Trabalho de Conclusão de Curso

Uma ferramenta para demarcação de áreas de interesse e monitoramento em tempo real de pessoas usando Aprendizado Profundo

David Silva Alexandre
dsa@ic.ufal.br

Orientadores:

Thales Miranda de Almeida Vieira
Eduarda Tatiane Caetano Chagas

Maceió, Agosto de 2022

David Silva Alexandre

Uma ferramenta para demarcação de áreas de interesse e monitoramento em tempo real de pessoas usando Aprendizado Profundo

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:

Thales Miranda de Almeida Vieira

Eduarda Tatiane Caetano Chagas

Maceió, Agosto de 2022

Catalogação na Fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico

Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

A381f Alexandre, David Silva.

Uma ferramenta para demarcação de áreas de interesse e monitoramento em tempo real de pessoas usando aprendizado profundo / David Silva Alexandre. – 2022.

62 f. : il.

Orientador: Thales Miranda de Almeida Vieira.

Co-orientadora: Eduarda Tatiane Caetano Chagas.

Monografia (Trabalho de conclusão de curso em Ciência da Computação) – Universidade Federal de Alagoas, Instituto de Computação. Maceió, 2022.

Bibliografia: f. 59-62.

1. Detecção de objetos (Aprendizado profundo). 2. Processamento eletrônico de dados em tempo real. 3. Python (Linguagem de programação de computador). I. Título.

CDU: 004.4’41

Agradecimentos

Primeiramente, eu gostaria de agradecer a toda a minha família, em especial a minha mãe, minha avó, minhas tias e meus QUATRO irmãos, por me concederem todo tipo de suporte, seja financeiro ou emocional. Sem seus sacrifícios eu jamais teria concluído essa graduação.

Também gostaria de agradecer a minha amiga Maria por todo amor e confiança depositados em mim nessa jornada, a minha amiga e parceira Bruna Damaris por sua singularidade, confiança e coragem, acima de tudo por todos os momentos que vivemos juntos durante essa caminhada, vou carregar essas memórias comigo até o dia em que eu deixar de existir. Agradeço também a minha amiga Eduarda Chagas, que é um dos maiores exemplos de perfeição que eu conheço, não sou capaz de expressar em palavras o quão importante você é pra mim Duda. Não posso deixar de agradecer ao meu amigo Samuel, por ser uma pessoa incrível e de um coração lindo. Aos meus amigos Thiago Cabral e José Silvino Neto, sou extremamente grato pelo amigo incrível que você é e por cada conversa maluca que tivemos Cabral, e muito obrigado por me passar em Estrutura de Dados Neto. Meu muito obrigado também para o meu amigo Marco Aurélio, por me acolher quando mais precisei. Obrigado Bruno =D.

Ao meu orientador Prof. Dr. Thales Vieira, o qual eu admiro muito e que com sua ótima didática foi um dos catalisadores da minha paixão por Visão Computacional.

À todos os funcionários do Instituto de Computação, em especial a Ana Ferreira, Lúcia e Marcelo Gusmão, muito obrigado por sempre estarem dispostos a me ajudar e a me atender muito bem, vocês são incríveis.

Por fim, agradeço à banca examinadora, pela leitura atenta, questionamentos e sugestões.

"... tudo está se transformando continuamente, não há nada fixo, nem permanente, pois em tudo há uma dependência de causas e condições.'

– SUTRA DO CORAÇÃO DA GRANDE SABEDORIA COMPLETA

David Alexandre

“A disciplina surge, ao fazer o que se tem que fazer, da maneira correta a fazer.”
– Monja Coen Rōshi

Resumo

Identificar pessoas é um problema relevante da área de Detecção de Objetos. Mais recentemente, surgiram técnicas para realizar essas detecções em tempo real, utilizando Redes neurais e Aprendizado profundo. Essas técnicas têm como grande vantagem a rapidez e a precisão com que conseguem classificar seres humanos em uma cena, o que são propriedades indispensáveis em aplicações de monitoramento por vídeo. Soluções baseadas em Aprendizado Profundo possuem uma ampla gama de aplicações. Desde que as Redes Neurais começaram a ser aplicadas com sucesso na área de Aprendizado Supervisionado, abordagens que fazem seu uso estão alcançando resultados a nível de estado da arte na solução de problemas de classificação(como reconhecimento de fala, sensoriamento remoto, dentre outros). Essas tecnologias podem ser combinadas com outras técnicas para resolver problemas geométricos do mundo real, como por exemplo monitorar a localização de um usuário usando óculos de realidade virtual. Nesse trabalho, relatamos o processo de desenvolvimento de uma ferramenta que detecta pessoas e infere se elas estão em uma área demarcada ou não. Esse programa visa possibilitar o monitoramento em tempo real de regiões customizadas pelo usuário, podendo ser utilizado para diversas finalidades. O sistema foi implementado na linguagem de programação *Python* que, além de fornecer praticidade, também possui uma vasta usabilidade de bibliotecas, métodos e funções muito utilizadas atualmente na área de Visão Computacional. Também utilizamos um modelo pré-treinado da Yolov5 para detectar pessoas com mais confiança e rapidez.

Palavras-chave: Detecção de Objetos; Aplicação em Tempo Real; Linguagem *Python*.

Abstract

The identification of people is a relevant problem in the area of Object Detection. More recently, techniques have emerged to perform these detections in real time, using Neural Networks and Deep Learning. These techniques have the great advantage of being able to quickly and accurately classify persons in a scene, which are indispensable properties in video monitoring applications. Deep Learning based solutions have a wide range of applications. Since Neural Networks began to be successfully applied in the area of Supervised Learning, approaches that make use of them are achieving state-of-the-art results in solving classification problems (such as speech recognition, remote sensing, among others). These technologies can be combined with other techniques to solve real-world geometric problems, such as tracking a user's location using virtual reality glasses. In this work, we report the process of developing a tool that detects people and infers whether they are in a demarcated area or not. This program aims to enable real-time monitoring of regions customized by the user, and can be used for various purposes. The system was implemented in the *Python* programming language, which provide practicality and also has a vast usability of libraries, methods and functions that are currently used in the area of Computer Vision. We also use a pre-trained Yolov5 model to detect people more confidently and quickly.

Keywords: Object Detection; Real Time Application; *Python* language.

Conteúdo

Lista de Figuras

1

Introdução

1.1 Motivação

O monitoramento de ambientes internos e externos usando câmeras de segurança é realizado durante horas todos os dias pelos mais diversos profissionais espalhados pelo globo. No mundo real, um ser humano geralmente precisa patrulhar a área e checar constantemente as imagens para inferir se há alguma anormalidade, o que é uma tarefa que demanda tempo e esforço. Entretanto, uma área de interesse pode apresentar diversos objetivos distintos a depender da problemática envolvida. Podemos ter em mente por exemplo, delimitar uma região da imagem na qual não desejamos que pessoa alguma se aproxime por questões de segurança corporativa, ou até mesmo, delimitar uma área de risco para a vida de um ser humano, como buracos ou demais situações que comprometeriam sua segurança pessoal.

A literatura apresenta certos trabalhos relacionados, alguns mais antigos, que remontam ao reconhecimento de pedestres (Szarvas et al., 2005) e outros mais modernos, que usam técnicas em constante evolução no estado da arte para garantir a segurança dos mesmos, como é o caso de (Khanna and Awasthi) e (Dominguez-Sanchez et al., 2017). Também temos trabalhos recentes onde a problemática envolve a distância entre pessoas(que surgiu devido a pandemia de COVID-19) (Darapaneni et al., 2022). No entanto, nenhuma das abordagens comentadas foca diretamente na definição de áreas de interesse.

Definir essas áreas pode ser uma tarefa complexa para um computador realizar automaticamente, pois existem inúmeras preocupações e objetivos por trás dessa definição. Por esse motivo, em nosso trabalho essa tarefa é realizada pelo usuário. Detectar objetos em uma cena também não é uma tarefa simples, mas abordagens mais modernas conseguem realizá-la com uma confiança aceitável, e algumas são capazes de fazer isso em tempo real usando hardware de custo relativamente baixo, como é o caso da Yolo (Redmon et al., 2016).

Desse modo, existem dois principais pontos nessas linhas de pesquisa que podem originar ótimos trabalhos inovadores:

- A necessidade de garantir a segurança de pessoas enquanto elas andam por ruas ou avenidas
- O uso de técnicas modernas para resolver problemas de Geometria Computacional

O primeiro ponto pode ser solucionado por meio do desenvolvimento de sistemas baseados em Redes Neurais Convolucionais e outras técnicas como foi mostrado anteriormente. Enquanto o segundo, consiste em utilizar essas abordagens combinadas com técnicas a fim de resolver problemas de Geometria Computacional.

Portanto, este trabalho agrega componentes de Aprendizado Profundo e Geometria Computacional para o desenvolvimento de uma ferramenta capaz de realizar monitoramento inteligente de pessoas.

Apresentamos, assim, o desenvolvimento de uma ferramenta, rápida e de boa qualidade, que possibilita inferir se uma pessoa cruzou ou não uma linha Poligonal previamente definida. Seu uso é capaz de fornecer ao usuário uma forma rápida e automática de monitoramento de áreas de interesse definidas por ele em tempo real.

1.2 Objetivo

O objetivo geral deste trabalho é propor e desenvolver uma ferramenta a ser executada em tempo real, para inferir se uma pessoa adentrou ou não uma região previamente definida. Essa região deve ser definida pelo usuário e o programa emite um alerta sonoro quando o espaço é invadido.

1.3 Solução proposta

Realizamos o uso de uma técnica moderna de Detecção de Objetos, para identificar pessoas em tempo real. Essa abordagem é a **Yolov5**, que é uma extensão da **Yolov4** (Bochkovskiy et al., 2020) e que utiliza o *Pytorch framework*. Enquanto o modelo detecta pessoas em tempo real, o usuário define uma região de interesse customizada usando uma interface gráfica interativa. Caso o Software detecte pessoas em uma região de interesse estabelecida, um alarme sonoro é executado. A ferramenta também pode ser utilizada em vídeos já gravados como mostrado em capítulos futuros deste trabalho.

Com o mouse, o usuário define no vídeo gerado por uma câmera a área que deseja monitorar, em seguida, o programa calcula o ponto médio inferior da caixa delimitadora gerada pela Yolo, este ponto é próximo ao pés das pessoas que aparecem no vídeo. Por fim, implementamos uma função para calcular se este ponto está dentro da área que demarcamos e em caso positivo um sinal sonoro é disparado

1.4 Contribuições

As contribuições deste trabalho são:

- A compreensão e implementação de algumas técnicas utilizadas em Geometria Computacional;
- A compreensão e uso de um dos principais métodos de Detecção de Objetos da literatura;
- A implementação de uma ferramenta que detecta pessoas e é usada para delimitar áreas de interesse de acordo com as preferências do usuário da aplicação.

Essas contribuições podem facilitar o processo de análise de vídeos pelo usuário, tornando tal tarefa automática.

1.5 Estrutura do texto

Este trabalho foi dividido em 5 capítulos e dois anexos. No capítulo 2 introduzimos algumas das principais técnicas e ferramentas disponíveis na literatura para Detecção de Objetos, além disso, também são descritos conceitos fundamentais de Aprendizado de Máquina e de Geometria Computacional. No capítulo 3 apresentamos a metodologia do trabalho desenvolvido. No capítulo 4 mostramos os resultados obtidos durante a realização das atividades. As funções implementadas ao longo do desenvolvimento do projeto, assim como uma breve explicação dos métodos que pertencem a Bibliotecas da linguagem *Python* utilizadas e um tutorial de como instalar os componentes necessários para utilizar o software, estão presentes no Anexo A, enquanto no Anexo B são descritos detalhes do uso da ferramenta para execução dos experimentos qualitativos. O Capítulo 5 exibe exemplos de uso da ferramenta. Por fim, no Capítulo 6 apresentamos as considerações finais do trabalho.

2

Fundamentação Teórica

Para que o trabalho proposto seja melhor compreendido, serão apresentadas as fundamentações teóricas neste capítulo. As informações a seguir foram obtidas por meio de revisão bibliográfica, abrangendo os principais conceitos e técnicas presentes na literatura.

2.1 Aprendizado de Máquina

A definição de Aprendizado de Máquina pode variar um pouco de acordo com o autor. Uma de suas definições mais completas considera a Aprendizagem de Máquina como um conjunto de métodos que podem detectar automaticamente padrões nos dados e, em seguida, usar os padrões descobertos para prever dados futuros ou realizar outros tipos de tomada de decisão (Murphy, 2012). De uma forma geral, esta área estuda algoritmos capazes de melhorar seu desempenho conforme aprendem através da experiência (Mitchell and Mitchell, 1997).

2.1.1 Fundamentos de Aprendizado de Máquina

Conceitos importantes norteiam essa área e são indispensáveis para o entendimento do trabalho em questão. O primeiro diz respeito aos paradigmas de aprendizado de máquina presentes na literatura. Esses paradigmas tem como objetivo tornar o "aprendizado"o mais adequado possível, a depender do tipo de problema, e consequentemente aumentar a eficiência dos algoritmos utilizados para resolvê-los. Especificamente para área de Processamento de Imagens, os dois paradigmas principais e que são amplamente utilizados no estado da arte(O'Shea and Nash, 2015), são:

(a) **Aprendizado supervisionado:** é a aprendizagem através de entradas pré-rotuladas, que atuam como alvos. Para cada exemplo de treinamento temos um conjunto de valores de entrada e um ou mais valores de saída associados. O objetivo desta forma de treinamento é reduzir o erro geral de classificação dos modelos, ou em outras palavras, aprender uma função que mapeia uma entrada para uma saída com base em pares de entrada-saída de exemplo.

(b) **Aprendizado não-supervisionado:** o conjunto de treinamento não inclui nenhum rótulo. Note que, como os dados não estão rotulados, o algoritmo deve encontrar determinados padrões nos dados. O sucesso geralmente é determinado pela capacidade do Modelo de reduzir ou aumentar uma função de custo associada.

Atualmente a maioria das tarefas de reconhecimento de padrões focadas em imagens utiliza a abordagem de Aprendizado Supervisionado, uma vez que a maioria dos métodos presentes na literatura, tratam essa questão como um problema de Classificação (apesar de algumas abordagens tratarem como um problema de Regressão).

A **Classificação** é uma das categorias de problemas mais importantes da área em questão, e o objetivo dos algoritmos para problemas deste tipo, é aprender uma regra geral que mapeie as entradas nas saídas corretamente. Para isso, os dados de entrada podem ser organizados como pares (x, y) onde $x \in X$ são os atributos a serem utilizados na determinação da classe de saída e $y \in Y$, a classe de saída(o atributo para o qual se deseja fazer a predição do valor da classe), sendo que em problemas de Classificação, Y é sempre categórico (Soofi and Awan, 2017). Um problema de Classificação pode ser definido (informalmente) como a busca por uma função matemática que permita associar corretamente cada exemplo X_i de um conjunto de dados a um único rótulo categórico, Y_i, denominado classe. Esta função, uma vez identificada, poderá ser aplicada a novos dados para prever suas respectivas classes.

O fluxo resumido de um problema de Classificação se inicia gerando-se dois subconjuntos disjuntos a partir de um conjunto de treinamento rotulado(para cada exemplo, conhecemos a sua respectiva classe). Esses dois conjuntos são o de treino(que é usado para treinar o modelo) e o de teste(geralmente é usado para comprovar que aquele algoritmo realmente funciona). Em seguida, é realizado o treinamento do modelo, onde o conjunto de treino é submetida ao modelo(classificador) para que seus parâmetros sejam calibrados de acordo com os dados apresentados. Após esta etapa, ocorre o estágio de predição de classes, na qual os exemplos da base de teste são apresentados para o modelo treinado para que este realize as predições.

Já a **Regressão** consiste em realizar aprendizado supervisionado a partir de dados numéricos, o que é similar a abordagem explicada anteriormente. A principal diferença entre essas duas técnicas, é a saída do Algoritmo: na Classificação, o resultado é categórico(1 ou 0, sim ou não, classe A ou classe B), enquanto na Regressão, o resultado é numérico(contínuo ou discreto).

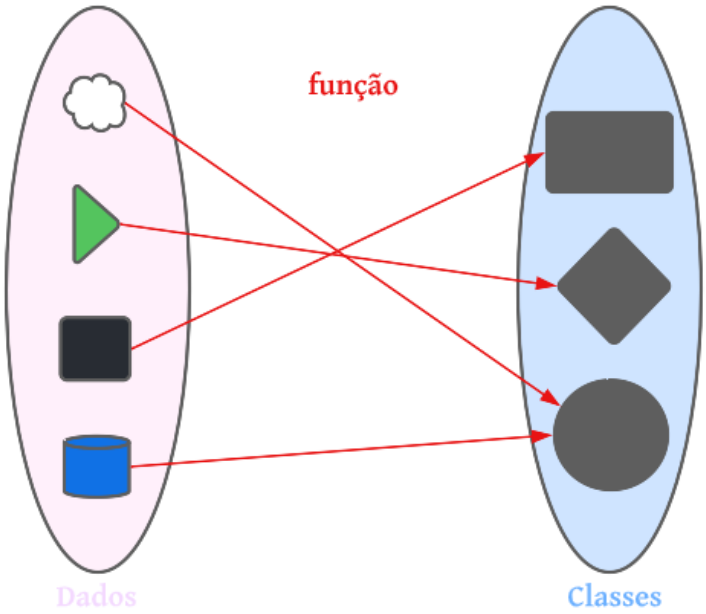

Figura 2.1: Exemplo de Classificação.

Existem algumas técnicas para avaliar a eficácia dos algoritmos de Aprendizado de Máquina, uma delas é a **Matriz de Confusão**, que oferece um detalhamento do desempenho do modelo de classificação, mostrando, para cada classe, o número de classificações corretas em relação ao número de classificações preditas pelo modelo. Com essa abordagem, visualizamos o número de **Falsos Positivo**(quando o resultado esperado é negativo, mas o modelo resulta em positivo), **Falsos Negativo**(quando o resultado esperado é positivo, mas o modelo resulta em negativo), **Verdadeiros Positivo**(quando o resultado esperado é positivo e o modelo resulta em positivo) e **Verdadeiros Negativo**(quando o resultado esperado é negativo e o modelo resulta em negativo).

Ao obtermos as informações descritas acima, podemos definir **Métricas de Avaliação** para o nosso modelo, sendo elas:

- **Acurácia:** indica uma performance geral do modelo, ou seja, dentre todas as classificações, quantas o modelo classificou corretamente.
- **Precisão:** dentre todas as classificações de classe Positivo que o modelo fez, quantas estão corretas.
- **Recall:** dentre todas as situações de classe Positivo como valor esperado, quantas estão corretas.
- **F1-Score:** é média harmônica entre a precisão e o recall.

2.1.2 Rede Neural Artificial

Uma rede neural artificial é um conjunto interconectado de elementos de processamento simples, conhecidos como unidades ou nós, cuja funcionalidade é baseada no neurônio animal. A capacidade de processamento da rede é armazenada nas forças de conexão entre unidades, que são conhecidas como pesos, obtidos através de um processo de adaptação ou aprendizado de um conjunto de padrões de treinamento (Gurney, 2018).

Como dito anteriormente, os nós da rede são baseados em neurônios biológicos, e cada sinapse é modelada como sendo um único número(conhecido como peso)(Müller et al., 1995). Na Figura 2.2, temos o exemplo de funcionamento de um único neurônio. Cada entrada é multiplicada por um determinado peso antes de ser enviada para o equivalente do corpo celular. Os sinais ponderados são somados a fim de fornecer uma ativação do nó. A função de ativação(soma), é comparada com um limiar; se a soma ultrapassar este valor, a unidade produz uma saída de alto valor(neste caso "1"), caso contrário, ela produz zero. O tamanho dos sinais é representado pela largura de suas setas correspondentes, os pesos são representados por símbolos de multiplicação em círculos e seus valores devem ser proporcionais ao tamanho do símbolo(considere somente pesos positivos para este exemplo). Esse modelo é conhecido como TLU(*Threshold Logic Unit*), e é o modelo mais simples(e historicamente o mais antigo de um neurônio artificial)(McCulloch and Pitts, 1943).

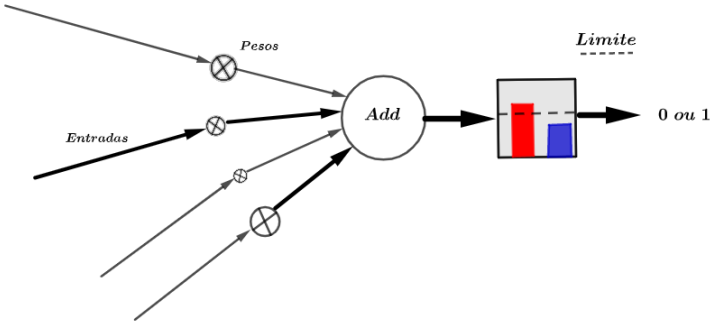

Figura 2.2: Neurônio Artificial Simples.

Matematicamente, este modelo pode ser descrito como um neurônio N, que recebe k sinais de entrada, $x \in \mathbb{R}^k$, possui um vetor com pesos w, um limite $t \in \mathbb{R}$ e uma função de ativação $\alpha : \mathbb{R} \to \mathbb{R}$. Sendo a entrada

$$N_{in} : \mathbb{R}^k \times \mathbb{R}^k \times \mathbb{R} \to \mathbb{R} \tag{2.1}$$

$$N_{in}(x, w, l) = x.w + l \tag{2.2}$$

e a saída

$$\alpha(x): \begin{cases} 1, & se \quad x \geqslant 0 \\ 0, & se \quad x < 0 \end{cases}$$

que apenas indica se o Neurônio foi ativado ou não.

Na Figura 2.3 temos um outro exemplo de Rede Neural Artificial. Cada nó é representado por um círculo, os pesos estão implícitos em todas as conexões. As unidades são organizadas em três estruturas, que são conhecidas como camadas. Perceba que no exemplo, cada sinal na camada de entrada emana para outros nós e passa por dois últimos nós, para então chegar a uma saída além da qual não é mais transformado. Essa estrutura (conhecida como *feedforward*) é apenas uma das várias disponíveis e normalmente é usada para colocar um padrão de entrada em uma das várias classes de acordo com o padrão de saídas resultante(utilizando o paradigma de aprendizagem supervisionado). Por exemplo, se a entrada consiste em uma codificação dos padrões de claro e escuro em uma imagem de letras manuscritas, a camada de saída pode conter 26 nós - um para cada letra do alfabeto - para sinalizar a qual classe o caractere de entrada corresponde. Isso seria feito alocando um nó de saída por classe e exigindo que apenas um desses nós fosse acionado sempre que um padrão correspondente fosse fornecido na entrada. Perceba que, dessa forma a rede ganha ênfase em aprender com a experiência.

Neurônios reais, podem, sob certas circunstâncias, modificarem suas forças sinápticas de modo que o comportamento de cada neurônio possa mudar ou se adaptar à sua entrada de estímulo particular. No caso dos neurônios artificiais, o equivalente a isso é a modificação dos valores dos pesos, que é realizada no processo de treinamento da rede. Em termos de processamento de informações, o "conhecimento"que a rede neural possui deve ser armazenado em seus pesos, que evoluem por um processo de adaptação ao estímulo de um conjunto de exemplos de padrões. Considerando nosso exemplo anterior de reconhecimento de letras, um "A", pode ser a entrada e a saída da rede comparada com o código de classificação para "A". Daí, o que vai definir como os pesos são alterados é a diferença entre os dois padrões de saída. Quando as atualizações de peso necessárias forem feitas, outro padrão é apresentado, a saída comparada com o destino e novas alterações feitas. Essa sequência de eventos é repetida iterativamente muitas vezes até que o comportamento da rede convirja de modo que sua resposta para cada padrão seja a mais próxima possível(Gurney, 2018).

Atualmente, temos milhares de aplicações para Redes Neurais, pois estas são capazes de adaptar-se ao meio que estão inseridas, aprendendo a realizar uma tarefa através de ajuste dos pesos de suas sinapses, de acordo com as amostras (Haykin, 2009). Geralmente os pesos são ajustados por um algoritmo de otimização, que comumente utiliza técnicas de retropropagação.

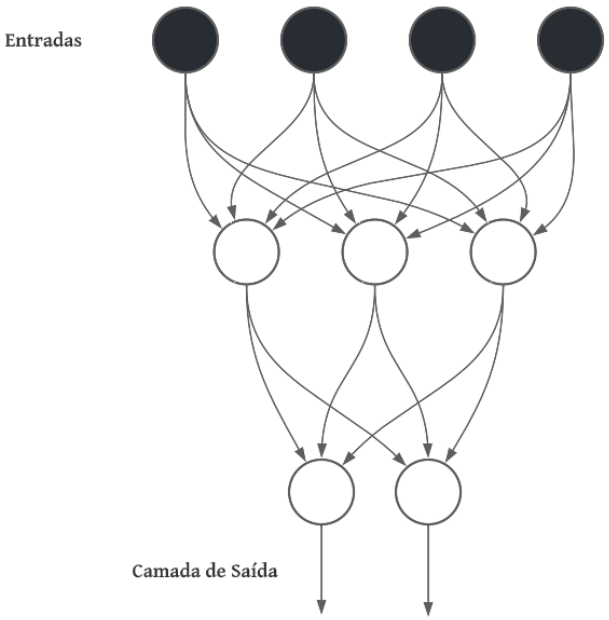

Figura 2.3: Exemplo de Rede Neural com mais de um Neurônio.

2.1.3 Treino de Redes Neurais

Em uma rede neural sem retroalimentação os neurônios não formam ciclos, portanto, podemos considerar a rede como um grafo conexo, acíclico e dirigido, onde os dados fluem nó a nó em uma direção única, e no fim fornecem uma saída. Para treinar a rede, devemos ajustar seus pesos e limites. Inicialmente os pesos estarão desajustados e a saída obtida irá diferir da saída esperada, mas somos capazes de mudar isto minimizando uma função de perda que nos dará novos resultados. Esses novos pesos são descobertos através da propagação de erros em sentido contrário ao fluxo do grafo em questão (Rumelhart et al., 1986).

Suponha então que queremos treinar uma rede N. No começo do treinamento os pesos w e o limiar t de N são escolhidos aleatoriamente e conforme o treinamento ocorre, os algoritmos do método do gradiente e retropropagação acham os valores mais adequados para que as saídas fornecidas sejam as mais corretas. Como mencionado anteriormente, isso é feito através da minimização de uma função de perda, basicamente, o método é capaz de encontrar os pesos ótimos w e t, caminhando iterativamente na direção do sentido oposto ao vetor gradiente $\nabla_w \text{loss}(y, \hat{y})$.

$$l : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R} \tag{2.3}$$

$$\nabla_w \text{loss}(\hat{y}, y) = \sum l(\hat{y}_i, y_i) \tag{2.4}$$

Para construir o vetor gradiente $\nabla_w \text{loss}(\hat{y}, y)$ precisamos de cada uma de suas derivadas

parciais, e é aí que o algoritmo de retropropagação entra. Esta técnica realiza a diferenciação automática para encontrar as derivadas parciais que constituem o vetor em questão. As condições de parada do algoritmo de retropropagação dependem da função de perda convergir ou da conclusão de uma determinada quantidade de épocas.

Como o método do gradiente original utiliza todas as amostras disponíveis sequencialmente em cada uma de suas iterações(e isso não é muito eficiente), algumas alterações foram feitas para melhorar sua performance. O estado da arte contém uma série de variações, uma delas é o método do gradiente estocástico. Esta operação realiza uma aproximação estocástica, onde as amostras de entrada são embaralhadas para cada época e as atualizações do vetor gradiente são feitas para uma única amostra (Klein et al., 2009).

Certos modelos definem formas de atualizar a taxa de aprendizado junto das iterações do método do gradiente. Com uma taxa de aprendizado dinâmica, pode-se ajustar melhor o algoritmo a fim de lidar com superfícies não convexas, evitando pontos de mínimo locais e uma convergência lenta do programa. Algumas dessas versões, como AdaGrad, RMSProp e Adam, tem sua taxa de aprendizado controlada pelo próprio algoritmo, e variam de acordo com a função de otimização ou a importância dos pesos. Em todo caso, o otimizador que você deve usar vai depender do problema que você tem. Consequentemente, se queremos uma convergência rápida e estamos lidando com uma rede neural profunda ou complexa, devemos escolher um dos métodos de taxa de aprendizado adaptável (Ruder, 2016).

2.1.4 Aprendizado Profundo

Apesar do termo *Deep Learning* estar extremamente popular nos dias atuais, ele não é um conceito novo no campo de Ciência da Computação. Os primeiros trabalhos nesta sub-área do Aprendizado de Máquina foram produzidos na década de 60, como é o caso de (Ivakhnenko, 1967) e (Ivakhnenko, 1968). No entanto, a idéia não engrenou devido a certos motivos, sendo o principal deles o poder computacional insuficiente na época.

Os modelos de *Deep Learning* consistem em Redes Neurais conhecidas como Redes Neurais Profundas, que usam camadas de neurônios matemáticos para processar dados, e atualmente tem as mais diversas aplicações, como por exemplo: compreender a fala humana, reconhecer objetos visualmente, entre outros. A informação é passada através de cada camada, com a saída da camada anterior fornecendo entrada para a próxima camada. A primeira camada em uma rede é chamada de camada de entrada, enquanto a última é chamada de camada de saída.

O principal motivo que diferencia essa abordagem de uma Rede Neural simples comentada em seções anteriores deste trabalho, é a quantidade de camadas entre as duas. No caso das Redes Neurais profundas, essa quantidade é extremamente numerosa. Essas camadas são conhecidas como camadas ocultas (ou *Hidden Layers*). Cada uma delas é tipicamente um algoritmo simples e uniforme contendo um tipo de função de ativação (Zhang et al.,

2021).

Na figura 2.4 temos a representação visual de uma Rede Neural profunda. A camada de entrada é representada pelos neurônios de cor azul, enquanto as camadas ocultas são representadas pela cor rosa, por fim, temos a camada de saída, com os neurônios pintados de verde claro.

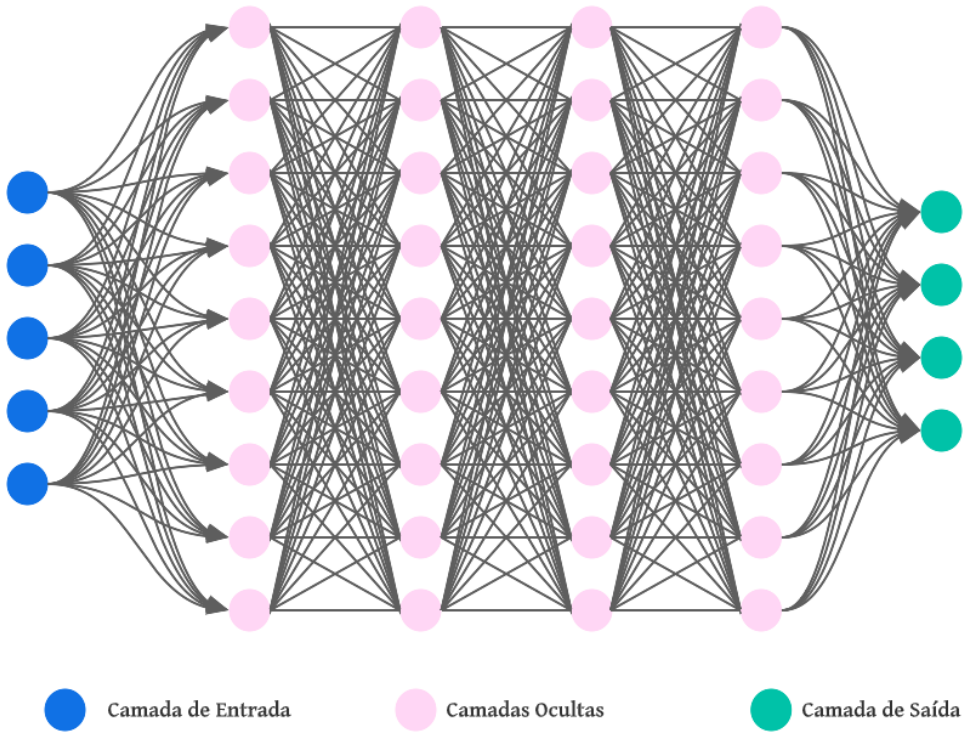

Figura 2.4: Exemplo de uma Rede Neural Profunda.

A extrema maioria das Redes Neurais da atualidade são desse tipo, na seção seguinte falaremos sobre uma delas e que é amplamente utilizada para o Processamento e Análise de imagens.

2.1.5 Redes Neurais Convolucionais

Dados podem ser separados em dois tipos distintos: estruturados e não-estruturados. O primeiro tipo é identificado quando o conjunto de dados está organizado por atributos, em outras palavras, os elementos estão organizados por características para cada amostra, por exemplo: O conjunto de dados de pessoas que recebem o Auxílio emergencial do governo, contém nome, endereço, número de telefone e algumas outras informações sobre cada uma desssas pessoas. Já os não-estruturados, são dados que ainda não tiveram suas características extraídas ou organizados para representar a informação contida neles.

Quando trabalhamos com dados não-estruturados é comum utilizar extratores de características para obter informações com uma boa estrutura. Um dos problemas tratados na área de processamento de imagem é a extração dessas características (LeCun et al., 1999). Muitos dos extratores para imagens utilizam filtros de convolução. No entanto, nem sempre um determinado tipo de convolução pode extrair características relevantes para um problema específico. Uma das formas de resolver isso e criar filtros mais adequados ao problema, é a utilização de Redes Neurais Convolucionais. Essas redes aprendem o conjunto de filtros que melhor se adapta às necessidades dos dados e do problema de classificação (LeCun et al., 1998).

A convolução é uma operação muito importante e com aplicações em diversas áreas, sendo fundamental na de processamento de imagens. Ela é usada para extrair atributos como contornos ou formas, filtrar ruídos, realçar, borrar e extrair propriedades da imagem, sendo definida pela seguinte equação:

$$f(t) = g(t) * h(t) = \int_{-\infty}^{\infty} g(\tau)h(t-\tau)\,d\tau \tag{2.5}$$

A operação acima consiste em relacionar o sinal de entrada g com outro sinal h para então obter um sinal de saída $(g*h)(\tau)$ que define o quanto g foi modificado por h (Gonzalez and Woods, 2009). No entanto, geralmente não é possível ter a representação de um sinal de forma contínua no computador. Logo, um sinal é representado de forma discreta através de uma matriz.

Redes Neurais baseadas em Convolução, tem sua arquitetura organizada como na figura abaixo: sequencialmente, uma imagem é dada de entrada para camadas de convolução, que aplicam filtros, obtendo mapas de *features* que são dados de entrada para outras camadas convolucionais. Em seguida, camadas densas são responsáveis por produzir a classificação da imagem.

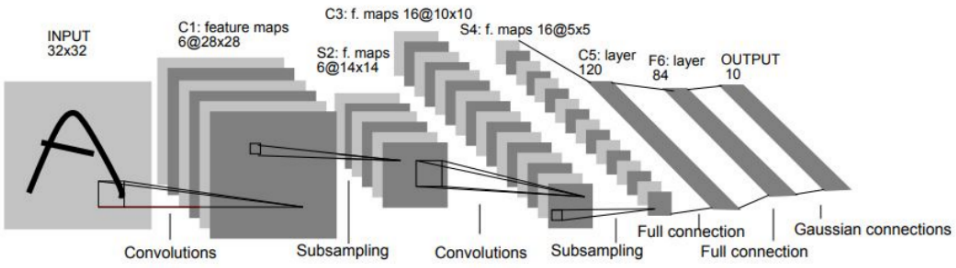

Figura 2.5: Arquitetura da LeNet-5 uma Rede Neural Convolucional (LeCun et al., 1999).

Uma camada de rede neural convolucional tem intenção de identificar padrões sem alterar a topologia dos dados e evitar sofrer com a dimensionalidade. Para encontrar padrões em imagens utilizamos filtros convolucionais que destaquem as características necessárias.

No entanto, criar um novo tipo de filtro que seja útil para resolver determinado problema demanda muito tempo e esforço. Uma forma de resolver esse problema, é usar estas redes para aprenderem de forma automática quais filtros são necessários através de um processo de otimização. Para isso, temos um conjunto de neurônios que compartilham pesos entre si atrelados a uma convolução em cada camada. Cada conjunto produz um ou mais mapas de características*(ou feature maps)* dependendo da quantidade de sinais de entrada recebidos em uma única amostra. Os neurônios nas camadas convolucionais, são organizados de forma a cada um ser um elemento na construção do mapa de características, assim, cada neurônio na camada convolucional recebe uma entrada atrelada a uma região da imagem anterior delimitada pelo tamanho dos filtros de convolução. Para exemplificar, considere um neurônio conectado a uma região 5x5, ele teria ligações com os seguintes pixels da imagem de entrada:

$$\begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \\ \cdots & p_{x-2,y+2} & p_{x-1,y+2} & p_{x,y+2} & p_{x+1,y+2} & p_{x+2,y+2} & \cdots \\ \cdots & p_{x-2,y+1} & p_{x-1,y+1} & p_{x,y+1} & p_{x+1,y+1} & p_{x+2,y+1} & \cdots \\ \cdots & p_{x-2,y} & p_{x-1,y} & p_{x,y} & p_{x+1,y} & p_{x+2,y} & \cdots \\ \cdots & p_{x-2,y-1} & p_{x-1,y-1} & p_{x,y-1} & p_{x+1,y-1} & p_{x+2,y-1} & \cdots \\ \cdots & p_{x-2,y-2} & p_{x-1,y-2} & p_{x,y-2} & p_{x+1,y-2} & p_{x+2,y-2} & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \end{bmatrix}$$

onde cada neurônio produz um único pixel para o *feature map* .

Geralmente, na arquitetura desse tipo de Rede, camadas convolucionais são seguidas de camadas de Acumulação(ou *Pooling*), que tem como função acumular o valor de uma vizinhança de pixels e dessa forma produzir uma imagem menor. Devemos lembrar que esse tipo de camada não possui pesos treináveis, ou seja, uma função de *pooling* basicamente substitui uma região com um resumo estatístico de saídas próximas. Por exemplo, uma operação de max pooling aplicada em uma região de pixels 2×2, produz um único pixel com o valor mais alto daquela região. Existem outros tipos de operações deste tipo, como por exemplo o average pooling, onde o pixel de saída tem como valor a média de todos os pixels da região, dentre outras (Wang and Shang, 2006).

2.1.6 Detecção de Objetos

Nos últimos anos, muitas abordagens avançadas de classificação, como redes neurais artificiais, conjuntos fuzzy e sistemas especialistas, têm sido amplamente aplicadas para classificação de imagens (Lu and Weng, 2007). De uma maneira geral, classificar imagens consiste em predizer qual objeto está aparecendo na cena. Se quisermos classificar fotos de cachorros por exemplo, precisamos fazer um classificador para detectar este tipo específico de animal. Agora consideremos um cenário em que temos um gato e um cachorro em uma

mesma fotografia. Como o nosso modelo detectaria um gato, se ele só conhece o que é um cachorro? Para resolver este problema, devemos treinar um classificador multi-rótulo que irá prever ambas as classes (cão e gato). No entanto, esse modelo ainda não saberia a localização dos animais na fotografia, ele nos diria apenas que temos um cão e um gato na cena em questão. O problema de identificar a localização de um objeto (dada a classe) em uma imagem é chamado de **problema de localização** (Amit et al., 2020).

Um outro problema surge quando a classe do objeto presente na imagem não é conhecida. Nesse cenário, temos que não apenas determinar a localização, mas também prever a classe de cada objeto em cena. Quando prevemos a localização do objeto junto com a classe a qual ele pertence, temos o que é conhecido como **Detecção de Objetos**. Em outras palavras, a detecção de objetos é a estimativa simultânea de categorias e localizações de instâncias de objetos em uma determinada imagem (Oksuz et al., 2020).

Com o passar do tempo os classificadores encontraram uma maneira mais simples e eficiente para essas questões (Borji et al., 2019). Em vez de prever a classe do objeto se baseando em toda a imagem, os algoritmos de classificação começaram a realizar tais inferências a partir de um retângulo(conhecido como *bounding box*) que contém o objeto. Métodos mais recentes, como a Yolo, fazem a inferência não só do objeto, mas também do próprio retângulo(*bounding box*) que o contém. São necessárias 5 variáveis para definir esse retângulo. Assim, para cada instância do objeto na imagem, devemos prever: o nome da classe, a coordenada de abscissa x, a ordenada y, uma altura e uma largura (Galvez et al., 2018). Para determinar essas variáveis, a maioria dos métodos começou trabalhando com janelas de tamanho fixo na imagem de entrada, a fim de determinar a localização desses retângulos. A solução utilizada era a de traçar várias dessas figuras na imagem para checar todas as partes possíveis dela. Atualmente, a maioria das abordagens que tratam o problema como de Classificação, trabalham com uma Busca Seletiva(*Selective Search*). Essa busca consiste no agrupamento hierárquico de regiões semelhantes da imagem a partir de características como cor, textura, tamanho e forma, que é obtido através de uma segmentação da imagem, com base na intensidade dos *pixels* como proposto por (Felzenszwalb and Huttenlocher, 2004). Um exemplo é exibido na figura 2.6

Figura 2.6: Exemplo de segmentação pelo método(Felzenszwalb and Huttenlocher, 2004).

Ao obtermos essas partes, é hora de colocar todos esses locais possíveis em uma rede neural, que é executado em cada um desses "pedaços". A rede em questão prevê a classe do objeto na janela(ou plano de fundo caso não tenha objeto em cena), e assim, sabemos tanto a classe quanto a localização dos objetos na imagem.

Desde que a Detecção de Objetos foi modelada como um **Problema de Classificação**, ótimas abordagens baseadas em Redes Neurais começaram a surgir. Ainda assim, existem algumas complicações(que não serão exploradas neste trabalho). A literatura apresenta diversas técnicas que utilizam Redes Neurais para Detectar Objetos. A seguir, faremos uma breve descrição dos principais modelos do estado da arte (Padilla et al., 2020).

(a) **Region-based Convolutional Neural Networks(R-CNN)** (Girshick et al., 2014)

Quando os primeiros modelos de classificação baseados em aprendizado profundo começaram a surgir, a expectativa era que a área de Classificação de imagens fosse imediatamente dominada por estes algoritmos baseados em Redes Neurais Convolucionais (CNNs). No entanto, as CNNs eram muito lentas e computacionalmente muito caras. Era impossível rodar essas Redes em tantos “arranjos” gerados por um detector de janela deslizante. A R-CNN resolve esse problema usando a Busca Seletiva, que reduz o número de caixas delimitadoras que são passadas para o classificador para cerca de 2.000 propostas de região. Como mencionado anteriormente, a Busca seletiva usa dicas locais como textura, intensidade, cor e/ou uma medida de interioridade, etc., para gerar todas as localizações possíveis do objeto,e assim, podemos usar essas caixas em nosso classificador.

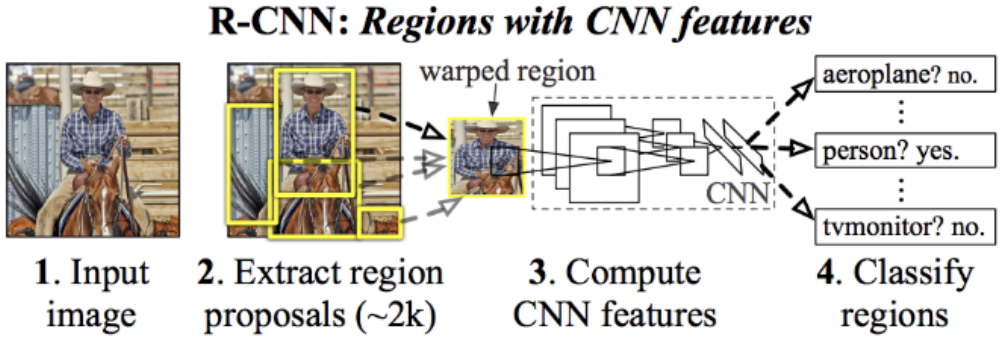

Figura 2.7: Arquitetura da de uma R-CNN (Girshick et al., 2015).

Vale lembrar que a CNN recebe uma entrada de tamanho fixo, portanto, as caixas são redimensionadas. Temos 3 passos importantes do R-CNN:

I. Execute a Busca Seletiva para gerar objetos prováveis.

II. Pegue os arranjos da imagem, que são as regiões de interesse da imagem e mande para a CNN, e em seguida rode SVM(Support Vector Machine) (Wang, 2005) para prever a classe de cada região proposta.

III. Otimize os arranjos treinando a regressão das caixas delimitadoras separadamente.

(b) **Spatial Pyramid Pooling(SPP-net)** (He et al., 2015)

Com o SPP-net, calculamos a representação com CNN para a imagem inteira apenas uma vez e podemos usar isso para agrupar recursos em regiões arbitrárias(sub-imagens) afim de gerar representações de comprimento fixo para treinar os detectores. Isso é feito executando uma operação do tipo Pooling apenas na seção dos mapas de característica na última camada convolucional da rede. Portanto, esse método evita computar repetidamente os recursos convolucionais, o que gera um pouco mais de velocidade para realizar as predições.

(c) **Fast R-CNN** (Girshick, 2015)

O Fast R-CNN usa as ideias do SPP-net e da R-CNN e corrige o problema principal no SPP-net, ou seja, possibilita o treinamento de ponta a ponta. Para propagar os gradientes por meio de agrupamento espacial, ele usa um cálculo de retropropagação simples que é muito semelhante ao cálculo de gradiente de agrupamento máximo, com exceção de que as regiões de agrupamento se sobrepõem e, portanto, uma célula pode ter um gradientes bombeando de várias regiões.

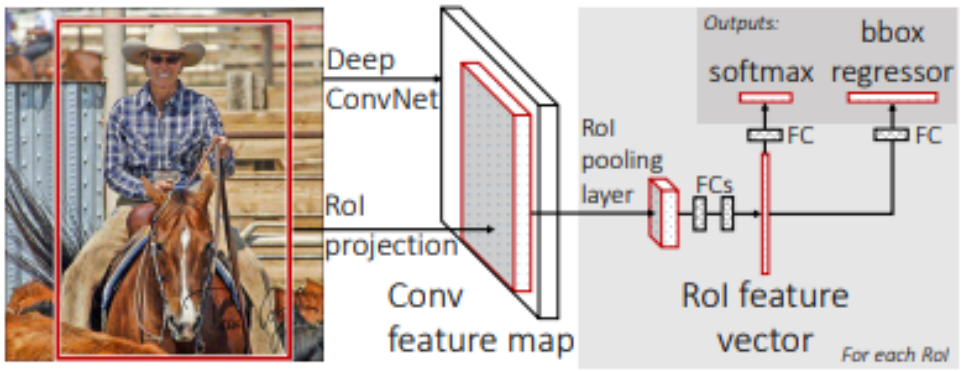

Figura 2.8: Arquitetura da Fast R-CNN (Girshick, 2015).

Além disso, esse modelo adiciona a regressão da *bounding box* ao próprio treinamento da rede neural. Com isso, a rede fica com duas *heads*, uma *head* de classificação e outra de regressão de *bounding box*. Esse objetivo multitarefa é um recurso importante do Fast-RCNN, pois não requer mais treinamento da rede de forma independente para classificação e localização. Essas duas mudanças reduzem o tempo total de treinamento e aumentam a precisão em comparação com os métodos anteriores devido ao aprendizado de ponta a ponta da CNN.

(d) **Faster R-CNN**(Ren et al., 2015)

Este método é ainda mais rápido, e o motivo disso é que ele substitui a Busca Seletiva utilizada nas abordagens anteriores, por mais uma rede neural convolucional, muito pequena, chamada Rede de Proposta de Região(RPN) para gerar regiões de interesse.

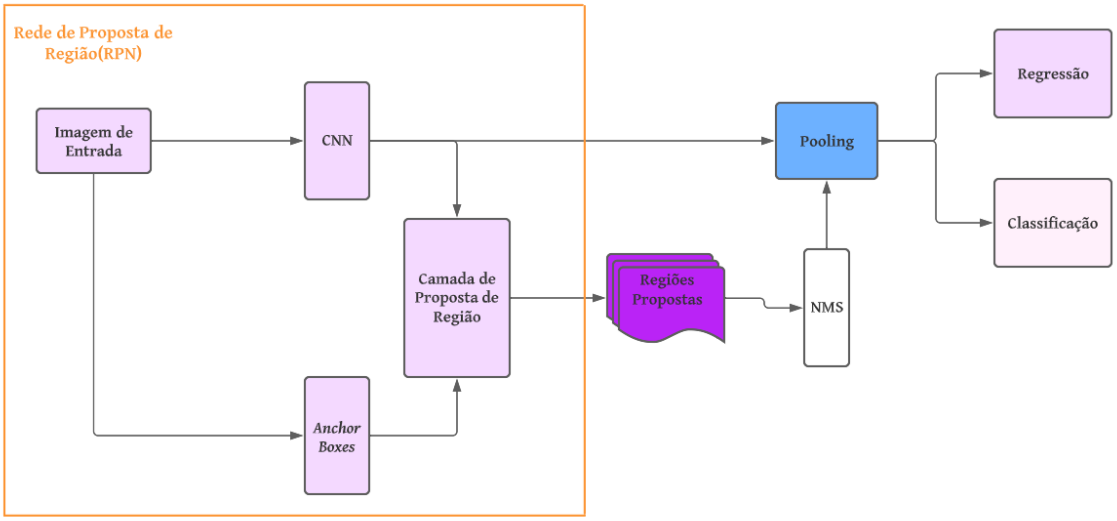

Figura 2.9: Arquitetura da Faster R-CNN.

Para lidar com as variações na proporção e escala dos objetos, o Faster R-CNN introduz a ideia de *anchor boxes*. Em cada local, o paper original usa 3 tipos de *anchor boxes* com escalas de 128x128, 256×256 e 512×512. Para a proporção, ele usa 1:1, 2:1 e 1:2. Portanto, no total, em cada local, temos 9 caixas nas quais a RPN prevê a probabilidade de ser plano de fundo ou primeiro plano. Aplicamos a regressão de *bounding box* para melhorar as *anchor boxes* em cada local. Daí, a RPN fornece caixas delimitadoras de vários tamanhos com as probabilidades correspondentes de cada classe. O Faster-RCNN é 10 vezes mais rápida que o Fast-RCNN com precisão semelhante dependendo do conjunto de Dados. Atualmente, o Faster-RCNN é um dos algoritmos de detecção de objetos mais precisos do estado da arte.

Conforme comentado anteriormente, todos os métodos acima tratam a Detecção como um problema de Classificação, construindo uma *pipeline* onde as primeiras propostas de objetos são geradas e, em seguida, essas propostas são enviadas para as *heads* de classificação/regressão. Entretanto, existem outras abordagens que tratam a Detecção de Objetos como um **Problema de Regressão**. Duas abordagens muito utilizadas atualmente, são o YOLO e o SSD.

(a) **You Only Look Once(YOLO)** (Redmon et al., 2016)

A YOLO divide cada imagem em uma grade de S × S e cada grade prevê um número N de *boxes* e uma confiança. A confiança reflete a precisão da *bounding box* e se a mesma realmente contém um objeto. Esse método também prevê a pontuação de classificação para cada caixa e para cada classe em treinamento. Com isso, somos capazes de combinar as classes para calcular a probabilidade de cada uma estar presente em uma *box* prevista. Assim, prevemos um número de caixas S × S × N no total. No entanto, a maioria dessas caixas tem pontuações de confiança baixas e, se definirmos um limite de confiança, de 25% por exemplo, podemos remover a maioria delas, conforme mostrado na figura 2.10.

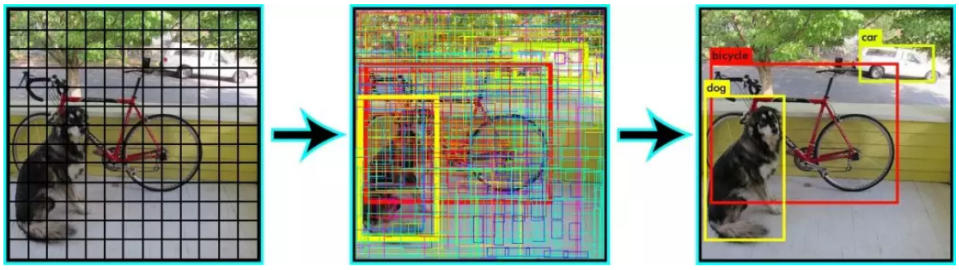

Figura 2.10: Captura de *bounding boxes* na YOLO (Redmon et al., 2016)

Note que em tempo de execução, precisamos rodar nossa imagem na CNN apenas uma vez. Logo, a YOLO é super rápida e pode ser executado em tempo real. Além disso, utilizando esse método só precisamos analisar a imagem completa uma vez, ao invés de várias propostas de região separadamente como nos métodos de classificação(essas informações contextuais ajudam a evitar falsos positivos).

(b) **Single Shot Detector(SSD)** (Liu et al., 2016)

O Single Shot Detector alcança um bom equilíbrio entre velocidade e precisão. Essa abordagem executa uma rede convolucional na imagem de entrada apenas uma vez e calcula um *feature map*. Em seguida, executa um pequeno *kernel* convolucional de tamanho 3×3 nesse *feature map* com o objetivo de prever as *bounding box's* e a probabilidade de classificação. O SSD também usa *anchor box's* em várias proporções e aprende o deslocamento em vez de aprender a caixa em si. Para lidar com a escala, o método em questão prevê as *bounding box's* após várias camadas convolucionais, e como cada camada convolucional opera em uma escala diferente, ela é capaz de detectar objetos de várias escalas.

Uma detecção de pessoas mais detalhada poderia ser realizada usando técnicas de detecção de pose como é o caso da **OpenPose**(Cao et al., 2017). A técnica em questão, usa uma

representação não paramétrica, chamada de *Part Affinity Fields(PAFs)*, para aprender a associar partes do corpo a indivíduos na imagem. Resumindo, a OpenPose usa a imagem inteira como entrada para uma CNN prever conjuntamente mapas de confiança para Detecção de partes do corpo e *PAFs* para associação parcial, em seguida, ela executa a etapa de análise, que roda um conjunto de correspondências bipartidas para associar candidatos a partes do corpo, e por fim montar as poses de corpo inteiro para todas as pessoas na imagem.

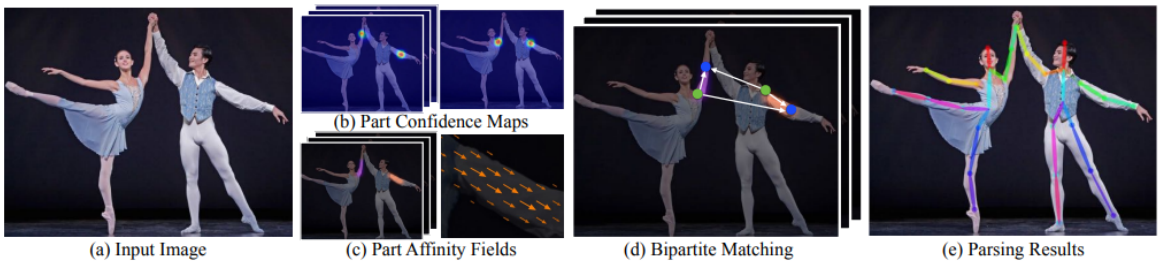

Figura 2.11: Visão geral do funcionamento da OpenPose (Cao et al., 2017).

2.2 Geometria Computacional

Quando analisamos a história da Geometria, é inegável que suas versões Egípicia e Grega foram obras primas da matemática aplicada. Sabe-se ainda, que as motivações originais para resolução de problemas geométricos, incluíam temas como: a necessidade de tributar terras com precisão e construir edifícios. No entanto, depois de muitos anos, e consequentemente vários estudos, a Geometria assume várias áreas de aplicação. Uma das áreas mais recentes, é a Computacional. O termo Geometria Computacional, aparece pela primeira vez em um artigo de 1975, onde os autores realizam um estudo dos aspectos computacionais da geometria no âmbito da análise de algoritmos (Shamos, 1978). Pode-se dizer então, que a Geometria Computacional é o estudo e o desenvolvimento de técnicas matemáticas necessárias para o projeto de algoritmos eficientes que resolvem problemas geométricos. Esses problemas são os mais diversos, incluindo o caixeiro viajante euclidiano, árvore geradora mínima, problemas de programação linear, entre muitos outros (Preparata and Shamos, 2012). Um desses problemas é o do Ponto em Polígono.

2.2.1 Polígonos Simples

Polígonos simples são figuras geométricas planas compostas apenas por uma linha fechada, que por sua vez é formada de segmentos de reta chamados de lados. Além disso, os lados não adjacentes dessa figura não se interceptam. A ilustração abaixo apresenta alguns exemplos a fim de facilitar o entendimento.

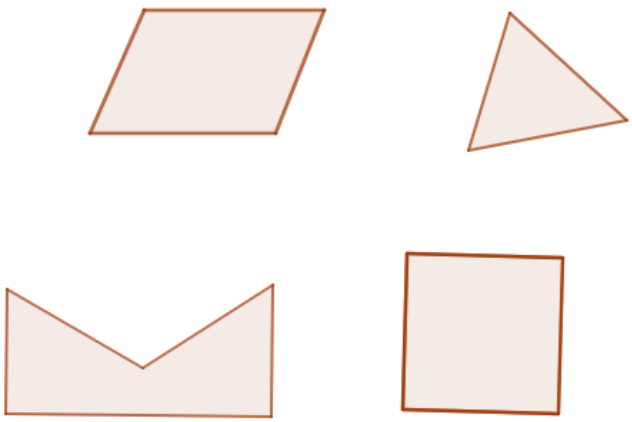

Figura 2.12: Exemplos de Polígonos Simples.

É importante destacar que tais formas devem possuir ao menos 3 Lados (segmentos de reta que formam o Polígono), Vértices (os pontos de intersecção entre dois Lados), Diagonais (segmentos de reta que ligam dois vértices não consecutivos) e Ângulos (que podem ser exteriores ou interiores a Figura, estes últimos formados por dois segmentos de reta adja-

centes no interior do Polígono, enquanto o primeiro é formado pelo prolongamento de um lado e o lado adjacente a ele).

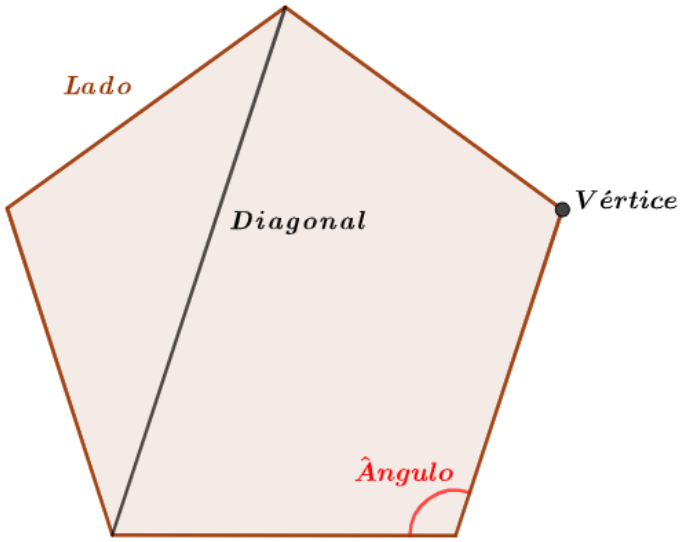

Figura 2.13: Principais elementos de um Polígono.

Podemos classificar esses Polígonos em quatro tipos principais, sendo eles:

(a) Convexos: possuem ângulos internos menores do que 180°. Geometricamente, dizemos que um Polígono é Convexo quando todos os pontos de um segmento de reta que possui as extremidades em seu interior também estão dentro dele.

(b) Não Convexos ou Côncavos: apresentam ângulos internos maiores do que 180°. Inferimos que um Polígono é Côncavo quando encontramos pelo menos um segmento de reta que possui as extremidades dentro da figura e ao mesmo tempo, um ponto fora dela.

(c) Regulares: possuem todos os lados e ângulos com medidas iguais. Todo Polígono regular simples é Convexo por definição.

(d) Irregulares: seus lados e ângulos são diferentes entre si.

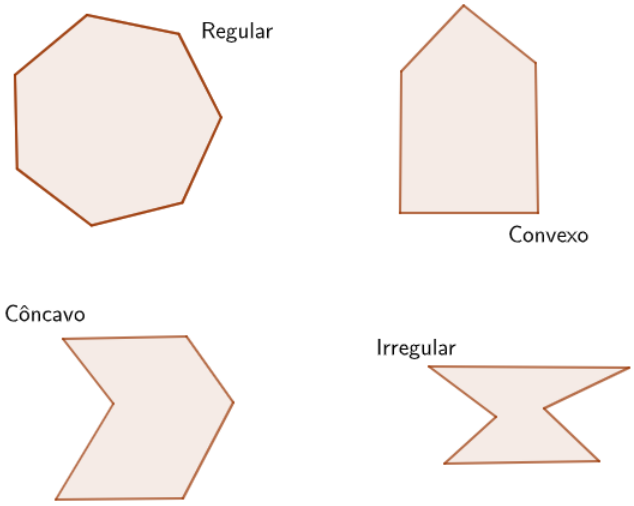

Figura 2.14: Exemplos de Tipos de Polígonos.

2.2.2 Localização de pontos em relação a polígonos

No problema do Ponto em Polígono, devemos determinar se um ponto p é interior, exterior ou está na fronteira de um determinado Polígono simples $P = p_1 p_2 \ldots p_n$. Note que, uma vez que uma linha poligonal fechada e simples é uma curva de Jordan e, portanto, separa o plano em duas regiões conexas e abertas, uma limitada e a outra não, ambas tendo a curva poligonal como fronteira(Henle, 1994).

A literatura apresenta duas soluções para o problema (Figueiredo and Carvalho, 2000). A primeira consiste em traçar uma semi-reta L na horizontal partindo de p, e daí contar seus pontos de intersecção com a linha poligonal. Se p coincidir com um destes pontos de interseção, concluímos que ele está na fronteira de P. Caso contrário, basta computar quantas vezes a semi-reta atravessa a poligonal. Sabe-se que no infinito, L se encontra na região exterior. Logo, se o número de cruzamentos for ímpar, o ponto p é interior; senão, p é exterior.

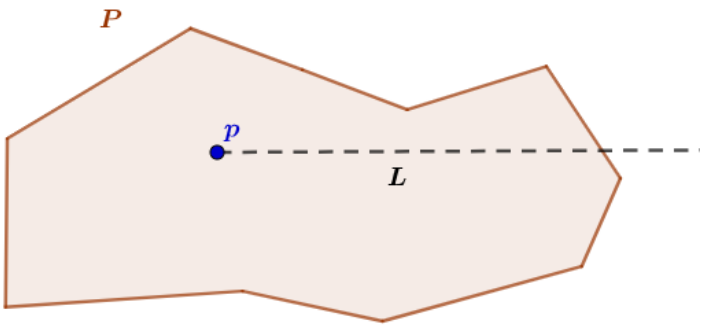

Figura 2.15: Traçando semi-reta na Horizontal a partir de p.

Uma outra solução é recorrer à noção de Índice de rotação. Dada uma linha poligonal fechada $P = p_1p_2\ldots p_n$(não necessariamente simples) e um ponto p não pertencente a ela, definimos o Índice de rotação de p em relação a P como

$$k = \frac{1}{2\pi}\sum_{i=1}^{n} \angle(p_ipp_{i+1})$$

onde $\angle(p_ipp_{i+1})$ representa o ângulo orientado do vetor pp_i para o vetor pp_{i+1} e onde temos $p_{n+1} = p_1$.

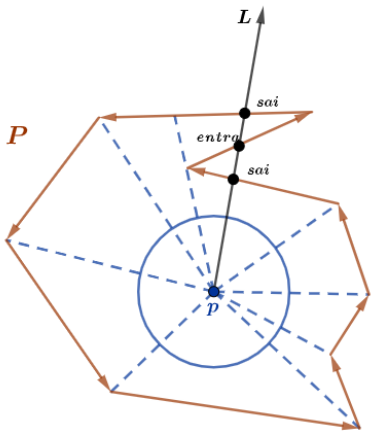

Figura 2.16: Calculando o índice de Rotação.

Perceba que cada ângulo orientado acima é igual ao comprimento do arco orientado obtido com a projeção de dois vértices consecutivos sobre um círculo de raio 1 centrado em p. Daí, a soma de todos os ângulos corresponde a um número inteiro de voltas no círculo. Portanto, k certamente é um número inteiro, e no caso específico de P determinar um polígono simples vale o seguinte resultado:

Teorema: Seja $P = p_1p_2\ldots p_n$ um poligono simples e seja p um ponto que não pertence à fronteira de P. Então o índice de rotação k de p é igual a 0, 1 ou -1 e

(a) p é interior se e somente se $k = \pm 1$

(b) p é exterior se e somente se $k = 0$.

2.2.3 Ponto Médio

Define-se ponto médio como o ponto que divide um segmento de reta exatamente no meio. Portanto, dados dois pontos no plano, de coordenadas $P = (x_1, y_1)$ e $Q = (x_2, y_2)$ definimos o ponto médio M através da equação

$$M = \left(\frac{(x_1 + x_2)}{2}, \frac{(y_1 + y_2)}{2} \right)$$

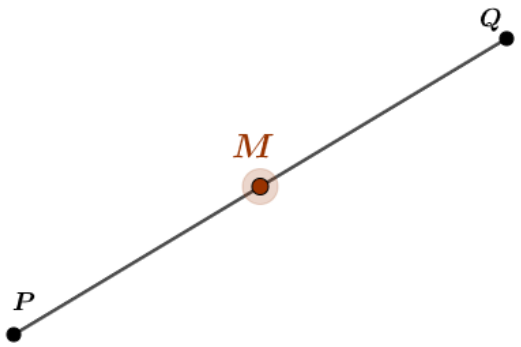

Figura 2.17: Representação Geométrica do Ponto Médio.

3

Metodologia

A metodologia da pesquisa desenvolvida consistiu em dois grandes momentos, a etapa teórica e a implementação das funcionalidades.

Para o desenvolvimento do projeto descrito neste trabalho, foram planejadas as seguintes etapas de execução.

3.1 Estudo dos Modelos de Detecção de Objetos

A partir de um conjunto de referências bibliográficas, foram estudados os principais métodos de detecção de Objetos presentes no estado da arte, e que melhor se adequavam ao problema em questão. Em nosso caso, a preocupação maior era conseguir detectar pessoas de uma forma confiável e rápida, já que a ferramenta pode ser executada em tempo real.

Depois de diversas pesquisas, reduzimos nossas opções para dois modelos: o Yolo (Redmon et al., 2016) e o OpenPose (Cao et al., 2017). Decidimos dar continuidade com o Yolov5, por ter uma funcionalidade rápida comparada com os outros métodos do estado da arte, além de uma boa precisão para detecção de pessoas. Em contrapartida, a OpenPose é completamente inviável para ser utilizado na maioria dos *Desktops* em Tempo Real. Apesar de detectar Key Points com mais precisão, o custo computacional é simplesmente muito alto (exigindo pelo menos 16G de RAM para funcionar de forma minimamente aceitável, além claro, de um processador de gerações mais atuais e com muitos núcleos) o que comprometeria muito o desempenho da ferramenta pensando na portabilidade. Para se ter uma ideia, testamos o OpenPose em uma máquina com um processador intel de quarta geração i7 e 8G de RAM e o resultado foi de 0,2 frames por segundo, com a imagem sumindo diversas vezes e reaperecendo em intervalos de minutos, o que é de fato inconcebível. Vale ressaltar que, como observado em (Bochkovskiy et al., 2020), as Redes Neurais modernas mais precisas não operam tão bem em tempo real, por exigirem um grande número de GPUs para

treinamento com um grande tamanho de mini-lote. A Yolov4 resolve esse problema apresentando uma arquitetura de baixo custo computacional que opera em tempo real em uma GPU convencional e para a qual o treinamento requer apenas uma GPU convencional, possibilitando uma boa execução em tempo real e uma detecção de boa precisão. Já versão 5 trás consigo melhorias excelentes nos mais diversos aspectos, os principais incluem a otimização do cálculo de predição das *bouding boxes*, além de duas substituições: a adição de uma 6x6 Conv2d no lugar da estrutura Focus utilizada em versões anteriores e a da estrutura SPP por uma SPPF como ilustra a figura 3.1. Essas substituições, e mais especificamente a última impactam diretamente no desempenho do modelo dobrando a sua velocidade como relatado.

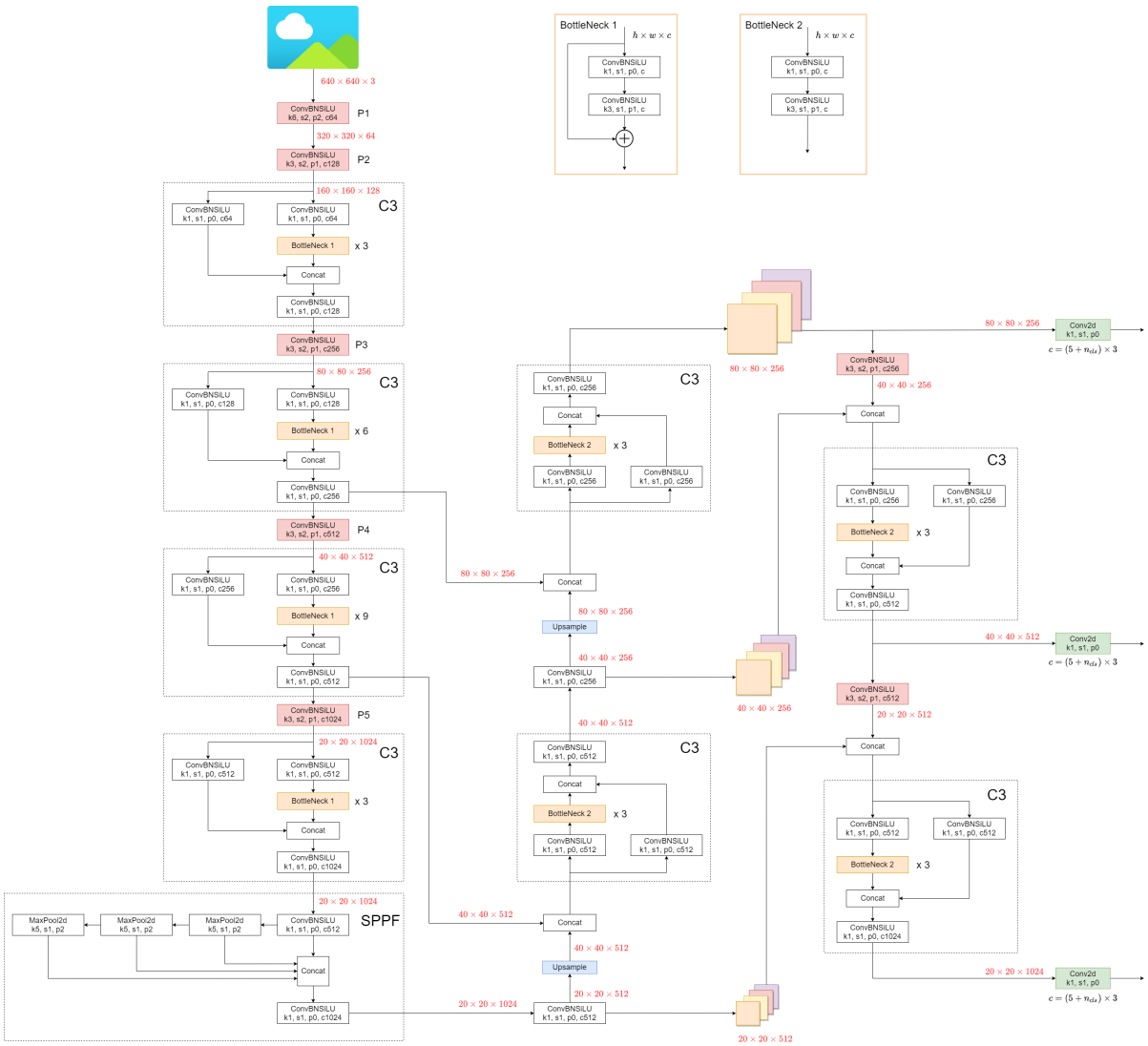

Figura 3.1: Arquitetura da Yolov5 disponível aqui.

3.2 Estudo das funções a serem implementadas

O estudo das funções a serem implementadas foi realizado a partir da análise de um conjunto de referências bibliográficas, visando ampliar os conhecimentos a cerca do tema proposto.

Foram estudados ao longo deste momento, temas como Aprendizado de Máquina, Redes Neurais Convolucionais, Analise e Processamento de Imagens, Geometria Computacional e a linguagem de programação **Python**. Inicialmente precisavamos descobrir uma forma de detectar unicamente pessoas, felizmente isso pode ser feito passando um comando específico como mostrado em seções futuras, em seguida, idealizamos um roteiro de funções que deveriam ser implementadas, incluindo: uma função para inferir se o ponto está dentro do Polígono, uma função para desenhar com o Mouse, outra funcionalidade capaz de emitir um alerta caso a linha poligonal fosse cruzada e por fim, alguma capaz de ilustrar de alguma forma o ponto médio(que optamos por representar como um pequeno círculo).

3.3 Implementação

Após o término da revisão bibliográfica na literatura existente, deu-se então início à implementação do trabalho, desenvolvido em **Python**.

Para tornar possível a proposta da ferramenta temos que detectar somente pessoas na cena. A Yolo já nos oferece essa opção de filtrar, para isso basta passar **-- classes 0** quando vamos realizar a inferência. Para dar continuidade, precisavamos descobrir uma forma de desenhar um Polígono simples utilizando o Mouse na interface do vídeo em tempo real ou gravado. Essa funcionalidade foi implementada utilizando a Biblioteca *OpenCV* (Bradski and Kaehler, 2000), muito famosa na área de Visão Computacional. Também foram usadas funções mais gerais da própria linguagem, por exemplo, as funções para registrar eventos do Mouse, como é o caso da *EVENTLBUTTONDOWN*, que foi usada para marcar os pontos quando um clique com o botão esquerdo acontecia.

O próximo passo seria conseguir as cordenadas dos vértices das *bouding boxes* geradas pela Yolov5 ao detectar pessoas, e então definir um ponto para representar a posição da pessoa. Neste caso, propomos usar o ponto médio da aresta inferior da *bounding box* como uma aproximação dos pés da pessoa como mostra a figura 3.2.

Como o Yolov5 já nos retorna esses dados, bastou apenas pegar a abcissa e ordenada inferior esquerda x_1 e y_1 da *bouding box*, assim como, as outras coordenadas à direita x_2 e y_2, realizar a soma e dividir por 2, para então obter o ponto médio. Com essa informação, precisamos agora da função para calcular se o ponto está dentro ou fora do Polígono simples.

A função para inferir se o ponto está dentro ou fora do Polígono foi implementada de acordo com a estratégia explicada na seção de Geometria Computacional. O método escolhido foi o de traçar uma semi-reta na horizontal. Escolhemos essa abordagem porque além

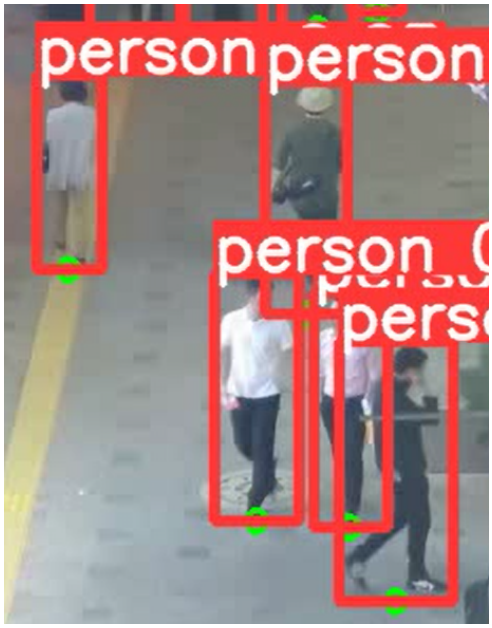

Figura 3.2: Exemplo do ponto médio como representação da pessoa.

da implementação ser mais simples, a descontinuidade do Índice de rotação na fronteira do polígono torna o segundo método altamente instável para pontos situados próximos à fronteira. Como lidamos com esses pontos sempre que uma pessoa se aproxima de uma área demarcada, não teríamos bons resultados de uma forma geral. Vale ressaltar que as duas abordagens resolvem o problema em tempo computacional $O(n)$ e por isso só temos a ganhar com o primeiro método.

Por fim, deveríamos disparar um alerta sonoro caso uma pessoa entrasse na área demarcada, o que foi feito utilizando o *mixer*, que é um dos componentes da biblioteca *pygame*. Um dos parâmetros que precisamos para utilizar esse componente é o tempo pois precisamos definir por quantos segundos o alerta sonoro ficará em execução, nesse caso optamos por 0,2 segundos, caso a pessoa permaneça o som se mantém, mas caso ela saia da área, o alerta sonoro para depois de 0.2 segundos. Para essas questões de tempo e duração, usamos a biblioteca *time*. As funcionalidades acima são utilizadas quando a ferramenta é executada em tempo real.

Já para os experimentos, ou em vídeo gravados, optamos por desenhar um ponto na posição dos pés das pessoas (que é definido como o ponto médio da aresta inferior da *bouding box*) e caso a linha poligonal seja cruzada, pintar esse ponto para vermelho(inicialmente a cor é verde). Usamos novamente a *OpenCV* aqui, mais especificamente a função *circle()*. Acreditamos que isso facilita o entendimento para fins experimentais.

3.4 Fluxogramas de Funcionamento do Algoritmo

Dependendo da execução, a ferramenta pode se comportar de duas maneiras distintas. O primeiro caso é quando queremos executar o programa em vídeos já gravados(como foi feito na realização dos experimentos). Para este caso, temos o Fluxograma de execução representado na figura 3.1

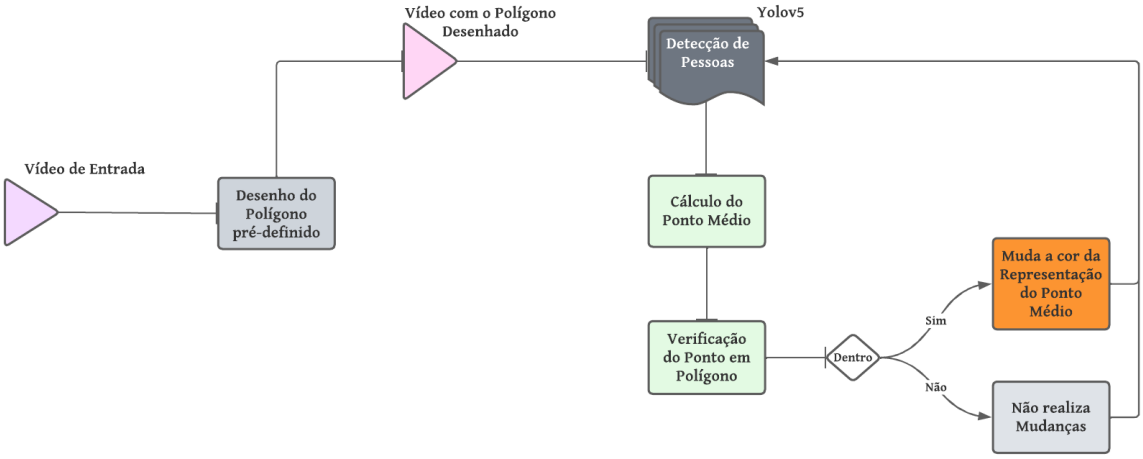

Figura 3.3: Execução da ferramenta em vídeos gravados

Uma outra possibilidade é a de executar as verificações em tempo real. Nesse caso, um alerta sonoro é emitido conforme demonstrado na Figura 3.2

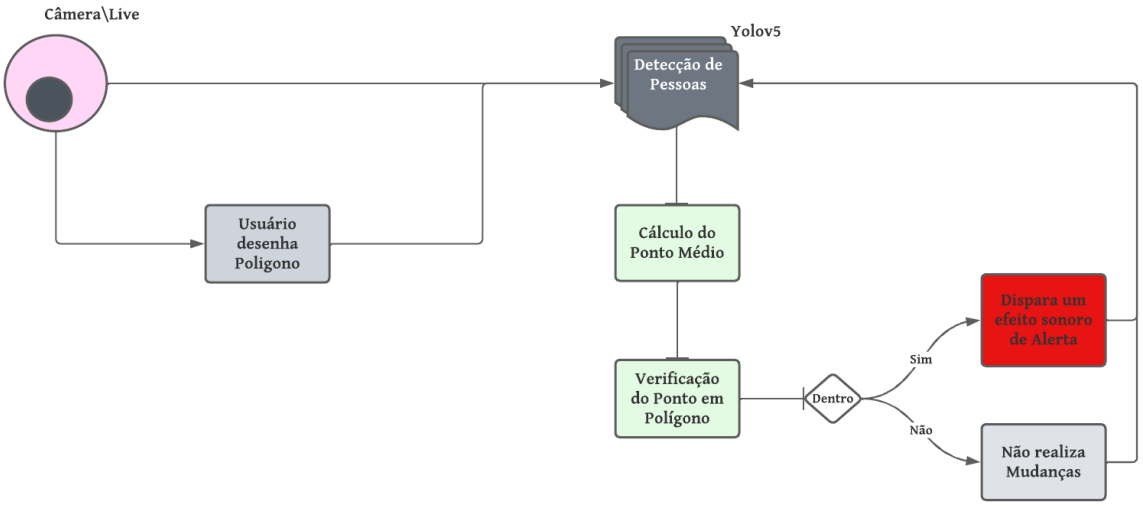

Figura 3.4: Funcionamento do Algoritmo em tempo real

3.5 Validação e preparação de manuais de uso e instalação

É de fundamental importância para tal projeto a verificação da qualidade software desenvolvido, portanto, um de nossos objetivos consistiu em validar as funcionalidades implementadas. Para isso, realizamos testes em diversos vídeos, que são amostras de *datasets* comentados posteriormente. Alguns desses testes estão presentes no capítulo a seguir para facilitar o entedimento.

Também foram desenvolvidos manuais de uso do programa, assim como, informações sobre a instalação. Um exemplo de uso do Software é mostrado no Capítulo 5. Todas as demais descrições se encontram no apêndice A deste trabalho.

4

Resultados

Realizamos alguns experimentos a fim de validar o funcionamento do software desenvolvido. Não houve necessidade de treinar um novo modelo, uma vez que os últimos disponibilizados pela Yolov5 e treinados com o *COCO dataset* (Lin et al., 2014)já detectam pessoas com uma boa precisão(como mostrado na seção *Training*) e atendem às nossas necessidades. Ao todo, foram realizados 80 experimentos em 40 vídeos, que foram extraídos dos *datasets*: MOTChallenge (Leal-Taixé et al., 2015), VOT2018 (Kristan et al., 2018), LaSOT (Large-scale Single Object Tracking) (Fan et al., 2019) e o KITTI (Geiger et al., 2012). Alguns desses experimentos foram realizados em tempo real, utilizando uma Webcam para gravar uma rua pouco movimentada, no entanto, a grande maioria foi realizada em vídeos gravados, e para tal, precisamos realizar algumas alterações no código fonte da ferramenta(para mais detalhes consulte o Anexo B deste trabalho. Todos os vídeos utilizados nos experimentos têm no mínimo 10 frames por segundo e possuem duração de 7 segundos a 3 minutos. O computador em que os experimentos foram realizados possui uma CPU integrada, com um processador intel i7 de quarta geração e 8G Memória RAM.

Abaixo temos a descrição detalhada de 5 destes experimentos com o objetivo de ilustrar o funcionamento do software, bem como comprovar suas funcionalidades. As figuras a seguir foram produzidas durante a realização do trabalho e servem para facilitar o entendimento. Nas imagens, o ponto médio é representado por um círculo pequeno, que muda de cor dependendo da situação, sendo verde quando o ponto está fora do Polígono e vermelho quando ele está dentro.

4.1 Experimentos qualitativos

(a) **Experimento 1:**

Para o Polígono exibido na Figura 4.1 de cor azul o software computou todas as vezes em que alguém entrou na área em questão corretamente. Utilizando o mesmo vídeo, também verificamos a funcionalidade do programa para um Polígono com coordenadas distintas, que é exibido na Figura 4.2.

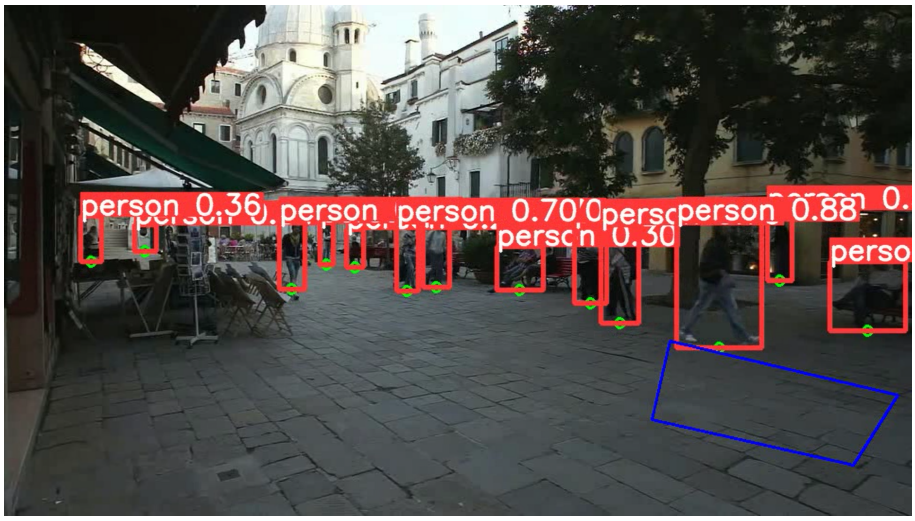

Figura 4.1: Experimento 1 parte 1

Neste caso, os resultados foram totalmente satisfatórios, onde o algoritmo contabilizou corretamente o número de pessoas que entraram na área selecionada.

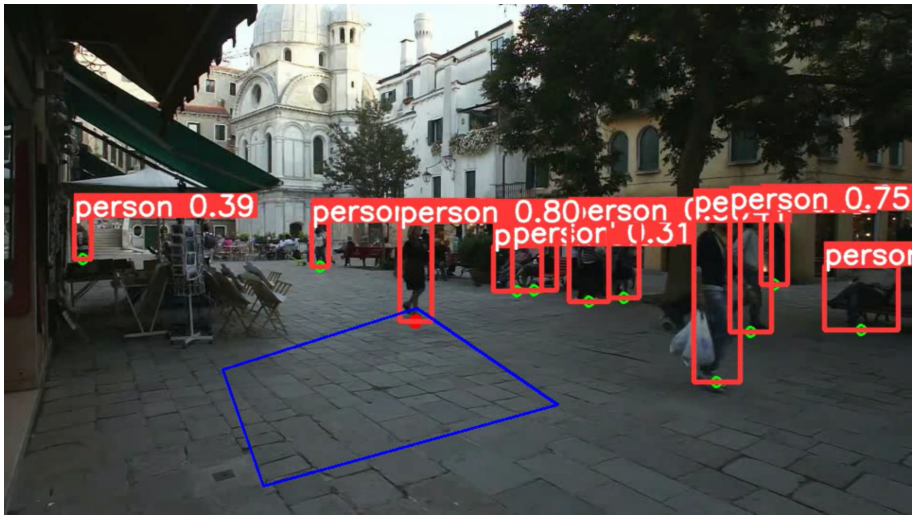

Figura 4.2: Experimento 1 parte 2

(b) **Experimento 2:**

Para o Polígono simples representado na Figura 4.3 o programa foi capaz de acertar a maioria das vezes em que as pessoas cruzaram a linha poligonal, falhando somente uma vez devido a posição do ponto médio(que realmente está fora do polígono, mas o pé direito do homem não).

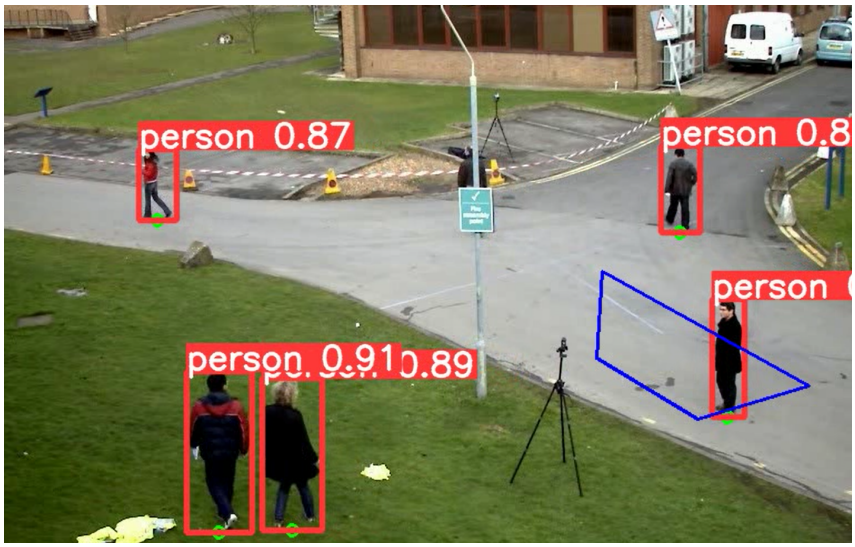

Figura 4.3: Experimento 2 parte 1

Também realizamos um teste utilizando as posições diferentes posições para criar o polígono exibido na Figura 4.4.

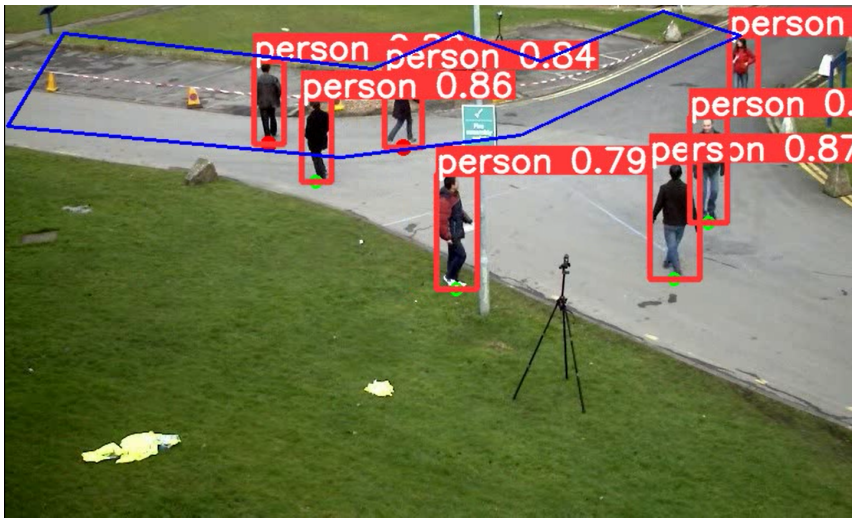

Figura 4.4: Experimento 2 parte 2

Os resultados foram satisfatórios e a ferramenta contabilizou corretamente o número de vezes que alguém cruzou a linha poligonal.

(c) **Experimento 3:**

Para o Polígono demonstrado na Figura 4.5 o algoritmo acertou todos os casos em que houve contato com a linha poligonal, inclusive nos mais complexos para a própria Yolo identificar pessoas, como mostra a figura acima.

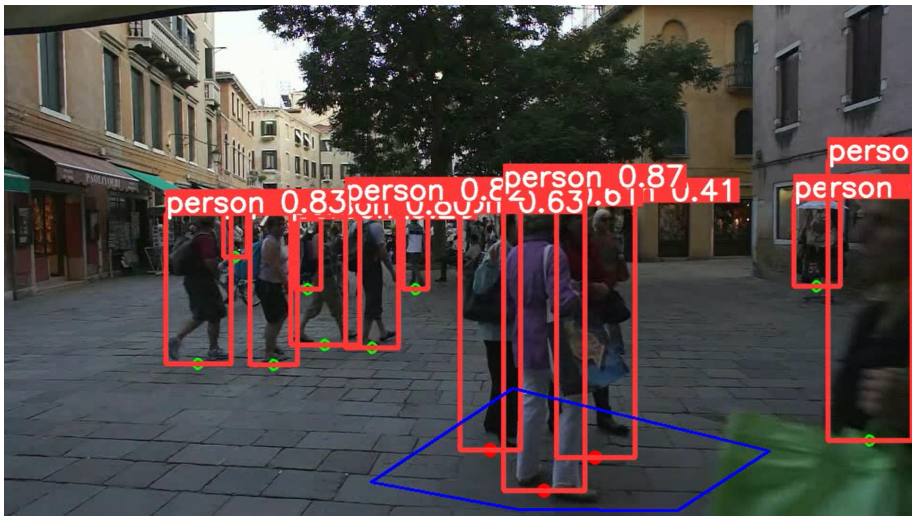

Figura 4.5: Experimento 3 parte 1

Também realizamos um teste utilizando as coordenadas mostradas na Figura 4.6. Os resultados também foram satisfatórios e a ferramenta contou corretamente o número de vezes em que alguma pessoa cruzou a linha poligonal.

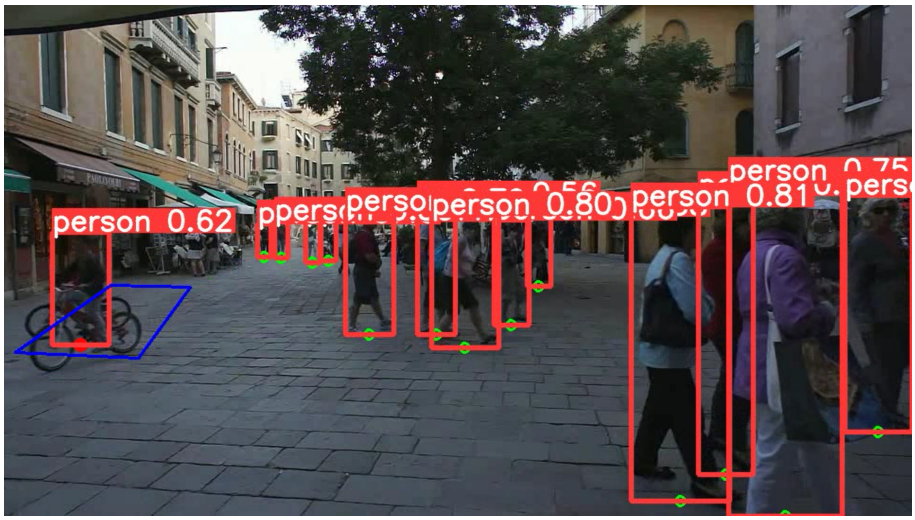

Figura 4.6: Experimento 3 parte 2

(d) **Experimento 4:**

Para o polígono simples exibido na Figura 4.7 o programa acertou todas as vezes em que alguém entrou na área demarcada.

Figura 4.7: Experimento 4 parte 1

Outro teste foi realizado usando os pontos ilustrados na Figura 4.8 Nesse caso o algoritmo também computou corretamente o número de vezes em que a linha poligonal foi invadida, por alguém. Vale ressaltar que isso representa uma excelente perfomance, pois mesmo em casos em que as caixas delimitadoras das pessoas se sobrepõe,o número de vezes em que a pessoa entrou no determinado espaço foi computado corretamente.

Figura 4.8: Experimento 4 parte 2

(e) **Experimento 5:**

Para o Polígono de Coordenadas exibidas na Figura 4.9 o programa mostrou um bom funcionamento, falhando apenas uma vez devido a proximidade das pessoas na cena. Por outro lado, podemos observar que algumas pessoas não foram detectadas nesse experimento, isso ocorre geralmente quando temos um número elevado de pessoas em um curto espaço, devido a limitações dos métodos de detecção de objetos.

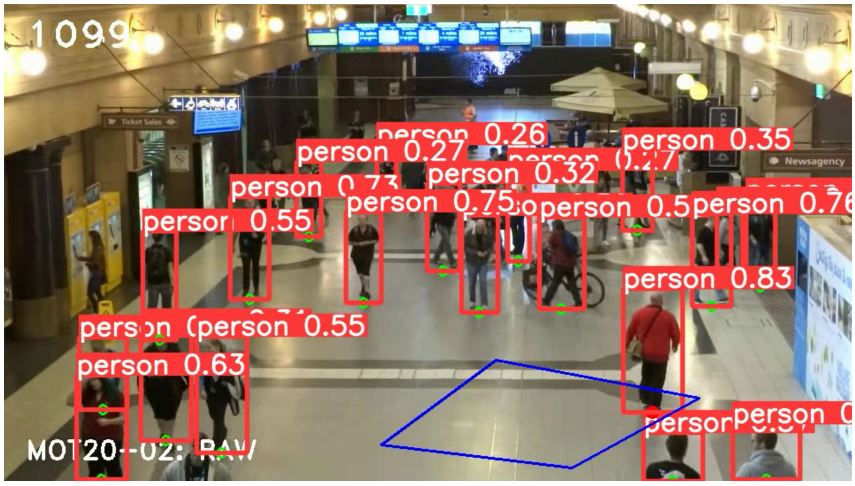

Figura 4.9: Experimento 5 parte 1

Também foram realizados testes para o Polígono que é ilustrado na Figura 4.10. Obtivemos esses pontos, a fim de ilustrar que nossa abordagem pode falhar em casos muito específcos e que seriam relativamente difíceis até mesmo para um ser humano inferir. Perceba que os pés da criança estão em sua maioria fora da caixa delimitadora, mas apenas uma pequena ponta do final de um dos pés está dentro da área, o que é realmente difícil de decidir.

Figura 4.10: Experimento 5 parte 2

4.2 Avaliação do Modelo

Para avaliar nosso programa de forma quantitativa, construímos uma matriz de confusão a partir dos resultados dos experimentos realizados. Ao todo, executamos o algoritmo em 40 vídeos, com dois polígonos sendo desenhados em cada um deles. Com os resultados construímos a seguinte matriz de confusão:

Predições Totais = 9384		Predição do Modelo	
		Sim	Não
Predição Correta	Sim	1296	104
	Não	8	8000

Figura 4.11: Matriz de confusão do modelo

Construímos a matriz acima nos baseando no número de pessoas detectadas que entraram no Polígono. Um Verdadeiro Positivo foi computado sempre que uma pessoa entrou no polígono e o algoritmo inferiu corretamente sua entrada, no caso dos Falsos Positivos, eles foram contados sempre que o modelo classificou um objeto como pessoa sem ser o caso, e esse objeto classificado errado cruzou a linha poligonal. Já os Falsos Negativos, ocorreram quando o modelo deveria ter contabilizado a entrada da pessoa no Polígono, mas não o fez. A maioria desses casos, ocorreu quando pessoas estavam muito próximas uma das outras, o que como dito anteriormente, gera problemas devido as caixas delimitadoras presentes. Uma outra possibilidade, mas que aconteceu com bem menos frequência, foram os casos apresentados no experimento 2 parte 1 e experimento 5 parte 2. Por fim, os Verdadeiros Negativos considerados, foram o número de vezes em que o modelo classificou uma pessoa e inferiu corretamente que ela estava fora do Polígono definido.

A ferramenta apresentou uma Acurácia de 98%, enquanto a Precisão foi de 99%, com um *Recall* de aproximadamente 92%, resultando em um *F1-Score* de 95%. No nosso caso, a principal métrica a ser levada em conta deve ser o *Recall*, pois para o nosso problema os Falsos Negativos são mais importantes, ou seja, o número de vezes em que a pessoa cruzou a linha poligonal e o modelo não contabilizou.

5

Demonstração de uso do Software

Nesta seção, demonstraremos como utilizar o programa desenvolvido para inferir se uma pessoa adentrou ou não a linha poligonal. Como comentado anteriormente, existem duas propostas para execução do Software, sendo a primeira em tempo real através de uma câmera conectada ao computador, e a segunda para lives disponíveis no Youtube. Neste capítulo mostraremos os dois casos.

Depois de ter instalado todas as dependências necessárias e seguido todos os passos presentes no Apêndice A, você deve decidir para qual finalidade quer executar o programa.

5.1 Executar em tempo real para lives no Youtube

Alguns vídeos no Youtube transmitem imagens de câmeras ao redor do mundo. Também é possível executar essa ferramenta para tais vídeos. No entanto, como a maioria dessas transmissões têm uma resolução muito alta, é extremamente recomendado o uso de uma *GPU*. Rode o comando abaixo no terminal:

python detect.py −−classes 0 −−source
https://www.youtube.com/watch?v=DjdUEyjx8GM −−weights yolov5s.pt

você deve substituir o link do exemplo acima pelo do vídeo que deseja executar. Ao executar o comando acima, duas janelas serão abertas, uma contendo um pequeno erro(provavelmente questões com a OpenCV) e a outra com o vídeo em tempo real. Com o botão esquerdo do mouse, clique nos vértices da figura que deseja desenhar. Depois de clicar na posição dos vértices desejados aperte a tecla **P** do seu teclado e a figura aparecerá na tela como mostrado na Figura 5.1.

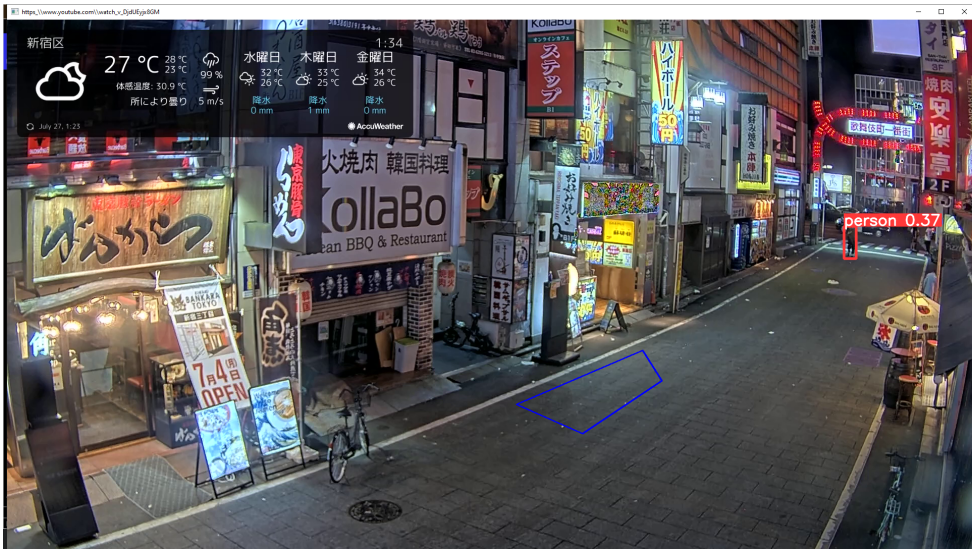

Figura 5.1: Uso da ferramenta para câmeras dispostas em lives no Youtube

Certifique-se de ter desenhado um Polígono Simples. O algoritmo disparará um alerta sonoro caso alguma pessoa adentre a linha poligonal.

5.2 Para executar em tempo real com uma câmera conectada

Para esta opção, devemos ter uma câmera conectada ao computador, além disso, é fortemente recomendado ter uma *GPU*. A forma de execução é simples, basta digitar:

python detect.py ——classes 0 ——source 0

no terminal ou no terminal do *Pycharm*(lembrando que você deve estar dentro da pasta **yolov5**) e apertar Enter. O programa irá abrir a camera conectada ao seu computador(caso contrário passe 1, ou 2 após o *source* substituindo o 0 no comando acima) e você poderá desenhar o Polígono na tela como anteriormente para lives do Youtube. A diferença é que neste caso, o algoritmo faz as predições e detectações para as imagens capturadas por sua câmera.

6

Conclusões e Trabalhos Futuros

Neste trabalho, apresentamos o desenvolvimento de uma ferramenta rápida e de boa precisão, que é capaz de detectar se uma pessoa entrou ou não em uma área demarcada em tempo real. O teste e a validação do sistema foram tarefas contínuas, e ocorreram ao longo do desenvolvimento do projeto, bem como o incremento e desenvolvimento de novas funcionalidades.

A ferramenta performou bem de uma forma geral, exceto em vídeos onde o volume de pessoas era muito grande(com mais de 40 pessoas aparecendo na cena ao mesmo tempo) ou quando executada em tempo real em cenas com muitas pessoas(devido as necessidades gráficas que não são atendidas pela CPU). Todos esses motivos se dão devido a ausência de poder de processamento e qualidade gráfica na máquina utilizada para os testes.

Esse programa tem diversas aplicações, desde o monitoramente de determinadas áreas de risco, a abordagens relacionadas a segurança de pedestres e proteção de propriedades (Khanna and Awasthi). Também podemos realizar pequenas alterações e utilizar a ferramenta para monitorar pesoas usando óculos VR em tempo real. Nesse caso, a problemática seria inferir uma área da qual o usuário não deve sair, dessa forma, preservamos sua segurança e o impedimos de colidir com algum objeto enquanto usa os óculos.

Em trabalhos futuros é indispensável a criação de uma interface gráfica, que ajudará não só na qualidade visual do projeto, mas também na usabilidade do software como um todo. Outra questão diz respeito a Portabilidade do Sistema, uma vez que no mundo moderno a grande maioria dos dispositivos é portátil, seria ideal ter um aplicativo como este funcionando em ambientes Android e/ou IoS.

Apêndice A

Manual de utilização do Software

A.1 Instalação de bibliotecas e ferramentas necessárias

Para que seja possível utilizar plenamente as funções desenvolvidas ao longo deste projeto, será necessário que algumas bibliotecas da *Python* sejam instaladas. Além disso, recomendamos fortemente o uso do *Pycharm* para a execução desta ferramenta.

Nossa primeira necessidade, é a linguagem de programação *Python 3.7* disponível em: Python 3.7. Depois de baixar, instale seguindo as configurações recomendadas.

Em seguida, instale a versão *Pycharm Community 2021.2* disponível neste link

Depois, clone o repositório do Projeto que está disponível no Git hub, e pode ser encontrado aqui

Quando terminar de seguir os passos acima, inicie a configuração do ambiente para tornar possível a execução do programa. Caso deseje executar utilizando o *Pycharm*, abra-o e selecione *File* e em seguida *Open*. Selecione o Repositório clonado e abra o projeto. Agora clique novamente em *File* e em seguida em *Settings* como ilustra a figura a seguir.

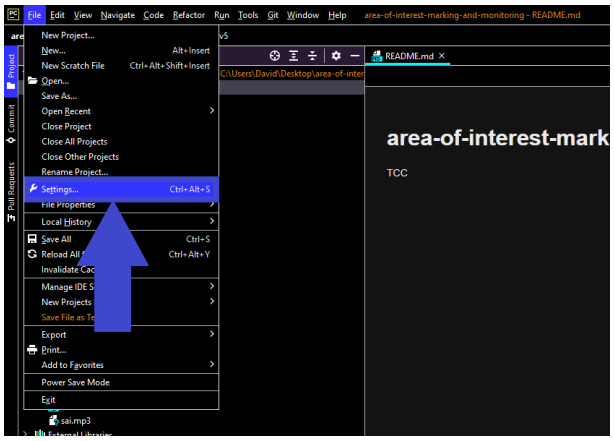

Figura A.1: Configurando o ambiente de execução

Depois, selecione a área *Project* e clique em *Python Interpreter*

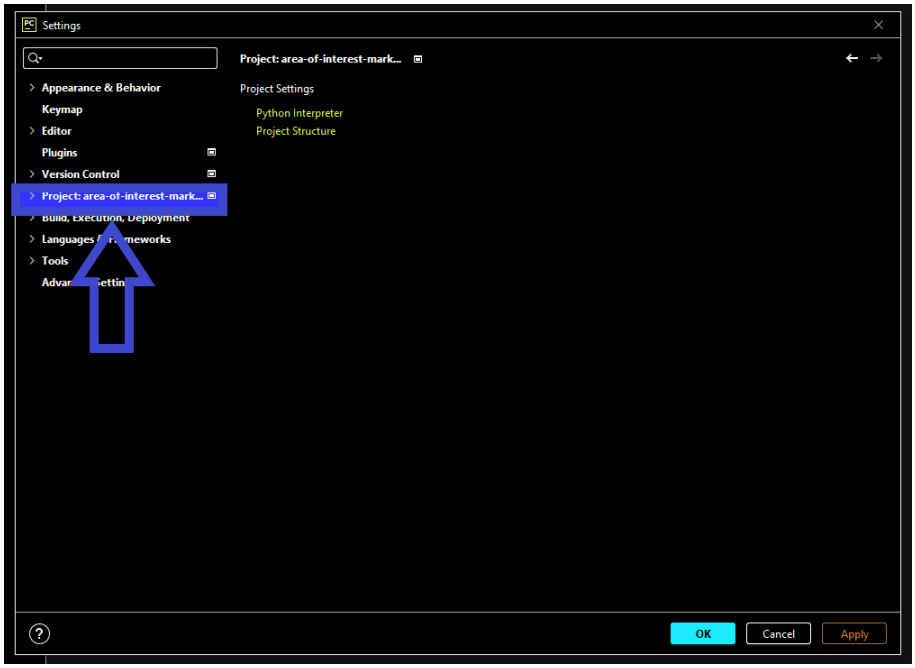

Figura A.2: Configurando o ambiente de execução

Na tela seguinte, clique na pequena engrenagem e depois em Add

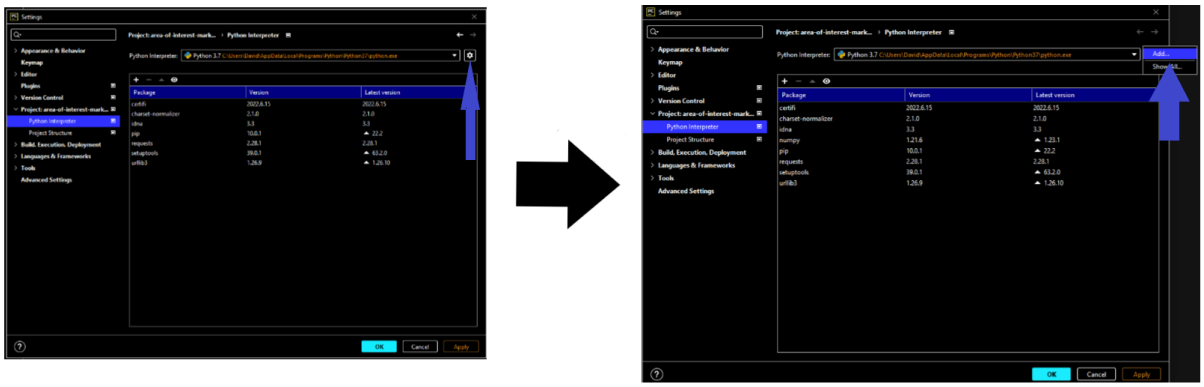

Figura A.3: Configurando o ambiente de execução

Selecione a opção *Virtualenv Enviroment* e em *Base Interpreter* coloque o local do arquivo executável do Python 3.7. Caso você tenha instalado a linguagem com as configurações sugeridas pelo site da mesma, o endereço será praticamente idêntico que aparece na imagem A.4 abaixo

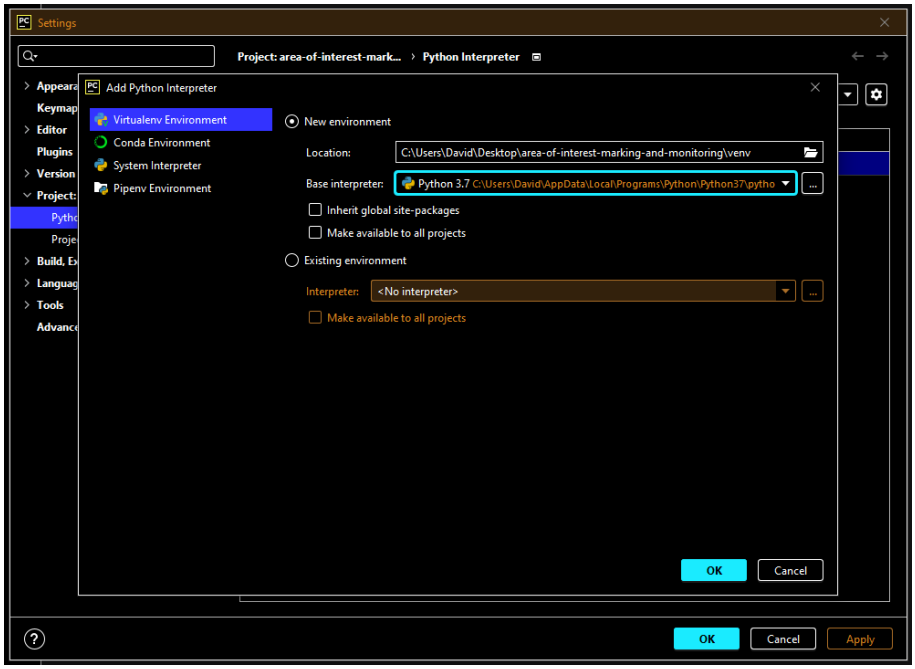

Figura A.4: Configurando o ambiente de execução

Para que o nosso programa seja executado de forma correta, precisamos de uma série de bibliotecas da linguagem *Python*. Para adicionar algumas delas, clique no pequeno sinal de mais que aparece na tela como mostrado na figura a seguir.

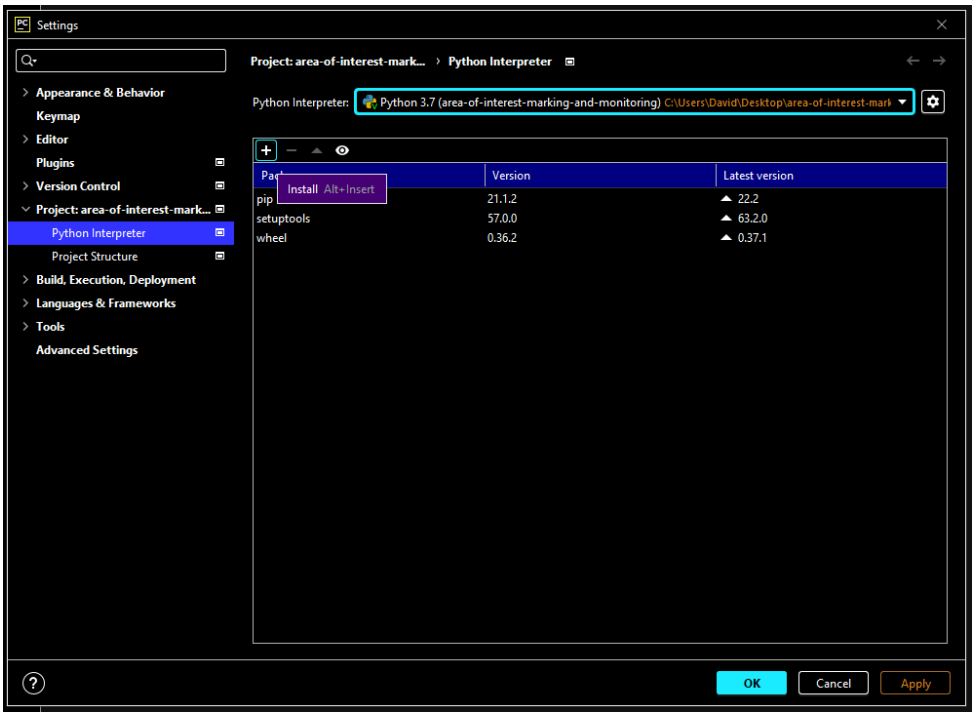

Figura A.5: Adicionando Bibliotecas

agora basta digitar o nome das bibliotecas que queremos adicionar, vamos começar pela *OpenCV*. Você pode realizar essa instalação digitando *cv2*, mas caso dê erro, digite *opencv-python* e depois clique em instalar pacote.

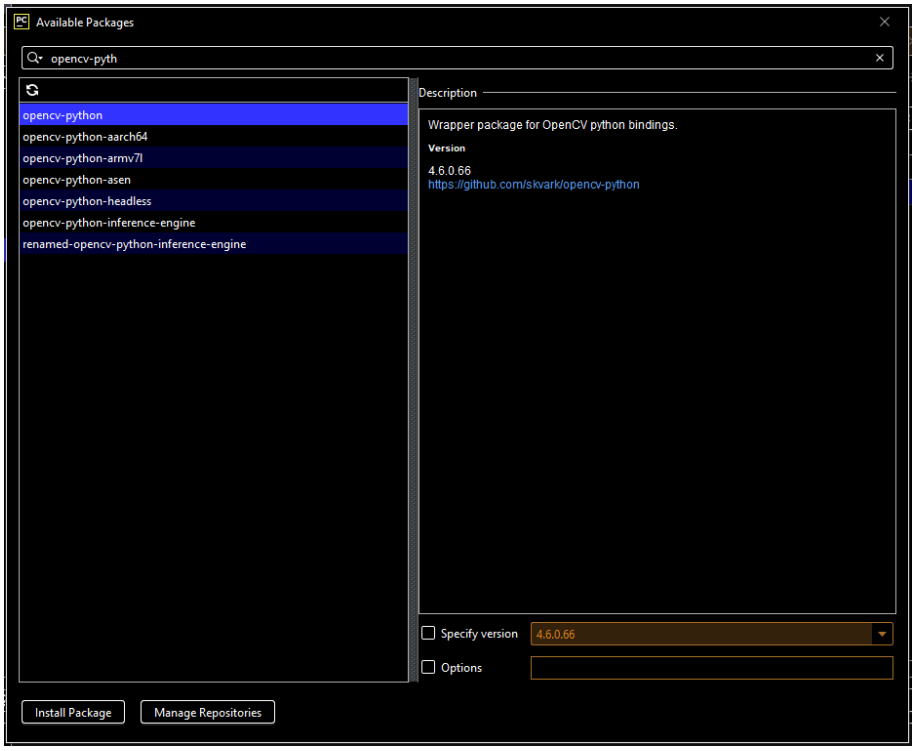

Figura A.6: Configurando o ambiente de execução

Você deve repetir o processo para as bibliotecas

pygame

time

Com as bibliotecas instaladas, é hora de instalar as dependências da *Yolov5*. Para isso, abra o terminal do *Pycharm* como mostrado na figura A.7 ou simplesmente pressione *Alt* + *F12*.

em seguida execute os dois comandos a seguir no terminal e aguarde

cd yolov5

pip install -r requirements.txt

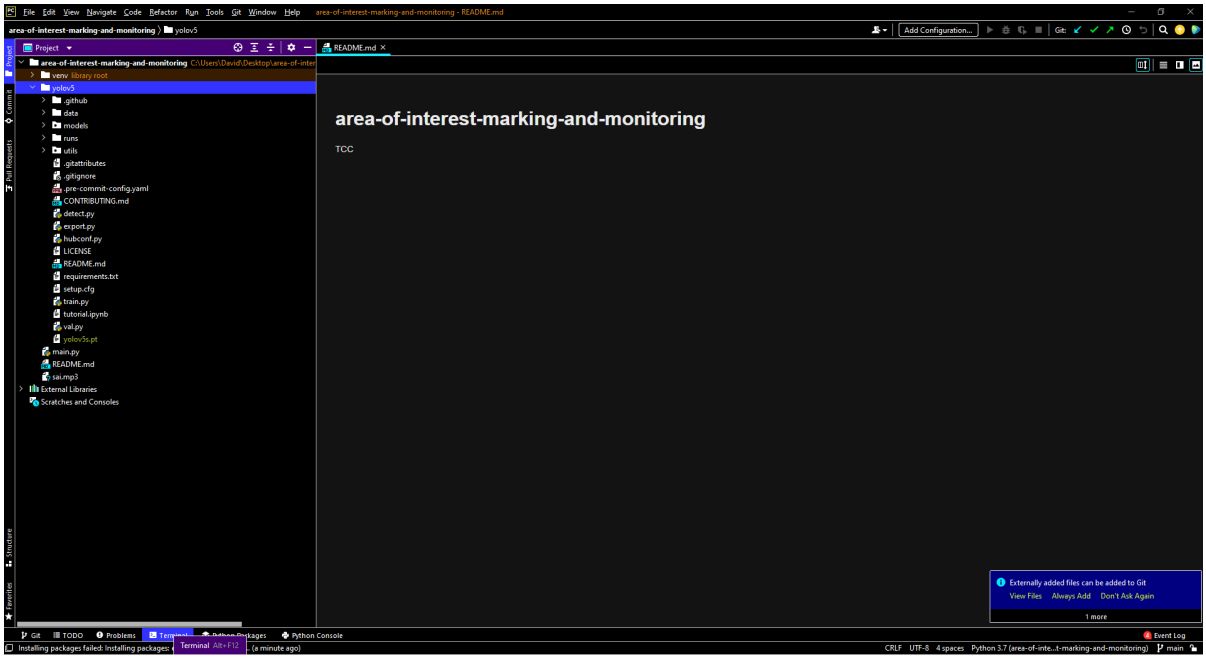

Figura A.7: Configurando o ambiente de execução

Depois que a instalação for concluída, você pode executar os comandos no terminal do *Pycharm* para rodar a ferramenta em tempo real.

Caso você deseje executar sem a IDE, clone o repositório e instale as seguintes bibliotecas em sua máquina

opencv

pygame

time

em seguida, vá até a pasta da yolov5 utilizando o comando

cd yolov5

no terminal do Windows e execute o comando a seguir

pip install -r requirements.txt

Ao seguir os passos acima sem nenhum problema, o ambiente está configurado e estamos prontos para executar a ferramenta.

A.2 Funções implementadas

Função

left_click_detect	*Detecta os cliques com o botão esquerdo do mouse, cada clique é uma posição que será utilizada para desenhar uma figura posteriormente. Ao clicar essa função adiciona as coordenadas a uma lista*

Exemplo de Uso

```
1 cv2.setMouseCallback('Frame', left_click_detect, points)
```

Parametros

event	Um valor booleano que assume valor positivo quando algum botão do mouse é pressionado
x	O valor da coordenada x onde o clique foi executado
y	Analogamente, o valor da coordenada y onde o clique foi executado
points	Lista dos pontos(cliques realizados) que são usados para desenhar a figura

Função

polygon_ray_casting_algorithm	*Função para calcular se um ponto está dentro do Polígono seguindo a metologia explicada no capítulo de Geometria Computacional.*

Exemplo de Uso

```
1 polygon_ray_casting_algorithm(polly, pm)
```

Parametros

polly	Lista contendo as coordendas do polígono
pm	Variável com as coordenadas x e y do que seria aproximadamente a posição do pé das pessoas na cena. A usamos para verificar se o ponto que ela forma está dentro ou fora do Polígono

A.3 Principais Métodos utilizados(por biblioteca)

Biblioteca

1	OpenCV

Método

EVENT_LBUTTONDOWN	Utilizada para ajudar a checar se houve clique com o botão esquerdo em algum momento.
VideoCapture(r"local_do_arquivo")	Usada para abrir um vídeo. Para isso, basta apenas passar o caminho absoluto do arquivo como parâmetro entre aspas duplas.
isOpened()	Usada para checar se o vídeo ainda está aberto, ou seja, se ainda está sendo executado.
read()	Usamos essa função para capturar cada frame do vídeo.
polylines(frame, polygon, flag, cor, t)	Função utilizada para desenhar o Polígono na tela. Essa função recebe como parâmetro: a imagem, um array com os pontos do polígono, uma flag para indicar se o Polígono é fechado ou não, uma cor e um tamanho para a espessura da linha poligonal.

circle(frame, pm(x), pm(y), r, cor, t)	Função utilizada para desenhar os círculos próximos aos pés das pessoas. Como parâmetros temos cada frame do vídeo, as coordenadas x e y do ponto médio, o raio do círculo, a cor e a espessura.

Biblioteca

```
1 pygame
```

Método

mixer.init()	Para iniciar o mixer da biblioteca.
mixer.music.load(r'local_do_arquivo')	Utilizada para carregar o arquivo de som a ser reproduzido.
mixer.music.play()	Usada para executar o arquivo de som quando alguém cruza a linha poligonal.

Biblioteca

```
1 time
```

Método

time.sleep(tempo_em_segundos)	Para definir por quanto tempo o arquivo de som deve ser executado. Usamos esta função para definir por quanto tempo o alerta sonoro deve ser executado quando a pessoa entra no Polígono. Escolhemos o tempo de 0.2 segundos.

Biblioteca

```
1 numpy
```

Método

numpy.int32()	Usado definir um array a partir dos pontos que formarão a figura a ser desenhada.

A.4 Opções de execução

A ferramenta pode ser executada de várias formas dependendo do valor passado após o *source* na linha de comando do terminal:

Comando

```
python detect.py --classes 0 --source 0
```

Valor passado:

0	Para executar a ferramenta em tempo real utilizando os frames capturados por sua **Webcam**.

Comando

```
python detect.py --classes 0 --source nome_do_arquivo.jpg
```

Valor passado:

nome_do_arquivo.jpg	Para executar a ferramenta em uma imagem específica na mesma pasta do *detect.py*.

Comando

```
python detect.py --classes 0 --source nome_do_arquivo.mp4
```

Valor passado:

nome_do_arquivo.mp4	Para executar a ferramenta em vídeos já gravados e no mesmo diretório do *detect.py*.

Comando

```
python detect.py --classes 0 --source pasta\nome_do_arquivo.mp4
```

Valor passado:

pasta\nome_do_arquivo.mp4	Para executar a ferramenta em Arquivos que estão em uma determinada pasta.

Comando

```
python detect.py --classes 0 --source pasta\*\nome_do_arquivo.mp4
```

Valor passado:

pasta*\nome_do_arquivo.mp4	Para executar a ferramenta em Arquivos Globais dentro de vários outros diretórios(*).

O programa funciona para os seguintes formatos de arquivos

- **Imagem**

  bmp, jpg, jpeg, png, tif, tiff, dng, webp, mpo

- **Vídeo**

  mov, avi, mp4, mpg, mpeg, m4v, wmv, mkv

Comando

```
python detect.py --classes 0 --source link_da_live
```

Valor passado:

link_da_live	Para executar a ferramenta em *lives* do Youtube.

Apêndice B

Uso da ferramenta para realização dos Experimentos qualitativos

Como mencionamos, os experimentos foram realizados em **vídeos já gravados**. A seguir demonstramos como foram realizados tais experimentos, utilizando o *Pycharm* como IDLE:

Inicialmente, precisamos passar o local do arquivo em vídeo que queremos abrir no arquivo *main.py* como ilustrado na figura B.1

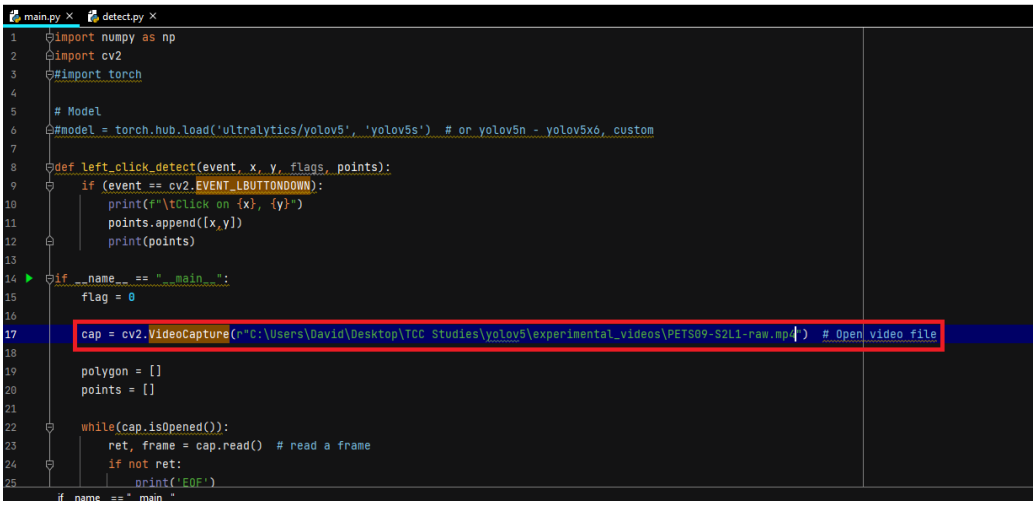

Figura B.1: Local do vídeo que queremos executar o modelo.

depois de fornecer o local, pressionamos o botão *Run* do *Pycharm* para que o vídeo seja executado. Assim que o vídeo abrir, clicamos na tela usando o botão esquerdo do mouse para desenhar o Polígono. Cada clique representa um vértice da Figura. Depois de clicar o número de vértices que desejamos ter, a tecla "P"no teclado é pressionada e uma figura será desenhada na tela através da conexão dos pontos de cada clique. Nos certificamos de desenhar um Polígono Simples em cada um dos experimentos, como no exemplo ilustrado na figura B.2

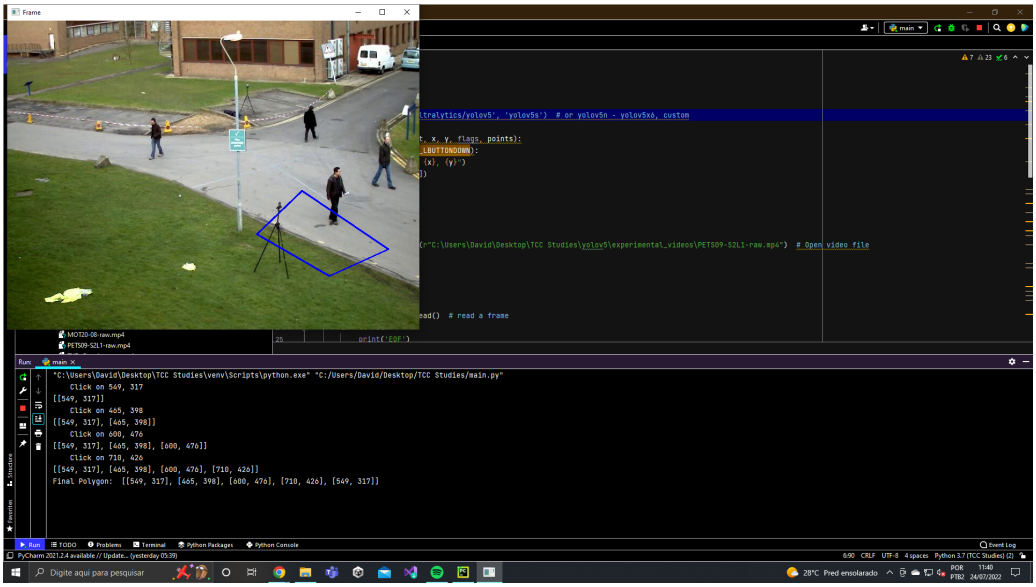

Figura B.2: Desenho de um Polígono Simples.

Depois de desenhar na tela, a interface contendo o vídeo é fechada e em seguida pegamos os pontos que o algoritmo retorna. Eles se encontram na saída fornecida pelo *Pycharm* como é exibido na figura B.3

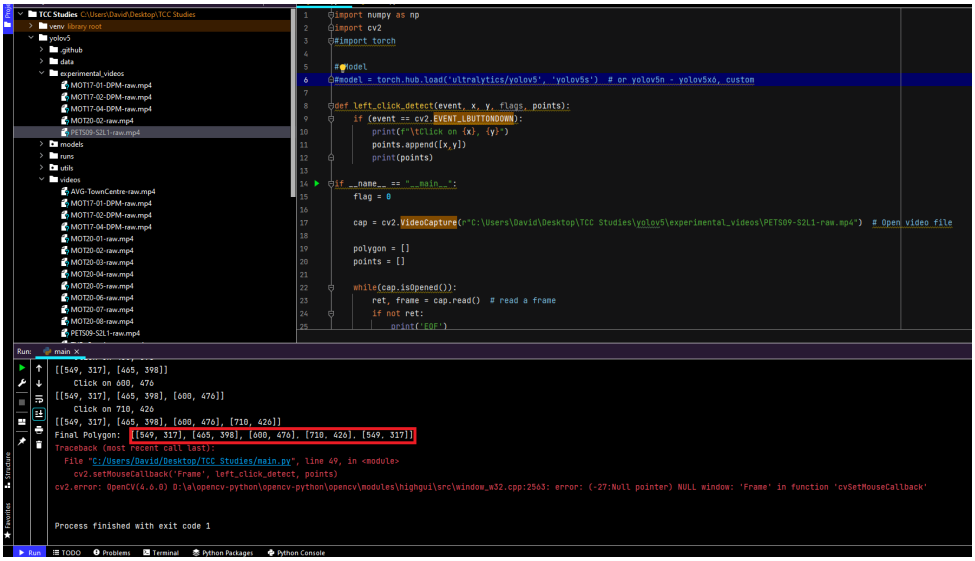

Figura B.3: Pegando as coordenadas do Polígono.

selecionamos estas coordenadas, copiamos e colamos em uma variável do *detect.py*. Como ilustrado em B.4

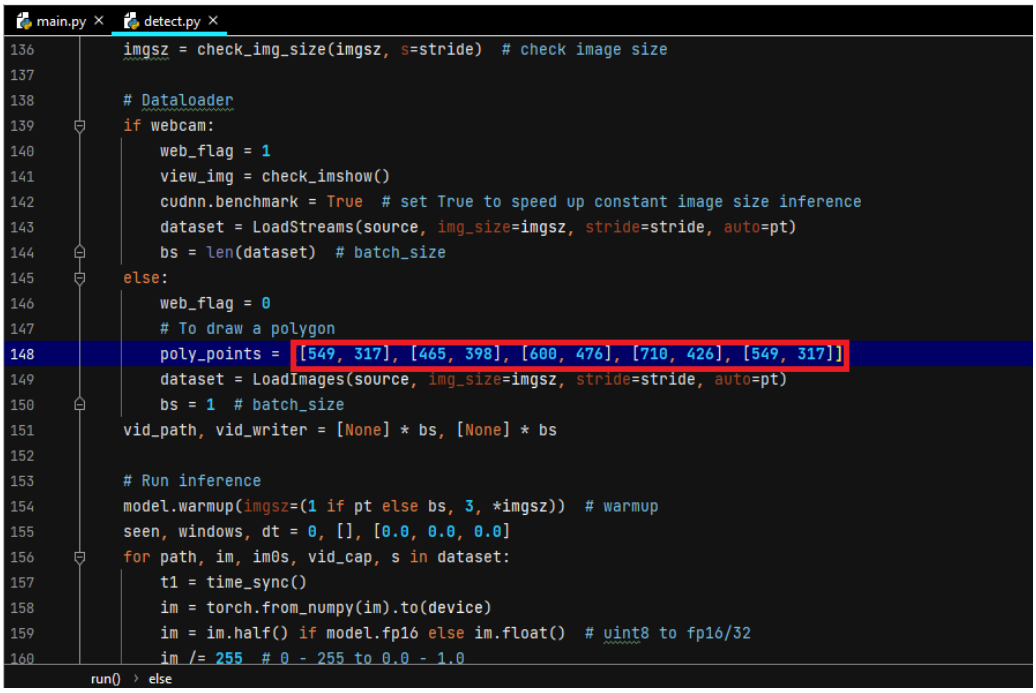

Figura B.4: Substituindo o valor da lista que representa o Polígono.

Ao substituir o valor da variável *poly_points* pelo valor recém copiado, o vídeo que será processado pela Yolov5 já contém o polígono desenhado e todas as informações necessárias para realizar as inferências, isso nos dá tempo e praticidade para realizar os experimentos qualitativos. Para rodar a ferramenta nesses vídeos, vamos até a pasta yolov5 e digitamos o seguinte comando:

python detect.py ——classes 0 ——source experimental_videos\PETS09-S2L1-raw.mp4

Note que ——**source experimental_videos\PETS09-S2L1-raw.mp4** é o caminho do arquivo, portanto, esse comando mudava caso o vídeo estivesse em um diretório diferente. Como utilizamos o mesmo diretório para armazenar os vídeos de teste, precisavámos mudar apenas o que aparece a partir do nome da pasta "experimental\videos". Para realizar os experimentos em cada arquivo, mudavámos seu local na *main.py*, repetíamos os passos comentados acima e apertavamos Enter. A figura B.5 ilustra um dos comandos que foram utilizados

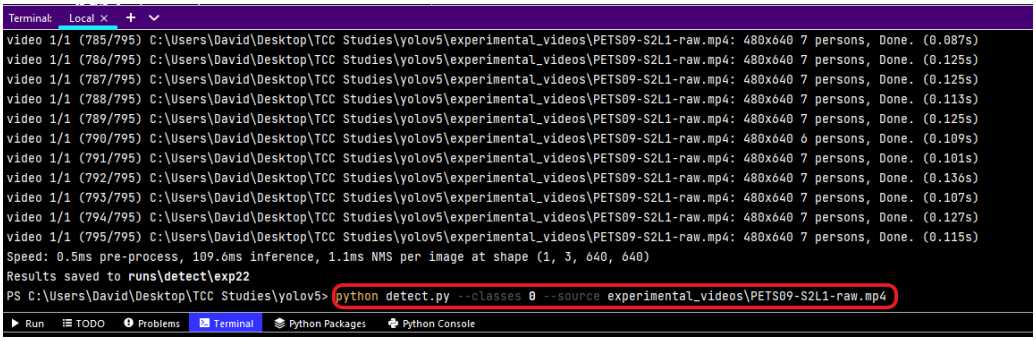

Figura B.5: Comando sendo escrito no terminal do *Pycharm*

depois que a execução é finalizada, os resultados são armazenados na pasta "runs\detect\exp". Esses vídeos de resultado já ilustram o funcionamento do algoritmo, exibindo o ponto médio em cor verde caso eles estejam fora do polígono e na cor vermelha caso eles adentrem na área demarcada. Um exemplo é mostrado na figura B.6

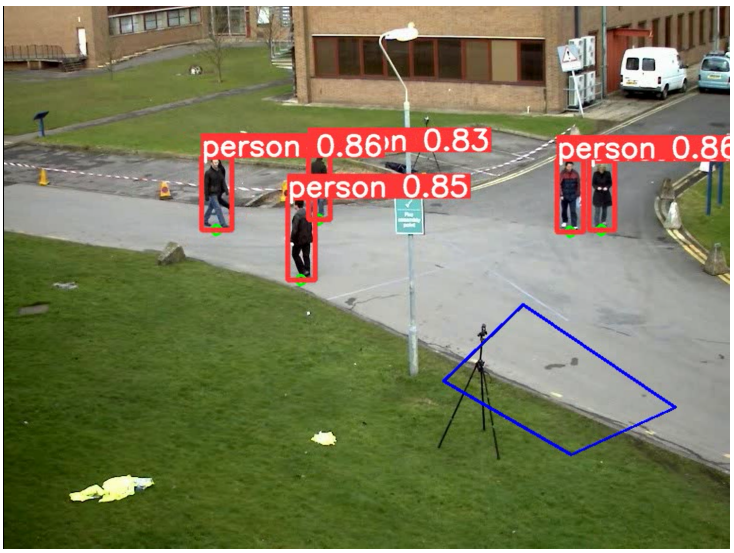

Figura B.6: Vídeo já processado e com as inferências realizadas

Referências bibliográficas

Yali Amit, Pedro Felzenszwalb, and Ross Girshick. Object detection. *Computer Vision: A Reference Guide*, pages 1–9, 2020.

Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

Ali Borji, Ming-Ming Cheng, Qibin Hou, Huaizu Jiang, and Jia Li. Salient object detection: A survey. *Computational visual media*, 5(2):117–150, 2019.

Gary Bradski and Adrian Kaehler. Opencv. *Dr. Dobb's journal of software tools*, 3:120, 2000.

Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7291–7299, 2017.

Narayana Darapaneni, Shrawan Kumar, Selvarangan Krishnan, Arunkumar Rajagopal, Anwesh Reddy Paduri, et al. Implementing a real-time, yolov5 based social distancing measuring system for covid-19. *arXiv preprint arXiv:2204.03350*, 2022.

Alex Dominguez-Sanchez, Miguel Cazorla, and Sergio Orts-Escolano. Pedestrian movement direction recognition using convolutional neural networks. *IEEE transactions on intelligent transportation systems*, 18(12):3540–3548, 2017.

Heng Fan, Liting Lin, Fan Yang, Peng Chu, Ge Deng, Sijia Yu, Hexin Bai, Yong Xu, Chunyuan Liao, and Haibin Ling. Lasot: A high-quality benchmark for large-scale single object tracking. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5374–5383, 2019.

Pedro F Felzenszwalb and Daniel P Huttenlocher. Efficient graph-based image segmentation. *International journal of computer vision*, 59(2):167–181, 2004.

Luiz Henrique Figueiredo and Paulo Cezar Pinto Carvalho. *Introdução à geometria computacional*, volume 13. IMCA, 2000.

Reagan L Galvez, Argel A Bandala, Elmer P Dadios, Ryan Rhay P Vicerra, and Jose Martin Z Maningo. Object detection using convolutional neural networks. In *TENCON 2018-2018 IEEE Region 10 Conference*, pages 2023–2027. IEEE, 2018.

Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3354–3361. IEEE, 2012.

Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Region-based convolutional networks for accurate object detection and segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 38(1):142–158, 2015.

Rafael C Gonzalez and Richard C Woods. *Processamento digital de imagens* . Pearson Educación, 2009.

Kevin Gurney. *An introduction to neural networks*. CRC press, 2018.

Simon Haykin. *Neural networks and learning machines, 3/E*. Pearson Education India, 2009.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE transactions on pattern analysis and machine intelligence*, 37(9):1904–1916, 2015.

Michael Henle. *A combinatorial introduction to topology*. Courier Corporation, 1994.

Alexey Grigorevich Ivakhnenko. The group method of data handling, a rival of the method of stochastic approximation. *Soviet Automatic Control*, 13(3):43–55, 1968.

Valentin Grigorevich Ivakhnenko, Alekse. *Cybernetics and forecasting techniques*, volume 8. American Elsevier Publishing Company, 1967.

Ujjwal Khanna and Anjali Awasthi. Neural network and internet of things implementation to aid pedestrian safety.

Stefan Klein, Josien PW Pluim, Marius Staring, and Max A Viergever. Adaptive stochastic gradient descent optimisation for image registration. *International journal of computer vision*, 81(3):227–239, 2009.

Matej Kristan, Ales Leonardis, Jiri Matas, Michael Felsberg, Roman Pflugfelder, Luka ˇCehovin Zajc, Tomas Vojir, Goutam Bhat, Alan Lukezic, Abdelrahman Eldesokey, et al. The sixth visual object tracking vot2018 challenge results. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, pages 0–0, 2018.

Laura Leal-Taixé, Anton Milan, Ian Reid, Stefan Roth, and Konrad Schindler. Motchallenge 2015: Towards a benchmark for multi-target tracking. *arXiv preprint arXiv:1504.01942*, 2015.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*, pages 319–345. Springer, 1999.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

Dengsheng Lu and Qihao Weng. A survey of image classification methods and techniques for improving classification performance. *International journal of Remote sensing*, 28(5): 823–870, 2007.

Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

Tom M Mitchell and Tom M Mitchell. *Machine learning*, volume 1. McGraw-hill New York, 1997.

Berndt Müller, Joachim Reinhardt, and Michael T Strickland. *Neural networks: an introduction*. Springer Science & Business Media, 1995.

Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

Kemal Oksuz, Baris Can Cam, Sinan Kalkan, and Emre Akbas. Imbalance problems in object detection: A review. *IEEE transactions on pattern analysis and machine intelligence*, 43 (10):3388–3415, 2020.

Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

Rafael Padilla, Sergio L Netto, and Eduardo AB Da Silva. A survey on performance metrics for object-detection algorithms. In *2020 international conference on systems, signals and image processing (IWSSIP)*, pages 237–242. IEEE, 2020.

Franco P Preparata and Michael I Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Michael Ian Shamos. *Computational geometry.* Yale University, 1978.

Aized Amin Soofi and Arshad Awan. Classification techniques in machine learning: applications and issues. *Journal of Basic & Applied Sciences*, 13:459–465, 2017.

Mate Szarvas, Akira Yoshizawa, Munetaka Yamamoto, and Jun Ogata. Pedestrian detection with convolutional neural networks. In *IEEE Proceedings. Intelligent Vehicles Symposium, 2005.*, pages 224–229. IEEE, 2005.

Lipo Wang. *Support vector machines: theory and applications*, volume 177. Springer Science & Business Media, 2005.

Zhou Wang and Xinli Shang. Spatial pooling strategies for perceptual image quality assessment. In *2006 International Conference on Image Processing*, pages 2945–2948. IEEE, 2006.

Aston Zhang, Zachary C Lipton, Mu Li, and Alexander J Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.