



APLICAÇÃO DE VISÃO COMPUTACIONAL PARA RASTREAMENTO E CONTAGEM DE VEÍCULOS EM RODOVIAS

Matheus Molin de Andrade

Projeto de Graduação apresentado ao Curso de Engenharia Eletrônica e de Computação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Flávio Luis de Mello

Rio de Janeiro

Outubro de 2020

APLICAÇÃO DE VISÃO COMPUTACIONAL PARA RASTREAMENTO E CONTAGEM
DE VEÍCULOS EM RODOVIAS

Matheus Molin de Andrade

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO
DE ENGENHARIA ELETRÔNICA E DE COMPUTAÇÃO DA ESCOLA
POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO
PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE
ENGENHEIRO ELETRÔNICO E DE COMPUTAÇÃO

Autor:


Matheus Molin de Andrade

Orientador:


Prof. Flávio Luis de Mello, D. Sc.

Examinador:


Diego Leonel Cadette Dutra, DSc.

Examinador:


Tatiana Sciammarella, MSc.

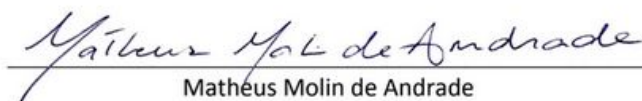
Rio de Janeiro – RJ, Brasil

Outubro de 2020

Declaração de Autoria e de Direitos

Eu, *Matheus Molin de Andrade* CPF 120.327.987-60, autor da monografia *aplicação de visão computacional para rastreamento e contagem de veículos em rodovias*, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.
2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e idéias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.
3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.
4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.
5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.
6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.
7. Por ser verdade, firmo a presente declaração.


Matheus Molin de Andrade

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

Escola Politécnica – Departamento de Eletrônica e de Computação

Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária

Rio de Janeiro – RJ CEP 21949-900

Este exemplar é de propriedade da Universidade Federal do Rio de Janeiro, que poderá incluí-lo em base de dados, armazenar em computador, microfilmear ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es).

DEDICATÓRIA

Dedico à minha esposa, família e amigos que nunca me deixaram desistir.

AGRADECIMENTO

Agradeço primeiramente a minha mãe em especial, que sempre me apoiou e tentou me dar a maior tranquilidade possível para que eu fizesse esse curso no meu tempo. Todo apoio que ela me dá na vida é sentido por mim e, se cheguei até aqui neste curso, foi por que ela foi meu norte.

Agradeço à minha esposa que sempre me trouxe incentivo para que eu mantivesse o ânimo para concluir este ciclo da minha vida, principalmente nos momentos em que estive mais fraco e com vontade de desistir. Ver o exemplo dela e a dedicação que ela dá aos objetivos foi o motor que me fez continuar.

Agradeço aos meus amigos da faculdade que tiraram o máximo do peso de fazer uma faculdade complexa e trabalhosa. Seja ao ajudarem a entender tópicos de matérias abstratas ou apenas por dedicarem alguns minutos para tomarem um café no Burguesão após um dia cansativo. No fim, as boas companhias do dia a dia que nos mantêm firmes. Eles sabem o quão foi tormentoso pra mim completar essa etapa e foram eles que me mantiveram inteiro por esses longos períodos.

Por fim agradeço aos professores e pesquisadores da UFRJ, que lutam para que a instituição se mantenha como um suspiro de ar puro em meio a um ambiente nocivo que está cada vez mais ganhando espaço neste país. Que possam seguir firmes, mostrando a importância da ciência para os nossos futuros.

RESUMO

Cidades estão cada vez maiores e com isso seus desafios também crescem, um destes é o tráfego. Extensos engarrafamentos fazem que todos os dias pessoas percam horas de seus dias para chegarem e voltarem de seus destinos. É importante que novas soluções para auxiliarem no planejamento de trânsito surjam. Usando visão computacional e ferramentas de *Deep Learning*, este projeto se propõe a estruturar uma ferramenta para contagem de veículos em variados tipos de rodovia e diferentes cenários gravados por câmeras de segurança. Com essa ferramenta será possível coletar dados de tráfego de uma rodovia para que estes sejam posteriormente analisados sem que haja necessidade de separar um indivíduo para esta tarefa. Para alcançar este objetivo foi utilizado *YOLO*, para identificação, junto com *DeepSORT*, para o rastreamento dos objetos que serão contabilizados ao atravessarem uma região de interesse. Por fim, o desempenho deste algoritmo foi comparado ao desempenho de voluntários na tarefa de contagem para verificar se este, além de acurado, se mostra superior a humanos na realização desta tarefa.

Palavras-Chave: YOLO, Tensorflow, Contagem de Veículos, Rastreamento de Objetos, Deep Learning, DeepSORT

ABSTRACT

Every day new challenges appear in the daily life of big cities, one of them is high traffic. Long traffic jams make people waste hours of the day reaching and getting back from their destinations. It's important that new solutions are created to improve the traffic reorganization in order to reduce traffic jams in these cities. Using computer vision and Deep Learning tools this project aims to create an algorithm capable of identifying, tracking and counting vehicles in roads, highways and streets, using videos from security cameras. This tool will collect data from traffic for further analysis without using a human in this process. To implement this algorithm YOLO was used together with DeepSORT, making the counting of vehicles possible within a region of interest from video images. Finally, the algorithm was tested in different situations to show whether the accuracy showed satisfactory results in comparison to humans in the same task.

Key-words: YOLO, Tensorflow, Vehicle Counting, Object Tracking, Deep Learning, DeepSORT

SIGLAS

CNN – *Convolutional Neural Network*

R-CNN – *Region CNN*

ReLU – *Rectified Linear Unit*

SORT - *Simple Real Time Tracker*

UFRJ – Universidade Federal do Rio de Janeiro

YOLO – *You Only Look Once*

Sumário

Capítulo 1	14
Introdução	14
1.1 – Tema	14
1.2 – Delimitação	14
1.3 – Justificativa	14
1.4 – Objetivos	15
1.5 – Metodologia	15
1.6 – Descrição	16
Capítulo 2	17
Fundamentação Teórica	17
2.1 – Redes Neurais	17
2.2 – Funções de Ativação	18
2.3 – Redes Convolucionais	21
2.4 – YOLO - You Only Look Once	25
2.5 – Non-maximum suppression	27
2.6 – YOLOv3 e YOLOv4	29
2.7 – TensorFlow	30
2.8 – Rastreamento de Objetos	31
2.9 – O DeepSort	33
2.10 – A rotina de associação de trilhas	35
2.11 – O processo de contagem de veículos	36
Capítulo 3	38
A implementação	38
3.1 – Arquitetura do algoritmo	38
3.2 – Implementação da detecção	42
3.3 – Implementação do rastreador	45
3.4 – Implementação do contador	49

Capítulo 4	53
Resultados	53
4.1 – Descrição do experimento de contagem	53
4.2 – Considerações sobre os resultados obtidos	54
4.3 – Resultados Obtidos	56
4.4 – Resultado em vídeo noturno	59
 Capítulo 5	 61
Conclusão	61
5.1 – Trabalhos Futuros	62
 Referências Bibliográficas	 63

Lista de Figuras

Figura 2.1 – Ilustração do modelo de um neurônio artificial	17
Figura 2.2 – Problema de segmentação utilizando função linear e não-linear	19
Figura 2.3: Interações Esparsas em redes convolucionais	22
Figura 2.4: Processo de convolução entre um input e o filtro, em duas dimensões.	23
Figura 2.5: Max Pooling com stride de 2.	24
Figura 2.6: Processo de extração e classificação após max pooling com strides.	24
Figura 2.7: Processo de classificação do YOLO.	26
Figura 2.8: Processo de non-maximum suppression	27
Figura 2.9: Intersection over Union (IOU)	28
Figura 2.10: Vetor criado para cada objeto identificado em um frame de entrada	29
Figura 2.11: Exemplificação de um tensor para imagens coloridas	30
Figura 2.12: Carro ocluso ainda identificável.	32
Figura 2.13: Carros oclusos (esquerda) e em câmeras de baixa qualidade (direita)	34
Figura 2.14: Mesmo carro com identificação trocada	37
Figura 3.1: Diagrama de Atividades do Algoritmo	39
Figura 3.2: Estrutura de diretórios do projeto	40
Figura 3.3: Diagrama da classe Sort	45
Figura 3.4: Definição da Região de Interesse	50
Figura 3.5: Verificação de retas concorrentes com método de sentido horário	51
Figura 4.1: Caminhonete gerando duas detecções	55
Figura 4.2: Vídeos analisados para o experimento de contagem	55
Figura 4.3: Gráfico de resultados referentes ao vídeo 1	57
Figura 4.4: Gráfico de resultados referentes ao vídeo 2	57
Figura 4.5: Gráfico de resultados referentes ao vídeo 3	58
Figura 4.6: Moto em vídeo noturno	59
Figura 4.7: Gráfico de resultados referentes ao vídeo noturno	60

Capítulo 1

Introdução

1.1 – Tema

O tema deste trabalho é a aplicação de ferramentas de visão computacional para realizar um projeto de contador de veículos a ser executado em vídeos de câmeras de segurança. Utilizando YOLO[38] como a ferramenta de detecção de veículos e DeepSORT[46] para o rastreamento, um algoritmo será proposto. Este algoritmo será testado e analisado, buscando resultados com o menor número de erros de contagem na comparação com humanos.

1.2 – Delimitação

Este estudo utilizará a rede disponibilizada pela equipe do YOLO que foi treinada com a base de dados COCO para 80 classes[24]. Dentre as 80 classes, as principais classes de veículos como carros, motos, bicicletas, ônibus e caminhões estão disponíveis. Apesar de todas as 80 classes disponibilizadas, este projeto reconhecerá e contabilizará apenas veículos.

1.3 – Justificativa

O deslocamento cotidiano é uma realidade de quem mora em cidades grandes ou até mesmo em áreas rurais, onde raramente se tem a possibilidade de estar próximo do local de trabalho. Com isso, horas são gastas em transportes públicos ou carros para que os destinos sejam alcançados, muitas vezes em longos engarrafamentos. Além de incentivar o transporte público, é importante que os governos tomem outras medidas mais rápidas para mitigar esse problema que está cada vez maior em metrópoles. A análise de tráfego para mudança de rotas e sinalizações, alteração de tempos de semáforos e planejamento de novas rodovias são

algumas das maneiras de melhorar, ainda que momentaneamente, a atual situação enfrentada cotidianamente por moradores destes locais[31]. Estes métodos são rápidos e mais simples quando comparados a criação de novos meios de transporte, por exemplo.

O planejamento dessas soluções mais rápidas passa intrinsecamente pelo conhecimento do local e dos dados de trânsito da região. Este projeto foca exatamente na em fornecer uma ferramenta capaz de analisar imagens de vídeo, obtidas de câmeras de tráfego ou segurança, e gerar dados para serem analisados durante esses projetos. Sem necessitar que várias pessoas sejam alocadas para a tarefa de coleta de dados, um algoritmo capaz de detectar e contar veículos em uma rodovia será proposto para essa função.

O reconhecimento de objetos é uma área bastante explorada na comunidade científica e tecnológica[23][6..], já existindo diversas ferramentas para este propósito. Portanto este projeto usará conceitos de *deep learning*, unindo modelos de redes neurais convolucionais[5] a técnicas de previsão de posicionamento para reconhecer, rastrear e contabilizar veículos em filmagem de ruas e rodovias.

1.4 – Objetivos

O objetivo deste projeto é propor um algoritmo capaz de contabilizar o número de veículos que passam por uma rodovia a partir da filmagem desta. Para isso os objetivos específicos são: (1) Analisar e utilizar uma rede neural de reconhecimento de objetos, (2) tratar o reconhecimento destes objetos de modo a prever o seu deslocamento para que seja possível (3) reconhecer que este objeto ultrapassou uma região de interesse e seja contabilizado.

1.5 – Metodologia

Primeiramente foi realizado um estudo das redes disponíveis para identificação de objetos, verificando a capacidade destas principalmente em dispositivos de baixo desempenho para que seja utilizado um computador pessoal ou até mesmo celulares.

A escolha do YOLO como a arquitetura de rede neural para a execução do projeto foi devido a sua rapidez na hora da detecção[6], além da alternativa do YOLO *Lite*[20] como uma solução mais rápida. A disponibilidade de execução em variados hardwares norteou também a linguagem e o *framework* a serem utilizados, chegando ao Tensorflow[1] que já possui essa portabilidade por também ter sua versão *lite*, voltada para dispositivos móveis.

Após isto, a implementação da rede YOLO no *framework* TensorFlow[1] foi realizada de modo a adaptá-la ao rastreador de objetos DeepSORT.

Por fim, o algoritmo final foi testado em vídeos de rodovias de diferentes qualidades e características. Realizou-se também uma pesquisa com 132 voluntários que foram expostos aos mesmos vídeos, comparando o resultado do algoritmo ao de seres humanos.

1.6 – Descrição

O capítulo 2 apresenta um panorama geral do que são redes neurais e como estas foram utilizadas num primeiro momento e como estas vêm sendo utilizadas atualmente. Além disso este capítulo introduz ferramentas como o TensorFlow, o YOLO e o DeepSORT, que serão utilizados para a implementação deste projeto.

No capítulo 3, o problema da contagem é mostrado, analisando os desafios existentes neste processo. A partir deste ponto a implementação do algoritmo é iniciada, mostrando passo a passo os processos de identificação do veículo até o processo de contabilização deste.

O capítulo 4 é onde o algoritmo é testado em quatro vídeos de cenários distintos para verificar o seu desempenho. Além disso também é mostrado o resultado de uma pesquisa que verificou a capacidade de voluntários de realizarem a mesma tarefa, para os mesmos vídeos, para que estes sejam comparados ao desempenho do algoritmo.

O capítulo 5 é o capítulo de conclusão deste projeto.

Capítulo 2

Fundamentação Teórica

2.1 – Redes Neurais

Para cumprir os objetivos especificados para este projeto, antes de entender as ferramentas que serão abordadas aqui, precisa-se ter um panorama do que é uma rede neural.

Desde 1943, quando Walter Pitts e Warren McCulloch publicaram o seu artigo Logical Calculus of the Ideas Immanent in Nervous Activity[28], onde eles propuseram um modelo matemático para representar neurônios biológicos, pesquisadores vem tentando aplicar um modelo de aprendizado a algoritmos. A tarefa de replicar a complexidade do cérebro humano a máquinas é o principal objetivo do que hoje é conhecido como redes neurais.

A primeira implementação prática que foi utilizada tendo seus conceitos estudados e aplicados até hoje é o algoritmo perceptron [41]. O perceptron é uma rede de duas camadas que através de ajuste em seus pesos internos é capaz de realizar tarefas simples como separar dois segmentos. Apesar de ser uma premissa muito inovadora, o perceptron demorou até que fosse amplamente utilizado em pesquisas e produtos. Um dos principais motivos foi quando em 1969 o livro intitulado *Perceptrons*[29] mostrou que não era possível realizar operações básicas como XOR usando perceptrons.

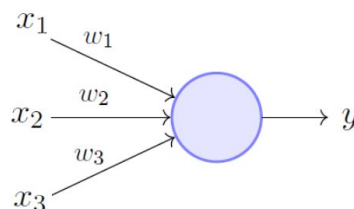


Figura 2.1 – Ilustração do modelo de um neurônio artificial

Fonte: Perceptron Paper [29]

O funcionamento do Perceptron é bem simples e pode ser resumido como uma função soma, onde no caso da figura 2.1 é dada por:

$$y = \sum_i^m X_i w_i + b$$

A constante b é conhecida por *bias* e será um peso definido individualmente para cada neurônio da rede. O *output* vai ser utilizado ou não, dependendo do seu valor e da função de ativação escolhida.

O surgimento do MLP (Multilayered Perceptron) foi o que de fato colocou os perceptrons no foco de pesquisas de toda comunidade científica[9]. Neste novo método a rede era composta de várias camadas que se comunicavam, passando os parâmetros ao longo dos neurônios além de corrigir os pesos utilizados por meio de iterações no processo. O treinamento da rede, que é o processo de achar os pesos ideais para cada neurônio, passou a utilizar o que hoje é conhecido como *feedforward* e *backpropagation*. Ambos métodos são essenciais para que a rede possa se adaptar às entradas, alterando os valores dos pesos internos de cada perceptron de modo eficaz. Os resultados na saída vão se adequando conforme o processo de treinamento vai acontecendo, alterando os pesos dos neurônios internos da rede, mesmo para problemas não-lineares complexos.[9]

2.2 – Funções de Ativação

Por mais que seja uma constante na história humana tentar comparar o cérebro às tecnologias mais complexas existentes de cada período, as redes neurais se baseiam bastante no que é conhecido sobre o funcionamento dos neurônios e suas sinapses. As funções de ativação fazem parte deste processo.

O entendimento atual sobre as sinapses e o funcionamento dos neurônios, como um todo, ainda não são bem estabelecidos, mas o funcionamento individual de cada neurônio, já é o bastante para compreender como os sinais elétricos e químicos alteram a relação intracelular do cérebro humano [13].

As funções de ativação são um paralelo de funcionamento de um neurônio

reagindo a uma sinapse. Uma excitação sináptica em um neurônio provocará uma despolarização deste. Se esta despolarização ultrapassar um limite pré-estabelecido, esse neurônio dará continuidade a transmissão dessa sinapse para um próximo neurônio. As funções de ativação são responsáveis por analisar a resposta de cada “neurônio” de uma rede neural e definir qual será o valor de seu *output*.

Além de restringir o comportamento da saída dos neurônios das camadas internas da rede, as funções de ativação também adicionam não linearidade às redes neurais. A função que realiza o cálculo dos pesos dos neurônios, como mencionado na seção anterior, é uma função soma. Ainda que a rede possua muitas camadas, uma função linear não será capaz de interpretar com acurácia casos reais, que costumam se comportar de maneira não linear, conforme mostrado na figura 2.2. As funções de ativação serão responsáveis, então, pela adição dessa etapa não linear, que fará com que a rede seja capaz de aprender problemas de natureza complexa, adequados para os desafios dos problemas reais.[44]

Apesar disso, as funções lineares também são aplicadas em situações específicas, principalmente em problemas de classificação. As funções lineares costumam estar presentes na última camada para fazer o mapeamento entre *input* e *output* em alguns tipos de rede.

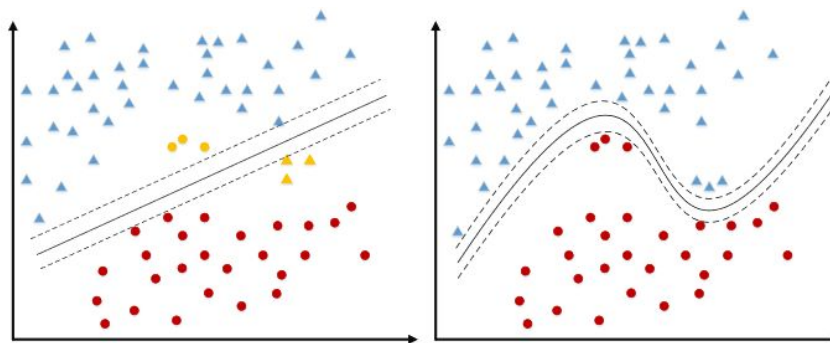


Figura 2.2 – Problema de segmentação utilizando função linear e não-linear

Fonte: Text Algorithm Classification [22]

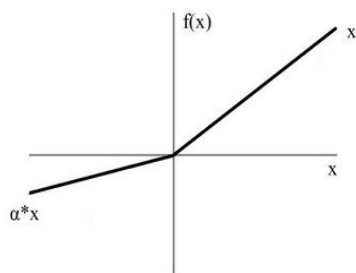
No caso do perceptron, a função de ativação utilizada era o degrau, apresentando um *output*, dado por:

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b \geq 0 \\ 0 & \text{else} \end{cases}$$

Com o tempo, essas funções foram sendo adequadas com o propósito de cada rede, seja para acelerar o processo de treinamento ou aumentar a precisão. Como o foco deste trabalho é o modelo YOLO[38] de rede neural, a função de ativação utilizada será a *leaky ReLU* [33].

A função *leaky ReLU* é uma variação da *Rectified Linear Unit* (ReLU). O ReLU apresenta uma função linear para os valores positivos e zero para os valores negativos. Como é uma função simples, não possuindo divisões ou exponenciação em seu cálculo, o ReLU é uma função de ativação considerada rápida [34]. Uma das limitações do ReLU, é o fato de durante o treinamento ele gerar saídas zero para entradas negativas. Essas saídas zeradas pode bloquear o processo de *backpropagation* porque os gradientes também serão zerados.

O *leaky ReLU* é uma resposta a essa característica de desaparecimento do gradiente [25]. Nessa versão, existe um parâmetro alfa que possui valores bem menores que um, fazendo com que o cálculo do gradiente não seja zero durante o processo de treinamento. Isso evita que o gradiente seja zerado durante as etapas de *feedforwarding* e *backpropagation*.



$$f(x) = \alpha x + x = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

2.3 – Redes Convolucionais

O cérebro humano é muito eficaz na hora de reconhecer padrões do cotidiano. Com a visão é possível identificar qualquer tipo de objeto desde que este já tenha sido visto de antemão. Basta que o objeto em questão tenha um padrão aproximado com um objeto em memória para que seja feita a associação. Se este padrão for muito diferente, pode ser complicado associar o que está sendo visto como um objeto conhecido. Um exemplo disso é uma bola de futebol americano e uma bola de futebol. Se somente um dos esportes é conhecido pelo indivíduo, provavelmente este teria dificuldade de imaginar que a outra bola também é para uso esportivo.

As Redes Neurais Convolucionais, também conhecidas como ConvNets, ou CNN (*Convolutional Neural Networks*), começaram a ganhar protagonismo nos últimos anos, no que tange paradigmas para visão computacional [2]. O principal motivo da adoção desse modelo de rede é devido sua capacidade em reconhecer padrões em imagens, como o cérebro humano, e à sua performance.

Pela definição em [5] Redes Neurais Convolucionais são redes que, como o próprio nome diz, usam a convolução no lugar da multiplicação matricial em pelo menos uma de suas camadas. Essa convolução ocorre entre a imagem de entrada e um filtro, ou *kernel*, que é a janela que percorre a imagem realizando as convoluções.

Para uma convolução no tempo discreto a equação a seguir pode ser utilizada:

$$\sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

como na análise de imagens existe um espaço bidimensional, essa equação será readaptada para o formato a seguir, onde I é a matriz *input* e K a matriz *kernel*:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n).$$

Redes neurais tradicionais usam a multiplicação matricial para fazer a relação entre o *input*, com seus pesos, e o *output*. Isso leva a necessitar que toda unidade de *output* interaja com toda unidade de *input* [5], guardando todos os parâmetros obtidos nessas multiplicações. As redes convolucionais evitam isso ao utilizar o que é chamado de interações esparsas e compartilhamento de parâmetros. As interações esparsas proporcionam às redes convolucionais a capacidade de reduzir sua complexidade, visto que para cada unidade de *output* não é necessário que todas as unidades de *input* sejam utilizadas, conforme é mostrado na figura 2.3.

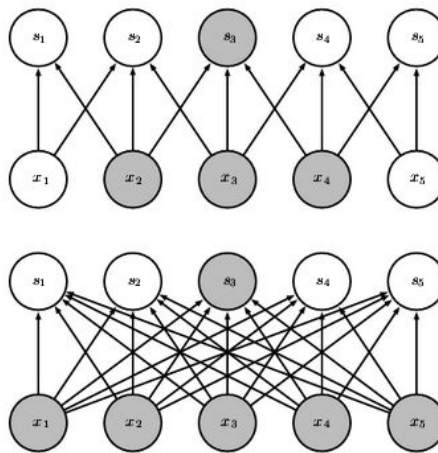


Figura 2.3: Interações Esparsas em redes convolucionais

Fonte: The Deep Learning Book, capítulo 9 [5]

Outra melhoria também relacionada com o que foi mencionado anteriormente é a reutilização de parâmetros. Esse é o nome dado ao fato de durante a convolução ser utilizado um mesmo filtro ao longo de todas as operações. Isto é, todos os *outputs* são obtidos através da convolução dos valores de entrada com um mesmo filtro, que é compartilhado por todas as entradas na obtenção dos outputs da camada.

Para facilitar o entendimento prático da convolução aplicada a imagens, a figura 2.4 exemplifica a etapa de convolução entre um *input* e o *kernel*, ou filtro. O *input* sendo uma imagem preto e branco será composto por vários pixels, que no exemplo são representados por valores entre 0 e 1, onde 1 representa o mais alto brilho e 0 representa sombra. Esse *input* é no formato 5x5 e cada quadrado representa um pixel da imagem.

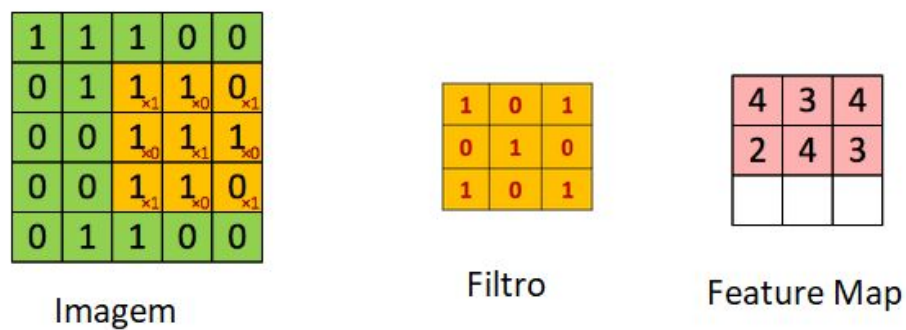


Figura 2.4: Processo de convolução entre um *input* e o filtro, em duas dimensões.

Fonte: The Data Science blog [3]

O *feature map* é o vetor obtido no resultado da operação de convolução entre a entrada e o filtro. Para completar este vetor, o filtro é deslocado ao longo da imagem no processo chamado de *striding*, que é definido com um valor inteiro que define quantos quadrantes serão deslocados para a obtenção do próximo valor a ser preenchido no *feature map*.

Depois de obter todo o feature map é realizado o processo de *pooling*, geralmente o max pooling, que subdivide o feature map em janelas e seleciona o maior valor de cada janela, descartando os outros valores. A figura 2.5 exemplifica visualmente o *max pooling*. Esse processo minimiza a complexidade além de diminuir a chance de *overfitting*[18], que é quando os neurônios passam a ser capazes de decorar o mapeamento de entrada e saída para todos os *inputs* disponíveis no treinamento, mas não conseguem prever para imagens fora daquelas utilizadas no processo de treino.

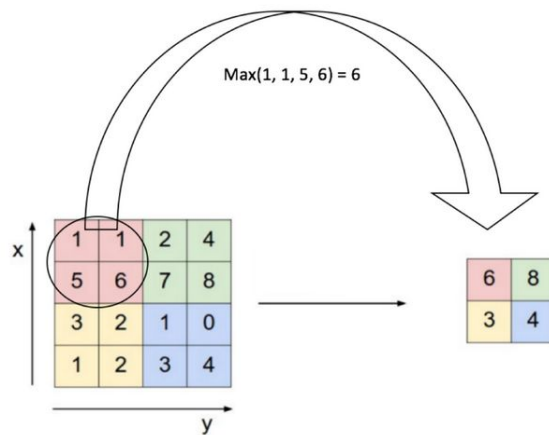


Figura 2.5: Max Pooling com stride de 2.

Fonte: The Data Science blog [3]

As técnicas especificadas nesta seção trazem um panorama geral do funcionamento das redes convolucionais. Estes métodos e técnicas, ou ainda outros, podem ser encontrados em [5][15][3] que trazem ao leitor interessado maior profundidade na explicação. Para sintetizar o processo de funcionamento das redes convolucionais tradicionais, a figura 2.6 é mostrada a seguir. Esta figura mostra as etapas envolvidas desde a entrada da imagem, passando pelas camadas internas das redes, até o vetor de saída que terá as informações das classes a serem identificadas.

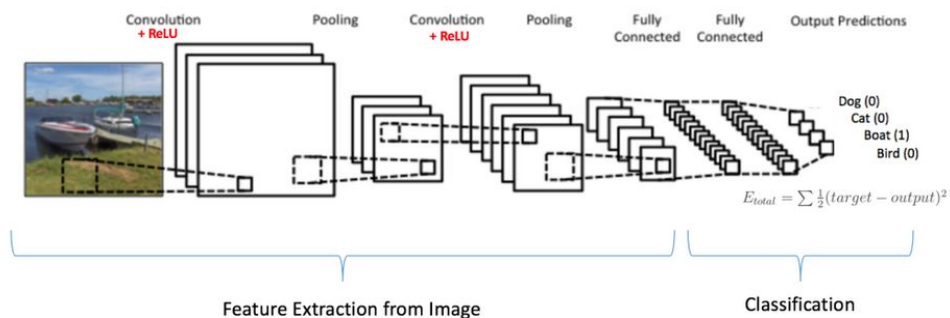


Figura 2.6: Processo de extração e classificação após max pooling com strides.

Fonte: The Data Science blog [3]

2.4 – YOLO - You Only Look Once

A contagem de veículos envolve diretamente o processo de detecção de objetos, isso porque o algoritmo precisa, primeiramente, reconhecer o que está sendo mostrado, identificando o tipo de objeto e delimitando sua área. Para isso que o YOLO será utilizado. Proposto primeiramente em 2016 [38], o YOLO (You Only Look Once) é uma rede neural única, composta de várias camadas e que funciona de um modo um pouco diferente das demais redes convolucionais que existem. O processo de detecção nas maiores das redes passa simplesmente pelo processo de *striding* e max pooling para chegar às classes de objetos mais prováveis, como foi abordado anteriormente.

Além de mais lento, essa abordagem também possui um erro maior na hora de delimitar as áreas das regiões de interesse e pode acarretar numa confusão entre o plano de fundo e os objetos que de fato estavam sendo identificados, isso porque são extraídos diversas regiões de uma mesma imagem, podendo gerar falsos positivos[38].

Já no YOLO, o processo de detecção, desde a fase de treinamento, usa a imagem completa para criar um contexto, e a partir disto regiões significativamente distintas uma das outras são criadas, conforme mostrado na figura 2.8. Isso é muito importante na hora de acelerar todo o processo, pois a rede passa a reconhecer não só o objeto-alvo mas também o contexto onde este objeto geralmente está inserido, como, por exemplo, os carros que quase sempre estão sobre rodovias asfaltadas, que são bem parecidas.

Essas abordagens diferentes permitiram a rede YOLO obter resultados superiores as redes anteriores, tornando o YOLO o estado da arte nas técnicas de visão computacional para identificação de objetos. O processo de detecção do YOLO começa com a subdivisão da imagem de entrada em subsetores. O número de subsetores é definido previamente e cada subsetor será responsável por detectar um número específico de objetos. As delimitações desses objetos são armazenadas em um *array* que será conhecido como *bounding box*. As *bounding boxes* são as caixas delimitadoras que serão previstas para cada objeto encontrado na imagem. Esse *array* conterá as dimensões da caixa, a posição dela na imagem, a classe de objeto e a probabilidade de

certeza da inferência.

Como afirmado anteriormente, cada subsetor terá seu número definido de objetos. A partir disso ficará nítido que algumas classes predominam em determinados conjuntos de subsetores, mostrando que naquela região provavelmente contém um objeto-alvo. Após a etapa de detecção é feito o processo de non-maximum suppression que será responsável por unificar e delimitar os conjuntos de *bounding boxes* que foram obtidos, como mostrado na figura 2.7.

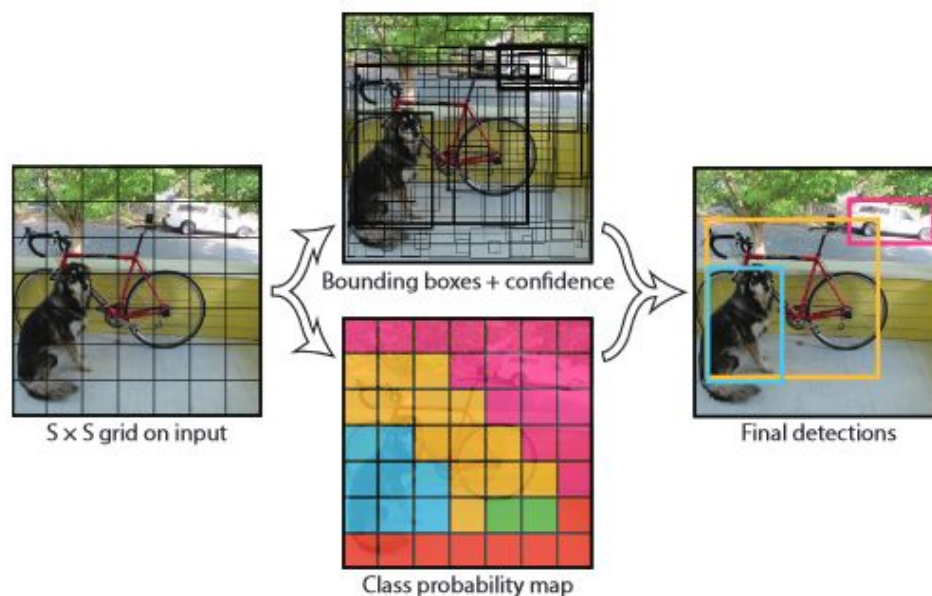


Figura 2.7: Processo de classificação do YOLO.

Fonte: Artigo YOLO [38]

2.5 – Non-maximum suppression

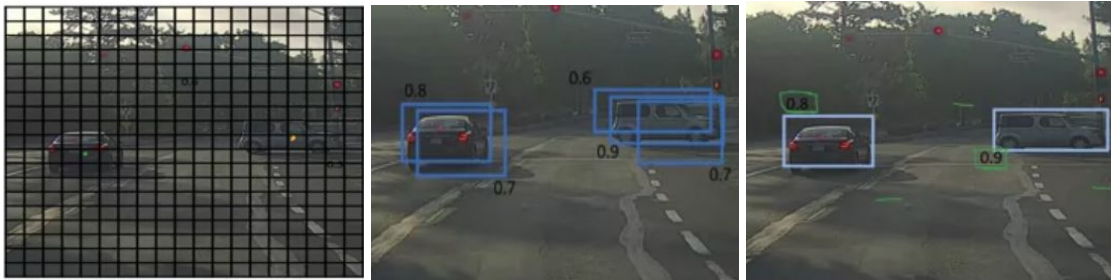


Figura 2.8: Processo de non-maximum suppression

Fonte: Coursera [32]

Para entender melhor o processo de Non-maximum suppression (NMS) é necessário voltar a forma como o YOLO vai atuar na imagem de entrada. Após dividir a imagem em um *grid*, como na figura 2.8, cada quadrado desse *grid* ficará responsável por detectar se dentro dele existe um objeto em potencial ou não. O limite de objetos permitidos por quadrado do *grid* é definido pelos *anchors*[16] que serão responsáveis por diferenciar as detecções dentro de um mesmo quadrado. Como muitas vezes os objetos são bem maiores que uma unidade de quadrado que compõe o *grid*, o algoritmo acaba detectando várias *bounding boxes* para o mesmo objeto.

O objetivo da detecção de objetos é identificar um *bounding box* para cada objeto, mas da forma como o YOLO funciona, é preciso tratar a sobreposição de *bounding boxes*. O non-maximal suppression é utilizado para selecionar a melhor *bounding box* dentre as disponíveis para um determinado objeto.

Algumas métricas são definidas na hora de selecionar os *bounding boxes* neste processo. Uma destas métricas é o *Intersect over Union*, que é um cálculo para medir o quanto dois *bounding boxes* estão sobrepostos. Um limite para essa sobreposição é definido para que o processo de NMS perceba se a sobreposição refere-se a um mesmo objeto definido duas vezes ou se são dois objetos muito próximos. O cálculo é realizado conforme ilustrado na figura 2.9.

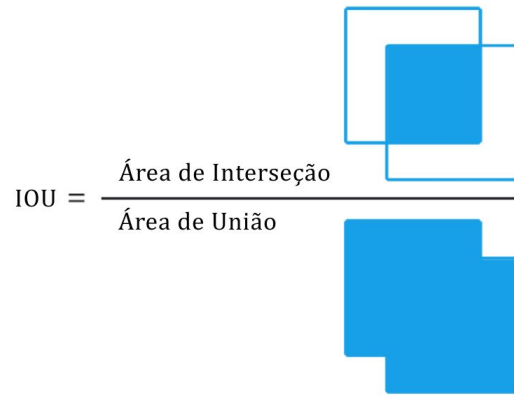


Figura 2.9: Intersection over Union (IOU)

Fonte: pyimagesearch [40]

O algoritmo NMS possui também um limite de probabilidade mínimo para que o *bounding box* seja considerado, todos os outros abaixo desse limite são descartados. Dentre os *bounding boxes* que restam é selecionado o de maior probabilidade e então verifica-se quais outros *boxes* estão com sobrepostos, respeitando o limite de IoU. Com o de maior probabilidade selecionado descarta-se todos os outros *boxes* que estão sobrepostos. Esse processo é repetido até que não haja mais boxes sobrepostos para um mesmo objeto.

O non-maximum suppression é essencial para a contagem de objetos, visto que este processo é o responsável por evitar que um mesmo objeto seja contabilizado diversas vezes. Por outro lado, a complexidade do algoritmo é $O(n^2)$ e dependendo do número de boxes sobrepostas o tempo de inferência pode aumentar em 10%, o que é um tempo considerável quando é buscada a detecção em tempo real [17].

Após a identificação de um objeto e o processo de non-maximum suppression, a rede do YOLO disponibiliza um vetor que contém as informações deste objeto, além da posição e formato da *bounding box* deste, como mostrado na figura 2.10.

As variáveis t_x e t_y correspondem à posição do objeto no *frame* e em conjunto com t_w e t_h que são, respectivamente, largura e altura, formam as variáveis que compõem a geometria de um *bounding box*. Além disso, existe a variável p_o que indica a probabilidade de nesse local haver um objeto e as probabilidades de classe (p_c), que vão indicar qual a classe desse objeto. A cada *frame* que possuam objetos

identificáveis vários vetores como este serão criados e estes vetores serão passados para o DeepSORT.

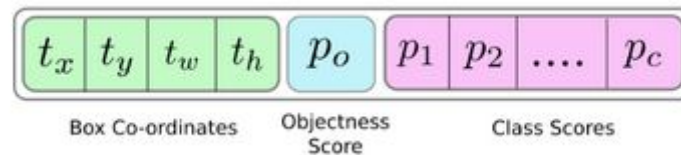


Figura 2.10: Vetor criado para cada objeto identificado em um *frame* de entrada

Fonte: PyLessons[4]

2.6 – YOLOv3 e YOLOv4

A primeira versão do YOLO[38] foi proposta em 2016 por Joseph Redmon e desde então tem sido adaptada e implementada por várias equipes com suas adaptações, seja no processo de treinamento ou no processo de inferência. No ano de 2020, Alexey Bochkovskiy propôs a quarta versão do YOLO [6], refinando e agilizando ainda mais o YOLO para dispositivos comerciais, sendo possível fazer inferências em tempo real e treinamentos de datasets complexos até mesmo em GPUs menos sofisticadas.

O YOLOv4 é uma reestruturação do seu antecessor, aplicando técnicas mais novas no processo de treinamento e preparação do dataset. A proposta da nova versão, comparado ao YOLOv3[37], é utilizar o mesmo detector que a terceira versão, mas alterar o processo de tratamento das imagens do dataset, tanto no *backbone*[45] quanto na etapa intermediária, antes da detecção. Todas informações, com detalhes, podem ser encontradas em [6]. O esquema da rede YOLOv4 é ilustrado em [39].

2.7 – TensorFlow

Implementar a rede neural proposta pelo YOLO, com toda a matemática envolvida quanto a geometria da rede com seus neurônios, processos de ativação, maxpooling e etc, pode ser muito custoso. Para facilitar este trabalho começaram a surgir várias bibliotecas e *frameworks* que já possuíam modelos de neurônios e métodos facilitadores implementados de antemão. O TensorFlow [1] é um desses *frameworks* que, de tanto evoluir, já é chamada de plataforma devido à grande quantidade de dispositivos compatíveis e possibilidades de aplicação.

Com a compatibilidade entre diferentes plataformas, suporte para modelos de inferência desde celulares até supercomputadores com múltiplas GPUs, treinamento em GPUs comerciais, facilidade de implementação de redes complexas e simples, bibliotecas prontas para paralelização e suporte ao deploy de redes para softwares em produção, o TensorFlow logo passou a ser utilizado pela comunidade tanto para pesquisas quanto para o desenvolvimento de novos produtos.

O nome TensorFlow é devido a forma em que os dados são estruturados através de tensores[7]. Desde valores escalares a arrays multidimensionais, todos os *inputs* e *outputs*, todo tipo de dado que passará pelas operações serão tensores de variados postos. Um exemplo clássico disso são imagens RGB que são representadas como tensores de posto três, no formato (H,W,3), com as informações de cada canal de cor para toda a imagem, conforme ilustrado na figura 2.11.

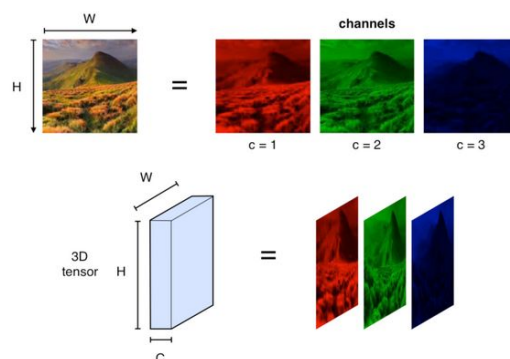


Figura 2.11: Exemplificação de um tensor para imagens coloridas

Fonte: Université PSL [27]

A modelagem da rede neural dentro do TensorFlow é muito mais fácil, já que com apenas uma linha de código é possível criar uma camada inteira de neurônios com todas suas conexões e função de ativação definida. No código a seguir, o Keras[21], que é executado em cima do TensorFlow, criará uma camada convolucional para *input* do tipo imagem, aplicando dez filtros diferentes, além de especificar o intervalo de deslocamento no stride, a função de ativação e o formato do *input*.

```
tf.keras.layers.Conv2D(10,(11,11),strides=4,activation='relu',input  
_shape=(227,227,3))
```

Além das facilidade de implementação da rede neural em forma de código, o TensorFlow também possibilita novas adaptações para dispositivos menos potentes. O chamado TensorFlow *lite*[43] é um *framework* com a proposta de utilizar aprendizado de máquina para internet das coisas e também para utilização mobile. O *framework* é responsável por converter o modelo tradicional da rede em TensorFlow para a versão *lite*. Esse modelo será transportado para o dispositivo desejado e já dentro do dispositivo, a *framework* gera um otimizador que adapta o modelo para arquitetura de processamento do dispositivo[43].

2.8 – Rastreamento de Objetos

Ao tratar um vídeo como a entrada de um algoritmo, é enviado para a rede neural uma sequência de imagens, denominada *frames*, que são quadros exibidos a cada fração de segundo do vídeo em questão. Isto é, o vídeo de entrada é tratado como uma sequência de *frames* e estes *frames* serão enviados a rede neural, que identifica os objetos presentes em cada *frame* independentemente.

O processo de identificação de objeto utilizado pelo YOLO já foi discutido anteriormente, mas não é possível realizar uma contagem desses objetos se, a cada *frame*, objetos novos são contabilizados e objetos antigos, que já foram identificados mas continuam por vários *frames*, passam a ser recontados. Para conseguir realizar esta tarefa, ainda é preciso de uma forma que, além de identificar um objeto-alvo

(como veículos e pessoas), saiba que um objeto identificado no atual *frame* já foi identificado anteriormente e, portanto, não pode ser contabilizado. Para chegar a esse objetivo é usado um rastreador.

O rastreador de objeto é uma ferramenta que acompanhará o mesmo objeto ao longo de toda a sua presença no vídeo, acompanhado-o *frame* por *frame* até que este não seja mais identificável. Os principais obstáculos deste processo são: oclusão, mudança de perspectiva e câmeras que se movimentam.

A oclusão é o nome que se dá quando um objeto é sobreposto por outro objeto ou então por algum elemento do ambiente. Isto é, um carro que está atrás de um ônibus ou embaixo de uma árvore, por exemplo. Até mesmo para o ser humano é impossível detectar alguns objetos se estes estiverem com um nível de oclusão alto. Apesar desta dificuldade, é preciso treinar a rede para que esta se adapte a algumas oclusões que acontecem com maior frequência, como o caso dos carros sobrepostos. Muitas vezes os ângulos das câmeras não permitirão que todas as faixas de uma rodovia sejam observadas de cima, com visão desobstruída, assim, a rede deve ser capaz de detectar partes de veículos, como mostrado na figura 2.12, para que não haja muitos erros no resultado final.



Figura 2.12: Carro ocluído ainda identificável.

Fonte: City Cameras SP [12]

O rastreamento de objeto pode ser do tipo objeto único ou multi objeto. O tipo multi objeto é desafiador por necessitar que o algoritmo seja capaz de acompanhar seu objeto desde que este entra no vídeo até que este saia, sem que se perca esse objeto de vista devido a oclusões e nem que confunda dois objetos que estejam

próximos. Abordagens diferentes para solucionar este problema já são conhecidas como *Meanshift*[11], *K-Means*[26], *Optical Flow*[42] e Filtros de Kalman[10].

Filtros de Kalman é uma ferramenta estatística usada para tentar aproximar o sinal ao longo do tempo, de uma variável ruidosa, ao valor real desta variável, tentando eliminar o ruído. A aplicação do filtro de Kalman para o acompanhamento da posição de um objeto detectado, é feita com o intuito de eliminar variações ruidosas da posição do objeto. Essas variações ocorrem por pequenas imprecisões na detecção, gerando centros de massa para cada posição que não estão exatamente na posição real da imagem.

A aplicação da teoria dos filtros de Kalman para os algoritmos de rastreamento já está implementada em uma biblioteca chamada DeepSORT[46]. Esta biblioteca é responsável por armazenar cada objeto detectado desde o primeiro *frame*, atualizando as posições destes ao longo dos *frames* subsequentes. Também é diferenciado objetos que já não estão mais no quadro, ou que acabaram de aparecer na imagem, de objetos que permaneceram desde *frames* anteriores.

2.9 – O DeepSort

O DeepSORT, que é um acrônimo para *Simple Online and Realtime Tracking*, é um algoritmo que aplica a teoria dos Filtros de Kalman para realizar a tarefa de rastrear um objeto através dos *frames* do vídeo de *input*. Essa tarefa é muito complexa e depende intrinsecamente de um detector de objetos que seja bastante preciso para que o resultado do rastreamento seja fiel e suficiente. Por mais que a rede seja bem treinada e o detector seja bastante preciso, muitas vezes as imagens de entrada não serão de qualidade. Muitas câmeras de segurança ou câmeras de rodovia sequer possuem imagem em alta definição, isso faz com que ao longo do vídeo alguns *frames* sejam compostos por borrões devido a incapacidade da câmera de captar um veículo em alta velocidade, por exemplo.

É normal que câmeras de rodovias estejam posicionadas em ângulos onde ocorrem oclusões constantes, como ilustrado na figura 2.13. Ainda que as redes estejam preparadas para detectar objetos oclusos, ainda assim pode-se ter queda no desempenho ou na precisão da identificação. Devido a oclusões e a baixa qualidade

das imagens, alguns objetos podem permanecer alguns *frames* sem serem identificados. Com a movimentação destes objetos, a nova identificação pode ser feita em um local já distante de onde foi a primeira. Isso impossibilita o algoritmo de rastreamento de associar esse objeto ao anterior, devido à algumas métricas internas que serão analisadas na seção 3.3.



Figura 2.13: Carros oclusos (esquerda) e em câmeras de baixa qualidade (direita)

Fonte: Autoria própria

O DeepSORT atua como um gerenciador de trilhas, onde cada objeto detectado representa uma trilha no algoritmo. Para cada trilha k o rastreador verifica quantos *frames* f_k se passaram desde que o objeto correspondente a esta trilha foi identificado. Se no *frame* atual um objeto identificado pode corresponder ao objeto desta trilha, o valor de f_k é zerado, se não houver correspondência, o valor de f_k é incrementado. Se f_k ultrapassar o limite f_{max} , essa trilha é deletada pois o algoritmo entende que este objeto não está mais na imagem.

Quando um *bounding box* é gerada pelo YOLO, as informações correspondentes ao centro da caixa (também chamado centróide), a proporção e a altura da caixa são passadas ao DeepSORT. Com estes valores o rastreador tratará estas variáveis através de uma rotina de associação que verificará, com o uso de Filtros de Kalman e algumas análises métricas, se este objeto corresponde a um objeto já rastreado anteriormente ou a um novo objeto. Se for um objeto antigo, o valor do centróide será atualizado e, comparando o número de *frames* entre a última detecção e esta atual, o algoritmo define velocidade, direção e sentido para o objeto, prevendo eventuais posições futuras deste. Caso este objeto seja um novo objeto, uma nova trilha é criada e as variáveis de centróide, proporção e altura são salvas.

2.10 – A rotina de associação de trilhas

Para associar as detecções vindas de novos *frames* a objetos que já estavam presentes em *frames* anteriores, o DeepSORT utiliza as previsões obtidas pelo filtro de Kalman. Para todas as trilhas o algoritmo realiza o cálculo da distância de Mahalanobis[14] entre a previsão do objeto da trilha e todas as *bounding boxes* do *frame* atual. Com as distâncias calculadas, associando os valores ao inverso da distribuição qui-quadrada é possível definir um limite de confiança para prever qual dos objetos identificados corresponde aquela trilha. As equações podem ser encontradas em [46].

Utilizar o cálculo de Mahalanobis para calcular as distâncias entre a previsão e as *bounding boxes* pode não ser tão preciso quanto esperado. A alta variação da posição das *bounding boxes* provoca uma incerteza bastante alta que prejudica os cálculos. Além disso, toda a discussão anterior sobre oclusões pode fazer com que um objeto ocluso jamais seja associado a uma trilha novamente. Para corrigir essas questões, o DeepSORT usa uma rede pré-treinada que é responsável por reconhecer a aparência dos objetos. De forma análoga ao cálculo de Mahalanobis, uma distância de cossenos considerando a aparência é realizada entre todos objetos. Essa análise permite que mesmo após pequenas oclusões o objeto possa voltar a ser identificado.

Apesar de abranger tanto distâncias estatísticas e similaridade entre os objetos, algumas questões ainda precisam ser discutidas. Se por acaso duas trilhas estiverem competindo pelo mesmo objeto, isto é, após todos os cálculos anteriores um mesmo objeto se configura como pertencente a duas trilhas diferentes, o DeepSORT associa este objeto a trilha que possui mais associações frequentes, pois esta possui menos incertezas associadas. Afinal, quanto maior o número de *frames* entre o *frame* atual e a última associação, maior a incerteza adquirida nas previsões obtidas pelo filtro de Kalman.

2.11 – O processo de contagem de veículos

Apesar de toda ferramenta estatística, a associação e contagem de objetos é uma tarefa muitas vezes pouco precisa na prática, ainda mais quando esta teoria é aplicada em um universo repleto de incertezas, com materiais e dados de qualidade duvidosa. Para mitigar estas incertezas é preciso entender a dinâmica de uma rodovia ou uma via pública.

Quando se pensa em contagem de veículos em uma rodovia não importa se um mesmo veículo passou por aquele trecho em horas diferentes do mesmo dia, ou se este fez um retorno e decidiu voltar pela mesma rua alguns segundos depois. Outra questão é que, apesar da quantidade de veículos ou pessoas presentes em uma mesma tela, é natural realizar a contagem em somente um local específico, para evitar que o mesmo carro seja contado duas vezes.

A contagem dos veículos ou pessoas se dá numa área de interesse (ROI, *region of interest*) onde uma linha é traçada na imagem para indicar ao algoritmo o local onde serão contabilizados os objetos que passarem por ali. Na prática, a existência de uma região de interesse vai limitar o impacto dos problemas citados anteriormente. Apesar de eventuais imagens de baixa qualidade e dificuldade de fazer que um objeto permaneça associado a mesma trilha, como o veículo apresentado na figura 3.1, basta que este objeto permaneça detectado e associado durante a região de interesse. Não importa quantas vezes um objeto deixar de ser identificado ou associado, desde que isso não ocorra na região de interesse. A contagem é feita de modo bastante preciso se a linha de ROI for selecionada em um local com menores chances de oclusão e em uma região mais próxima da câmera, possibilitando ao detector e ao *tracker* um resultado mais satisfatório, ainda que somente durante uma dezena de *frames*.

A contagem é feita de maneira simples e será explicitada no código em uma seção posterior, mas basicamente são duas funções que detectam se há interseção entre um centróide de alguma trilha e a linha da região de interesse. Caso haja a interseção, o contador é incrementado e essa *bounding box*, associada a uma trilha do *tracker*, não será contada novamente.

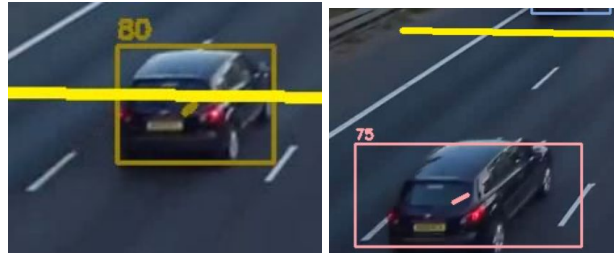


Figura 2.14: Mesmo carro com identificação trocada

Fonte: Autoria Própria

Capítulo 3

A implementação

3.1 – Arquitetura do algoritmo

O diagrama de atividades da figura 3.1 mostra as etapas do processo de contagem e detecção. Primeiramente, o algoritmo inicia a rede, analisando as *flags* inseridas via linha de comando e abre o arquivo de vídeo. Esse arquivo é manipulado com a biblioteca OpenCV[8] e o usuário escolhe a região de interesse onde será realizada a contagem dos veículos. Com a rede YOLO carregada do modelo presente no projeto e a região de interesse definida, começa o processo de loop que fará as detecções, a filtragem de *bounding boxes* com o non-maximum suppression e as detecções no rastreador. Esse processo é realizado enquanto existirem novos frames disponíveis no arquivo de entrada.

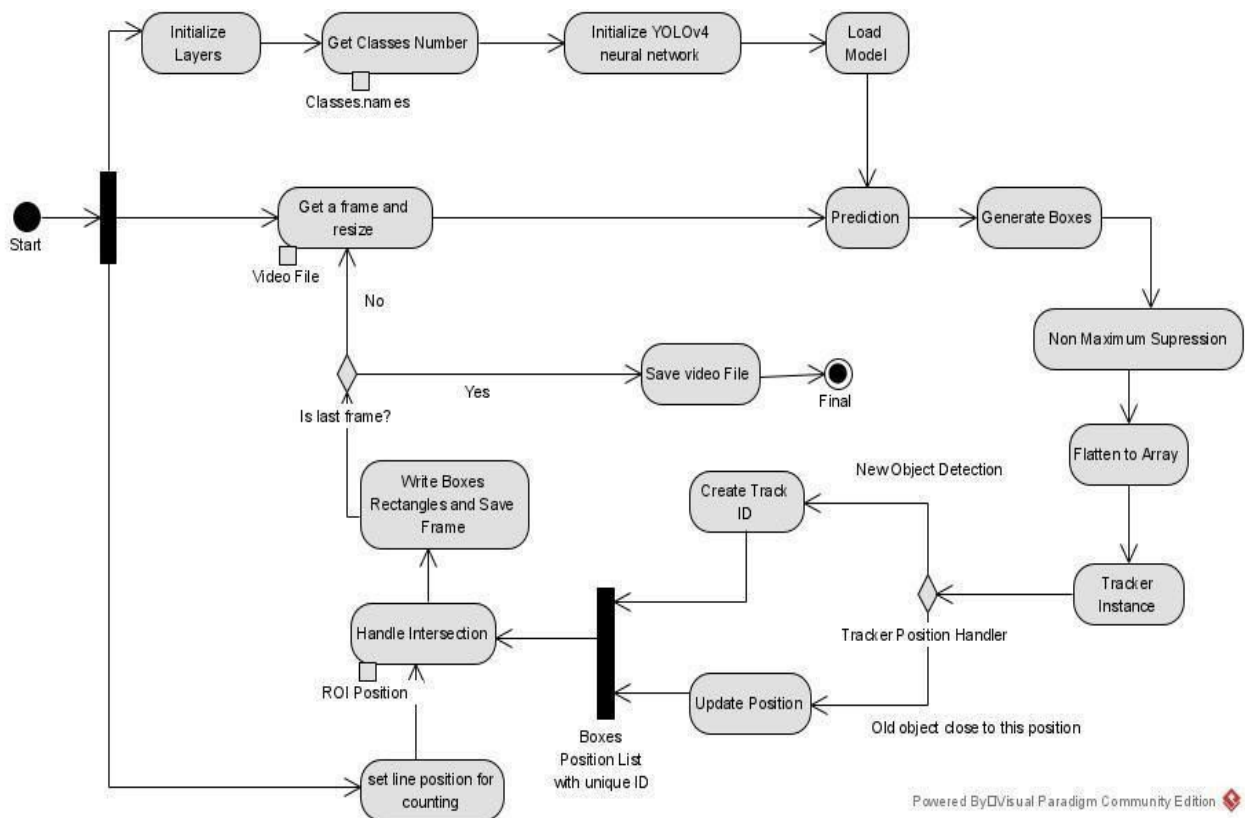


Figura 3.1: Diagrama de Atividades do Algoritmo

Fonte: Autoria própria

Todo o código está disponibilizado no GitHub deste projeto [30], além do Google Drive contendo os pesos necessários para a inicialização da rede e utilização do algoritmo. Para configurar os arquivos para a execução é necessário clonar o repositório do GitHub, realizar o download dos pesos da rede do link disponibilizado no GitHub, adicionando-os a pasta checkpoints. Os código a seguir clona o repositório:

```
$ git clone https://github.com/molimat/YOLOv4-Counter-in-TF
```

Este algoritmo exige que algumas bibliotecas estejam instaladas no computador do usuário para que este possa ser executado. Os pré-requisitos estão todos listados no arquivo requirements.txt e requirements-gpu.txt que estão disponíveis na pasta raiz do projeto. O ambiente usado para a implementação do YOLO

é um ambiente em Python, que deve ser uma versão superior ao Python 3.7 e está disponível tanto para Windows, Linux e MacOS. Para configurar o ambiente pode-se utilizar o comando pip para fazer o download de todos os pré-requisitos necessários. O comando para instalar os pré-requisitos é este abaixo:

```
$ pip install -r requirements.txt
```

O diretório raiz do projeto contém o código principal para a execução do algoritmo, que é o arquivo `detectvideo_counter.py`. Na pasta raiz também existem arquivos de manipulação e preparação de pesos da rede para os diferentes modos de execução disponíveis neste projeto, como transformação do modelo de peso darknet para o TensorFlow ou Tensorflow *lite*. Além disso três subdiretórios estão presentes, o primeiro é o subdiretório `android` que contém arquivos necessário para executar o código em dispositivos mobile, o segundo é o `core`, onde estão todos os arquivos com funções necessárias para a execução do algoritmo. Nesse diretório estão as implementações do DeepSORT, no arquivo `sort.py` e a implementação da rede YOLO, no arquivo `yolov4.py`. O terceiro diretório se chama `data` e nele estão os arquivos de imagem e vídeo utilizados para o teste do algoritmo. Além disso as posições dos anchors, e as classes utilizadas também ficam neste diretório. A figura 3.2 mostra esta estrutura.



Figura 3.2: Estrutura de diretórios do projeto

Fonte: Autoria própria

O início do processo é feito com uma chamada via linha de comando onde o arquivo `detectvideo_counter.py` é iniciado contendo flags que especificam os parâmetros necessários para o tratamento do *input* e definem o tipo de rede a ser utilizada. Um exemplo do comando é mostrado a seguir:

```
$ python detectvideo_counter.py --weights
./checkpoints/yolov4-416 --video ./data/video3.mp4
```

Dentro do arquivo `detectvideo_counter.py` pode ser verificado todas as opções de parâmetros de inicialização possíveis para a execução. Dentre esses tem-se:

- *Framework*: Vai definir qual tipo de TensorFlow será utilizado, a versão normal para desktops ou a versão *lite* otimizado para sistemas com processador operando em ARM. Valores possíveis são “--tf” ou “--tflite”.
- *Weights*: Recebe o local onde estão salvos os pesos dos neurônios da rede a ser utilizada
- *Tiny*: Escolhe a versão do YOLO a ser utilizada, dependendo dos pesos selecionados. A versão *lite* é mais rápida, porém menos acurada. Para selecionar a versão tiny basta usar “--tiny”.
- *Vídeo*: O arquivo de vídeo de *input*
- *IOU*: É o limite aceito para a métrica de Intersection over Union, que é a área usada no non-maximum suppression para eliminar *bounding boxes* muito sobrepostas. Qualquer valor real entre 0 e 1 é possível.
- *Score*: É o limite aceito para a probabilidade de certeza que o YOLO retorna para um objeto identificado. Qualquer valor real entre 0 e 1 é possível.

3.2 – Implementação da detecção

As primeiras linhas de código presente no arquivo `detectvideo_counter.py` se referem ao tratamento do arquivo de entrada e inicialização de parâmetros necessários para a rede neural. A função `ConfigProto()` junto com a função `InteractiveSession()` serão responsáveis por inicializar as configurações do TensorFlow para a máquina que está executando o código, recolhendo informações de placas de vídeos ou processadores disponíveis, além de especificações de cada um desses.

Após a configuração da sessão, as flags são inicializadas dependendo do tipo de arquivo de entrada, das flags adicionadas na chamada via linha de comando e das classes disponíveis no arquivo `"/data/classes/coco.names"` que são as classes treinadas em cima do COCO Dataset. Para lidar com o arquivo de entrada e mostrar algumas etapas para o usuário, o algoritmo usa a biblioteca OpenCV que permite desenhar linhas, *bounding boxes*, extrair *frames* e informações de arquivos de vídeos, além de salvar o arquivo de saída. As últimas linhas do processo de inicialização são para a seleção da região de interesse. Um *frame* qualquer do vídeo é mostrado ao usuário e este deve desenhar uma linha onde será realizado o processo de contagem. A figura 3.7, que está na seção 3.4 mostra esse processo.

O código a seguir exemplifica o que foi abordado nos parágrafos anteriores:

```
config = ConfigProto()
config.gpu_options.allow_growth = True
session = InteractiveSession(config=config)
STRIDES, ANCHORS, NUM_CLASS, XYSCALE = utils.load_config(FLAGS)
input_size = FLAGS.size
video_path = FLAGS.video

vid = cv2.VideoCapture(video_path)
height = int(vid.get(cv2.CAP_PROP_FRAME_HEIGHT))
width = int(vid.get(cv2.CAP_PROP_FRAME_WIDTH))
fps = int(vid.get(cv2.CAP_PROP_FPS))
```

```
_ , frame_to_roi = vid.read()
frame_to_roi = Image.fromarray(frame_to_roi)
vid.set(1, 1)
```

Dependendo de quais flags foram utilizadas na execução via linha de comando do arquivo, as etapas a seguir serão diferentes. Para o caso do TensorFlow *lite* é necessário uma conversão dos pesos da rede, que são salvos no modelo de tensores do padrão TensorFlow. As linhas a seguir realizam esta conversão:

```
interpreter = tf.lite.Interpreter(model_path=FLAGS.weights)
interpreter.allocate_tensors()
```

Caso o modelo de rede padrão seja escolhido, as linhas a seguir serão responsáveis por inicializar a rede com os pesos disponibilizados para o algoritmo. No caso deste trabalho, os pesos gerados com o COCO Dataset e disponibilizados pela equipe Darknet para o YOLOv4.

```
saved_model_loaded = tf.saved_model.load(FLAGS.weights,
tags=[tag_constants.SERVING])
infer = saved_model_loaded.signatures['serving_default']
```

Com a rede inicializada, o algoritmo já está pronto para o processo de detecção que ocorrerá de modo igual para cada *frame* do vídeo de entrada. Uma estrutura de loop é então iniciada e durará enquanto houver novos *frames*. O OpenCV vai retirar um *frame* por vez e cada *frame* será redimensionado para os valores aceitos pela rede. Para o modelo do TensorFlow *lite*, a inferência se dá de uma forma diferente e necessita de um interpretador[43] que transformará a saída da rede neural em inferências interpretáveis. Já para o TensorFlow basta usar o modelo carregado anteriormente, neste código chamado de *infer*, que executará a inferência conforme os parâmetros de rede inicializados previamente.

```

if FLAGS.framework == 'tflite':
    interpreter.set_tensor(input_details[0]['index'], image_data)
    interpreter.invoke()
    pred = [interpreter.get_tensor(output_details[i]['index']) for i in
range(len(output_details))]
else:
    batch_data = tf.constant(image_data)
    pred_bbox = infer(batch_data)
    for key, value in pred_bbox.items():
        boxes = value[:, :, 0:4]
        pred_conf = value[:, :, 4:]

```

Após a predição, é necessário realizar a etapa de non-maximum suppression conforme foi abordado na seção 2.5. Este processo é executado em uma função chamada de `combined_non_max_suppression` que é disponibilizada pelo próprio TensorFlow, necessitando apenas que os parâmetros sejam informados no formato correto, conforme é verificado no código a seguir:

```

boxes, scores, classes, valid_detections = tf.image.combined_non_max_suppression(
    boxes=tf.reshape( boxes, (tf.shape(boxes)[0], -1, 1, 4)),
    scores=tf.reshape(pred_conf, (tf.shape(pred_conf)[0], -1, tf.shape(pred_conf)[-1])),
    max_output_size_per_class=50,
    max_total_size=50,
    iou_threshold=FLAGS.iou,
    score_threshold=FLAGS.score
)

```

A função anterior retorna quatro listas contendo as coordenadas dos *bounding boxes*, o nível de certeza da inferência, chamado *score*, as classes presentes no *frame* analisado e o número de detecções válidas que respeitaram os limites estabelecidos. Esses quatro itens precisam ser convertidos para listas do tipo `numpy[35]` para serem utilizados nas etapas posteriores, conforme será visto na próxima seção.

3.3 – Implementação do rastreador

O algoritmo de rastreamento é composto pela classe Sort que pode ser verificada na figura 3.3, a seguir:

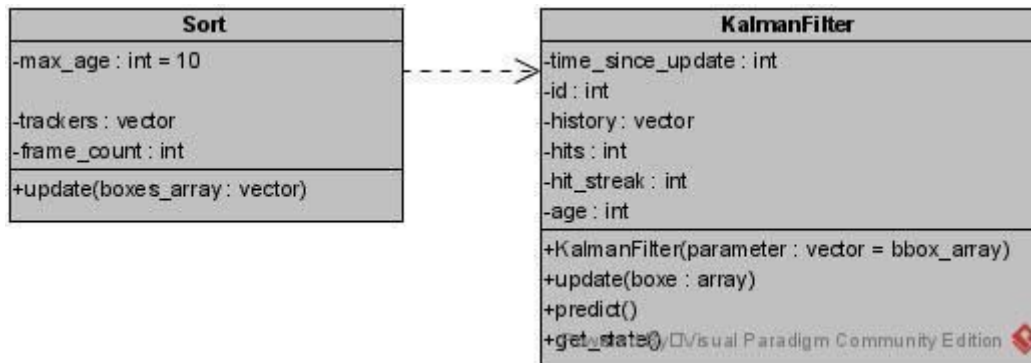


Figura 3.3: Diagrama da classe Sort

Fonte: Autoria própria

A classe Sort vai ser a responsável por manipular todas as etapas do rastreamento, isto é, todo objeto que for identificado pelo YOLO será passado para a classe Sort como um *input*. Este *input* nada mais é do que as coordenadas dos *bounding boxes* geradas para o *frame* atual do vídeo.

```
bboxes = [boxes.numpy(), scores.numpy(), classes.numpy(), valid_detections.numpy()]
dets = utils.prepare_for_tracking(frame, bboxes)
```

O código anterior é a etapa de preparação para a inicialização do objeto da classe Sort. Como explicado na seção 3.2, o detector retorna uma lista de *bounding boxes* para cada *frame*, esta lista é convertida para o modelo *numpy* e passada para a função *prepare_for_tracking*, junto com o *frame* a qual a lista pertence.

A função *prepare_for_tracking* é a integração das duas etapas entre a detecção e o rastreamento. A primeira parte da função é feita para obter as alturas e larguras do *frame* em questão. Após isto, a função enumerará as classes disponíveis para a etapa de detecção, estas classes estão presentes no arquivo *coco.names*, que é gerado para

a etapa de treinamento da rede que não é abordada no escopo deste trabalho. A partir deste ponto há o remapeamento das coordenadas de cada *bounding box* para o formato utilizado no DeepSORT, que é diferente do formato usado pelo YOLO. Com os *bounding boxes* reformatadas, uma lista é criada com todas as novas coordenadas, além da respectiva classe identificada e a probabilidade da previsão. Essa lista é exatamente no formato necessitado pela classe Sort que terá seu método `update` chamado em seguida.

```
tracks = tracker.update(dets)
```

O método `update` é o único a ser utilizado para executar o Sort, a partir dele todas as etapas são feitas. Para entender na prática como todo o processo exposto na seção 2.8 funciona, é necessário verificar as etapas principais dentro do código.

```
class Sort(object):
```

```
    def __init__(self,max_age=5,min_hits=1):
```

A classe é iniciada com parâmetros de *max_age* e *min_hit* que são os parâmetros responsáveis pelo máximo tempo em que uma trilha pode existir sem identificar seu objeto e o número mínimo de detecções do objeto da trilha para que essa trilha seja retornada pela classe, respectivamente.

```
    pos = self.trackers[t].predict()[0]
```

Antes de tratar as detecções do *frame* atual, o *tracker* efetuará a previsão dos locais esperados para as trilhas já existentes em memória. Essa previsão é feita através dos filtros de Kalman, que são chamados através do método *predict()*, para detectar a trajetória de cada trilha associada ao *tracker*. Além disso esse método guardará informações de quantos *frames* o objeto está permanecendo no *frame*, além de saber durante quantos *frames* ele está sendo identificado em sequência.

```
    matched, unmatched_dets, unmatched_trks =  
    associate_detections_to_trackers(dets,trks)
```

Após a previsão para as trilhas existentes, ocorre o processo de associação, onde cada detecção, vetor *dets*, será comparada com as trilhas, vetor *trks*, disponíveis.

```
for d,det in enumerate(detections):
    for t,trk in enumerate(trackers):
        iou_matrix[d,t] = iou(det,trk)
```

O código anterior compara o valor de IOU entre cada *bounding box* detectado no *frame* atual e os *bounding boxes* das previsões das trilhas existentes. Esse valor é o que vai limitar se um objeto vai ser associado a uma trilha existente ou se será necessário criar uma trilha para o objeto.

Com os valores de IOU calculados para todas as trilhas que foram associadas às detecções do *frame* atual, resta saber quais são as trilhas que não foram associadas a nenhuma detecção ou quais objetos que não foram associados a nenhuma trilha. Ou seja, é necessário verificar quais trilhas que estavam acompanhando objetos que já saíram do vídeo, e quais objetos são novos no vídeo. Além das questões acima, também é preciso filtrar as associações que obtiveram um valor de IOU abaixo do limite. O código a seguir é o responsável pelos tratamentos mencionados:

```
for d,det in enumerate(detections):
    if(d not in matched_indices[:,0]):
        unmatched_detections.append(d)

for t,trk in enumerate(trackers):
    if(t not in matched_indices[:,1]):
        unmatched_trackers.append(t)

for m in matched_indices:
    if(iou_matrix[m[0],m[1]]<iou_threshold):
        unmatched_detections.append(m[0])
        unmatched_trackers.append(m[1])
```

Após os tratamentos feitos, a função `associate_detections_to_trackers` nos retorna três listas contendo as associações, os objetos que não foram associados e as trilhas com objeto não identificado no *frame*. As trilhas associadas passam por uma função chamada `update` que será responsável por alterar o vetor de estados do filtro de Kalman com os valores associados no *frame* atual, isso é o que permite o filtro adaptar suas previsões com os valores reais observados pelo detector.

```
for t, trk in enumerate(self.trackers):
    if (t not in unmatched_trks):
        d = matched[np.where(matched[:, 1] == t)[0], 0]
        if len(d) > 0:
            trk.update(dets[d, :][0])
```

Para finalizar a etapa de rastreamento é preciso incrementar o contador das trilhas que não tiveram seus objetos identificados no *frame* atual, isso permite que seja possível excluir trilhas que já não possuem identificação de seus objetos a muitos *frames*, que representam objetos que já não estão mais no vídeo. Além disso, é necessário criar trilhas para os objetos não associados para que estes possam ser previstos a partir do próximo *frame*. Os trechos de código a seguir são responsáveis por isso.

```
trk = KalmanBoxTracker(dets[i, :])
self.trackers.append(trk)
```

A trilha é iniciada e então adicionada à lista de trilhas disponíveis. Após isso, essa lista é filtrada dentro dos parâmetros definidos para o projeto (`max_hit` e `min_hit`). No caso do rastreador de veículos foi definido que ao menos duas detecções daquele objeto tenham ocorrido para que este seja retornado como um objeto rastreado. Essa prática evita que falsos positivos, que não costumam ocorrer várias vezes no mesmo local, sejam rastreados e contabilizados.

```
for trk in reversed(self.trackers):
    d = trk.get_state()[0]
```

```

        if((trk.time_since_update < self.max_age) and (trk.hit_streak >= self.min_hits or
self.frame_count <= self.min_hits)):

if(trk.time_since_update > self.max_age):
    self.trackers.pop(i)

if(len(ret)>0):
    return np.concatenate(ret)
return np.empty((0,5))

```

A função `update` então retorna as *bounding boxes* geradas pelo rastreador, que serão utilizadas no contador para checar interseções entre estas e a região de interesse.

3.4 – Implementação do contador

O contador é a etapa mais simples de todo o processo, ele é responsável por interpretar quando um segmento de reta é intersectado por um ponto. Este segmento de reta é definido no início do algoritmo, quando uma linha é escolhida como a região de interesse do vídeo analisado. A figura 3.4 mostra a região sendo escolhida e as tuplas com os valores de início e fim da reta (x, y) sendo enviadas ao contador. A partir da definição da região de interesse o algoritmo verificará todos os objetos rastreados e comparará a localização do centróide destes com a equação da reta que representa a região de interesse.

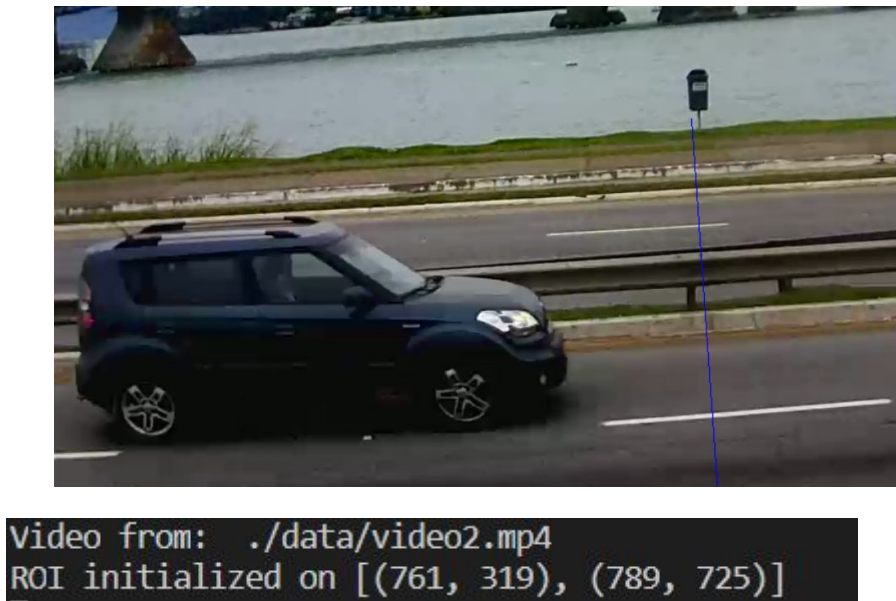


Figura 3.4: Definição da Região de Interesse

Fonte: City Cameras SP [12]

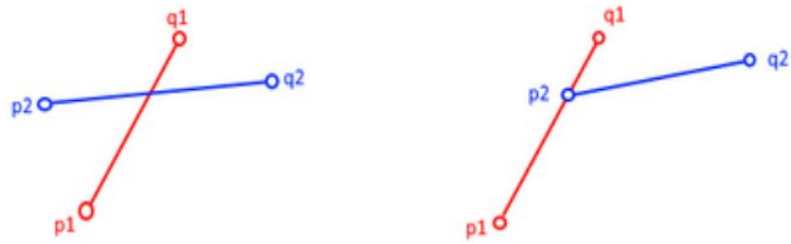
Para calcular a interseção, os valores com as coordenadas das bounding boxes geradas pelo rastreador são extraídos do vetor dets da seção anterior. Após os tratamentos desses valores serão calculadas as trajetórias feitas pelos objetos nas últimas duas detecções. Estas trajetórias, junto com a reta da região de interesse, serão passados para a função intersect que será a responsável pela contagem. O código a seguir mostra a implementação do processo abordado, o ponto p0 e o ponto p1 representam, respectivamente o centróide da detecção atual e o centróide da detecção anterior:

```
if indexIDs[i] in previous:
    previous_box = previous[indexIDs[i]]
    (x2, y2) = (int(previous_box[0]), int(previous_box[1]))
    (w2, h2) = (int(previous_box[2]), int(previous_box[3]))
    p0 = (int(x + (w-x)/2), int(y + (h-y)/2))
    p1 = (int(x2 + (w2-x2)/2), int(y2 + (h2-y2)/2))

if utils.intersect(p0, p1, line[0], line[1]):
    counter += 1
```

A função intersect é a função para verificação da contagem. Ela compara o sentido de rotação entre diferentes pontos para isso.

Primeiro Caso:



Segundo Caso:

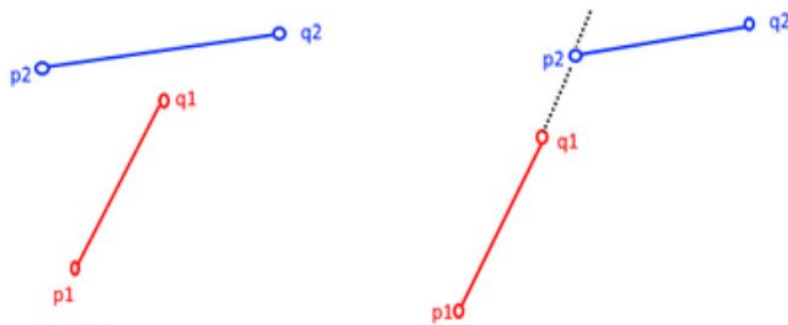


Figura 3.5: Verificação de retas concorrentes com método de sentido horário

Fonte: Universidade de Glasgow [36]

Em resumo, duas retas são concorrentes quando o sentidos da rotação de três pontos selecionados difere do sentido de outros três pontos selecionados, obedecendo a regra de seleção dos pontos conforme a figura 3.5. O código abaixo executa esta tarefa comparando os pontos da trajetória do veículo nos últimos dois *frames* com os pontos da região de interesse.

```
def intersect(A,B,C,D):  
    return ccw(A,C,D) != ccw(B,C,D) and ccw(A,B,C) != ccw(A,B,D)  
def ccw(A,B,C):  
    return (C[1]-A[1]) * (B[0]-A[0]) > (B[1]-A[1]) * (C[0]-A[0])
```

Todos os processos explanados ao longo desta seção ocorrem para cada *frame* de vídeo fornecido pelo arquivo de entrada, sendo repetidamente executados até o fim do arquivo de vídeo ou até o cancelamento feito pelo usuário.

Capítulo 4

Resultados

4.1 – Descrição do experimento de contagem

Para verificar a confiabilidade do algoritmo foi realizado um experimento de contagem para analisar se o valor obtido pelo contador se aproxima do valor real e o como o contador se sai quando comparado ao desempenho de seres humanos.

O experimento foi realizado em um notebook sem placa de vídeo, o que fez com que o processo de contagem fosse extremamente lento, na velocidade de 1 FPS. A infraestrutura empregada contém as seguintes características:

- Notebook ASUS ZenBook Flip S UX370UA
- Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2712 Mhz, 2 Core(s)
- 8 GB 2133MHz LPDDR3 onboard RAM
- Windows 10
- Python 3.7
- TensorFlow v2

Os vídeos foram obtidos do YouTube ou de câmeras de segurança disponíveis ao público no sistema City Cameras[12] da cidade de São Paulo. Para o propósito da análise foram escolhidos três vídeos de um minuto cada. Os vídeos possuem qualidades diferentes e orientações variadas entre si proporcionando cenários diferentes para evitar enviesamentos.

Os vídeos foram disponibilizados através do Google Form para a análise. Cento e trinta e dois voluntários obtidos dentre o grupo de Facebook do Centro de Tecnologia da UFRJ, família e amigos realizaram a contagem de veículos presentes em cada um dos três vídeos. Eles foram orientados a contar sem pausar ou voltar o vídeo, simulando um ambiente de contagem em tempo real.

4.2 – Considerações sobre os resultados obtidos

Para obter os valores de controle das contagens, cada vídeo foi visualizado diversas vezes em velocidade 50% menor que a real para a contagem dos veículos. Os valores de veículos verificados em cada vídeo foram de:

- Vídeo 1 - 110 veículos
- Vídeo 2 - 107 veículos
- Vídeo 3 - 160 veículos

Mesmo ao obter esses valores exatos para cada vídeo, é completamente admissível pequenas variações, visto que não foi dito em momento algum a região de interesse a ser contabilizada. Isso causa variação na contagem pois, para o algoritmo, o carro só será contabilizado se passar pela região de interesse, não importando se este aparece no vídeo alguns *frames* antes de finalizar a gravação. Já para os voluntários isso não foi especificado. Mesmo assim esse efeito provoca não mais que algumas unidades a mais ou a menos na contagem.

Outra questão importante de mencionar é a importância de entender como o algoritmo se comporta em diferentes cenários. Motos e caminhonetes, por exemplo, podem ser desafiadores em câmeras de baixa qualidade. O primeiro por ser geralmente menor acaba não sendo rastreado por não ficar dentro do limite estabelecido para IOU de rastreamento, o segundo por variações na detecção, podem gerar dois objetos simultâneos, um para caminhão e outro para carro, conforme observado na figura 4.1, onde o objeto 584 é detectado corretamente como uma caminhonete e o 611 que detecta apenas a parte frontal como sendo a de um carro. Como essas duas detecções possuem *bounding boxes* de formatos distintos, os dois são registrados como objetos.

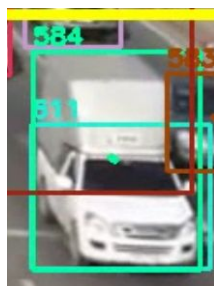


Figura 4.1: Caminhonete gerando duas detecções

Fonte: Autoria própria

Apesar de todas as variáveis que envolvem o processo de contagem de veículos, esse experimento tem como objetivo verificar a qualidade da contagem de seres humanos comparando-os com um algoritmo de visão computacional. Além disso é interessante avaliar o quão acurado o algoritmo pode ser mesmo com variações como os erros apontados no parágrafo anterior. Isto é, mesmo com as imprecisões, o quão próximo ao valor correto o algoritmo pode ser ao analisar rodovias que seriam muito custosas para serem contabilizadas por seres humanos.

Cada um dos vídeos está representado a seguir para que seja possível visualizar as questões que serão abordadas na seção 4.3. A figura 4.2 possui capturas de telas dos três vídeos, com o vídeo 1 a esquerda e o vídeo 3 a direita.



Figura 4.2: Vídeos analisados para o experimento de contagem

Fonte: Autoria própria

4.3 – Resultados Obtidos

Para facilitar a visualização, as mais de 100 respostas obtidas pelo formulário foram plotadas em um gráfico de dispersão, onde o eixo horizontal representar cada voluntário e o eixo vertical o valor contabilizado por este voluntário. Além disso, foram adicionadas duas retas: verde e vermelha, que representam respectivamente o valor verificado pelo processo de contagem repetitiva em velocidade reduzida e o valor contabilizado pelo algoritmo.

No primeiro vídeo, a média dos valores contados pelos voluntários foi de 103, apresentando um erro relativo de -6.3%. Já para o algoritmo o valor foi de 114, representando um erro relativo de 3.6%. Por mais que os valores não sejam tão diferentes e nenhum dos métodos obtenha um erro muito grande, é interessante analisar dois pontos. O primeiro é que ainda assim o algoritmo teve uma acurácia melhor, comparando com os voluntários. O segundo ponto é que, conforme é possível perceber na figura 4.3, o desvio padrão é bem grande, apresentando um valor de 29. A falta de precisão presente nos resultados mostram que nem sempre os seres humanos são confiáveis ao analisar contagem de veículos em rodovias. Esse resultado é agravado em rodovias com maior número de veículos e faixas e não tão presente em rodovias com menos faixas, conforme o resultado do segundo vídeo, conforme a figura 4.4.

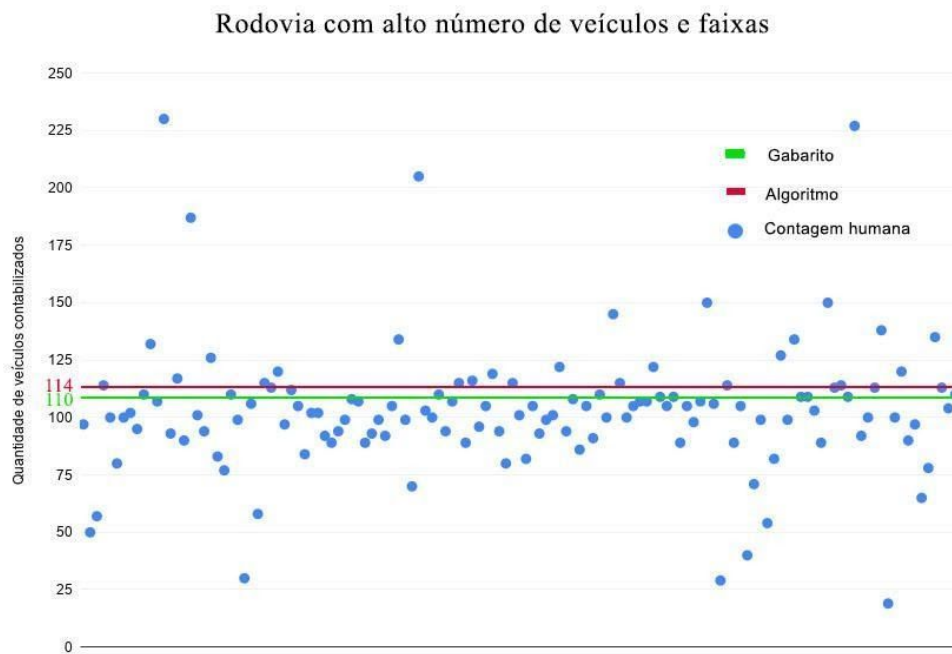


Figura 4.3: Gráfico de resultados referentes ao vídeo 1
 Fonte: Autoria própria

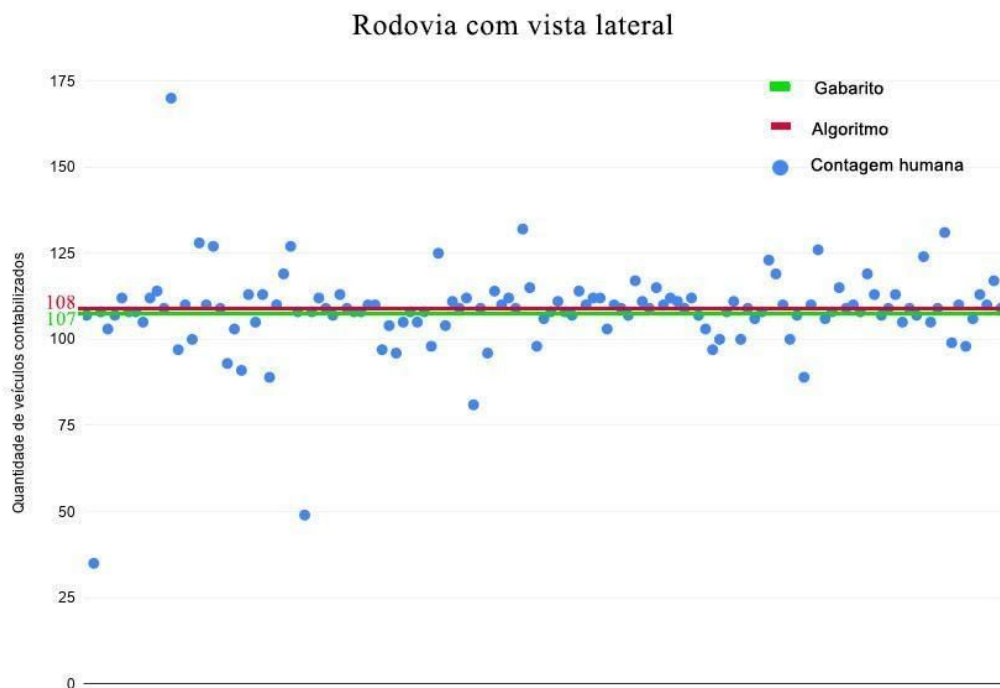


Figura 4.4: Gráfico de resultados referentes ao vídeo 2
 Fonte: Autoria própria

A análise dos dados do vídeo 2 mostra um desvio padrão de 14.4 e erro relativo de 0.5% para os voluntários, que apresentaram uma média de 107.5. Isso mostra que para rodovias com menos faixas é mais fácil realizar o processo de contagem. Além desse fator, este vídeo estava orientado de um modo que a rodovia estava sendo vista de lado, não de cima. Porém não é possível tirar muitas conclusões sobre a influência das orientações por falta de mais dados. Para o algoritmo, houve quase um acerto completo do gabarito, tendo sido impedido apenas pela contagem de uma bicicleta, que não foi levada em consideração na formulação do gabarito.

Para o vídeo 3, conforme pode ser visto na figura 4.5, o gabarito também apresentou um resultado excelente, com um erro relativo de apenas 0.6%. Por mais que talvez esse valor tenha sido alcançado através de uma contagem não necessariamente perfeita, com alguns veículos omitidos no processo e talvez alguns duplicados, o resultado mostra que o algoritmo possui um resultado superior ao dos voluntários. Estes apresentam um desvio padrão de 38 e um erro relativo de -12%, o pior resultado dos três vídeos analisados. O motivo para isso pode ser o fato desta rodovia ter um alto volume de veículos em alta velocidade, presentes em seis faixas constantemente trafegadas, o que torna uma tarefa difícil para o cérebro humano que mostra dificuldades em focar e interpretar diferentes pontos ao mesmo tempo.

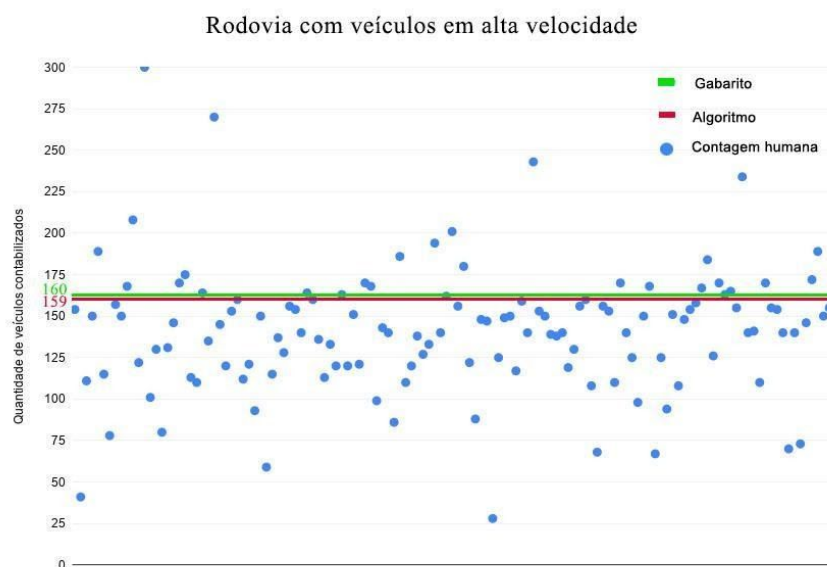


Figura 4.5: Gráfico de resultados referentes ao vídeo 3

Fonte: Autoria própria

4.4 – Resultado em vídeo noturno

Por mais que não faça parte do experimento e não exista comparação com a capacidade humana, o algoritmo também foi exposto a um vídeo gravado por um celular durante a noite para verificar os resultados obtidos. Ficou claro que a rede possui dificuldades em situações de baixa luminosidade, principalmente para reconhecer humanos e motos, o que é perfeitamente compreensível, visto que para humanos em baixa luminosidade é difícil definir onde termina a pessoa e onde começa o background. Já para motos isto também ocorre, principalmente quando o farol pode ser facilmente confundido com qualquer fonte luminosa mesmo por olhos humanos, conforme a figura 4.6, a seguir.



Figura 4.6: Moto em vídeo noturno

Fonte: Autoria própria

A figura 4.7 apresenta resultados com erro relativo de -38,4%. Conforme mencionado anteriormente, as motos presentes no vídeo são um dos principais motivos deste erro, já que estas parecem nove vezes no vídeo e não são reconhecidas pelo algoritmo. Talvez um processo de treinamento reforçando estes objetos neste contexto já seria uma melhoria considerável para o desempenho da rede.

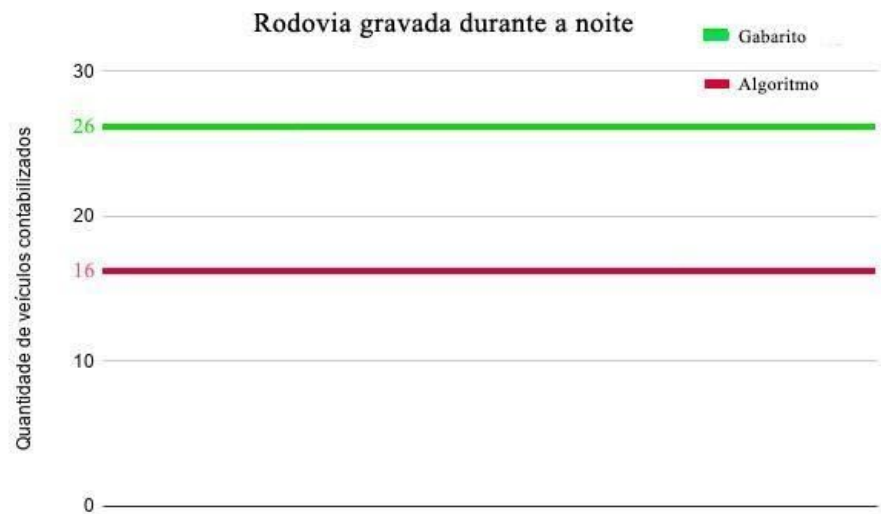


Figura 4.7: Gráfico de resultados referentes ao vídeo noturno
Fonte: Autoria própria

Os vídeos contendo os resultados, até para este caso da gravação noturna, estarão disponibilizados no google drive mencionado no GitHub[30] para que possam ser verificados.

Capítulo 5

Conclusão

Ainda que este algoritmo realize a contagem com alguns erros, conforme foi mencionado na seção anterior, ele se aproxima com eficácia ao valor real verificado para os exemplos analisados. No comparativo à qualidade humana de realizar esta mesma tarefa, fica claro que o algoritmo pode ser muito mais eficaz. Apesar da média da contagem humana se aproximar do valor real, o desvio padrão apresentado é muito grande, mesmo para um pequeno trecho de vídeo analisado.

Para cálculos efetuados em projetos de tráfego de cidades ou grandes eventos é claro que longos vídeos deverão ser analisados para que os dados necessários para o planejamento sejam extraídos. Este trabalho mostra que para estes casos, mesmo com um grande número de pessoas disponíveis, a alta variância do desempenho humano pode proporcionar resultados imprecisos, sendo preferível a utilização deste algoritmo para esta tarefa. Sem mencionar o fato de que este algoritmo, se executado em uma máquina de alto desempenho, pode analisar longos vídeos em um menor tempo, devido à capacidade de executá-lo com taxa de quadros superiores a 30 fps.

Para futuros projetos que utilizarem o código deste trabalho algumas melhorias podem ser implementadas na rede, como treinar uma rede específica para o cenário do trânsito brasileiro, focando especificamente na classe de objetos que abrange veículos. Treinamento para câmeras de baixa qualidade e noturnas podem melhorar o resultado final.

5.1 – Trabalhos Futuros

A rede utilizada neste projeto foi a rede disponibilizada pela equipe do YOLOv4, que foi treinada previamente com 80 classes padrões pertencentes ao COCO Dataset. A existência de classes que não pertencem ao escopo deste trabalho pode gerar um processo de inferência mais lento do que uma rede específica para veículos. Portanto, um trabalho futuro possível seria o processo de retreinamento da rede, gerando uma rede específica para veículos.

Outro fator a ser pesquisado é uma possível obtenção de dados de trânsito a partir dos objetos rastreados em vídeo, tais como: velocidade, tempo de engarrafamento e contagem específica para cada classe. A existência desses dados pode ser interessante para os projetos de planejamento urbano que utilizarem este material.

Referências Bibliográficas

- [1] ABADI, Martín et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. **arXiv preprint arXiv:1603.04467**, 2016.
- [2] ANDERSON, Michael; IANDOLA, Forrest; KEUTZER, Kurt. **Quantifying the energy efficiency of object recognition and optical flow**. California Univ Berkeley Dept Of Electrical Engineering And Computer Sciences, 2014.
- [3] **An Intuitive Explanation of Convolutional Neural Networks**. The data science blog. 11 ago. 2016. Disponível em:
<https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>. Acesso em: 28 jun. 2020
- [4] BALSYS, Rokas. YOLOv3 Explained. **Pylessons**, 2019. Disponível em:
<https://pylessons.com/YOLOv3-introduction/>
- [5] BENGIO, Yoshua; GOODFELLOW, Ian; COURVILLE, Aaron. **Deep learning**. Massachusetts, USA:: MIT press, 2017.
- [6] BOCHKOVSKIY, Alexey; WANG, Chien-Yao; LIAO, Hong-Yuan Mark. **YOLOv4: Optimal Speed and Accuracy of Object Detection**. arXiv preprint arXiv:2004.10934, 2020.
- [7] BOWEN, Ray M.; WANG, Chao-cheng. **Introduction to vectors and tensors**. Courier Corporation, 2008.
- [8] BRADSKI, Gary; KAEHLER, Adrian. OpenCV. **Dr. Dobb's journal of software tools**, v. 3, 2000.

- [9] CASTRO, Leandro Nunes. Análise e síntese de estratégias de aprendizado para redes neurais artificiais. **Campinas: FEEC, UNICAMP. Dissertação de Mestrado-Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas**, p. 250, 1998. p. 12, 14, 16, 17
- [10] CHEN, Xi; WANG, Xiao; XUAN, Jianhua. **Tracking multiple moving objects using unscented Kalman filtering techniques**. arXiv preprint arXiv:1802.01235, 2018.
- [11] CHENG, Yizong. **Mean shift, mode seeking, and clustering**. *IEEE transactions on pattern analysis and machine intelligence* 17.8 (1995): 790-799.
- [12] **City Cameras**. São Paulo, 2019. Disponível em: <https://www.citycameras.prefeitura.sp.gov.br/>. Acesso em: 05 ago. 2020.
- [13] **Deep Learning Book**. Data Science Academy, 2019. Disponível em: <http://deeplearningbook.com.br/o-neuronio-biologico-e-matematico/>. Acesso em: 30 jun. 2020.
- [14] DE MAESSCHALCK, Roy; JOUAN-RIMBAUD, Delphine; MASSART, Désiré L. The mahalanobis distance. **Chemometrics and intelligent laboratory systems**, v. 50, n. 1, p. 1-18, 2000.
- [15] DERTAT, Arden. **Applied Deep Learning - Part 4: Convolutional Neural Networks**. 08 nov. 2017. Disponível em: <https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2>. Acesso em: 28 jun. 2020
- [16] GAO, Mingyu et al. Adaptive anchor box mechanism to improve the accuracy in the object detection system. **Multimedia Tools and Applications**, v. 78, n. 19, p. 27383-27402, 2019.

- [17] GOSWAMI, Subrata. **Reflections on Non Maximum Suppression (NMS)**. 13 jan. 2020. Disponível em:
<https://medium.com/@whatdhack/reflections-on-non-maximum-suppression-nms-d2fce148ef0a>. Acesso em: 30 jun. 2020
- [18] HAWKINS, Douglas M. The problem of overfitting. *Journal of chemical information and computer sciences*, v. 44, n. 1, p. 1-12, 2004.
- [19] HE, Kaiming et al. **Spatial pyramid pooling in deep convolutional networks for visual recognition**. *IEEE transactions on pattern analysis and machine intelligence*, v. 37, n. 9, p. 1904-1916, 2015.
- [20] HUANG, Rachel; PEDOEEM, Jonathan; CHEN, Cuixian. YOLO-LITE: a real-time object detection algorithm optimized for non-GPU computers. In: **2018 IEEE International Conference on Big Data (Big Data)**. IEEE, 2018. p. 2503-2510.
- [21] KETKAR, Nikhil. Introduction to keras. In: **Deep learning with Python**. Apress, Berkeley, CA, 2017. p. 97-111.
- [22] KOWSARI, Kamran et al. Text classification algorithms: A survey. **Information**, v. 10, n. 4, p. 150, 2019.
- [23] LIN, Tsung-Yi et al. Feature pyramid networks for object detection. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. 2017. p. 2117-2125.
- [24] _____. Microsoft coco: Common objects in context. In: **European conference on computer vision**. Springer, Cham, 2014. p. 740-755.
- [25] LU, Lu et al. Dying relu and initialization: Theory and numerical examples. **arXiv preprint arXiv:1903.06733**, 2019.

- [26] MACQUEEN, James et al. Some methods for classification and analysis of multivariate observations. In: **Proceedings of the fifth Berkeley symposium on mathematical statistics and probability**. 1967. p. 281-297.
- [27] MARC, Lelarge. Deep Learning Course. **Université PSL**, 2019. Disponível em: https://www.di.ens.fr/~lelarge/dldiy/slides/lecture_6/index.html#1
- [28] MCCULLOCH, Warren S.; PITTS, Walter. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, v. 5, n. 4, p. 115-133, 1943.
- [29] MINSKY, M. Papert. S., Perceptrons. 1969.
- [30] MOLIN, Matheus. **YOLOv4-Counter-in-TF**. GitHub. 2020. Disponível em: <https://github.com/molimat/YOLOv4-Counter-in-TF>. Acessado em: 03 de outubro de 2020.
- [31] MOREIRA, M. E. P.; NETO, W. A. P. Proposição de um método de levantamento e análise de dados para um diagnóstico de um corredor viário urbano. **PLURIS 2006**, 2006.
- [32] NG, Andrew. Non-maximum suppression. **Coursera**, 2018. Disponível em: <https://www.coursera.org/lecture/convolutional-neural-networks/non-max-suppression-dvrjH>
- [33] NWANKPA, Chigozie et al. Activation functions: Comparison of trends in practice and research for deep learning. **arXiv preprint arXiv:1811.03378**, 2018.
- [34] _____. Activation functions: Comparison of trends in practice and research for deep learning. **arXiv preprint arXiv:1811.03378**, 2018.

- [35] OLIPHANT, Travis E. **A guide to NumPy**. USA: Trelgol Publishing, 2006.
- [36] PROSSER, Patrick. Geometric Algorithms. **University of Glasgow**, 2000. Disponível em: <http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf>. Acesso em: 06 set. 2020.
- [37] REDMON, Joseph; FARHADI, Ali. **Yolov3**: An incremental improvement. arXiv preprint arXiv:1804.02767, 2018.
- [38] _____. You only look once: Unified, real-time object detection. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. 2016. p. 779-788.
- [39] ROEDER, Lutz. **YoloV4 Scheme**. Disponível em: <https://lutzroeder.github.io/netron/?url=https%3A%2F%2Fraw.githubusercontent.com%2FAlexeyAB%2Fdarknet%2Fmaster%2Fcfg%2Fyolov4.cfg>. Acesso em: 30 jun. 2020
- [40] ROSEBROCK, Adrian. Intersection over Union (IoU) for object detection. **Pyimagesearch**, 2016. Disponível em: <https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/>. Acesso em: 25 set. 2019.
- [41] ROSENBLATT, Frank. The perceptron: a probabilistic model for information storage and organization in the brain. **Psychological review**, v. 65, n. 6, p. 386, 1958.
- [42] SUN, Shijun; KIM, Yongmin. **Motion estimation within a sequence of data frames using optical flow with adaptive gradients**. U.S. Patent n. 6,480,615, 12 nov. 2002.
- [43] TensorFlow. **TensorFlow Lite guide**, 2016. Disponível em: <https://www.tensorflow.org/lite/guide>. Acesso em: 15 jul. 2020.

- [44] VANDITI, Jain. **Everything you need to know about “Activation Functions” in Deep learning models**. 30 dez. 2019. Disponível em:
[https://towardsdatascience.com/everything-you-need-to-know-about-activation-f
unctions-in-deep-learning-models-84ba9f82c253](https://towardsdatascience.com/everything-you-need-to-know-about-activation-functions-in-deep-learning-models-84ba9f82c253)
- [45] WANG, Chien-Yao et al. **CSPNet: A new backbone that can enhance learning capability of cnn**. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops. 2020. p. 390-391.
- [46] WOJKE, Nicolai; BEWLEY, Alex; PAULUS, Dietrich. Simple online and realtime tracking with a deep association metric. In: **2017 IEEE international conference on image processing (ICIP)**. IEEE, 2017. p. 3645-3649.