\n"
161.        "Average number of features: {}\n"
162.        "Average number of matches: {}\n"
163.        "Average ratio matches/features: {}\n"
164.        "Average ratio features/feature detection time: {}\n"
165.        "Total time: {}s\n\n"
166.        .format(avg_fps, avg_kpdes_time, avg_match_time,
167.                avg_guess_time, avg_draw_time, avg_num_features,
168.                avg_num_matches, avg_ratio_matches_features,
169.                avg_ratio_features_time, total_time))
170.
171. cap.release()
```

## VI. Teste do modelo *YOLOv3* (*CPU* e *GPU*)

```
1.  # Import necessary libraries
2.  import time
3.  import cv2
4.  import numpy as np
5.
6.  # Constants
7.  confidence_threshold = 0.5
8.  threshold = 0.3
9.
10. # COCO class labels that the model was trained to detect
11. labels = ['person', 'bicycle', 'car', 'motorbike', 'aeroplane', 'bus',
12.           'train', 'truck', 'boat', 'traffic light', 'fire hydrant',
13.           'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog',
14.           'horse', 'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe',
15.           'backpack', 'umbrella', 'handbag', 'tie', 'suitcase', 'frisbee',
16.           'skis', 'snowboard', 'sports ball', 'kite', 'baseball bat',
17.           'baseball glove', 'skateboard', 'surfboard', 'tennis racket',
18.           'bottle', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
19.           'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot',
20.           'hot dog', 'pizza', 'donut', 'cake', 'chair', 'sofa',
    'pottedplant',
21.           'bed', 'diningtable', 'toilet', 'tvmonitor', 'laptop', 'mouse',
22.           'remote', 'keyboard', 'cell phone', 'microwave', 'oven',
    'toaster',
23.           'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors',
24.           'teddy bear', 'hair drier', 'toothbrush'
25.           ]
26.
27. # Get a list of colors to represent the classes
28. np.random.seed(5)  # Set a seed so that we always get the same colors
29. colors = np.random.randint(0, 255, size=(len(labels), 3), dtype="uint8")
30.
31. # Load the YOLO model from disk
32. net = cv2.dnn.readNetFromDarknet("./yolo_model/yolov3.cfg",  # config
33.                                 "./yolo_model/yolov3.weights")  # weights
34.
35. # Enable gpu processing - To disable just remove these two lines
36. net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
37. net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)
38.
39. # Open video stream
40. vs = cv2.VideoCapture("./resources/traffic_video_sample.mkv")
41.
42. # Auxiliary variables
43. num_frames = 0
44. total_object_detections = 0
45. pre_network_times = []
```

```
post_network_times = []
post_processing_times = []
warm = False

# Get starting time
start_time = time.time() * 1000

# Loop over frames
while True:
    # Get the next frame
    ret, frame = vs.read()

    # If the video is over we stop
    if not ret:
        break

    # The first time we process using the gpu takes a lot longer than after
    # it's been running for a while. As such we do a warmup before starting
    # the actual test
    if not warm:
        # Determine only the output layer names that we need from YOLO
        layer_names = net.getLayerNames()
        layer_names = [layer_names[i[0] - 1]
                       for i in net.getUnconnectedOutLayers()]
        # Construct a blob from the frame
        blob = cv2.dnn.blobFromImage(frame, 1 / 255.0, (416, 416),
                                     swapRB=True, crop=False)
        # Set it as input to the neural network
        net.setInput(blob)
        # Run it through the network
        layerOutputs = net.forward(layer_names)
        # Only warmup once, before processing the first frame
        warm = True

    # Get the time before running the image through the neural network
    pre_network_times.append(time.time() * 1000)
    layer_names = net.getLayerNames()
    layer_names = [layer_names[i[0] - 1]
                   for i in net.getUnconnectedOutLayers()]
    blob = cv2.dnn.blobFromImage(frame, 1 / 255.0, (416, 416),
                                 swapRB=True, crop=False)
    net.setInput(blob)
    layerOutputs = net.forward(layer_names)
    # Get the time after we're done with the network part and before
    # processing the results
    post_network_times.append(time.time() * 1000)

    # Lists of detected bounding boxes, confidences, and class IDs
    boxes = []
    confidences = []
    class_ids = []

    # Loop over each of the layer outputs
    for output in layerOutputs:
        # Loop over the detections
        for detection in output:
            scores = detection[5:]
            class_id = np.argmax(scores)  # Get the id
            class_name = labels[class_id]  # Get the label
            confidence = scores[class_id]  # Get the confidence

            # If this detection's confidence is higher than our set
            # confidence threshold then we consider it relevant and process
it
            if confidence > confidence_threshold:
                # Scale the bounding box coordinates back relative to the
                # size of the image, keeping in mind that YOLO actually
                # returns the center (x, y)-coordinates of the bounding
```

```
                # box followed by the boxes' width and height
                (H, W) = frame.shape[:2]
                box = detection[0:4] * np.array([W, H, W, H])
                (centerX, centerY, width, height) = box.astype("int")

                # Get the bounding box top left coordinates
                startX = int(centerX - (width / 2))
                startY = int(centerY - (height / 2))

                # Update lists of bounding box coordinates, confidences,
                # and class IDs
                boxes.append([startX, startY, int(width), int(height)])
                confidences.append(float(confidence))
                class_ids.append(class_id)

    # Apply non-maxima suppression to suppress weak,
    # overlapping bounding boxes
    indexes = cv2.dnn.NMSBoxes(boxes, confidences, confidence_threshold,
                               threshold)

    # If at least one detection exists
    if len(indexes) > 0:
        indexes = indexes.flatten()
        # Add the number of relevant detections to the total
        total_object_detections += len(indexes)
        # Loop over the indexes we are keeping
        for i in indexes:
            # Get the bounding box coordinates
            (startX, startY) = (boxes[i][0], boxes[i][1])
            (w, h) = (boxes[i][2], boxes[i][3])

            # Draw the bounding box
            color = [int(c) for c in colors[class_ids[i]]]
            cv2.rectangle(frame, (startX, startY),
                          (startX + w, startY + h), color, 8)

            # Draw the class label and confidence
            text = "{}: {:.4f}".format(labels[class_ids[i]],
confidences[i])
            cv2.putText(frame, text, (startX, startY - 10),
                        cv2.FONT_HERSHEY_SIMPLEX, 1, color, 4)

    # Get time after we finish processing this frame
    post_processing_times.append(time.time() * 1000)
    # Increase total number of frames
    num_frames += 1

# Get time after we finish processing every frame
end_time = time.time() * 1000

# Process data and output results
sum_network_time = 0
sum_results_processing_time = 0

for i in range(num_frames):
    sum_network_time += post_network_times[i] - pre_network_times[i]
    sum_results_processing_time += \
        post_processing_times[i] - post_network_times[i]

total_time = round((end_time - start_time) / 1000, 3)
avg_fps = round(num_frames / total_time, 3)
avg_network_time = round((sum_network_time / num_frames) / 1000, 3)
avg_result_processing_time = \
    round((sum_results_processing_time / num_frames) / 1000, 3)
avg_object_detections = round(total_object_detections / num_frames)

print("Average fps: {}\n"
      "Average network processing time: {}\n"
```

```
        "Average result processing time: {}\n"
        "Average number of relevant object detections: {}\n"
        "Total number of relevant object detections: {}\n"
        "Number of frames: {}\n"
        "Total time: {}".format(avg_fps, avg_network_time,
                                avg_result_processing_time,
                                avg_object_detections,
                                total_object_detections,
                                num_frames, total_time))

vs.release()
```

## VII. Teste do modelo *SSD* (*CPU* e *GPU*)

```
# Import necessary libraries
import time
import cv2
import numpy as np

# Constants
confidence_threshold = 0.2

# Class labels MobileNet SSD was trained to detect
labels = ["background", "aeroplane", "bicycle", "bird", "boat",
          "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
          "dog", "horse", "motorbike", "person", "pottedplant", "sheep",
          "sofa", "train", "tvmonitor"
          ]

# Get a list of colors to represent the classes
np.random.seed(5)  # Set a seed so that we always get the same colors
colors = np.random.uniform(0, 255, size=(len(labels), 3))

# Load the SSD model from disk
net =
   cv2.dnn.readNetFromCaffe("./ssd_model/MobileNetSSD_deploy.prototxt.txt",
                            "./ssd_model/MobileNetSSD_deploy.caffemodel")

# Enable gpu processing - To disable just remove these two lines
net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)

# Open video stream
vs = cv2.VideoCapture("./resources/traffic_video_sample.mkv")

# Auxiliary variables
num_frames = 0
total_object_detections = 0
pre_network_times = []
post_network_times = []
post_processing_times = []
warm = False

# Get starting time
start_time = time.time() * 1000

# Loop over frames
while True:
    # Get the next frame
    ret, frame = vs.read()

    # If the video is over we stop
    if not ret:
```

```
49.         break
50.
51.     # The first time we process using the gpu takes a lot longer than after
52.     # it's been running for a while. As such we do a warmup before starting
53.     # the actual test
54.     if not warm:
55.         # Resize the frame to 300x300 pixels, normalize it and construct a
    blob
56.         blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)),
57.                                      0.007843, (300, 300), 127.5)
58.         # Set it as input to the neural network
59.         net.setInput(blob)
60.         # Run it through the network
61.         detections = net.forward()
62.         # Only warmup once, before processing the first frame
63.         warm = True
64.
65.     # Get the time before running the image through the neural network
66.     pre_network_times.append(time.time() * 1000)
67.     blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)),
68.                                  0.007843, (300, 300), 127.5)
69.     net.setInput(blob)
70.     detections = net.forward()
71.     # Get the time after we're done with the network part and before
    processing
72.     # the results
73.     post_network_times.append(time.time() * 1000)
74.
75.     # Loop over the detections
76.     for i in np.arange(0, detections.shape[2]):
77.         confidence = detections[0, 0, i, 2]  # Get the confidence
78.
79.         # If this detection's confidence is higher than our set confidence
80.         # threshold then we consider it relevant and process it
81.         if confidence > confidence_threshold:
82.             # Increase the total number of relevant objects detected
83.             total_object_detections += 1
84.             class_id = int(detections[0, 0, i, 1])  # Get the id
85.             class_name = labels[class_id]  # Get the label
86.
87.             # Scale the bounding box coordinates back,
88.             # relative to the size of the frame
89.             (H, W) = frame.shape[:2]
90.             box = detections[0, 0, i, 3:7] * np.array([W, H, W, H])
91.             (startX, startY, endX, endY) = box.astype("int")
92.
93.             # Draw the bounding box
94.             color = colors[class_id]
95.             cv2.rectangle(frame, (startX, startY), (endX, endY), color, 8)
96.
97.             # Draw the class label and confidence
98.             text = "{}: {:.4f}".format(class_name, confidence)
99.             cv2.putText(frame, text, (startX, startY - 10),
100.                         cv2.FONT_HERSHEY_SIMPLEX, 1, color, 4)
101.
102.     # Get time after we finish processing this frame
103.     post_processing_times.append(time.time() * 1000)
104.     # Increase total number of frames
105.     num_frames += 1
106.
107. # Get time after we finish processing every frame
108. end_time = time.time() * 1000
109.
110. # Process data and output results
111. sum_network_time = 0
112. sum_results_processing_time = 0
113.
114. for i in range(num_frames):
```

```
115.     sum_network_time += post_network_times[i] - pre_network_times[i]
116.     sum_results_processing_time += \
117.         post_processing_times[i] - post_network_times[i]
118.
119. total_time = round((end_time - start_time) / 1000, 3)
120. avg_fps = round(num_frames / total_time, 3)
121. avg_network_time = round((sum_network_time / num_frames) / 1000, 3)
122. avg_result_processing_time = \
123.     round((sum_results_processing_time / num_frames) / 1000, 3)
124. avg_object_detections = round(total_object_detections / num_frames)
125.
126. print("Average fps: {}\n"
127.       "Average network processing time: {}\n"
128.       "Average result processing time: {}\n"
129.       "Average number of relevant object detections: {}\n"
130.       "Total number of relevant object detections: {}\n"
131.       "Number of frames: {}\n"
132.       "Total time: {}".format(avg_fps, avg_network_time,
133.                               avg_result_processing_time,
134.                               avg_object_detections,
135.                               total_object_detections,
136.                               num_frames, total_time))
137.
138. vs.release()
```

## VIII. Teste do modelo *Mask R-CNN* (*CPU* e *GPU*)

```
1.  # Import necessary libraries
2.  import time
3.  import cv2
4.  import numpy as np
5.
6.  # Constants
7.  confidence_threshold = 0.5
8.  mask_threshold = 0.7
9.
10. # COCO class labels that the model was trained to detect
11. labels = ['person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
    'train',
12.           'truck', 'boat', 'traffic light', 'fire hydrant', 'street sign',
13.           'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog',
    'horse',
14.           'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'hat',
15.           'backpack', 'umbrella', 'shoe', 'eye glasses', 'handbag', 'tie',
16.           'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball', 'kite',
17.           'baseball bat', 'baseball glove', 'skateboard', 'surfboard',
18.           'tennis racket', 'bottle', 'plate', 'wine glass', 'cup', 'fork',
19.           'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich', 'orange',
20.           'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake',
    'chair',
21.           'couch', 'potted plant', 'bed', 'mirror', 'dining table',
    'window',
22.           'desk', 'toilet', 'door', 'tv', 'laptop', 'mouse', 'remote',
23.           'keyboard', 'cell phone', 'microwave', 'oven', 'toaster', 'sink',
24.           'refrigerator', 'blender', 'book', 'clock', 'vase', 'scissors',
25.           'teddy bear', 'hair drier', 'toothbrush'
26.           ]
27.
28. # Get a list of colors to represent the classes
29. np.random.seed(5)  # Set a seed so that we always get the same colors
30. colors = np.random.randint(0, 255, size=(len(labels), 3), dtype="uint8")
31.
32. # Load the Mask R-CNN model from disk
```

```
net = cv2.dnn.readNetFromTensorflow(
    "./mask_rcnn_model/frozen_inference_graph.pb",  # weights
    "./mask_rcnn_model/mask_rcnn_inception_v2_coco_2018_01_28.pbtxt")  # config

# Enable gpu processing - To disable just remove these two lines
net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)

# Open video stream
vs = cv2.VideoCapture("./resources/traffic_video_sample.mkv")

# Auxiliary variables
num_frames = 0
total_object_detections = 0
pre_network_times = []
post_network_times = []
post_processing_times = []
warm = False

# Get starting time
start_time = time.time() * 1000

# Loop over frames
while True:
    # Get the next frame
    ret, frame = vs.read()

    # If the video is over we stop
    if not ret:
        break

    # The first time we process using the gpu takes a lot longer than after
    # it's been running for a while. As such we do a warmup before starting
    # the actual test
    if not warm:
        # Construct a blob from the frame
        blob = cv2.dnn.blobFromImage(frame, swapRB=True, crop=False)
        # Set it as input to the neural network
        net.setInput(blob)
        # Run it through the network
        (boxes, masks) = net.forward(["detection_out_final",
                                      "detection_masks"])
        # Only warmup once, before processing the first frame
        warm = True

    # Get the time before running the image through the neural network
    pre_network_times.append(time.time() * 1000)
    blob = cv2.dnn.blobFromImage(frame, swapRB=True, crop=False)
    net.setInput(blob)
    (boxes, masks) = net.forward(["detection_out_final",
                                  "detection_masks"])
    # Get the time after we're done with the network part and
    # before processing the results
    post_network_times.append(time.time() * 1000)

    # Loop over the detections
    for i in range(0, boxes.shape[2]):
        confidence = boxes[0, 0, i, 2]  # Get the confidence

        # If this detection's confidence is higher than our set confidence
        # threshold then we consider it relevant and process it
        if confidence > confidence_threshold:
            # Increase the total number of relevant objects detected
            total_object_detections += 1
            class_id = int(boxes[0, 0, i, 1])  # Get the id
            class_name = labels[class_id]  # Get the label

```

```
100.             # Scale the bounding box coordinates back, relative to the size
101.             # of the frame, and calculate the dimensions of the bounding
    box
102.             (H, W) = frame.shape[:2]
103.             box = boxes[0, 0, i, 3:7] * np.array([W, H, W, H])
104.             (startX, startY, endX, endY) = box.astype("int")
105.             boxW = endX - startX
106.             boxH = endY - startY
107.
108.             # Get the segmentation mask
109.             mask = masks[i, class_id]
110.             # Resize to the dimensions of the bounding box
111.             mask = cv2.resize(mask, (boxW, boxH),
112.                               interpolation=cv2.INTER_CUBIC)
113.             # Create a binary mask considering our set threshold
114.             mask = (mask > mask_threshold)
115.
116.             # Get the region of the image that corresponds to the mask
117.             region = frame[startY:endY, startX:endX][mask]
118.
119.             # Blend this class's color with the region under the mask
120.             color = colors[class_id]
121.             blended = ((0.4 * color) + (0.6 * region)).astype("uint8")
122.
123.             # Put the blended region over the original frame
124.             frame[startY:endY, startX:endX][mask] = blended
125.
126.             # Draw the bounding box
127.             color = [int(c) for c in color]
128.             cv2.rectangle(frame, (startX, startY), (endX, endY), color, 8)
129.
130.             # Draw the class label and confidence
131.             text = "{}: {:.4f}".format(class_name, confidence)
132.             cv2.putText(frame, text, (startX, startY - 10),
133.                         cv2.FONT_HERSHEY_SIMPLEX, 1, color, 4)
134.
135.     # Get time after we finish processing this frame
136.     post_processing_times.append(time.time() * 1000)
137.     # Increase total number of frames
138.     num_frames += 1
139.
140. # Get time after we finish processing every frame
141. end_time = time.time() * 1000
142.
143. # Process data and output results
144. sum_network_time = 0
145. sum_results_processing_time = 0
146.
147. for i in range(num_frames):
148.     sum_network_time += post_network_times[i] - pre_network_times[i]
149.     sum_results_processing_time += \
150.         post_processing_times[i] - post_network_times[i]
151.
152. total_time = round((end_time - start_time) / 1000, 3)
153. avg_fps = round(num_frames / total_time, 3)
154. avg_network_time = round((sum_network_time / num_frames) / 1000, 3)
155. avg_result_processing_time = \
156.     round((sum_results_processing_time / num_frames) / 1000, 3)
157. avg_object_detections = round(total_object_detections / num_frames)
158.
159. print("Average fps: {}\n"
160.       "Average network processing time: {}\n"
161.       "Average result processing time: {}\n"
162.       "Average number of relevant object detections: {}\n"
163.       "Total number of relevant object detections: {}\n"
164.       "Number of frames: {}\n"
165.       "Total time: {}".format(avg_fps, avg_network_time,
166.                               avg_result_processing_time,
```

```
                                    avg_object_detections,
                                    total_object_detections,
                                    num_frames, total_time))

vs.release()
```

## IX. Teste do modelo *ENet* (*CPU* e *GPU*)

```
# Import necessary libraries
import time
import cv2
import numpy as np

# Classes that ENet was trained to detect
labels = ['unlabeled', 'road', 'sidewalk', 'building', 'wall', 'fence',
          'pole', 'traffic light', 'traffic sign', 'vegetation', 'terrain',
          'sky', 'person', 'rider', 'car', 'truck', 'bus', 'train',
          'motorcycle', 'bicycle']

# List of the colors to represent the classes
colors = [[0, 0, 0], [81, 0, 81], [244, 35, 232], [70, 70, 70],
          [102, 102, 156], [190, 153, 153], [153, 153, 153],
          [250, 170, 30], [220, 220, 0], [107, 142, 35],
          [152, 251, 152], [70, 130, 180], [220, 20, 60],
          [255, 0, 0], [0, 0, 142], [0, 0, 70], [0, 60, 100],
          [0, 80, 100], [0, 0, 230], [119, 11, 32]]
colors = np.array(colors, dtype="uint8")

# Load the ENet model from disk
net = cv2.dnn.readNet("./enet_model/enet-model.net")

# Enable gpu processing - To disable just remove these two lines
net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)

# Open video stream
vs = cv2.VideoCapture("./resources/traffic_video_sample.mkv")

# Auxiliary variables
num_frames = 0
total_object_detections = 0
pre_network_times = []
post_network_times = []
post_processing_times = []
warm = False

# Get starting time
start_time = time.time() * 1000

# Loop over frames
while True:
    # Get the next frame
    ret, frame = vs.read()

    # If the video is over we stop
    if not ret:
        break

    # The first time we process using the gpu takes a lot longer than after
    # it's been running for a while. As such we do a warmup before starting
    # the actual test
    if not warm:
        # Prepare the image
```

```
56.         img = cv2.resize(frame, (1024, 512))
57.         blob = cv2.dnn.blobFromImage(img, 1 / 255.0, (1024, 512), 0,
58.                                      swapRB=True, crop=False)
59.         # Set it as input to the neural network
60.         net.setInput(blob)
61.         # Run it through the network
62.         output = net.forward()
63.         # Only warmup once, before processing the first frame
64.         warm = True
65.
66.     # Get the time before running the image through the neural network
67.     pre_network_times.append(time.time() * 1000)
68.     img = cv2.resize(frame, (1024, 512))
69.     blob = cv2.dnn.blobFromImage(img, 1 / 255.0, (1024, 512), 0,
70.                                  swapRB=True, crop=False)
71.     net.setInput(blob)
72.     output = net.forward()
73.     # Get the time after we're done with the network part and before
74.     # processing the results
75.     post_network_times.append(time.time() * 1000)
76.
77.     (number_classes, height, width) = output.shape[1:4]
78.
79.     # Output class ID map is num_classes x height x width
80.     # Take the argmax to find the class label with the highest probability
81.     # for each pixel
82.     class_map = np.argmax(output[0], axis=0)
83.
84.     # Map each of the class IDs to its corresponding color
85.     mask = colors[class_map]
86.
87.     # Resize the mask to match the original size
88.     mask = cv2.resize(mask, (img.shape[1], img.shape[0]),
89.                       interpolation=cv2.INTER_NEAREST)
90.
91.     # Perform a weighted combination of the input frame with the mask
92.     img_od_masks = ((0.3 * img) + (0.7 * mask)).astype("uint8")
93.
94.     # Make sure that the processed image is taller than a set value so that
95.     # the text that we will write next is readable
96.     min_height = 300
97.     if frame.shape[0] < min_height:
98.         img_od_masks = cv2.resize(img_od_masks,
99.                         (int(min_height /
100.                           (frame.shape[0] / frame.shape[1])),
101.                          min_height))
102.     else:
103.         img_od_masks = cv2.resize(img_od_masks,
104.                             (frame.shape[1], frame.shape[0]))
105.
106.     # Initialize the legend visualization image with 25 pixels for each
    class
107.     num_classes = len(np.unique(class_map))
108.     row_height = 25
109.     legend = np.full(((number_classes * row_height), 150, 3), 33,
110.                      dtype="uint8")
111.
112.     # Get the detected class's names
113.     class_names = []
114.     for id in np.unique(class_map):
115.         class_names.append(labels[id])
116.
117.     # Draw the ENet class names + colors on the legend
118.     i = 0
119.     for label in sorted(class_names):
120.         id = labels.index(label)
121.         color = [int(c) for c in colors[id]]
122.         cv2.putText(legend, labels[id], (5, (i * row_height) + 17),
```

```
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (204, 204, 204), 1,
                    cv2.LINE_AA)
        cv2.rectangle(legend, (100, (i * row_height)),
                      (150, (i * row_height) + row_height), tuple(color), -
1)
        i += 1

    # Resize the legend to the height of the output image while keeping
    # its aspect ratio
    legend = cv2.resize(legend,
                    (int(legend.shape[1] /
                    legend.shape[0] * img_od_masks.shape[0]),
                    img_od_masks.shape[0]))
    # Join the output image and the legend side by side
    img_od_masks = np.concatenate((legend, img_od_masks), axis=1)
    # Add an alpha channel
    img_od_masks = np.c_[img_od_masks,
                         np.full((img_od_masks.shape[0],
                                  img_od_masks.shape[1], 1), 255)]

    # Get time after we finish processing this frame
    post_processing_times.append(time.time() * 1000)
    # Increase total number of frames
    num_frames += 1

# Get time after we finish processing every frame
end_time = time.time() * 1000

# Process data and output results
sum_network_time = 0
sum_results_processing_time = 0

for i in range(num_frames):
    sum_network_time += post_network_times[i] - pre_network_times[i]
    sum_results_processing_time += \
        post_processing_times[i] - post_network_times[i]

total_time = round((end_time - start_time) / 1000, 3)
avg_fps = round(num_frames / total_time, 3)
avg_network_time = round((sum_network_time / num_frames) / 1000, 3)
avg_result_processing_time = \
    round((sum_results_processing_time / num_frames) / 1000, 3)
avg_object_detections = round(total_object_detections / num_frames)

print("Average fps: {}\n"
      "Average network processing time: {}\n"
      "Average result processing time: {}\n"
      "Average number of relevant object detections: {}\n"
      "Total number of relevant object detections: {}\n"
      "Number of frames: {}\n"
      "Total time: {}".format(avg_fps, avg_network_time,
                              avg_result_processing_time,
                              avg_object_detections,
                              total_object_detections,
                              num_frames, total_time))

vs.release()
```

## X. Teste dos modelos *ENet + Mask R-CNN* (*CPU* e *GPU*)

```
 # Import necessary libraries
import time
import cv2
```

```
4. import numpy as np
5.
6. # Classes that ENet was trained to detect
7. enet_labels = ['unlabeled', 'road', 'sidewalk', 'building', 'wall', 'fence',
8.                'pole', 'traffic light', 'traffic sign', 'vegetation',
9.                'terrain', 'sky', 'person', 'rider', 'car', 'truck', 'bus',
10.               'train', 'motorcycle', 'bicycle']
11.
12. # List of the colors to represent the ENet classes
13. enet_colors = [[0, 0, 0], [81, 0, 81], [244, 35, 232], [70, 70, 70],
14.               [102, 102, 156], [190, 153, 153], [153, 153, 153],
15.               [250, 170, 30], [220, 220, 0], [107, 142, 35],
16.               [152, 251, 152], [70, 130, 180], [220, 20, 60],
17.               [255, 0, 0], [0, 0, 142], [0, 0, 70], [0, 60, 100],
18.               [0, 80, 100], [0, 0, 230], [119, 11, 32]]
19. enet_colors = np.array(enet_colors, dtype="uint8")
20.
21. # Load the ENet model from disk
22. enet_net = cv2.dnn.readNet("./enet_model/enet-model.net")
23.
24. # Enable gpu processing - To disable just remove these two lines
25. enet_net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
26. enet_net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)
27.
28. # Classes that Mask R-CNN was trained to detect
29. maskrcnn_labels = ['person', 'bicycle', 'car', 'motorcycle', 'airplane',
30.                    'bus', 'train', 'truck', 'boat', 'traffic light',
31.                    'fire hydrant', 'street sign', 'stop sign',
32.                    'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse',
33.                    'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe',
34.                    'hat', 'backpack', 'umbrella', 'shoe', 'eye glasses',
35.                    'handbag', 'tie', 'suitcase', 'frisbee', 'skis',
36.                    'snowboard', 'sports ball', 'kite', 'baseball bat',
37.                    'baseball glove', 'skateboard', 'surfboard',
38.                    'tennis racket', 'bottle', 'plate', 'wine glass', 'cup',
39.                    'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple',
40.                    'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog',
41.                    'pizza', 'donut', 'cake', 'chair', 'couch',
42.                    'potted plant', 'bed', 'mirror', 'dining table',
   'window',
43.                    'desk', 'toilet', 'door', 'tv', 'laptop', 'mouse',
44.                    'remote', 'keyboard', 'cell phone', 'microwave', 'oven',
45.                    'toaster', 'sink', 'refrigerator', 'blender', 'book',
46.                    'clock', 'vase', 'scissors', 'teddy bear', 'hair drier',
47.                    'toothbrush']
48.
49. # Load the Mask R-CNN model from disk
50. maskrcnn_net = cv2.dnn.\
51.     readNetFromTensorflow("./mask_rcnn_model/frozen_inference_graph.pb",
52.           "./mask_rcnn_model/mask_rcnn_inception_v2_coco_2018_01_28.pbtxt")
53.
54. # Constants
55. maskrcnn_confidence_threshold = 0.5
56. maskrcnn_mask_threshold = 0.7
57.
58. # Enable gpu processing - To disable just remove these two lines
59. maskrcnn_net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
60. maskrcnn_net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)
61.
62. # Open video stream
63. vs = cv2.VideoCapture("./resources/traffic_video_sample.mkv")
64.
65. # Auxiliary variables
66. num_frames = 0
67. total_object_detections = 0
68. pre_network_times = []
69. post_network_times = []
70. post_processing_times = []
```

```
warm = False

# Get starting time
start_time = time.time() * 1000

# Loop over frames
while True:
    # Get the next frame
    ret, frame = vs.read()

    # If the video is over we stop
    if not ret:
        break

    # The first time we process using the gpu takes a lot longer than after
    # it's been running for a while. As such we do a warmup before starting
    # the actual test
    if not warm:
        # ENet
        # Prepare the image
        min_height = 300
        if frame.shape[0] < min_height:
            img = cv2.resize(frame,
                      (int(min_height / (frame.shape[0] / frame.shape[1])),
                       min_height))
        else:
            img = frame
        blob = cv2.dnn.blobFromImage(img, 1 / 255.0, (1024, 512), 0,
                                     swapRB=True, crop=False)
          # Set it as input to the neural network
          enet_net.setInput(blob)
          # Run it through the network
          output = enet_net.forward()

          # Mask R-CNN
          blob = cv2.dnn.blobFromImage(frame, swapRB=True, crop=False)
          maskrcnn_net.setInput(blob)
          (boxes, masks) = maskrcnn_net.forward(["detection_out_final",
                                                 "detection_masks"])

          # Only warmup once, before processing the first frame
          warm = True

    # Get the time before running the image through the neural networks
    pre_network_times.append(time.time() * 1000)

    # Process it with ENet first
    min_height = 300
    if frame.shape[0] < min_height:
        img = cv2.resize(frame,
                  (int(min_height / (frame.shape[0] / frame.shape[1])),
                   min_height))
    else:
        img = frame

    blob = cv2.dnn.blobFromImage(img, 1 / 255.0, (1024, 512), 0,
swapRB=True,
                                 crop=False)
    # Set it as input to the neural network
    enet_net.setInput(blob)
    # Run it through the network
    output = enet_net.forward()

    # Process the original frame with Mask R-CNN to get the instance masks
    blob = cv2.dnn.blobFromImage(frame, swapRB=True, crop=False)
    maskrcnn_net.setInput(blob)
    (boxes, masks) = maskrcnn_net.forward(["detection_out_final",
                                           "detection_masks"])
```

```
138.
139.     # Get the time after we're done with the network part and before
140.     # processing the results
141.     post_network_times.append(time.time() * 1000)
142.
143.     # Output class ID map is num_classes x height x width
144.     # Take the argmax to find the class label with the highest probability
145.     # for each pixel
146.     class_map = np.argmax(output[0], axis=0)
147.
148.     # Get the list of classes that ENet detected so that we can build a
149.     # legend later
150.     enet_classes_detected = []
151.     for id in np.unique(class_map):
152.         if enet_labels[id] not in enet_classes_detected:
153.             enet_classes_detected.append(enet_labels[id])
154.
155.     # Map each of the class IDs to its corresponding color
156.     ss_mask = enet_colors[class_map]
157.
158.     # Resize the mask to match the original size
159.     ss_mask = cv2.resize(ss_mask, (frame.shape[1], frame.shape[0]),
160.                          interpolation=cv2.INTER_NEAREST)
161.
162.     # Perform a weighted combination of the input frame with the mask
163.     img_od_masks = ((0.3 * frame) + (0.7 * ss_mask)).astype("uint8")
164.
165.     # Loop over the Mask R-CNN detections
166.     maskrcnn_classes_detected = []
167.     for i in range(0, boxes.shape[2]):
168.         confidence = boxes[0, 0, i, 2]  # Get the confidence
169.
170.         # If this detection's confidence is higher than our set confidence
171.         # threshold then we consider it relevant and process it
172.         if confidence > maskrcnn_confidence_threshold:
173.             class_id = int(boxes[0, 0, i, 1])
174.             class_name = maskrcnn_labels[class_id]
175.             # Add to the list of classes detected that will be shown
176.             # in the legend
177.             if class_name not in maskrcnn_classes_detected:
178.                 maskrcnn_classes_detected.append(class_name)
179.
180.             # Scale the bounding box coordinates back, relative to the size
181.             # of the frame, and calculate the dimensions of the bounding
    box.
182.             (H, W) = frame.shape[:2]
183.             box = boxes[0, 0, i, 3:7] * np.array([W, H, W, H])
184.             (startX, startY, endX, endY) = box.astype("int")
185.             boxW = endX - startX
186.             boxH = endY - startY
187.
188.             # Get the segmentation masks needed to build the visualization
189.             # we want. Each object detected will have a semi-transparent
190.             # white overlay on top and will be distinguished from the
    others
191.             # with a white outline.
192.             border = 4  # Number of outline pixels
193.             mask = masks[i, class_id]
194.             # Get a mask the size of the detected object and one larger
195.             # according to the border value.
196.             # We will work with images and masks larger than the bounding
    box
197.             # by the number of border pixels so that we don't have to scale
198.             # down the original detection segmentation.
199.             mask_small = cv2.resize(mask, (boxW, boxH),
200.                                     interpolation=cv2.INTER_CUBIC)
201.             mask_small = (mask_small > maskrcnn_mask_threshold)
202.             mask_big = cv2.resize(mask,
```

```
                                    (boxW + border * 2, boxH + border * 2),
                                    interpolation=cv2.INTER_CUBIC)
            mask_big = (mask_big > maskrcnn_mask_threshold)

            # Create an all black image with the size of the bigger mask
            # and paint the masked pixels white
            aux = np.zeros((boxH + border * 2, boxW + border * 2, 4),
                           dtype="uint8")
            aux[mask_big != 0] = [255, 255, 255, 255]

            # Get the region of the original frame corresponding to this
            # detection's bounding box
            roi = frame[startY:endY, startX:endX, :]
            # Add an alpha channel so that it matches the masks
            roi = cv2.cvtColor(roi, cv2.COLOR_BGR2BGRA)
            # Paint it black outside the masked pixels so that we only have
            # the area under the mask
            roi[mask_small == 0] = [0, 0, 0, 0]

            # Add padding to the region so that its dimensions match the
            # original frame without stretching
            vertical = int((aux.shape[0] - roi.shape[0]) / 2)
            horizontal = int((aux.shape[1] - roi.shape[1]) / 2)
            roi = cv2.copyMakeBorder(src=roi, top=vertical,
bottom=vertical,
                                     left=horizontal, right=horizontal,
                                     borderType=cv2.BORDER_CONSTANT,
                                     value=[0, 0, 0, 0])

            # Now we have an image with the bigger mask pixels all white
and
            # an image with the smaller mask pixels with the original frame
            # colors.
            # Both images have the same dimensions and are slightly larger
            # than the bounding box.
            # Get the mask of the region where we have the original frame
            # colors. We do this so that we have a mask with the same
            # dimensions of the resized roi image
            m = roi[:, :, 3] > 0
            # Paint that region black on the image with the all white
            # segmentation so that we get only the outline
            aux[m] = [0, 0, 0, 0]
            # Resize it back to the dimensions of the original bounding box
            # This way we will overlay the outline over the original
detection
            # segmentation, keeping the original dimensions and without
ever
            # scaling down the "inside" of the detection segmentation
            aux = cv2.resize(aux, (boxW, boxH),
interpolation=cv2.INTER_CUBIC)
            # Get the mask of the final outline image
            m = aux[:, :, 3] > 0

            # Add a channel to the output image so that it matches the
masks
            # and overlays
            img_od_masks = cv2.cvtColor(img_od_masks, cv2.COLOR_BGR2BGRA)
            # Get the region of the original image that corresponds to the
            # detection segmentation
            region = img_od_masks[startY:endY, startX:endX][mask_small]

            # Blend white with the region under the mask
            color = np.array([[255, 255, 255, 255]])
            blended = ((0.2 * color) + (0.8 * region)).astype("uint8")
            # Copy the detection segmentation and the outline to the output
            # image
            img_od_masks[startY:endY, startX:endX, :][mask_small] = blended
            img_od_masks[startY:endY, startX:endX, :][m] = aux[m]
```

```
            # Write class label and confidence
            text = "{0}: {1}%".format(class_name, round(confidence * 100,
1))
            # Find out the ideal font size and thickness
            desired_text_height = img_od_masks.shape[0] * 0.02
            font_size = 0.1
            while cv2.getTextSize(text, cv2.FONT_HERSHEY_SIMPLEX,
font_size,
                                  max(int(font_size * 3), 1))[0][1] <
desired_text_height:
                font_size += 0.1
            font_size = round(font_size, 1)
            thickness = max(int(font_size * 3), 1)
            # Find out the final text size, draw a rectangle behind it to
make
            # sure that it is always readable and then write it
            text_size = cv2.getTextSize(text, cv2.FONT_HERSHEY_SIMPLEX,
                                        font_size, thickness)
            text_w = text_size[0][0]
            text_h = text_size[0][1]
            text_x = int(startX - (text_w - (endX - startX)) / 2)
            text_y = int(startY - 5)
            cv2.rectangle(img_od_masks, (text_x, text_y - text_h),
                          (text_x + text_w, text_y), (50, 50, 50, 255), -1)
            cv2.putText(img_od_masks, text, (text_x, text_y),
                        cv2.FONT_HERSHEY_SIMPLEX, font_size,
                        (255, 255, 255, 255), thickness, cv2.LINE_AA)

    # Initialize the legend visualization image with 25 pixels for each
class
    row_height = 25
    legend = np.full(((len(enet_classes_detected) * row_height), 150, 3),
33,
                     dtype="uint8")
    legend = np.c_[legend, np.full((legend.shape[0], legend.shape[1], 1),
                                   255)].astype("uint8")

    # Draw the ENet class names + colors on the legend
    i = 0
    for label in sorted(enet_classes_detected):
        id = enet_labels.index(label)
        color = [int(c) for c in enet_colors[id]]
        color.append(255)
        cv2.putText(legend, label, (5, (i * row_height) + 17),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (204, 204, 204), 1,
                    cv2.LINE_AA)
        cv2.rectangle(legend, (100, (i * row_height)),
                      (150, (i * row_height) + row_height), tuple(color), -
1)
        i += 1

    # Resize the legend to the height of the output image while keeping
    # its aspect ratio
    legend = cv2.resize(legend,
                 (int(img_od_masks.shape[0] /
                 (legend.shape[0] / legend.shape[1])),
                 img_od_masks.shape[0]))

    # Add an alpha channel if it's missing
    if img_od_masks.shape[2] == 3:
        img_od_masks = np.c_[img_od_masks,
                             np.full((img_od_masks.shape[0],
                                      img_od_masks.shape[1], 1), 255)]

    # Join the output image and the legend side by side
    img_od_masks = np.concatenate((legend, img_od_masks), axis=1)

    # Get time after we finish processing this frame
```

```
326.     post_processing_times.append(time.time() * 1000)
327.     # Increase total number of frames
328.     num_frames += 1
329.
330. # Get time after we finish processing every frame
331. end_time = time.time() * 1000
332.
333. # Process data and output results
334. sum_network_time = 0
335. sum_results_processing_time = 0
336.
337. for i in range(num_frames):
338.     sum_network_time += post_network_times[i] - pre_network_times[i]
339.     sum_results_processing_time += \
340.         post_processing_times[i] - post_network_times[i]
341.
342. total_time = round((end_time - start_time) / 1000, 3)
343. avg_fps = round(num_frames / total_time, 3)
344. avg_network_time = round((sum_network_time / num_frames) / 1000, 3)
345. avg_result_processing_time = \
346.     round((sum_results_processing_time / num_frames) / 1000, 3)
347. avg_object_detections = round(total_object_detections / num_frames)
348.
349. print("Average fps: {}\n"
350.       "Average network processing time: {}\n"
351.       "Average result processing time: {}\n"
352.       "Average number of relevant object detections: {}\n"
353.       "Total number of relevant object detections: {}\n"
354.       "Number of frames: {}\n"
355.       "Total time: {}".format(avg_fps, avg_network_time,
356.                               avg_result_processing_time,
357.                               avg_object_detections,
358.                               total_object_detections,
359.                               num_frames, total_time))
360.
361. vs.release()
```

# Anexo B: Questionários utilizados nos testes com utilizadores

## I. *SUS* (*System Usability Scale*)

***System Usability Scale***



| | Strongly disagree | | | | Strongly agree |
|---|---|---|---|---|---|
| 1. I think that I would like to use this system frequently | 1 | 2 | 3 | 4 | 5 |
| 2. I found the system unnecessarily complex | 1 | 2 | 3 | 4 | 5 |
| 3. I thought the system was easy to use | 1 | 2 | 3 | 4 | 5 |
| 4. I think that I would need the support of a technical person to be able to use this system | 1 | 2 | 3 | 4 | 5 |
| 5. I found the various functions in this system were well integrated | 1 | 2 | 3 | 4 | 5 |
| 6. I thought there was too much inconsistency in this system | 1 | 2 | 3 | 4 | 5 |
| 7. I would imagine that most people would learn to use this system very quickly | 1 | 2 | 3 | 4 | 5 |
| 8. I found the system very cumbersome to use | 1 | 2 | 3 | 4 | 5 |
| 9. I felt very confident using the system | 1 | 2 | 3 | 4 | 5 |
| 10. I needed to learn a lot of things before I could get going with this system | 1 | 2 | 3 | 4 | 5 |

# II. Questionário acerca das preferências pessoais

1. Nome *

2. Idade *

3. Género *

*Mark only one oval.*

- ◯ Masculino
- ◯ Feminino
- ◯ Outro

4. Classifique a utilidade dos métodos de visualização para visualizar os resultados obtidos. *

*Mark only one oval per row.*

| | 1 (pouco útil) | 2 | 3 | 4 | 5 | 6 | 7 (muito útil) |
|---|---|---|---|---|---|---|---|
| Grelha Normal | ◯ | ◯ | ◯ | ◯ | ◯ | ◯ | ◯ |
| Grelha Variável | ◯ | ◯ | ◯ | ◯ | ◯ | ◯ | ◯ |
| Pilha | ◯ | ◯ | ◯ | ◯ | ◯ | ◯ | ◯ |
| Espiral | ◯ | ◯ | ◯ | ◯ | ◯ | ◯ | ◯ |

5. Ordene os métodos de visualização de acordo com a sua preferência. (Grelha Normal - A; Grelha Variável - B; Pilha - C; Espiral - D) *

6. Identifique o(s) aspeto(s) mais importantes que o levaram a determinar a ordem acima. *