UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

RAIMUNDO VIDAL DE SOUSA JUNIOR

ESTUDO DE MÉTODOS DE INTELIGÊNCIA COMPUTACIONAL PARA DETECÇÃO DE HUMANOS EM IMAGENS DE WEBCAM

FORTALEZA, CE

2018

RAIMUNDO VIDAL DE SOUSA JUNIOR

ESTUDO DE MÉTODOS DE INTELIGÊNCIA COMPUTACIONAL PARA DETECÇÃO DE HUMANOS EM IMAGENS DE WEBCAM

Monografia apresentada ao Curso de Engenharia Elétrica da Universidade Federal do Ceará, como requisito parcial à obtenção do título de graduado em Engenharia Elétrica

Orientador: Prof. Dr. Arthur Plinio de Souza Braga

FORTALEZA

2018

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

V692e Sousa Junior, Raimundo Vidal de.
Estudo de métodos de inteligência computacional para detecção de humanos em imagens de webcam / Raimundo Vidal de Sousa Junior. – 2018.
80 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Tecnologia. Curso de Engenharia Elétrica. Fortaleza. 2018.
Orientação: Prof. Dr. Arthur Plinio de Souza Braga.

1. Redes Neurais Convolucionais Profundas. 2. Histograma de Gradientes Orientados. 3. Caffe. 4. Mobilenet. 5. Single Shot Multibox Detector. I. Titulo.

CDD 621.3

RAIMUNDO VIDAL DE SOUSA JUNIOR

ESTUDO DE MÉTODOS DE INTELIGÊNCIA COMPUTACIONAL PARA DETECÇÃO DE HUMANOS EM IMAGENS DE WEBCAM

Esta monografia foi julgada adequada para a obtenção do grau de Graduado em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Aprovada em: ___/___/_____.

BANCA EXAMINADORA

_____
Prof. Dr. Arthur Plinio de Souza Braga (Orientador)

_____
Prof. Dr. Bismark Claure Torrico

_____
Msc. Eng. Magno Prudêncio de Almeida Filho

AGRADECIMENTOS

Aos meus pais Maria das Graças e Raimundo Vidal pelo apoio, incentivo e principalmente por todos os sacrifícios que fizeram para que eu pudesse ter a educação que tive, sem eles não chegaria a lugar algum.

Aos meus familiares, principalmente minha madrinha Genecilda, por ter me criado desde pequeno, minha irmã Nair por me apoiar desde sempre e minha irmã Viviana por me ensinar a ser uma pessoa melhor.

Aos meus irmãos que a vida me deu Alex Sousa e Afonso Morais, por todo o apoio que me deram, toda a alegria que me proporcionaram e pelos 24 anos de convivência que me enriqueceram como pessoa e fortaleceram nossa amizade, cujos laços se estenderão por toda a minha vida.

Aos meus amigos Matheus Jonathan, Tobias Valentim, Matheus Nogueira, Clayton Paiva, Dalmo Mendes, Renan Sousa, Felipe Guedes, Felipe Porto, Alysson Santos, Gabriel Sampaio e Herivelton Távora, por terem me ajudado nos momentos difíceis, me acompanhado nas madrugadas de estudo e por me proporcionarem momentos de alegria e descontração que pra sempre carregarei comigo na memória e no coração.

A todos os demais amigos e companheiros de curso: Enzo, Fábio, Edmundo, Letícia, Mosquetti e Mozart, pela amizade e por todos os momentos de alegria.

Aos participantes da banca examinadora pelo tempo e dedicação.

“Não existe triunfo sem perda, não há vitória sem sofrimento, não há liberdade sem sacrifício”. (TOLKIEN, J.R.R, **Senhor dos Anéis**, 2003)

RESUMO

Este trabalho apresenta a implementação de dois métodos diferentes de detecção de humanos, um por meio de uma ferramenta do *Matlab* que utiliza histogramas de gradientes orientados alimentando uma máquina de vetor de suporte, e outro utilizando uma rede neural convolucional profunda. São apresentados conceitos básicos necessários para a compreensão dos métodos, incluindo definições de redes neurais artificiais, convoluções e cálculo de gradientes. A estrutura do trabalho se divide em duas partes: *PeopleDetector*, utilizado no primeiro método de detecção, e *Mobilenet-SSD*, utilizado no segundo método. Na primeira parte, são explicados os conceitos necessários para a compreensão da ferramenta de detecção e seus parâmetros. Na segunda parte são apresentadas definições básicas de redes neurais convolucionais e é apresentada a arquitetura *Caffe* e os modelos *MobileNet* e *Single Shot MultiBox Detector* utilizados para implementação. O primeiro método é implementado no *Matlab* e o segundo método é implementado em Python utilizando a biblioteca *OpenCV* e o *Imutils*. O *PeopleDetector* funciona melhor em situações em que os humanos na imagem estão em posição vertical, mas ainda assim apresenta alguns falso-positivos. A rede neural convolucional profunda apresentou excelente resultados tanto em precisão quanto em velocidade, mostrando-se uma possível opção para aplicações embarcadas.

Palavras-chave: Histograma de Gradientes Orientados. Máquina de Vetores de Suporte. Redes Neurais Convolucionais Profundas. Caffe. MobileNet-SSD.

ABSTRACT

This undergraduate thesis presents the implementation of two different human detection algorithms, one using a Matlab tool that uses histograms of oriented gradients feeding a support vector machine, and other using a deep convolutional neural network. Basic concepts that are necessary to comprehend the methods, like neural network definitions, convolutions and gradient calculation, are presented. The basic structure of this thesis can be divided into two parts: Peopledetector, utilized on the first detection method, and Mobilenet-SSD, utilized on the second one. The necessary concepts to the comprehension of the detection tool and its parameters are presented on the first part. On the second part, the Caffe architecture and the Mobilenet and Single Shot Multibox Detector models are explained in more details. The first method is implemented on Matlab while the second algorithm is programmed in Python using the OpenCV and Imutils libraries. The PeopleDetector works better on situations that involves humans standing in upright positions, and even then, the method presents some false positive detections. The deep convolutional neural network presented excellent results in precision and speed, showing as a possible option for embedded applications.

Keywords: Histogram of Oriented Gradients. Support Vector Machines. Deep convolutional neural network. Caffe. MobileNet-SSD.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ABNT	Associação Brasileira de Normas Técnicas
HOG	Histogram of Oriented Gradients
SVM	Support Vector Machine
ROI	Region of Interest
SSD	Single Shot MultiBox Detector
IoU	Intersection over Union
COCO	Common Objects in Context
VOC2012	Visual Object Classes Challenge 2012
GPAR	Grupo de Pesquisa em Automação, Controle e Robótica
mAP	Mean Average Precision

SUMÁRIO

CAPÍTULO 01: INTRODUÇÃO

De acordo com Dollar et al. (2011):

> "Pessoas estão entre os componentes mais importantes do ambiente de uma máquina e dotá-las da habilidade de interagir com pessoas é um dos mais interessantes e potencialmente úteis desafios". (Dollar et al, 2011)

Visto isso, é possível afirmar que a detecção de humanos e a visão computacional são campos de pesquisa que podem ser aplicados em diversas áreas como robótica, entretenimento, vigilância, cuidado de idosos e indexação baseada em conteúdo.

Existem diversas formas de realizar a detecção de humanos, porém um método promissor é a aprendizagem profunda que, segundo Voulodimos *et al* (2017), tem apresentado um desempenho melhor que os antigos métodos que compunham o estado da arte, principalmente nas áreas de visão computacional. Visto isso, um modelo de rede neural convolucional terá sua performance comparada com um método de detecção de humanos mais antigo.

O primeiro capítulo serve de introdução para o trabalho e é subdividido em quatro tópicos: objetivos, motivação, metodologia e estrutura do trabalho.

1.1. Objetivos

Este trabalho tem como objetivo estudar e implementar dois métodos de detecção de humanos diferentes utilizando como fonte de dados imagens obtidas a partir de uma webcam. Um método utilizando histograma de gradientes orientados e outro utilizando uma rede neural convolucional profunda.

1.2. Motivação

A motivação inicial desse projeto era implementar um método de detecção de humanos com o intuito de utiliza-lo para realizar o planejamento de trajetória de um robô, porém ao realizar uma pesquisa extensiva sobre os algoritmos existentes na literatura, identificou-se um novo problema: a escolha do melhor método de detecção de humanos para a

aplicação. Visto isso, o tema do projeto foi modificado para a implementação de dois métodos de detecção diferentes e a sua comparação.

1.3. Metodologia

O trabalho abrangerá dois métodos principais de detecção de humanos: o primeiro é um detector de humanos, chamado de *PeopleDetector*, desenvolvido pela equipe do *Matlab* e presente na toolbox de processamento de imagem que utiliza um histograma de gradientes orientados (HOG) para alimentar uma máquina de vetores de suporte (SVM), o segundo é um modelo de rede neural convolucional profunda aplicada na arquitetura *Caffe*, utilizando *Python* e *Opencv*,

1.4. Estrutura do Trabalho

No Capítulo 1, são apresentados objetivos, motivação e metodologia utilizados no desenvolvimento do trabalho

No Capítulo 2, é explicado brevemente o primeiro método de detecção de humanos utilizado, o *PeopleDetector* e seus parâmetros.

No Capítulo 3, são introduzidos conceitos básicos necessários para a compreensão de redes neurais convolucionais, bem como a arquitetura utilizada para implementação dos modelos.

No Capítulo 4, são explicados dois modelos bem difundidos na literatura, um utilizado para classificar objetos e outro utilizado para detectar objetos, bem como o modelo utilizado para a implementação do segundo método de detecção.

No Capítulo 5, serão apresentados detalhes da implementação dos dois métodos de detecção de humanos.

Na parte final é apresentado a conclusão, bem como as linhas de trabalhos futuros.

CAPÍTULO 02: *PeopleDetector*

O *Matlab* possui diversas bibliotecas muito poderosas para inúmeras aplicações, uma dessas bibliotecas se chama *Computer Vision System Toolbox*, e nela estão presentes inúmeras ferramentas de processamento de imagem, das mais simples, como calibradores de câmera estéreo, aos mais complexos, como arquiteturas de redes neurais convolucionais profundas.

Dentre as ferramentas presentes na biblioteca supracitada, existe o *PeopleDetector*, um detector de pessoas pré-treinado que utiliza um histograma de gradientes orientados para extração de características da imagem que então alimentam uma máquina de vetores de suporte que, após intenso treinamento, classifica a imagem em duas classes, humano ou não-humano.

Existem diversos parâmetros que podem ser alterados na ferramenta para adaptar o processamento de forma a melhor detectar as pessoas nas condições do experimento realizado, porém, antes de explica-los, será explanado brevemente as etapas do processamento de imagem implementado nessa ferramenta.

Esse capítulo é dividido nos seguinte subtópicos: a seção 2.1 apresenta os conceitos de Histograma de gradientes orientados; A seção 2.2 descreve Máquina de Vetores de Suporte; A seção 2.3 trata do método PeopleDetector, onde são resumidos os parâmetros e características da ferramenta do *Matlab*; A seção 2.4 resume o método e o exemplifica com o auxílio de imagens para facilitar a compreensão.

2.1. Histograma de Gradientes Orientados (HOG)

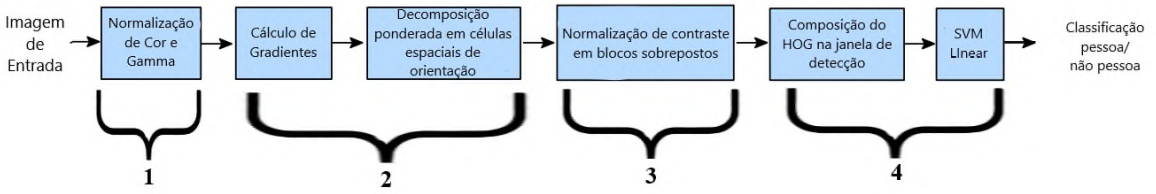
A detecção de humanos é uma tarefa bastante complexa. Segundo Dalal e Triggs (2005, p. xi):

> Detectar humanos em imagens é uma tarefa desafiadora devido a sua aparência variável e a vasta gamas de posições que eles podem assumir. A primeira necessidade é um conjunto de características robusta que permita que o formato de humanos seja discriminado claramente, mesmo em fundos desordenados e de iluminação ruim (Dalal e Triggs, 2005)

O histograma de gradientes orientados foi a forma encontrada por Dalal e Triggs (2005, p. xi) para obter as características da imagem que seriam então utilizadas para alimentar a máquina de vetores de suporte. Nos tópicos seguintes serão explanados brevemente etapas do processo de extração dessas características.

No método a imagem é varrida por uma janela de detecção e, em cada posição de sobreposição, o processo descrito pela Figura 1 abaixo é aplicado e aquela janela é então classificada como uma porção da imagem que contém um humano ou não. Após a composição do resultado da sobreposição da janela de detecção em todos as posições da imagem, um algoritmo de supressão de não-máximo é aplicado para eliminar as detecções redundantes e sobrepostas.

Figura 1 – Extração de HOG e classificação de imagem



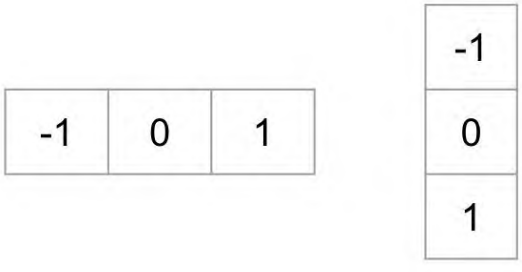
Fonte: Dalal e Triggs, 2005

Percebe-se que cada iteração possui basicamente 4 etapas principais, denotadas pelos números na Figura 1, pré-processamento, bastante comum em aplicações de processamento de imagens, cálculo de gradiente e decomposição em um histograma, normalização dos histogramas e classificação utilizando a máquina de vetor de suporte. As últimas três etapas serão brevemente explicadas nos subtópicos seguintes.

2.1.1. Cálculo do Histograma de Gradientes

Cada janela de detecção é subdividida em células. O cálculo de gradientes é, então, feito aplicando os seguintes filtros na imagem conforme a Figura 2:

Figura 2– Filtros aplicados para cálculo de gradiente



Fonte: Satya Mallick, 2016

O vetor da esquerda é o filtro referente ao gradiente no eixo x, ou eixo horizontal, e o da direita referente ao gradiente no eixo y, ou eixo vertical. O resultado da aplicação desse filtro numa porção da imagem nada mais é que a diferença entre a intensidade de pixels vizinhos.

Cada pixel da imagem resultante do gradiente terá um valor de gradiente no eixo x e um valor de gradiente no eixo y, é calculado então a intensidade e direção do vetor gradiente naquele ponto da seguinte forma:

$$|g| = \sqrt{g_x^2 + g_y^2} \quad (1)$$

$$\theta = \tan^{-1}(\frac{g_y}{g_x}) \quad (2)$$

Sendo: g é o módulo do vetor gradiente, g_x e g_y são os gradientes no eixo x e y respectivamente e θ é o ângulo do gradiente.

É formada, então, uma matriz de módulos e uma matriz de direções correspondentes para cada pixel de cada célula da janela de detecção. Os gradientes são então ponderados num histograma com ângulos fixos (0, 20º, 40º,60º,80º,100º,120º,140º,160º) de forma que cada vetor gradiente contribuirá para o respectivo ângulo do histograma de gradiente. Ou seja, visto que o ângulo do módulo ou será igual a um dos ângulos do histograma ou estará entre dois deles, o modulo de cada gradiente será decomposto baseado em seu ângulo e somado em uma ou duas das posições do gradiente.

Ao fim do processo acima, os valores de gradiente dos pixels da célula são convertidos em 2 matrizes, uma composta de magnitudes e a outra composta de seus respectivos ângulos e então esses valores são ponderados em um vetor de apenas 9 valores, cada um sendo a contribuição dos pixels da célula referente a cada um dos 9 ângulos predefinidos.

2.1.2. Normalização

Antes de explicar a normalização utilizada no algoritmo desenvolvido por Dalal e Triggs, será explicada uma normalização mais simples. A norma L2 é um método que consiste em dividir os valores de cada elemento de um vetor pelo módulo do vetor em si, que é calculado segundo as equações a seguir.

$$Y = \frac{x}{|x|} \quad (3)$$

$$|x| = \sqrt{\sum_{k=1}^{n} |x_k|^2} \quad (4)$$

Sendo: Y é o vetor x normalizado.

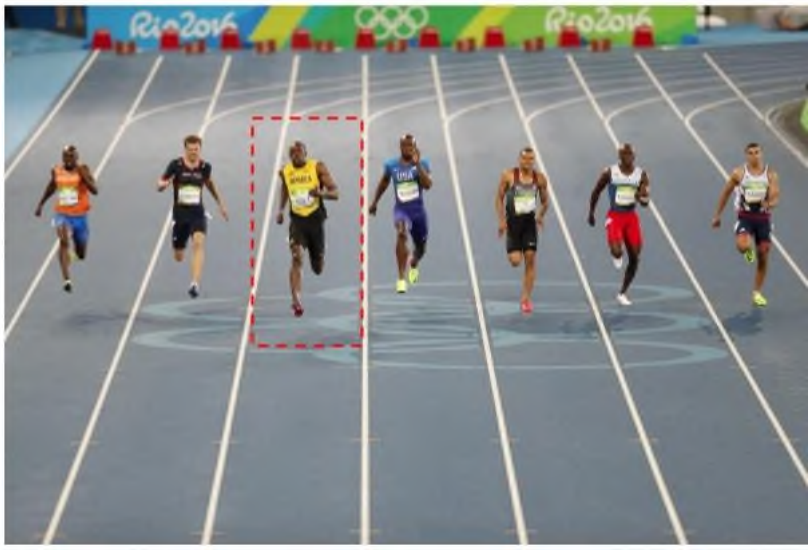
O método utilizado por Dalal e Triggs é uma variação do exemplo acima e chama-se L2-Hys, ele consiste numa normalização igual a superior, porém limitando o valor máximo dos elementos do vetor para 0,2, em seguida é aplicada uma segunda normalização utilizando o vetor resultante da primeira normalização.

Essa normalização é aplicada de quatro em quatro células, logo será aplicada em vetores de 36x1 elementos visto que cada célula possui um gradiente composto de 9 elementos. O resultado de cada normalização é então concatenado e utilizado para alimentar uma máquina de vetores de suporte que, após treinamento, classifica a janela de detecção que representa área da imagem de entrada, como humano ou não-humano.

2.1.3. Resumo do método

O algoritmo supracitado pode ser um pouco complexo de entender sem um auxílio gráfico, nesse subtópico será resumido o algoritmo etapa a etapa e serão apresentadas algumas imagens com o intuito de facilitar a compreensão do método. Vale ressaltar que todos os exemplos serão utilizando os parâmetros, como tamanho da janela de detecção e células, definidos na ferramenta do *Matlab*.

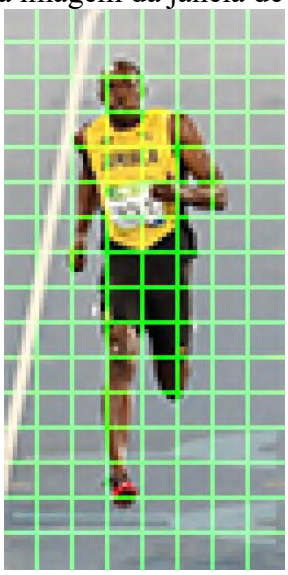
Figura 3– Exemplo de janela de detecção em imagem



Fonte: Satya Mallick, 2016

Na Figura 3, é possível ter a demonstração gráfica da janela de detecção parada em um local da imagem, lembrando que essa janela varre toda a imagem em busca por humanos. Após a janela parar em um local, são iniciados os processos que foram descritos anteriormente.

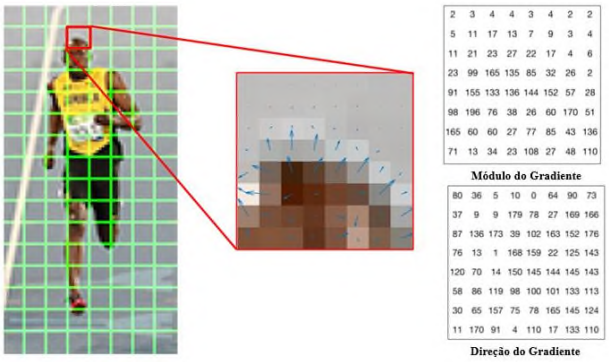
Figura 4– Divisão da imagem da janela de detecção em células



Fonte: Satya Mallick, 2016

A imagem delimitada na janela de detecção é então dividida em células de 8x8 pixels cada, conforme a Figura 4, é calculado, então, o gradiente para cada um dos 64 pixels de cada uma das 128 células. A imagem abaixo exemplifica a matriz de módulos e orientações dos 64 gradientes de 1 célula, bem como mostra graficamente o sentido dos gradientes.

Figura 5– Matrizes de módulo e orientação do gradiente



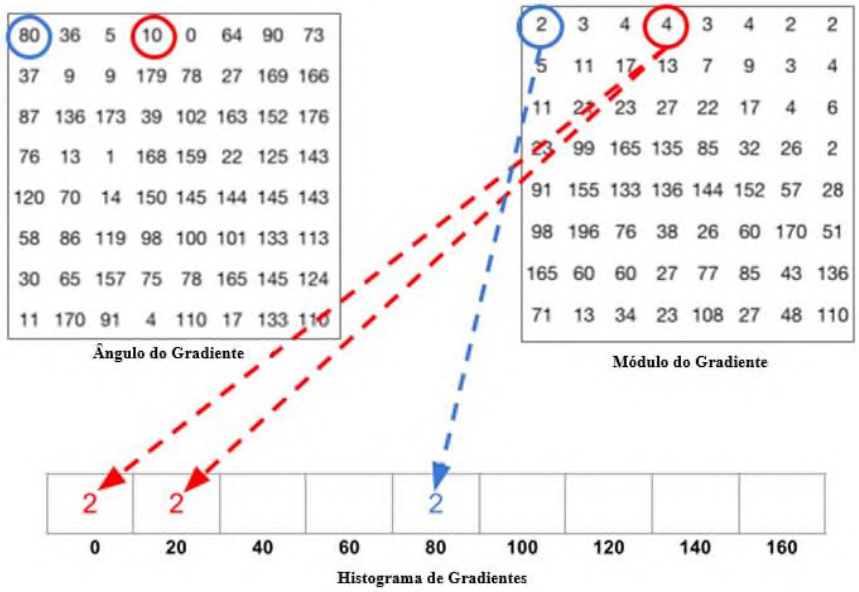
Fonte: Satya Mallick, 2016

Com as informações de ângulo e direção do gradiente, é possível agora decompor o módulo dos gradientes nos ângulos pré-definidos para gerar o histograma de gradientes.

A Figura 6 exemplifica esse processo, na matriz da esquerda estão dispostos os ângulos de orientação dos gradientes de cada pixel, e no da direita seus respectivos módulos. É possível constatar os cálculos explanados anteriormente ao observar, por exemplo, os valores

no gradiente para o grau 0 e grau 20. A magnitude dos dois é igual a dois, visto que a magnitude do gradiente de 10º é 4 e 10º está igualmente distante de 0º e 20º, portanto contribui igualmente para os dois.

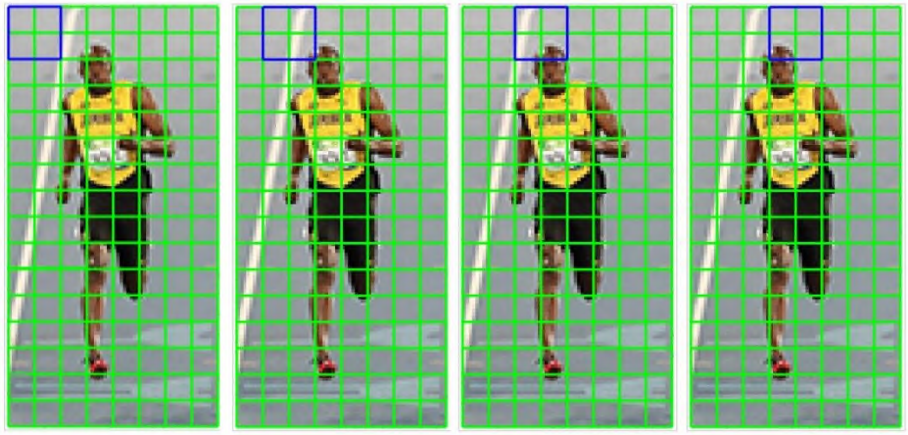
Figura 6– Exemplo de ponderação de gradientes para montagem do histograma.



Fonte: Satya Mallick, 2016

Por último, é realizada a normalização sobreposta dos histogramas de gradientes referentes aos blocos de 2x2 células. A Figura 7 exemplifica a varredura do bloco por toda a imagem. Por fim, o resultado de todas essas normalizações é concatenado no vetor de atributos.

Figura 7– Exemplo de ponderação de gradientes para montagem do histograma.



Fonte: Satya Mallick, 2016

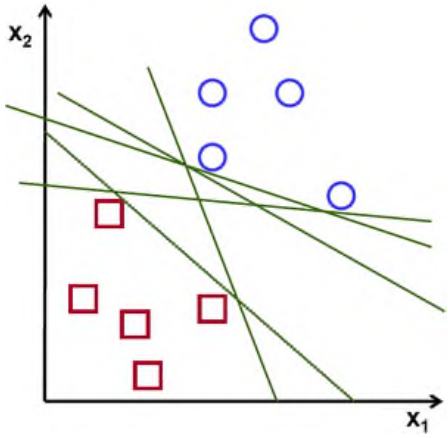
2.2. Máquina de Vetores de Suporte (SVM)

A Máquina de Vetor de Suporte é um método de aprendizagem supervisionada muito utilizado em problemas de regressão e classificação e é baseada no conceito de planos de

decisões que definem uma linha de separação entre duas classes, comumente chamada de hiperplano. Nesse algoritmo, os dados são plotados num espaço com n dimensões, cada uma representando os n parâmetros, e o objetivo do método é delimitar o hiperplano que melhor separa os dados em duas classes.

A título de simplificação, será utilizado de exemplo um modelo que possui apenas dois parâmetros, $\boldsymbol{x_1}$ e $\boldsymbol{x_2}$, e os vetores serão classificados em duas classes: círculo e quadrado.

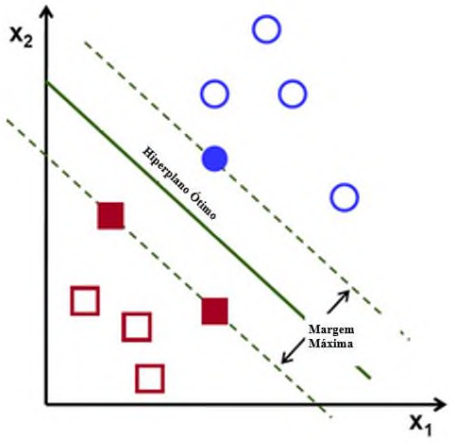
Figura 8– Exemplo de possíveis hiperplanos de uma SVM



Fonte: Rohith Gandhi, 2018

Na Figura 8, é possível ver que existem diversos hiperplanos que separam o espaço e delimitam corretamente as duas classes. O objetivo principal do algoritmo de SVM é encontrar o hiperplano que melhor separa as duas classes, e isso é feito selecionando o hiperplano que possui a maior margem entre os extremos de exemplo positivo e negativo, conforme a Figura 9.

Figura 9 – Exemplo de hiperplanos ótimo de uma SVM



Fonte: Rohith Gandhi, 2018

Para isso, a margem é calculada como a distância entre o hiperplano ótimo e os vetores de suporte, definido como os pontos de cada classe mais próximos do hiperplano, representados na imagem como os quadrados e o círculo preenchidos. Esse conceito pode ser estendido para problemas com mais dimensões, como no caso do algoritmo utilizado que possui, conforme mencionado anteriormente, 3780 parâmetros.

2.3. PeopleDetector

Agora que o algoritmo de extração de atributos por meio do histograma de gradientes orientados foi brevemente explicado, é possível dar mais detalhes sobre a ferramenta de detecção de pessoas presente na biblioteca de processamento de imagens do *Matlab*, que utiliza um histograma de gradientes orientados para alimentar uma SVM já treinada pela equipe do *Matlab*.

Ao utilizar essa ferramenta, cria-se um objeto no *Matlab* com algumas características que podem ser alteradas para melhorar a detecção, no subtópico a seguir, essas características serão explicadas brevemente.

2.3.1. Propriedades modificáveis

Figura 10 – Propriedades presentes no detector

```
detector =

  System: vision.PeopleDetector

  Properties:
        ClassificationModel: 'UprightPeople_96x48'
    ClassificationThreshold: 2.3
                    MinSize: []
                    MaxSize: []
                ScaleFactor: 1.01
               WindowStride: [8 8]
           MergeDetections: true
                     UseROI: false
```

Fonte: o próprio autor

A imagem acima foi retirada do *Matlab* depois de definir o objeto detector como sendo a ferramenta de detecção de pessoas, percebe-se que existem 8 propriedades presentes, cada uma delas altera fatores importantes no processo de detecção de humanos.

2.3.1.1 ClassificationModel

Essa propriedade pode assumir dois valores: *UprightPeople_128x64* e *UprightPeople_96x48*, ao alterar essa propriedade, altera-se também o modelo utilizado como um todo, pois a ferramenta possui dois modelos, um treinado com uma janela de detecção de 128x64 pixels e, consequentemente, imagens de treinamento de mesmo tamanho, e outro treinado com uma janela de detecção de 96x48 pixels. Como o modelo detecta pessoas no interior da janela de detecção, utilizar o modelo treinado com uma janela menor tenderá a ser mais eficiente em situações que os humanos a serem detectados tenham dimensões menores na imagem.

2.3.1.2 ClassificationThreshold

Essa propriedade é um escalar não negativo que indica o limite de classificação de pessoa, basicamente, quanto maior o seu valor, mais rigoroso é o algoritmo de classificação nas janelas de detecção. É útil para reduzir o número de falso positivos nas detecções aumentando o seu valor.

2.3.1.3 MinSize

Um vetor contendo 2 elementos, largura e altura em pixels, do tamanho da menor região contendo um humano. Pode ser utilizado para reduzir o poder de processamento necessário para a execução do código. Muitas vezes é interessante definir essa propriedade como o tamanho da janela de detecção escolhida no *ClassificationModel*.

2.3.1.4 MaxSize

Um vetor contendo 2 elementos, largura e altura em pixels, do tamanho da maior região contendo um humano. Pode ser utilizado para reduzir o poder de processamento necessário para a execução do código quando é possível restringir o tamanho, em pixels, que o humano poderá assumir na situação de utilização do programa.

2.3.1.5 ScaleFactor

A ferramenta detecta humanos de tamanhos, em pixels, específicos, essa propriedade é um escalar não negativo e com valor mínimo de 1,0001 que define o passo a ser

dado de forma incremental no tamanho detectável de humano, partindo do *MinSize* e atingindo o *MaxSize*. A utilização de baixos valores para essa propriedade melhora a precisão do algoritmo, mas aumenta consideravelmente o tempo de processamento.

2.3.1.6 WindowStride

Essa propriedade pode ser modificada utilizando como entrada um vetor [x y] ou um escalar a, ao inserir o escalar, a ferramenta entende que $x=y=a$. Define o passo a ser dado, no eixo x e eixo y, pela janela de detecção. A diminuição dessa propriedade melhora a precisão, mas compromete o tempo de processamento.

2.3.1.7 MergeDetections

Essa propriedade booleana, aceita valores de verdadeiro ou falso, e ela controla se, ao detectar múltiplos humanos na mesma imagem, a caixa que os delimita na imagem deve ser mesclada quando houverem sobreposições, ou não. Ao selecionar verdadeiro, caso existam 2 humanos muito próximos um do outro, o algoritmo retornará apenas uma caixa delimitadora contendo os dois humanos dentro dela, em vez de duas caixas individuais.

2.3.1.8 UseROI

Essa propriedade booleana, aceita valores de verdadeiro ou falso, e ela controla se foi delimitada uma região de interesse na imagem, de modo a detectar humanos apenas nessa região. Caso essa propriedade seja verdadeira, ao chamar a função, é necessário informar qual é a região de interesse (ROI)

2.3.2. Argumentos da Ferramenta

Essa ferramenta possui, basicamente, três argumentos de entrada: A imagem em si, a região de interesse, quando a propriedade UseROI for verdadeira, e o modelo a ser utilizado, que pode ser *UprightPeople_128x64* e *UprightPeople_96x48*

A saída dessa ferramenta é feita por meio de um objeto, que para detecção de humanos, possui, basicamente, dois argumentos de saída, supondo que o algoritmo detectou n humanos na imagem: n escalares com a porcentagem de confiança de haver realmente um

humano em cada uma das n janelas e uma matriz n x 4, onde n é o número de pessoas detectadas na imagem e as outras 4 dimensões definem a aresta superior direita, x e y, e a largura e altura da janela contendo o humano.

CAPÍTULO 03: REDES NEURAIS CONVOLUCIONAIS PROFUNDAS

As redes neurais profundas são uma evolução das redes neurais artificiais e foram responsáveis por vários avanços em diversas áreas, como visão computacional, reconhecimento de fala, entre outros. Para a detecção de humanos foi utilizado um modelo de rede neural convolucional profunda, mas antes de explicar os seus detalhes, serão explanados conceitos básicos e necessários para a compreensão da rede, começando por definições de redes neurais artificiais, seguido da distinção entre redes rasas e redes profundas, após isso, definições matemáticas do processo em redes neurais convolucionais e finalmente a arquitetura utilizada para implementação da rede.

Esse capítulo é dividido nos seguintes subtópicos: A seção 3.1 apresenta conceitos básicos sobre as redes neurais, como o neurônio, funções de ativação, camadas e métodos de aprendizagem; A seção 3.2 explana conceitos básicos para a compreensão da estrutura de uma rede convolucional; A seção 3.3 explica a arquitetura utilizada para implementar o modelo da rede bem como seus parâmetros e principais funções.

3.1. Redes Neurais Artificiais

Segundo Simon Haykin (2001), a definição de rede neural artificial é a seguinte:

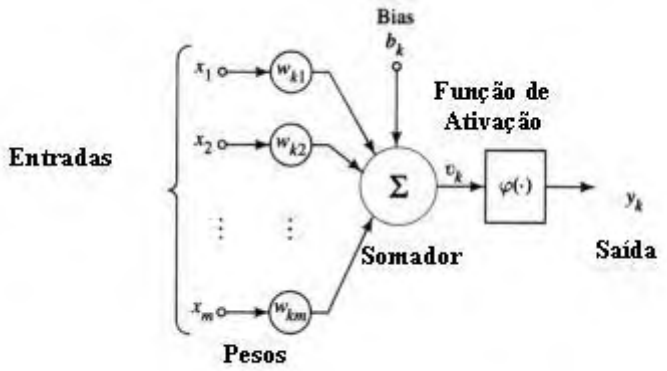
> Uma rede neural é um processador maciçamente paralelamente distribuído constituído de unidades de processamento simples, que têm a propensão natural para armazenar conhecimento experimental e torná-lo disponível para uso. Ela se assemelha ao cérebro em dois aspectos: o conhecimento é adquirido pela rede a partir de seu ambiente através de um processo de aprendizagem e forças de conexões entre neurônios, conhecidas como pesos sinápticos, são utilizadas para armazenar o conhecimento adquirido. (Simon Haykin, 2001)

Existem diversos tipos de redes neurais artificiais, mas uma característica comum a todas elas, é a existência do processo de aprendizado que, assim como no cérebro humano, se dá através da experiência. As unidades de processamento simples referidas no trecho acima são chamadas de neurônios e são a essência das redes artificiais.

3.1.1 Neurônio

O neurônio é a unidade básica de processamento das redes neurais artificiais e é fundamental para a sua operação. O seu modelo matemático foi criado utilizando como inspiração o neurônio humano e pode ser definido conforme a Figura 11.

Figura 11– Diagrama de blocos do modelo matemático do neurônio



Fonte: Simon Haykin, 2001

Segundo Haykin, pode-se separar o modelo em três elementos básicos:

- Um conjunto de sinapses constituído pelas entradas, representada pela letra *x*, e seus respectivos pesos, pela letra w*;*
- Um somador que adiciona o resultado das entradas ponderado com seus respectivos pesos e, por último, um valor de *bias* que representa a participação daquele neurônio como um todo no resultado da rede, definido como a letra b;
- uma função de ativação que insere não-linearidades no processo.

A saída do neurônio será o resultado dos seguintes processos: Primeiramente realiza-se uma soma de todas os valores de entrada, utilizando o peso de cada entrada como ponderação, que pode ser entendido como a representatividade daquela entrada na saída do neurônio. A esse resultado adiciona-se um valor de *bias* e então aplica-se uma não-linearidade, também conhecida como função de ativação. Esse processo está exemplificado na equação (8) considerando um neurônio k, que possui m entradas.

$$y_k = \varphi((\sum_{j=1}^{m} w_{kj}.x_j) + b_k)$$

(5)

3.1.2 Função de ativação

As funções de ativação têm fundamental importância nas redes neurais artificias e tem como principal finalidade inserir não-linearidades no processo, que caso não existissem, de

certa forma, limitariam o aprendizado da rede pois elas seriam compostas puramente de operações lineares.

Existem diversas funções de ativação popularmente utilizadas nas diferentes arquiteturas das redes neurais, os modelos estudados utilizam a ativação linear retificada (ReLU) que, segundo Krizhevsky, Sutskever e Hinton:

> Redes neurais convolucionais profundas com ReLu treinam várias vezes mais rápido que seus equivalentes utilizando tangente hiperbólica. (Krizhevsky, Sutskever e Hinton, 2012)

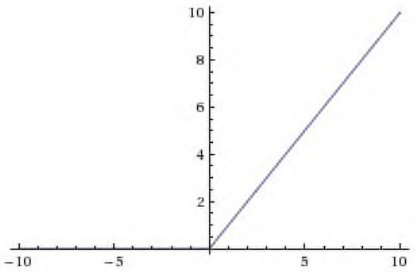
3.1.2.1 ReLu

A função de ativação linear, ou ReLU, é matematicamente simples e pode ser descrita da seguinte forma:

$$ReLU(x) = \max(0, x) \tag{6}$$

Percebe-se então que, basicamente, a única operação que a função ReLU realiza é zerar o valor de todos os neurônios menores que 0. Valer ressaltar que o ReLu é uma função simples, diferenciável e com derivada simples, 0 para valores não positivos e 1 para valores positivos, e por conta disso, facilitará a descida de gradiente, que será explicada a diante.

Figura 12– Representação gráfica da função ReLU



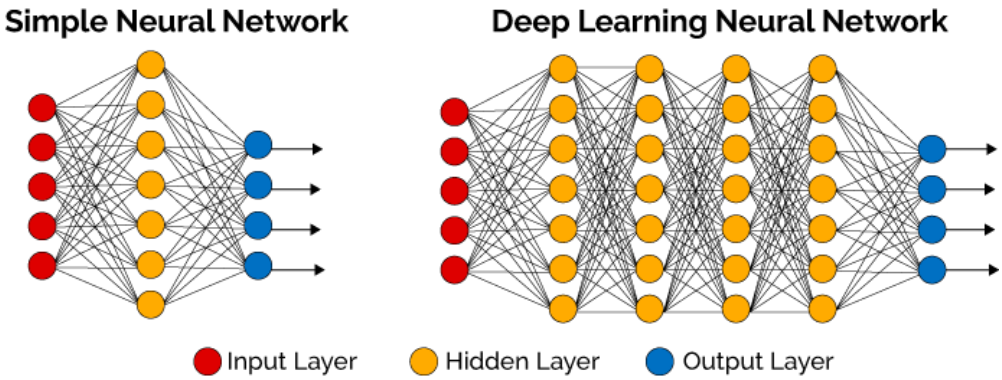
Fonte: Dans Becker, 2018

3.1.3 Camadas

As redes neurais artificiais organizam-se em camadas de neurônio, a primeira camada é chamada de camada de entrada, a última camada é chamada de camada de saída e as camadas entre as duas são chamadas de camadas ocultas. A principal diferença entre uma rede

neural rasa e uma rede neural profunda é o número de camadas oculta, este possui diversas camadas enquanto aquele possui poucas.

Figura 13– Diferenças entre redes rasas e profundas



Fonte: Data Science Academy, 2018

A Figura 13 demonstra as camadas de duas redes diferentes, a da esquerda representa uma rede rasa e a da direita uma rede profunda. Cada círculo na imagem é um neurônio e cada linha é uma conexão com outro neurônio e obviamente terá um peso atrelado. É possível notar que desde a camada de entrada, as saídas dos neurônios de uma camada são as entradas dos neurônios na camada seguinte até chegar na camada de saída. Vale ressaltar, porém, que nem todo modelo de rede neural é como o da imagem, em que todos os neurônios de uma camada são conectados com todos os neurônios da camada seguinte.

3.1.4 Softmax

O *softmax* é uma função derivada da regressão logística e muitas vezes é implementado como a camada final de redes neurais de classificação de imagens onde múltiplas classes estão presentes. A função consiste em aplicar a função exponencial ponto a ponto do vetor de entrada e normalizar o resultado obtido utilizando a soma das exponenciais dos elementos do vetor, conforme a equação abaixo.

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}} \tag{7}$$

A título de exemplificação, supõem-se que foi uma rede foi projetada para classificar imagens e foram definidas três possíveis classes para as imagens, a última camada antes do *softmax* será projetada para ser uma camada com três neurônios, ou um vetor de três elementos. A função *softmax* normalizará os valores dos três neurônios de modo que ao final do processo, a soma dos elementos será igual a 1 e esses valores poderão ser encarados como a

probabilidade de encontrar cada classe na imagem de entrada. Supondo que, num dado momento, os valores dos três neurônios fossem 5,1, 3,3 e 2,5, o cálculo seria feito da seguinte forma:

$$Y = [5{,}1\ 3{,}3\ 2{,}5] \xrightarrow{sofmax} S(Y) = \left[\frac{e^{5,1}}{e^{5,1}+e^{3,3}+e^{2,5}}\ \frac{e^{3,3}}{e^{5,1}+e^{3,3}+e^{2,5}}\ \frac{e^{2,5}}{e^{5,1}+e^{3,3}+e^{2,5}}\right] \quad (8)$$

$$Y = \left[\frac{164{,}02}{203{,}32}\frac{27{,}11}{203{,}32}\frac{12{,}18}{203{,}32}\right] = [0{,}81\ 0{,}13\ 0{,}06] \quad (9)$$

No início do treinamento, o resultado dessa camada será praticamente randômico, dependendo apenas dos métodos de inicialização dos parâmetros da rede, mas após o processo de treinamento, onde o erro entre as reais classes presentes na imagem e esse vetor *softmax* será minimizado por meio dos ajustes de pesos e *biases*, os valores dessa camada se aproximarão às reais probabilidades de cada classe estar contida na imagem.

3.1.5 Aprendizagem

Em termos gerais, o processo de aprendizagem de uma rede neural artificial é composto de um algoritmo iterativo que computa o resultado esperado da rede com o resultado real, para isso é necessário um banco de dados para treinamento devidamente classificado. A título de simplificação, a explicação geral sobre aprendizagem será voltada para um problema de classificação de imagem.

Após ter seu modelo definido, a rede neural terá uma série de parâmetros modificáveis, pesos e *bias*, que alterarão a sua saída. Durante o processo de treinamento, a rede comparará, iterativamente, o valor estimado pela rede com o valor real e utilizará essa diferença para alterar o valor dos pesos e *bias*.

Supondo uma rede que classifique imagens em duas classes, imagem que contem cachorros e que não contem cachorros. Durante o processo de treinamento a rede calculará a probabilidade de a imagem conter um cachorro e comparará com a classificação real da imagem, essa diferença, ou uma função dessa diferença chamada de função de custo, será a ferramenta utilizada para guiar a modificação dos parâmetros da rede. Um método bastante empregado para a realização desse processo se chama descida de gradiente e será brevemente explicado a seguir.

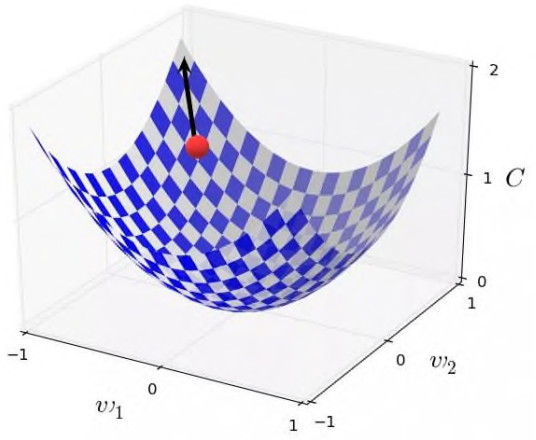
3.1.5.1 Descida de Gradiente

A otimização dos parâmetros da rede pode ser reduzida a um problema matemático bastante comum, o cálculo do mínimo global de uma função. Ainda utilizando o exemplo

anterior, a função de custo pode ser calculada em relação a última camada da rede, que nos daria a probabilidade da presença das classes. Como todas as camadas são combinações das anteriores, é possível calcular, utilizando a regra da cadeia, a função custo em relação às camadas anteriores, consequentemente, é possível calcular a função custo em relação aos pesos e *bias*. Basta agora, minimizar a função custo fazendo alterações nos valores de pesos e *bias*.

O gradiente de uma função é definido como a direção percorrida que acarretará no seu maior crescimento. Em outras palavras, o gradiente mede quais variações na entrada implicarão no maior crescimento na saída da função. Supondo que o neurônio de saída seja dependente apenas de dois neurônios de entrada e, portanto, só possuam dois valores de peso, w_1 e w_2, seria possível plotar uma função custo que seria influenciada por esses dois parâmetros, como a exposta abaixo.

Figura 14– Visualização de Gradiente



Fonte: Michael Nielsen, 2018

A Figura 14 representa graficamente a direção do gradiente de um ponto específico na função custo, representado pela seta preta e esfera vermelha. Os algoritmos de descida de gradiente utilizam o negativo do gradiente da função custo em relação aos pesos multiplicado por um fator de aprendizado, conforme a equação abaixo que retrata a iteração t do processo.

$$(w_t)_i = (w_{t-1})_i - \eta . \nabla C(w_{t-1})_i \quad (10)$$

A equação (10) nos mostra que o valor dos pesos é atualizado utilizando seu valor anterior, valores que indicam a posição da esfera na figura, e o negativo do gradiente multiplicado pelo fator de aprendizagem η.

A imagem acima representa um exemplo simples de apenas duas variáveis, mas o conceito pode ser estendido para hiperplanos de inúmeras variáveis. É importante mencionar

que existem diversos métodos de cálculo da função de custo e da descida de gradiente, os métodos específicos utilizados pela rede serão descritos no próximo capítulo.

3.2. Redes Convolucionais

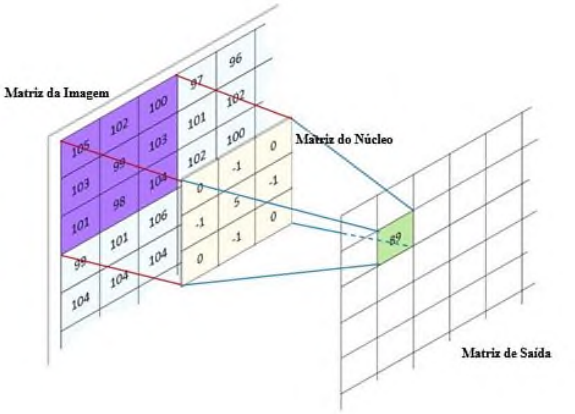
Como exposto no início do capítulo, a rede utilizada é uma rede neural artificial convolucional profunda, rede neural artificial pois é composta de neurônios matemáticos organizados em camadas conectadas, profunda pois é composta de muitas camadas ocultas entre a camada de entrada e a de saída, e convolucional pois sua estrutura é composta, primordialmente por camadas de convolução. A seguir serão explicados conceitos necessários para a compreensão de redes convolucionais.

3.2.1 Convolução

3.2.1.1 Definição geral

A convolução de imagens é uma operação matemática simples e consiste no produto interno entre uma matriz dos pixels da imagem e um matriz núcleo, ou kernell. O núcleo varre toda a imagem de entrada e cada produto interno, obtido da sobreposição de uma porção da imagem e o núcleo, se torna um pixel na imagem de saída, conforme exemplificado na Figura 15.

Figura 15– Exemplo de Convolução em imagem



Fonte: Utkarsh Sinha, 2018

Como o núcleo da Figura 15 é uma matriz 3x3, cada pixel de saída será o resultado

do produto interno de uma porção correspondente a uma matriz 3x3 da entrada e o núcleo. Esse núcleo é movido a cada iteração e seleciona mais 9 pixels para gerar outro pixel de saída. A matriz resultado da convolução é comumente chamada de mapa de características.

Núcleos diferentes produzem um efeito diferente na imagem de saída e realça características diferentes da imagem de entrada. O efeito de diferentes núcleos pode ser evidenciado na Figura 16.

Figura 16– Efeito de diferentes núcleos de convolução

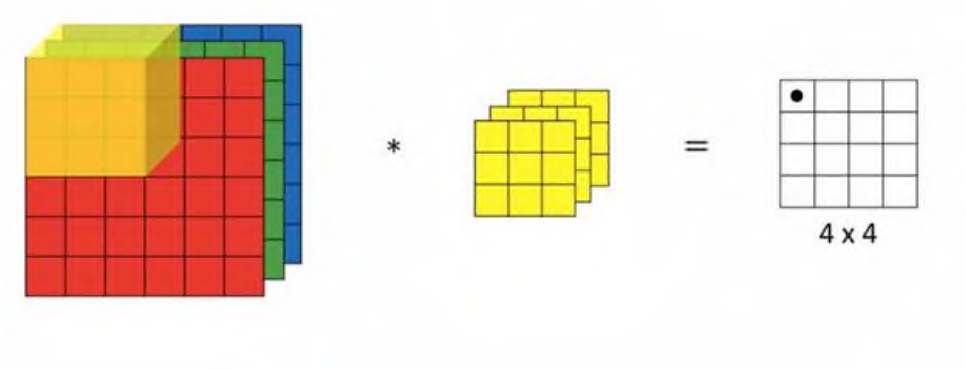
Fonte: Utkarsh Sinha, 2017

3.2.1.2 Convolução em volumes

Uma imagem monocromática de tamanho 300x300 pixels podem ser entendidas como uma matriz 300x300 em que cada valor representa a intensidade de cada pixel da imagem, porém para imagens coloridas esse não é o caso, pois existem 3 matrizes, uma parada cada cor fundamental, então existem mudanças sutis na forma de entender graficamente o resultado da convolução.

A figura abaixo representa o processo de convolução de uma imagem colorida de tamanho 6x6 pixels, pode-se entender a imagem como um cubo de dimensões $h\ x\ w\ x\ c$, onde h é a altura da imagem em pixels, w é a largura da imagem em pixels e c representa o número de canais da imagem. Como a imagem possui três canais, um para cada cor, o núcleo da convolução também terá uma profundidade igual a três.

Figura 17– Convolução em volume



Fonte: Bruno Klein, 2013

A sobreposição do núcleo com a primeira porção da imagem de entrada, será gerado o primeiro pixel da imagem de saída. Percebe-se que o processo de convolução além de destacar características da imagem, reduz a quantidade de pixels e consequentemente a quantidade de dados na rede.

Vale ressaltar que esse processo, muitas vezes, é aplicado diversas vezes na mesma imagem, com núcleos distintos, originando diversas imagens de saída que podem ser passadas à diante na rede, aumentando assim o número de parâmetros ajustáveis na rede, visto que cada convolução terá um núcleo e cada núcleo pode assumir infinitos valores.

A convolução pode ser implementada como camada de uma rede neural convolucional. Nessa camada, cada pixel da imagem representa um neurônio e o valor de cada elemento do núcleo utilizado na convolução representa um peso, de forma análoga ao modelo de neurônio apresentado, porém percebe-se os neurônios não estão todos conectados entre si e sim apenas na região tocada pelo núcleo, de modo que os neurônios do canto superior esquerdo da imagem não estarão conectados com os neurônios do canto inferior direito, por exemplo.

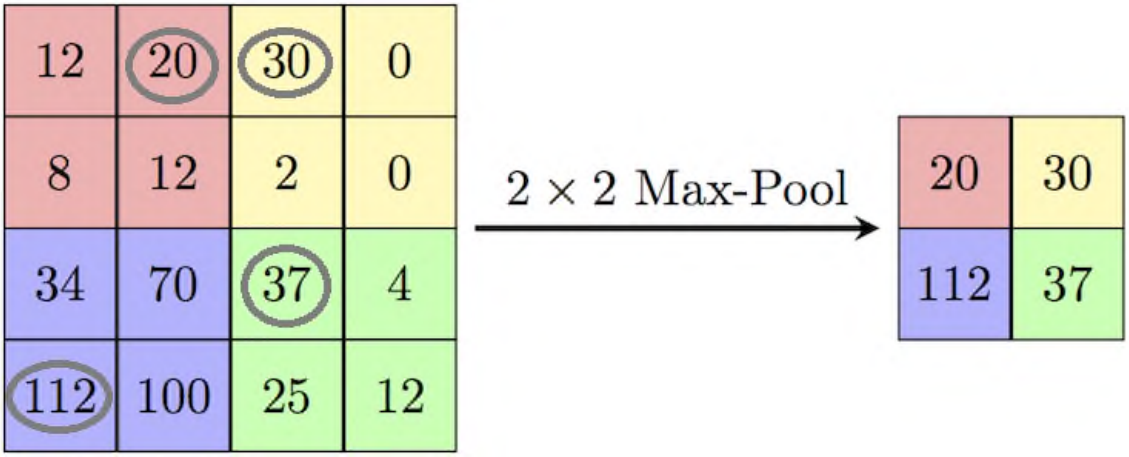
3.2.2 Pooling

Ao processar grandes imagens com muitos pixels, é comum reduzir o número de neurônios do sistema, com o intuito de diminuir o tempo de processamento, o número de parâmetros a serem aprendidos pela rede e o poder de processamento requerido, aplicando métodos de *pooling,* também chamado de agregação.

A forma com a qual o *pooling* funciona é bem parecida com uma convolução, pois ela também varre a imagem de entrada em grupos de neurônios, porém, a principal diferença é que os neurônios de saída não são resultados de um produto interno e sim de uma função de *pooling*, que pode ser uma função máximo, mínimo, média, entre outras. Basicamente, define-

se o tamanho do núcleo, o método de *pooling* e o passo com o qual o núcleo varrerá a imagem, e após isso, o algoritmo de *pooling* reduzirá os neurônios de saída. A imagem abaixo demonstra o processo de *pooling* utilizando o método de máximo num núcleo 2x2 numa imagem de entrada de 16 neurônios de entrada com o passo 2, percebe-se que o núcleo varre a imagem de entrada andando de 2 em 2 neurônios e selecionando o valor máximo da área como valor do neurônio de saída.

Figura 18– Método de p*ooling* utilizando função máximo e núcleo 2x2



Fonte: Jay Ricco, 2017

3.3. Arquitetura utilizada

Existem inúmeras arquiteturas diferentes e bem difundidas nas quais é possível construir modelos de redes neurais profundas, como *Tensorflow*, *Caffe*, *Pytorch*, *Keras* entre outras. A arquitetura utilizada na implementação da rede foi a *Caffe*, uma arquitetura com bastante rapidez, modularidade e disponibilidade de material na literatura. Nos subtópicos abaixo serão descritos brevemente as características da arquitetura.

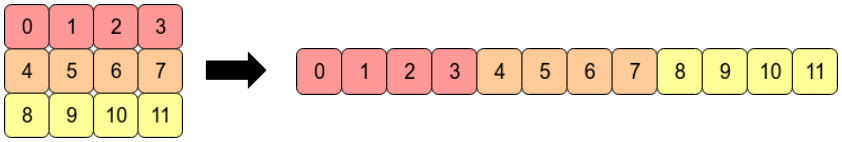
3.3.1 Dados

Uma das principais vantagens do *Caffe* é a possibilidade de utilização tanto da *GPU*, e de seus *cuda cores*, quanto do *CPU*, isso é implementado tanto nas estruturas de dados quanto em praticamente todas as funções e camadas.

A *Caffe* trata dados na forma de *Blobs*, que é um *wrapper* que envolve os dados e os dão características de objeto, além de permitir a sincronização entre *GPU* e *CPU* e suas

respectivas memorias. Matematicamente um *Blob* é apenas uma matriz densa de dados alocada na memória de forma contígua, semelhante à forma com que os dados são armazenados em C.

Figura 19– Representação de armazenamento contíguo na memória



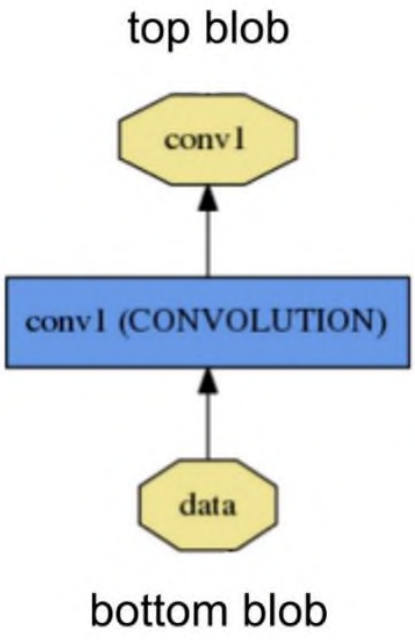
Fonte: Alex Riley, 2014

A *Caffe* utiliza *Blobs* como fonte unificada de dados, tanto para armazenar conjuntos de imagens para treinamento, como parâmetros de modelos, imagens de entrada, etc. De forma geral, as camadas são alimentadas por *Blobs* na sua entrada e emitem *Blobs* como saída.

3.3.2 Camadas

De forma geral, as camadas da *Caffe* recebem um *blob* como entrada e emitem um *blob* de saída, conforme ilustrado na Figura 20.

Figura 20– Representação de uma camada de convolução genérica



Fonte: YANGQING et al, 2014

Na estrutura *Caffe* já existem desenvolvidas diversos tipos de camadas e a forma de às definir, às inicializar e às utilizar. A linguagem padrão para definir qualquer camada na rede

é a *Google Protocol Buffer*, também conhecido como *protobuf*, que permite definir modelos complexos numa linguagem legível para humanos e compatível com múltiplas linguagens, como python e C++.

Figura 21– Representação de uma camada de convolução genérica

```
layer {
  name: "conv0"
  type: "Convolution"
  bottom: "data"
  top: "conv0"
  param {
    lr_mult: 1.0
    decay_mult: 1.0
  }
  param {
    lr_mult: 2.0
    decay_mult: 0.0
  }
  convolution_param {
    num_output: 32
    pad: 1
    kernel_size: 3
    stride: 2
    weight_filler {
      type: "msra"
    }
    bias_filler {
      type: "constant"
      value: 0.0
    }
  }
}
```

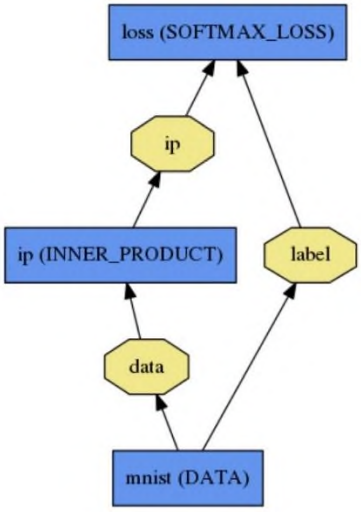
Fonte: O próprio autor.

As linhas de código em *protobuf* acima representam a definição de uma camada de convolução presente na rede utilizada, ela é a primeira camada após a camada de entrada, que foi rotulada como “data”. É possível perceber que os parâmetros da camada são definidos de forma intuitiva e legível e, para essa camada, é possível concluir o seguinte:

- A camada que alimenta, denotada como “*bottom”*, é a própria camada de entrada “*data*”
- A camada de saída é chamada de “conv0”
- São aplicados 32 filtros de convolução, definido como “*num_output*”
- O núcleo de convolução é uma matriz 3x3, definido em *“kernel_size”*
- O passo com o qual o núcleo varre a imagem é igual a dois, denotado por “*stride*”
- O método de inicialização dos pesos da convolução utilizado é o chamado *msra*

Também é comum camadas que possuem 2 entradas diferentes, conforme visto na imagem abaixo, nesse exemplo insere-se como entrada na camada *softmax* tanto os dados processados provenientes das camadas da rede como a classificação do objeto para comparar o resultado previsto com o real.

Figura 22– Exemplo de camada com duas entradas



Fonte: YANGQING et al, 2014

3.3.2.1 Tipos de Camada

Existem diversos tipos de camadas possíveis de se definir e utilizar na estrutura *Caffe*. Abaixo serão brevemente explicadas algumas camadas mais importantes utilizadas no modelo da rede.

3.3.2.1.1 Convolução

A função dessa camada é convoluir a camada de entrada com um conjunto de filtros cujos valores dos pesos são aprendidos durante o processo de treinamento.

Os principais parâmetros configuráveis da camada são os seguintes:

- *Num_output*: Número de filtros a serem aplicados na camada de entrada e aprendidos durante o processo de treinamento;
- *Kernel_size*: Tamanho do núcleo de convolução a ser aprendido, o núcleo é sempre quadrado então ao inserir um valor n a camada entenderá que a matriz deverá ter tamanho n x n;

- *Pad*: Define o número de pixels a serem adicionados, implicitamente, nas extremidades da imagem no momento da convolução
- *Lr_mult* e *Decay_mult*: Define a taxa de aprendizado e taxa de decadência dos parâmetros do filtro e dos *bias*;
- *Stride*: Define o passo, em neurônios, ou pixels, com o qual a camada será varrida pelo núcleo;
- *Weight_filler*: Define o método de inicialização dos parâmetros do núcleo;
- *Bias_filler*: Define o método de inicialização do *bias*.

A camada possui como entrada um *array* de dimensão $n\ x\ c_i\ x\ w_i\ x\ h_i$, onde c_i é o número de canais, w_i e é a largura em pixels e h_i é a altura em pixels. Como saída, a camada apresenta um *array* de dimensão $n\ x\ c_o\ x\ w_o\ x\ h_o$, onde c_o é o número de filtros aplicados e definidos na camada, e $\boldsymbol{w_o = \frac{w_i + 2.pad - kernel_{size}}{stride} + 1}$, e o w_o é calculado de forma análoga.

3.3.2.1.2 Pooling

A função dessa camada é aplicar o método de *pooling* na camada de entrada.
Os principais parâmetros configuráveis da camada são os seguintes:

- *Num_output*: Número de filtros a serem aplicados na camada de entrada;
- *Kernel_size*: Tamanho do filtro utilizado no processo de *pooling*, é possível especificar um filtro não quadrado utilizando os parâmetros *kernel_h* e *kernel_w* para definir a altura e largura, em pixels, do filtro, respectivamente;
- *Pool*: define o método de *pooling* a ser utilizado, entre o máximo, média e estocástico;
- *Pad*: Define o número de pixels a serem adicionados, implicitamente, às extremidades da imagem no momento do *pooling*;
- *Stride*: passo, em pixels ou neurônios, com o qual o filtro varrerá a camada de entrada.

A camada é idêntica à camada de convolução quanto ao cálculo de tamanho de entrada e saída.

3.3.2.1.3 ReLU

A função dessa camada é aplicar a função de ativação ReLU na camada de entrada. A camada só possui um parâmetro alterável e opcional:

- *Negative_slope*: Especifica se a parte negativa da camada deverá ser vazada ou apenas igualada a zero.

As dimensões da camada de saída são idênticas as da camada de entrada, e, além disso, ainda permite a utilização de método de *in-place computation* que permite que o *blob* de entrada e o de saída seja o mesmo para conservar o consumo de memória.

3.3.2.1.4 Flatten

A função dessa camada de utilidade é transformar uma camada de entrada de dimensões $n\ x\ c_i\ x\ w_i\ x\ h_i$ num vetor simples de dimensão $n\ x\ (c_i\ x\ w_i\ x\ h_i)$. Essa camada não possui parâmetros alteráveis. Esse processo é útil para preparar as ultimas camadas da rede para uma possível camada de produto interno ou classificador *softmax*.

3.3.2.1.5 Softmax

A função dessa camada é aplicar a regressão multinomial logística, *softmax*, na camada de entrada. como o *softmax* trata-se apenas de uma normalização dos valores de entrada utilizando a função exponencial, o tamanho da camada de saída é igual ao tamanho da camada de entrada. Essa camada não possuí parâmetros alteráveis.

3.3.3 Processos da Rede

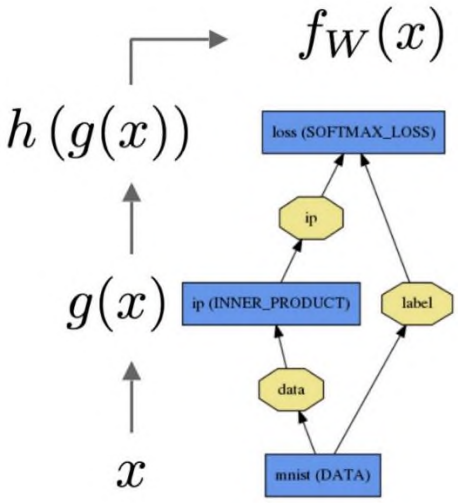
O processo de operação de uma rede neural profunda pode ser separado, basicamente, em duas etapas, o *foward* e o *backward*. O *foward* computa a saída da rede em resposta à uma imagem, passando por todas as camadas até chegar à camada de saída. Já o *backward* calcula o gradiente, dado o erro entre o resultado previsto e o resultado real, da camada de saída e utilizando a Regra da Cadeia, calcula esse gradiente em relação a todas as

outras camadas anteriores e seus parâmetros, como por exemplo seus pesos e *biases* ou valores de núcleo das camadas de convolução.

3.3.3.1 Foward

Conforme dito anteriormente, ao realizar o *foward*, a rede calcula a saída da entrada passando por todas as camadas, de baixo para cima, partindo da camada inicial até a camada final, conforme a Figura 23 exemplifica.

Figura 23– Etapa *foward* vista como composição de funções



Fonte: YANGQING et al, 2014

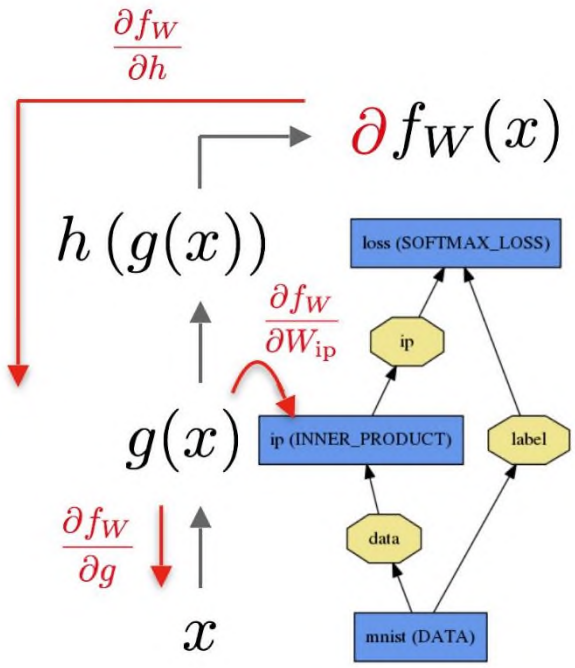
É possível entender essa etapa de *foward* como uma composição de funções, onde a camada de entrada representaria os valores de entrada da função e cada camada posterior representasse mais uma composição desse resultado. No caso da imagem acima, o resultado do *foward* seria equivalente à composição da função h, *softmax*, aplicada à composição da função g, camada completamente conectada de produto interno, aplicada aos dados de entrada. Essa analogia é válida para redes neurais profundas, porém o *foward* seria a composição de muitas funções.

Existe uma distinção entre a etapa de *foward* no processo de treinamento da rede e após o treinamento: em redes neurais já implementadas, o processo de *foward* simplesmente calcula a saída dada a entrada, já durante o processo de treinamento, o *foward*, além disso, calcula o erro, ou função custo, dessa saída em relação ao resultado real.

3.3.3.1 Backward

A etapa *backward* começa de cima para baixo e utiliza a função custo calculada na etapa *foward* para calcular o gradiente da função em respeito à camada de saída e, em seguida utiliza a regra da cadeia para calcular o gradiente em respeito a todas as outras camadas e seus respectivos parâmetros e, consequentemente, o gradiente de todo o modelo, esse processo é comumente chamado de *back-propagation*.

Figura 24– Etapa *backward* em rede simples



Fonte: YANGQING et al, 2014

3.3.4 Solver

O *solver* orquestra o processo de treinamento e é responsável por chamar os processos de *foward* e *backward*, bem como utilizar os valores de gradiente, gerados no processo de *backward*, para atualizar os parâmetros da rede, pesos e *bias*, bem como os núcleos das camadas de convolução. A responsabilidade é, dessa forma, dividida entre o solver que supervisiona o processo de otimização e gera as atualizações dos parâmetros, e os processos da rede, que geram a função de custo e os gradientes.

Basicamente o *solver* é reponsável por:

- Criar a estrutura de treinamento e avaliação da rede, baseado no modelo e hiperparâmetros definidos pelo usuário;
- Iterativamente otimizar a rede chamando os processos de *foward* e *backward* e atualizando os parâmetros;
- Avaliar, periodicamente, a rede de teste criada por ele mesmo;
- Criar *snapshots* do modelo e estado do *solver* durante todo o processo de otimização, é possível utilizar um *snapshot* para retornar o modelo e *solver* para o estado que estava anteriormente.

E, em cada iteração o *solver* realiza as seguintes ações:

- Chama o processo de *foward* para calcular a saída e o erro;
- Chama o processo de *backward* para calcular os gradientes;
- Atualiza os parâmetros da rede utilizando os valores do gradiente, dependendo do tipo de método utilizado para o *solver*;
- Atualiza o estado do *solver*.

3.3.4.1 Métodos do *solver*

Cada vez que uma imagem é alimentada à rede em processo de treinamento, são gerados os valores de saída e de erro. É de primordial importância ressaltar que o processo de atualização de parâmetros não utiliza cada erro gerado para gerar os gradientes e consequentemente alterar os parâmetros, em vez disso, é calculado a média dos erros de todo o conjunto de dados e só então é calculado o gradiente e os parâmetros são atualizados, conforme ilustrado na equação abaixo.

$$L(W) = \frac{1}{|D|}\sum_{i}^{|D|} fw\left(X^{(i)}\right) + \lambda_r(W) \quad (11)$$

Sendo: $fw\left(X^{(i)}\right)$ é o erro da entrada $X^{(i)}$ e $\lambda r(W)$ é um termo de regularização com peso λ.

Geralmente o conjunto de dados utilizado para o treinamento é bastante extenso e calcular o erro de cada elemento levaria bastante tempo e consumiria bastante poder de processamento, para contornar esse problema, o *solver* divide randomicamente o conjunto de

dados em lotes, ou *mini-batch*, dessa forma a fórmula para cálculo do erro seria a seguinte aproximação estocástica:

$$L(W) \approx \frac{1}{|N|}\sum_{i}^{|N|} fw\left(X^{(i)}\right) + \lambda_r(W)$$

(12)

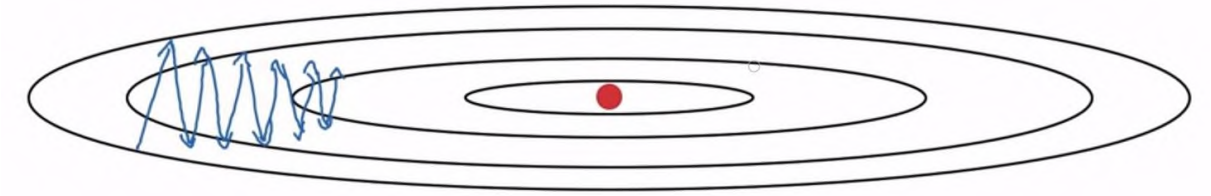
Sendo: N é uma porção do conjunto de dados de treinamento escolhida de tal modo que N << D.

3.3.4.1.1 RMSProp

Existem seis métodos para o *solver* implementados na arquitetura *caffe*, o método utilizado no modelo utilizado é o *RMS Props*. O método consiste na utilização de uma média exponencial ponderada móvel para amenizar as grandes variações no processo de descida de gradiente.

Conforme apresentado na equação (10), o processo de descida de gradiente aplica o negativo do gradiente da função custo em respeito a um parâmetro multiplicado pela taxa de aprendizagem. Um problema dessa operação simples pode ser exemplificado na imagem abaixo.

Figura 25– Exemplo de descida de gradiente oscilatória



Fonte: Andrew et al, 2014

A Figura 25 representa a curva de nível de uma função de custo onde existem dois parâmetros, w_1 no eixo vertical e w_2 no eixo horizontal e onde o círculo vermelho representa o ponto de mínimo local. A aplicação do negativo do gradiente pode levar, nesse caso, à uma descida de gradiente oscilatória que, consequentemente, iria requerer várias iterações para o treinamento.

O algoritmo do *RMSProp* proposto por Tielman e Hilton, possui o seguinte equacionamento matemático, expandindo para um hiperplano com n variáveis de peso e na iteração t.

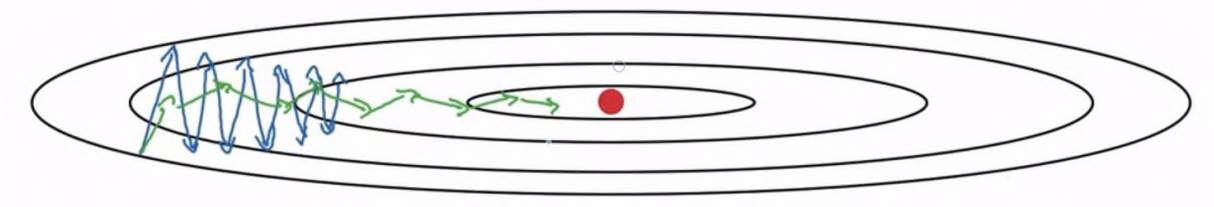
$$MS((W_t)_i) = \delta * MS((W_{t-1})_i) + (1-\delta) * (\nabla L(W_t))_i^2 \quad (13)$$

$$(W_{t+1})_i = (W_t)_i - \alpha * \left(\frac{(\nabla L(W_t))_i}{\sqrt{MS((W_t)_i)}}\right) \quad (14)$$

A equação (13) nada mais é que a média exponencial ponderada móvel do quadrado do gradiente da função custo em respeito aos pesos, no caso ao peso W_i, onde δ representa apenas o peso da média. Percebe-se que a equação (14) é uma variação da equação (10) de descida de gradiente, onde a única diferença é que o valor a ser subtraído do peso é dividido pelo valor da raiz quadrada da média exponencial ponderada móvel na iteração t.

O efeito desse método na descida de gradiente é uma amenização na atualização dos pesos que possuem gradiente com maior módulo, visando diminuir as oscilações. Isso se deve ao fato de que o termo a ser subtraído do peso será dividido pela média exponencial ponderada dos módulos dos gradientes, logo parâmetros que possuem gradientes maiores serão atualizados por fatores divididos por números maiores. A demonstração gráfica desse método pode ser exemplificada na Figura 26.

Figura 26– Comparação entre *rmsprop* e descida de gradiente comum



Fonte: Andrew et al, 2014

CAPÍTULO 04: MODELO DE REDE UTILIZADO

Para a detecção de pessoas, utilizou-se um modelo de rede neural convolucional profundo pré-treinado chamado de *Mobilenet-SSD*, o modelo pode ser visto como a composição de dois modelos, o *Mobilenet* e o *Single Shot Multibox Detector*. Antes de explanar com mais detalhes o modelo utilizado, os dois modelos originários serão brevemente explicados.

A estrutura desse capítulo é dividida nos seguintes subtópicos: A seção 4.1 explica o modelo da rede de classificação de imagens *Mobilenet*; A seção 4.2 explana o modelo da rede de detecção que servirá como camadas finais do modelo utilizado; A seção 4.3 elucida o modelo utilizado para a detecção de humanos que consiste numa fusão dos dois modelos anteriores.

4.1. Mobilenet

Mobilenet é um modelo de rede neural convolucional profunda desenvolvida por pesquisadores da *Google inc*. cujo principal diferencial é a eficiência, baixa latência e pequeno tamanho visando aplicações em dispositivos móveis como celulares e controladores embarcados. (HOWARD *et al*, 2017)

O modelo serve para aplicações de classificação de imagens em uma das 1000 classes pré-definidas na construção do modelo e no treinamento, sendo uma delas definida como *background* para representar imagens que não possuem nenhuma das outras 999 classes. Mais especificamente, a imagem alimentada à rede gerará um vetor de 1000 elementos em que cada um deles é a probabilidade de a classe respectiva estar presente na imagem. Para isso, utiliza-se como camada de saída uma camada *softmax*.

A título de exemplo, supondo que em vez de 1000 existissem apenas 4 classes, sendo elas: cachorros, gatos, humanos e vazio, e fosse alimentada uma imagem contendo cachorros e gatos, o resultado esperado seria valores altos nos elementos do vetor que representam a probabilidade de cachorros e gatos e baixos nos elementos que representam a probabilidade de encontrar humanos e vazio.

Grande parte da rapidez da rede deve-se à forma com a qual ela calcula as convoluções na imagem. O modelo utiliza a chamada *depthwise seperable convolution* e utilizando núcleos de tamanho 3x3 utiliza entre 8 e 9 vezes menos poder de processamento que as convoluções comuns. (HOWARD *et al*, 2017)

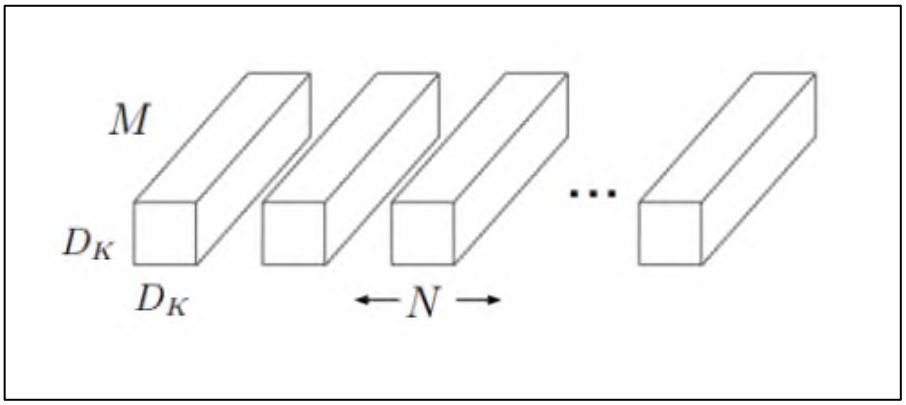
4.1.1 Depthwise Seperable Convolution

Grande parte da rapidez da rede deve-se à forma com a qual ela calcula as convoluções na imagem. O modelo utiliza a chamada *depthwise seperable convolution* e utilizando núcleos de tamanho 3x3 utiliza entre 8 e 9 vezes menos poder de processamento que as convoluções comuns. (HOWARD *et al*, 2017)

O método consiste, basicamente em dividir a convolução em duas etapas: a primeira, chamada de *depthwise convolution* que nada mais é que uma convolução aplicada em um único canal da imagem, em vez de ser feita nos M canais, a primeira etapa é então seguida da *pointwise convolution* que consiste numa convolução 1x1 que cria a combinação linear dos mapas de características gerados pela primeira etapa. (HOWARD, Andrew G. et al (2017))

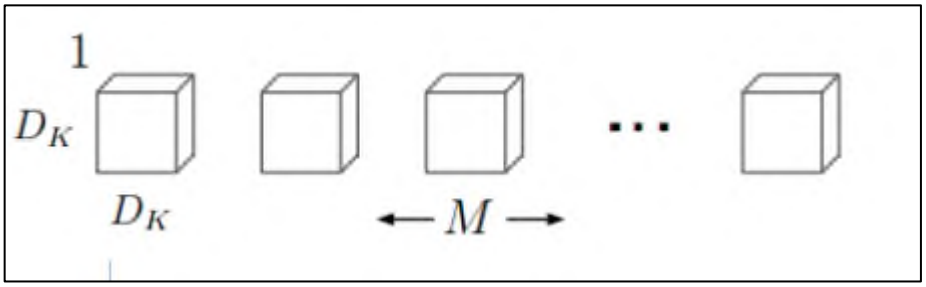
A Figuras 27, Figura 28 e Figura 29 representam a comparação entre o método comum de convolução e o método utilizado, ambos aplicando N filtros de convolução numa imagem de M canais.

Figura 27– Convolução comum



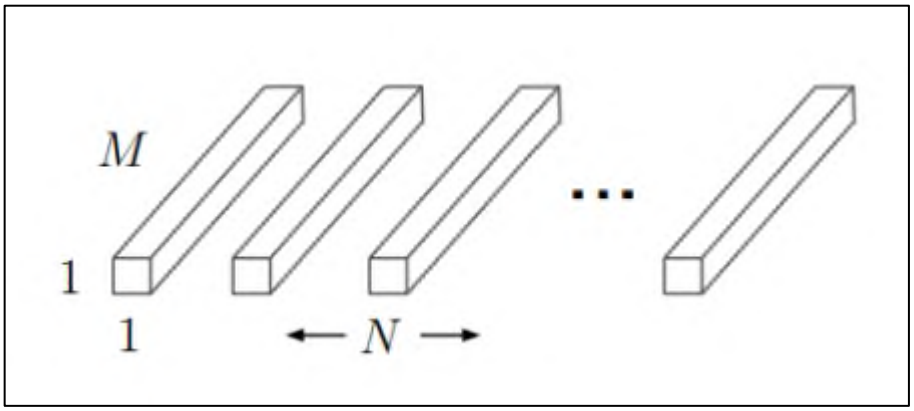
Fonte: HOWARD *et al*, 2017.

Figura 28– *Depthwise Convolution*



Fonte: HOWARD *et al*, 2017.

Figura 29– *Pointwise Convolution*



Fonte: HOWARD *et al*, 2017

As equações a seguir demonstram a diferença entre o número de multiplicações necessárias para calcular o método comum e o *depthwise seperable convolution* (HOWARD, Andrew G. et al (2017)):

$$D_k . D_k . M . N . D_F . D_F \quad (15)$$

A equação (15) representa o custo de uma convolução utilizando o método padrão, D_k é a largura e altura da imagem medida em filtros de convolução, em outras palavras, o número de passos que o filtro percorrerá até varrer toda a imagem naquela direção, D_F é o tamanho do filtro, M é o numero de canais da imagem e N é o número de filtros diferentes aplicados.

$$D_k . D_k . M . D_F . D_F \quad (16)$$

$$1.1. M . N . D_F . D_F \quad (17)$$

As equações (16) e (17) representam o custo da *depthwise convolution* e da *pointwise convolution*, é possível então calcular o custo da *depthwise seperable convolution* conforme a equação (18) e fazer a comparação com o custo da convolução normal (15) conforme mostrado na equação (19):

$$D_k . D_k . M . D_F . D_F + M . N . D_F . D_F \quad (18)$$

$$\frac{(D_k . D_k . M . D_F . D_F + M . N . D_F . D_F)}{D_k . D_k . M . N . D_F . D_F} = \frac{(D_k . D_k + N) . M . D_F . D_F)}{D_k . D_k . M . N . D_F . D_F} = \frac{(D_k . D_k + N)}{D_k . D_k . N} = \frac{1}{N} + \frac{1}{D_k^2} \quad (19)$$

Percebe-se então que a *depthwise separable convolution*, em geral, necessita muito menos multiplicações que a convolução comum, e, como a multiplicação é um processo

bastante dispendioso, isso implica numa velocidade de processamento por vezes maior. É importante ressaltar que esse ganho na velocidade tem um custo na precisão da rede, porém, segundo, Andrew G. Howard et al, o custo é bastante pequeno:

> Mobilenet usa convoluções 3x3 do tipo *depthwise separable* convolution o que usa entre 8 e 9 menos computações que convoluções normais por apenas uma pequena redução da precisão, conforme visto na tabela 4. (HOWARD, Andrew G. et al, 2017)

Figura 30– Comparação entre convolução comum e a *depthwise separable convolution*

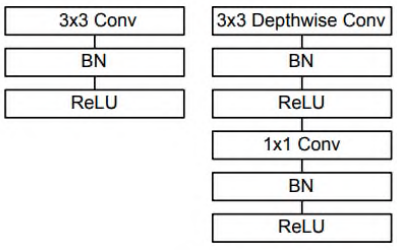
Modelo	ImageNet Acurácia	Milhões de Mult/Soma	Milhões de Parâmetros
Conv MobileNet	71.7%	4866	29.3
MobileNet	70.6%	569	4.2

Fonte: HOWARD et al, 2017.

A Figura 30 representa a tabela 4 mencionada na citação e compara a performance entre o modelo *mobilenet* utilizando a convolução comum, representada na tabela como *Conv MobileNet*, e o modelo utilizando a *depthwise separable convolution*. É possível perceber a redução drástica no número de parâmetros e no número de operações de multiplicação e adição com um preço pequeno na acurácia da rede.

Outro ponto importante é que após cada uma das duas etapas da convolução são aplicadas camadas de normalização e ativação, ou seja, a cada convolução são aplicadas duas camadas de não linearidades e de normalização, conforme ilustrado na Figura 31.

Figura 31– Diferença entre convolução normal e *depthwise separable convolution*



Fonte: HOWARD et al, 2017.

4.1.2 Camadas do modelo

Figura 32– Camadas presentes no modelo *Mobilenet*

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Fonte: HOWARD et al, 2017..

Na Figura 32 encontram-se todas as camadas definidas na rede neural, do fundo ao topo. Percebe-se que a grande maioria delas são camadas de convolução, separadas em *depthwise* e *pointwise,* e que possuem um número crescente de filtros, a primeira camada aplica 32 filtros e a última aplica 1024. Lembrando que as funções de ativação ReLU não estão especificadas, mas são inseridas após cada camada de convolução.

Percebe-se também, que não foram utilizadas muitas camadas de *pooling*, isso se deve ao fato de que as convoluções foram realizadas utilizando passo dois, e isso naturalmente diminui o tamanho da imagem no processo.

4.1.3 Resumo do funcionamento do modelo

Em resumo, a rede recebe uma imagem RGB de tamanho 224x224 e a passa por diversas camadas de convolução que progressivamente diminuem a dimensão da largura e altura da imagem e aumentam sua profundidade devido ao crescente número de filtros, inserindo não-linearidades ao longo do caminho. Após 14 camadas de convolução completa os

dados que tinham tamanho 224x224x3 passam a ter tamanho 7x7x1024. Após isso os dados passam por uma camada de *pooling* por média utilizando um núcleo 7x7 que reduzem as dimensões para 1x1x1024. Em seguida os dados alimentam uma camada completamente conectada, onde todos os neurônios de saída estão completamente conectados com todos os neurônios de entrada, que consiste, basicamente, de uma multiplicação do vetor de dados, de dimensão 1x1024 por uma matriz de pesos de dimensões 1024x1000, o resultado dessa operação será um vetor 1x1000. Por último, essa camada alimenta uma camada *softmax* que é a camada de saída da rede.

Cada camada de convolução possui entre centenas e milhões de parâmetros aprendíveis que são otimizados durante o processo de descida de gradiente, conforme mencionado anteriormente, utilizando o erro entre a classificação estimada e a real e propagando a média dos erros de um grupo de imagens para as camadas anteriores utilizando a Regra da Cadeia.

4.2. Single Shot Multibox Detector (SSD)

O modelo *SSD* é utilizado para detecção em vez de classificação, isso significa que as imagens inseridas terão como saída não apenas a probabilidade de a classe ser encontrada e sim sua localização, por esse motivo as imagens utilizadas para treinamento da rede precisam conter duas informações: a classificação dos objetos que estão contidos na imagem e suas *ground truth boxes*, caixas que delimitam o objeto classificado na imagem.

Segundo Wei Liu e colegas, 2016:

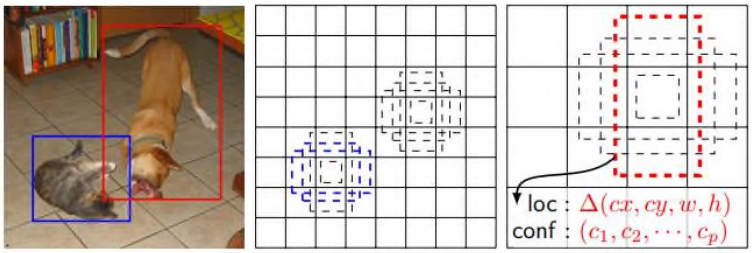
> "Nós introduzimos o SSD, um detector de múltiplas categorias em única imagem mais rápido e significativamente mais preciso que o detector que representava o estado da arte de detectores de única imagem (YOLO) e tão preciso quanto as técnicas mais lentas que aplicavam propostas de regiões explicitas e pooling"

O SSD é composto por um modelo de classificação de imagens, treinado previamente e truncado antes das camadas de classificação, alimentando 6 camadas de convolução cujos mapas de características tem tamanho decrescente e, consequentemente, resolução decrescente, criando assim um conjunto de mapas mais bem adaptados para detectar objetos de diferentes tamanhos.

Além disso, foram definidas entre quatro e seis caixas de detecção padrão para cada mapa de características, com tamanhos e proporções diferentes para ampliar a variedade de formas a serem detectadas. As caixas padrões são definidas em posições fixas no centro das

células da imagem e o algoritmo expõe como saída a variação na posição e no tamanho da caixa responsável pela sua célula.

Figura 33– Exemplo das caixas de detecção padrão em diferentes resoluções



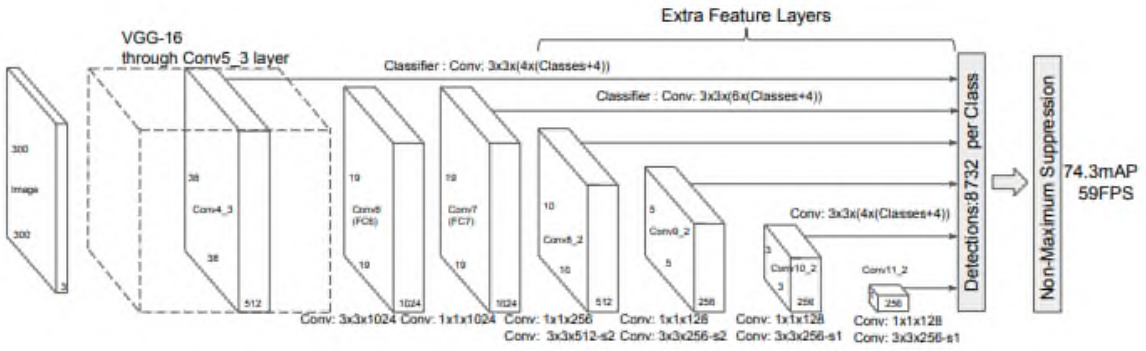
Fonte: LIU *et al*, 2016.

A Figura 33 é composta de 3 imagens: a da esquerda representa uma imagem juntamente com suas *ground truth boxes*; a imagem do meio representam graficamente as possíveis caixas de detecção padrão nas células que contém algum objeto num mapa de características com resolução maior; a imagem da direita representa as caixas de detecção padrão num mapa de características que possui resolução menor.

Percebe-se que os mapas de características, ou camadas de convolução, que possuem maior resolução são mais bem adaptados para detectar objetos menores na imagem, enquanto que as camadas com menor resolução são melhores para detectar objetos maiores, e que uma mesma imagem pode conter objetos detectados por duas camadas diferentes.

4.2.2 Camadas do modelo

Figura 34– Camadas do modelo SSD



Fonte: LIU *et al*, 2016.

Uma rede neural convolucional treinada para a classificação de imagens tem as camadas de classificação truncadas e então alimenta o modelo acima, a Figura 34 retrata as camadas adicionais propostas pelo modelo SSD, percebe-se que a camada envolvida pela caixa de linha tracejada, Conv4_3, representa a última camada do modelo de classificação. É possível notar a existência de 6 camadas de convolução de dimensões decrescentes, conforme mencionado anteriormente. A última camada da rede é uma camada de supressão de não-máximo

4.2.3 Resumo do funcionamento do modelo

Cada célula do mapa de característica possui caixas de detecção padrão, e a rede calcula, para cada caixa de detecção padrão de cada célula, a probabilidade de estar contida naquela caixa cada uma das classes definidas no modelo e a variação de posição, largura e altura que a caixa padrão terá de sofrer para que a classe detectada esteja contida corretamente e completamente em seu interior. A variação necessária é representada por $\Delta x, \Delta y, \Delta w \; e \; \Delta h$ (LIU *et al*, 2016)

O cálculo da presença das classes e da variação da caixa de detecção padrão é feita por meio da convolução do mapa com filtros de tamanho 3x3xMxN, onde M é a profundidade da camada, N é o número de filtros aplicados e os valores de *stride* e *pad* da convolução são iguais a um com o intuito de manter o resultado da convolução com mesma largura e altura da camada de entrada. Ademais, o número de filtros aplicados para serem detectadas K classes num mapa de característica que possui C caixas de detecção padrão é dado por: **(K+4).C**, pois são necessários K+4 filtros, uma para a presença de cada classe e mais quatro para a variação da caixa padrão, para cada uma das C caixas presentes no mapa.

Portanto, a dimensão da camada após a convolução será igual a $\boldsymbol{WxHx((K+4).C)}$, ou seja, cada uma das WxH células possui um vetor de dimensão $\boldsymbol{(K+4).C}$ atrelado, cujos elementos representam as probabilidades e variações das caixas padrões explanados acima.

As detecções de todas as células de todas as camadas de convolução são então alimentadas a uma camada de supressão de não-máximo que remove todas as detecções sobrepostas a fim de manter apenas uma caixa delimitadora por objeto contido na imagem.

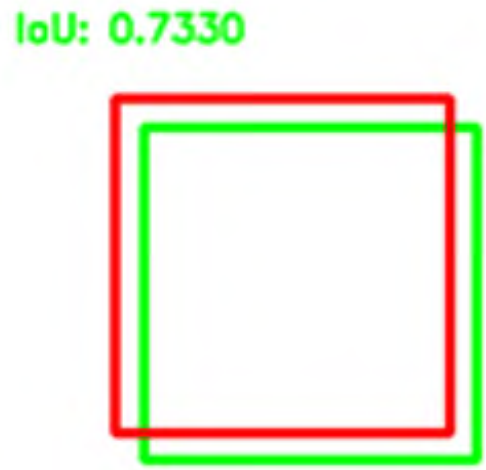
Um passo importante do treinamento desse modelo é a atribuição das informações das *ground truth boxes* a saídas específicas no conjunto de caixas de detecção padrão definidas,

só então é possível iniciar o treinamento da rede. O erro da rede é a média ponderada entre o erro *softmax* e o erro da localização das caixas contendo as classes.

4.2.3.1 Supressão de não-máximo

O método de supressão de não-máximo utiliza como critério para remoção de uma detecção o *Intersection over Union* (IoU), que consiste, basicamente, na porcentagem de intersecção das áreas em relação a união de duas figuras geométricas. O cálculo do IoU pode ser feito utilizando a equação $\boldsymbol{IoU} = \frac{\mathbf{A_i}}{\boldsymbol{A_u}}$, onde $\boldsymbol{A_i}$ é a área de intersecção entre as duas caixas de detecção e $\boldsymbol{A_u}$ é a área de união.

Figura 35– Exemplo de IoU elevado



Fonte: Adrian Rosebrock, 2016

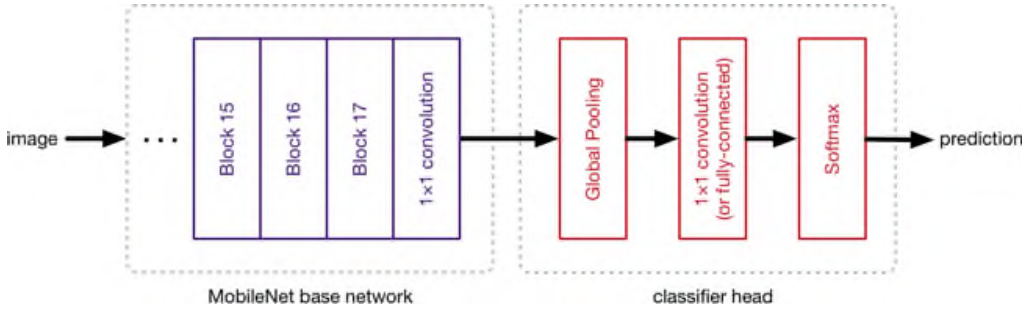
O algoritmo consiste, basicamente, em três passos: primeiramente suprime-se todas as detecções com probabilidade menor que um valor definido no modelo e, enquanto houverem detecções a serem avaliadas como previsões ou não:

- Seleciona-se a detecção que possui maior probabilidade e a define como uma das previsões a serem emitidas na saída;
- Todas as detecções que possuam IoU maior ou igual a 0.5 com a detecção escolhida são suprimidas.

4.3. Mobilenet-SSD

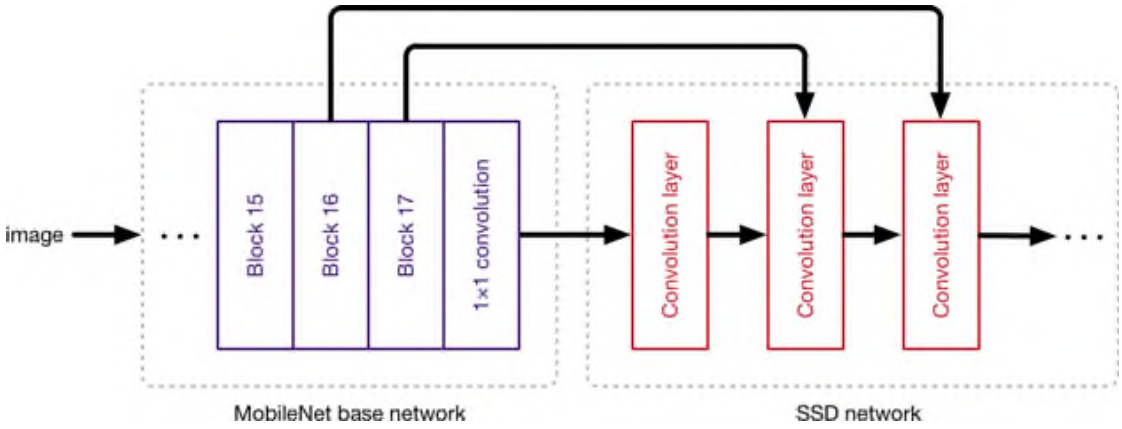
O modelo utilizado para a implementação do segundo algoritmo de detecção de pessoas foi um modelo pré-treinado desenvolvido na arquitetura *Caffe* e é chamado de *Mobilenet-SSD*. Ele consiste, basicamente, num modelo SSD cuja rede que alimenta as camadas anteriores às de localização são retiradas de uma rede *Mobilenet* treinada previamente e truncada antes de suas camadas de classificação.

Figura 36– Exemplo das camadas finais de uma rede *mobilenet* genérica



Fonte: Matthijs Hollemans, 2018

Figura 37– Exemplo das camadas finais de uma rede *mobilenet* alimentando uma rede SSD



Fonte: Matthijs Hollemans, 2018

A figura 36 e Figura 37 retratam a diferença básica nas camadas finais da estrutura de uma rede *Mobilenet* utilizada para classificação de imagens, na Figura 36, e para alimentar uma rede SSD, na Figura 37. Percebe-se que a rede Mobilenet serve de fundação para o modelo, provendo o processo de extração de características.

4.3.1 Camadas do modelo

A tabela abaixo mostra todas as camadas de convolução presentes na rede, desde a camada da imagem, chamada de "*Data"*. As camadas expostas são referentes apenas ao processamento da imagem, logo não foram incluídas as convoluções responsáveis por calcular a probabilidade de presença das classes e a deformação da caixa de detecção padrão.

Tabela 01 – Camadas de Convolução do Mobilenet-SSD.

#	Nome	Tamanho do Filtro	Tamanho de Saída	Número de Caixas de Detecção Padrão
1	Data	-	300x300x3	-
2	Conv0	3x3x3x32	150x150x32	-
3	Conv1/dw	3x3x32x32	150x150x32	-
4	Conv1	1x1x32x64	150x150x64	-
5	Conv2/dw	3x3x64x64	75x75x64	-
6	Conv2	1x1x64x128	75x75x128	-
7	Conv3/dw	3x3x128x128	75x75x128	-
8	Conv3	1x1x128x128	75x75x128	-
9	Conv4/dw	3x3x128x128	38x38x128	-
10	Conv4	1x1x128x256	38x38x256	-
11	Conv5/dw	3x3x256x256	38x38x256	-
12	Conv5	1x1x256x256	38x38x256	-
13	Conv6/dw	3x3x256x256	19x19x256	-
14	Conv6	1x1x256x512	19x19x512	-
15	Conv7/dw	3x3x512x512	19x19x512	-
16	Conv7	1x1x512x512	19x19x512	-
17	Conv8/dw	3x3x512x512	19x19x512	-
18	Conv8	1x1x512x512	19x19x512	-
19	Conv9/dw	3x3x512x512	19x19x512	-
20	Conv9	1x1x512x512	19x19x512	-
21	Conv10/dw	3x3x512x512	19x19x512	-
22	Conv10	1x1x512x512	19x19x512	-
23	Conv11/dw	3x3x512x512	19x19x512	-
24	**Conv11**	**1x1x512x512**	**19x19x512**	**3**
25	Conv12/dw	3x3x512x512	10x10x512	-
26	Conv12	1x1x512x1024	10x10x1024	-
27	Conv13/dw	3x3x1024x1024	10x10x1024	-
28	**Conv13**	**1x1x1024x1024**	**10x10x1024**	**6**
29	Conv14_1	1x1x1024x256	10x10x256	-
30	**Conv14_2**	**3x3x256x512**	**5x5x512**	**6**
31	Conv15_1	1x1x512x128	5x5x128	-
32	**Conv15_2**	**3x3x128x256**	**3x3x256**	**6**
33	Conv16_1	1x1x256x128	3x3x128	-
34	**Conv16_2**	**3x3x128x256**	**2x2x256**	**6**
35	Conv17_1	1x1x256x64	2x2x64	-
36	**Conv17_2**	**3x3x64x128**	**1x1x128**	**6**

Fonte: elaborada pelo autor.

As camadas provenientes do modelo *mobilenet* treinado são vinte oito primeiras camadas, contando com a camada de data, as demais camadas são oriundas do modelo SSD. As linhas destacadas em negrito representam os mapas de características que possuem caixas de detecção padrão, consequentemente, são responsáveis por alimentar as camadas de convolução que extraem as detecções geradas naquela camada.

Ademais percebe-se que a estrutura do modelo é bastante similar aos modelos expostos na figura 32 e na figura 34, com pequenas alterações como a diminuição do tamanho

dos mapas de características e o aumento do número de caixas de detecção padrão definidas para as camadas.

4.3.2 Treinamento do modelo

O *Mobilenet-SSD* supracitado trata-se de um modelo pré-treinado, disponível para download, código aberto, e publicado utilizando a licença MIT, ou X11. Uma licença que permite que pessoas que obtenham uma cópia do software a utilizem sem restrição, incluindo, sem limitações de direitos, utilizar, copiar, modificar, publicar, distribuir, sublicenciar e/ou vender copias do software. O único requerimento para a utilização do modelo é a inclusão da licença no código, e, basicamente a única restrição presente na licença é a ausência de garantias oriundas dos desenvolvedores do software (The MIT License, 2018).

A rede foi treinada pelos seus desenvolvedores em dois bancos de dados distintos, primeiramente utilizou-se o *Common Objects in Context* (COCO), um banco de imagens disponível na internet que contém mais de 200 mil imagens devidamente etiquetadas, em seguida utilizou-se o *Visual Object Classes Challenge 2012* (VOC2012), um banco de dados contendo 11530 imagens contendo nelas 27450 objetos e suas respectivas *ground truth boxes*.

A rede foi treinada para detectar as seguintes 20 classes, sendo elas subdivididas em quatro categorias (VISUAL Object Classes Challenge 2012 (VOC2012)):

- Humanos: humano;
- Animais: pássaro, gato, vaca, cachorro, cavalo e ovelha;
- Veículos: avião, bicicleta, barco, ônibus, carro, motocicleta e trem;
- Móveis: garrafa, cadeira, mesa de jantar, vaso de planta, sofá e monitor.

CAPÍTULO 05: IMPLEMENTAÇÃO DOS MÉTODOS

Foram implementados dois métodos de detecção de humanos diferentes, cada um deles utilizando uma técnica distinta. O primeiro deles, explicado na seção 2, foi implementado no *Matlab*. Enquanto que o segundo método foi implementado em python.

As implementações de ambos os métodos foram realizadas num computador que possui um processador *Intel i5 4690k*, 16 *gigabytes* de memória *ram* e uma placa de vídeo *Nvidia GTX 1070*. Todas as imagens foram obtidas da webcam *Logitech HD C270*

Esse capitulo é dividido em três subtópicos, o primeiro deles apresenta o código utilizado para a implementação do primeiro método de detecção, o segundo deles explica o código utilizado para a utilização do segundo método e o último mostra uma breve comparação entre os dois métodos.

5.1. Primeiro método: *PeopleDetector*

A função *PeopleDetector*, explicada na seção 2, é simples de implementar pois já é uma ferramenta do próprio *matlab*, seus parâmetros são facilmente alterados, desde que seja realizado o passo de *release* do objeto.

Primeiramente criou-se um código para efetuar a detecção de humanos em imagens inseridas no programa, para tentar encontrar valores ótimos dos parâmetros da ferramenta. Um dos problemas encontrado foi o grande acréscimo no tempo computacional para pequenas variações nos parâmetros, por exemplo: utilizando 1,001 como valor para o *ScaleFactor*, modificar o valor do *WindowStride* de [8 8], que é o valor padrão, para [7 7] provocou um aumento no tempo necessário para computar o resultado da imagem de 2s para 8,7s, um tempo 4,35 vezes maior. Vale ressaltar que as simulações foram feitas num computador com poder de processamento relativamente alto.

Para a obtenção das imagens da webcam, utilizou-se a função *Webcam*, implementada no *Matlab* e que permite a interface com uma série de webcams amplamente utilizadas no mercado. A função *Webcam* permite também alterar características da imagem como resolução, contraste, brilho, entre outros.

O algoritmo final consiste, basicamente, em duas etapas: a etapa de inicialização e o loop principal. Na etapa de inicialização, que ocorre no início do código, são definidos todos os parâmetros, tanto da webcam, quanto do *PeopleDetector*. No loop principal os seguintes processos ocorrem, enquanto o loop não for quebrado:

- Obtenção de uma foto da webcam;
- Redução da imagem para uma imagem 300x300;
- Inserção da imagem como argumento de entrada da *PeopleDetector*;
- Obtenção do argumento de saída;
- Desenho da caixa que contém o humano detectado, quando detectado.

O código utilizado encontra-se abaixo com todas as linhas comentadas em negrito.

```
%%
clear all;
close all;
%% Definição dos parâmetros da Câmera
cam=webcam;    % Definição a câmera “cam”, utilizando a biblioteca webcam
cam.Contrast=30 % Definição da contraste da câmera
cam.Resolution='320x240' %Definição da resolução da imagem extraída da câmera

%% Definição dos parâmetros do PeopleDetector
detector=vision.PeopleDetector;    % Definição do detector chamado “detector”
detector.ClassificationModel='UprightPeople_96x48'; % Definição do modelo
detector.release; % Esse comando é necessário para modificar os parâmetros
% da    ferramenta depois de definida no programa
detector.ScaleFactor=1.001; % Definição do ScaleFactor
detector.ClassificationThreshold=1.4; %Definição do classification threshold
detector.WindowStride=[8 8]; %Definição do window stride
detector.MinSize=[96 48] %Definição do minsize
detector.MaxSize=[300 300] %Definição do maxsize
go=1; % criação de variável temporária para formação de loop
%% Loop Principal
while go==1 %Loop principal
    % Obtenção da imagem.

    I = snapshot(cam); % Extrai o frame atual da câmera e o aloca na variável I

    I=imresize(I,[300 300]); % Redimensiona a imagem para diminuir o tempo de
% processamento
    % Detecção.

    [bboxes, scores] = step(detector, I);    % Esse comando obtém as detecções da
% Imagem e as coordenadas das caixas que o delimitam
    I = insertObjectAnnotation(I, 'rectangle', bboxes, scores);    % Insere os retângulos
na
% imagem original
    image(I) % Mostra a imagem original com os retângulos inseridos
    drawnow;

end % Fim do loop principal
```

Os parâmetros que afetam a robustez do modelo também afetam drasticamente o tempo de processamento, foram testadas dezenas de valores diferentes para os parâmetros, mas nenhum conjunto de configurações se mostrou rápido o suficiente para a ponto de ser viável a implementação em um controlador embarcado, cujo poder de processamento é bastante menor.

Os parâmetros implicaram no resultado obtido, e que levam cerca de 1,5s para processar cada imagem, conforme vistos no código comentado acima, foram os seguintes:

- *ClassificationModel = 'UprightPeople_96x48'*;
- *ScaleFactor* = 1.001;
- *ClassificationThreshold* = 1.4;
- *WindowStride* = [8 8];
- *MinSize* = [96 48]
- *MaxSize* = [300 300]

5.2. Segundo Método: *Mobilenet-SSD*

A implementação do segundo método foi feita utilizando a linguagem de programação *python*, porém é importante mencionar duas bibliotecas importantíssimas para o desenvolvimento do algoritmo, a primeira é o OpenCV, uma biblioteca código aberto de visão computacional com ferramentas poderoras, e a segunda é a *Imutils,* uma biblioteca código aberto com diversas ferramentas uteis para processamento de imagens e com licença MIT.

5.2.1 Bibliotecas auxiliares

5.2.1.1 *Opencv*

A versão do *Opencv* utilizada foi a versão 3.3, e ela possui uma biblioteca específica para implementação de redes neurais compatível com a arquitetura *Caffe,* entre muitas outras. Abaixo serão descritas brevemente as funções da biblioteca utilizadas no algoritmo, lembrando que todas as definições serão dadas considerando que está sendo utilizado o modelo *Mobilenet-SSD*.

- *dnn.blobFromImage*: Converte uma imagem em um *blob*. A função também realiza, opcionalmente, o recorte da imagem e a subtração de valores médios e/ou a mudança de escala da imagem;
- *dnn.readNetFromCaffe*: Carrega o modelo treinado da rede;
- *net.setInput*: Define o *blob* a ser inserido na rede, o prefixo da função, net, na verdade é o nome dado à rede no momento em que o comando *dnn.readNetFromCaffe* é utilizado;
- *net.forward*: Calcula o passo *foward* e traz como resultado um *array* contendo n vetores, um para cada detecção, com 6 elementos cada, sendo um a identificação da classe detectada, um a confiabilidade e quatro representando a caixa de detecção que contém a classe.

O OpenCV pode ser obtido no link a seguir: https://opencv.org/releases.html, e qualquer versão superior à versão 3.3 poderá ser utilizada.

5.2.1.2 *Imutils*

As funções utilizadas no algoritmo estão descritas abaixo.

- VideoStream: Inicia uma transmissão de vídeo, o grande diferencial dessa função e o principal motivo para a utilização dessa biblioteca é a eficiência da transmissão de vídeo, que utiliza *multithreading* para aumentar a velocidade de transmissão. Outro ponto bastante importante é a compatibilidade com o microcontrolador *raspberry pi* e sua câmera. Na função é possível escolher de onde serão transmitidos os frames, da webcam ou da câmera do raspberry;
- Stream.Read: lê o frame atual da transmissão de vídeo, na verdade o prefixo da função, “Stream”, é o nome definido pelo usuário à transmissão de vídeo;
- Resize: Redimensiona a imagem.

O *Imutils* pode ser obtido no link a seguir: https://github.com/jrosebr1/imutils

5.2.2 Algoritmo principal

Assim como o método anterior, o algoritmo pode ser dividido em duas etapas: a primeira etapa, sendo a inicialização e a segunda etapa o loop principal. Existe, porém, uma peculiaridade no algoritmo, conforme foi tratado nas seções anteriores, o modelo *mobilenet-ssd* foi treinado para detectar 20 classes diferentes, sendo humanos uma delas.

Não existe como adaptar o modelo para detectar apenas humanos sem treiná-lo novamente, o que demanda muito tempo e poder computacional, em vez disso, serão definidas duas listas no python, uma contendo todas as classes para que possa ser feita a correlação entre o número da classe presente no vetor detecção emitido na saída da rede, que é um número de 0 a 20, com sua respectiva classe, e outra contendo todas as classes que não são a classe humano, essa segunda lista é criada com o intuito de ignorar as detecções cuja classe pertence à lista. Esse passo é incorporado na primeira parte do algoritmo.

Na segunda parte, o loop principal, a rede:

- Obtém um frame da transmissão de vídeo;
- Transforma o frame em um blob;
- Alimenta o blob à rede;
- Recebe a saída da rede, que contém todas as classes detectadas na imagem e as respectivas caixas que as englobam;
- Desenha uma caixa ao redor das classes detectadas que não estão presentes na lista a ser ignorada e que possuem confiança maior que 60%, esse valor é escolhido no próprio algoritmo e pode ser alterado, diferentemente do valor definido na camada de supressão de não-máximo, que é de 20%;
- Mostra o frame novo com as classes desenhadas.

Esse loop é repetido até que o botão “q” do teclado seja pressionado. O algoritmo de detecção processa imagens com uma velocidade de aproximadamente 16 frames por segundo. Além disso o método se mostra bastante robusto a diferentes condições de iluminação e oclusão, conseguindo detectar humanos com bastante precisão.

O código utilizado é encontrado abaixo com comentários em negrito.

```
from imutils.video import VideoStream #Importação das bibliotecas necessárias
from imutils.video import FPS
import numpy as np
import imutils
import time
import cv2

# O comando abaixo cria um array de classes definidos na mesma ordem que
# foram definidas no modelo para ser feita a correlação

CLASSES = ["nada", "avião", "bicicleta", "pássaro", "barco",
           "garrafa", "onibus", "carro", "gato", "cadeira", "vaca", "mesa",
           "cachorro", "cavalo", "motocicleta", "humano", "vaso", "ovelha",
           "sofa", "trem", "monitor"]

# O comando abaixo cria um array de classes a serem ignoradas, basicamente todas
as classes com exceção do humano

IGNORE = ["nada", "aviao", "bicicleta", "pássaro", "barco", "garrafa", "onibus", "carro",
"gato", "cadeira", "vaca","mesa", "cachorro", "cavalo", "motocicleta", "vaso", "ovelha",
"sofa", "trem", "monitor"]

# O comando abaixo cria 21 cores aleatórias a serem utilizadas para pintar o
# quadrado de detecção

CORES = np.random.uniform(0, 255, size=(len(CLASSES), 3))

# Carrega o modelo de rede da mobilenetSSD, que estão contidos em arquivos
# Baixados no link https://github.com/chuanqi305/MobileNet-SSD

net = cv2.dnn.readNetFromCaffe("MobileNetSSD_deploy.prototxt.txt",
"MobileNetSSD_deploy.caffemodel")

vs = VideoStream(src=0).start() # Inicia o stream de vídeo

time.sleep(2.0) # Insere uma pausa de 2 segundos para evitar problemas na
  # inicialização da câmera
```

```
# Loop principal
while True:
# Carrega frame da camera e define largura maxima para 400
    frame = vs.read()
    frame = imutils.resize(frame, width=400)
# Conversão da imagem em blob
    (h, w) = frame.shape[:2]
# O comando abaixo transforma uma versão da imagem com tamanho 300x300
# em um blob utilizando uma subtração média de cores de 127.5 seguida de um
# fator de escala de 0,007843, esses valores foram os mesmos utilizados nas
# imagens no processo de treinamento do modelo.
    blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)),0.007843, (300, 300),
127.5)
    net.setInput(blob) # Insere o blob na rede
    detections = net.forward() # Calcula o forward da rede e grava as detecções no
# Array chamado “detections”
    # Loop nas detecções
    for i in np.arange(0, detections.shape[2]): # Loop em todas as detecções
encontradas
        confidence = detections[0, 0, i, 2] # Extração da confiança

        if confidence > 0.6: # Elimina as detecções com confiança menor de 60%
            idx = int(detections[0, 0, i, 1]) # Extrai o número da classe detectada

            if CLASSES[idx] in IGNORE: # Ignora as classes que não são humanos
                continue

            box = detections[0, 0, i, 3:7] * np.array([w, h, w, h]) # Extrai os índices
da
# etiqueta

            (startX, startY, endX, endY) = box.astype("int") # Calcula as
coordenadas
# da caixa contendo o humano
# Os comandos abaixo criam a etiqueta com o nome da classe e a confiança
# A etiqueta é inserida na parte superior do objeto porém se não houver espaço
# a etiqueta será inserida abaixo, isso é feito na linha y=start – 15 if start Y-15> 15
# else startY + 15
            label = f'{CLASSES[idx]}: {confidence * 100:.2f}%'
            cv2.rectangle(frame, (startX, startY), (endX, endY),
                          CORES[idx], 2)
            y = startY - 15 if startY - 15 > 15 else startY + 15
            cv2.putText(frame, label, (startX, y), cv2.FONT_HERSHEY_SIMPLEX,
0.5, CORES[idx], 2) # Insere a etiqueta na imagem
    cv2.imshow("Frame", frame) # Mostra o frame de saída com etiqueta
# Os commandos abaixo quebram o loop caso a tecla “q” seja pressionada
    key = cv2.waitKey(1) & 0xFF
    if key == ord("q"):
        break # Saída do loop principal
```

5.2.3 Comparação visual

Abaixo imagens que comparam visualmente os dois métodos, as fotos estão organizadas lado a lado com a mesma foto sendo alimentada para os dois métodos, na da esquerda utilizando o *people detector* e na da direita utilizando a *mobilenet-SSD*

A figura 38 compara o resultado utilizando imagens de humanos em pé e com todo o corpo aparecendo na imagem.

Figura 38– Comparação entre os dois algoritmos com humanos na vertical

Fonte: o próprio autor

Percebe-se que em imagens onde o humano está em pé e completamente visível, o primeiro método consegue fazer detecções, embora ainda possua muitos falso-positivos. O segundo método conseguiu detectar com perfeição quase todas os humanos na imagem.

A figura 39 apresenta os resultados utilizando imagens onde o humano ou não aparece por inteiro na imagem ou está em uma posição não completamente vertical.

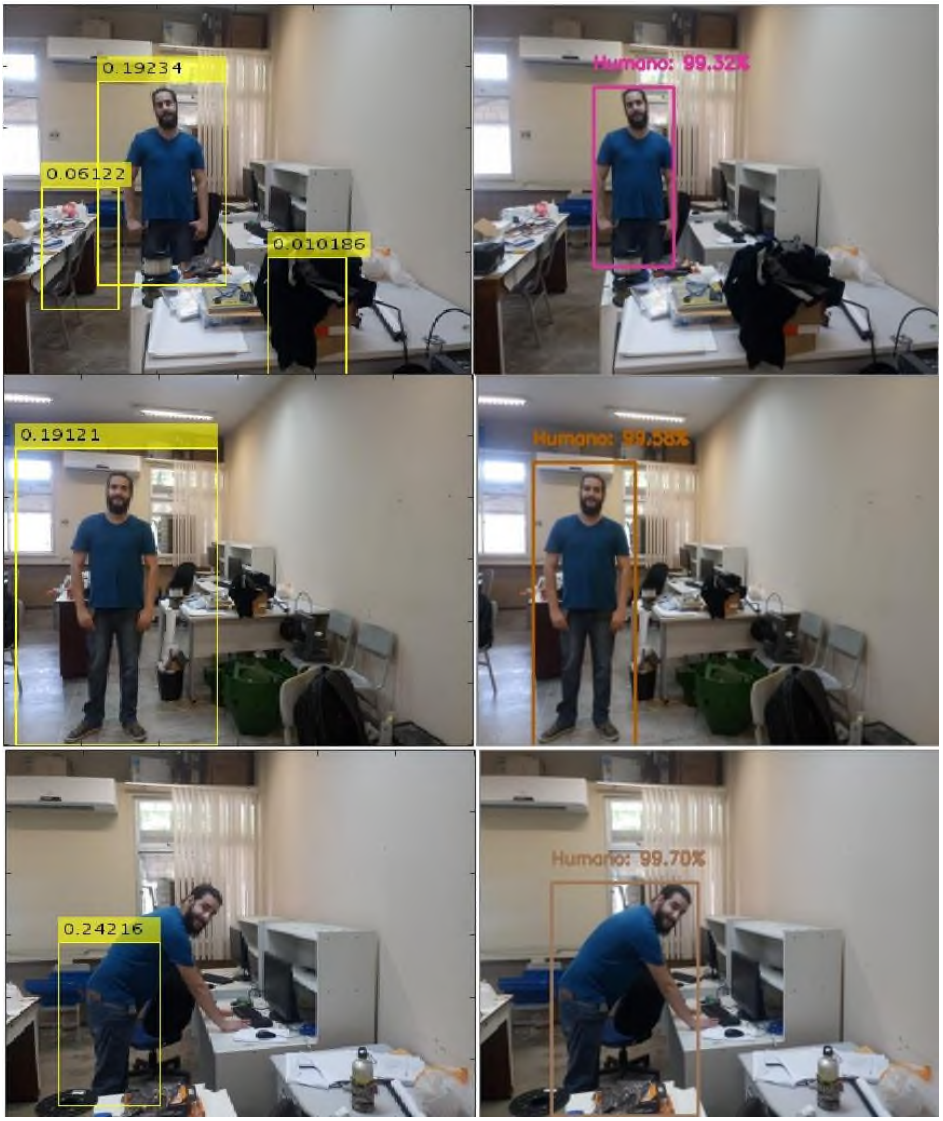
Figura 39– Comparação entre os dois algoritmos com humanos em outras posições

Fonte: o próprio autor

Ao utilizar imagens com humanos em posições diversas, percebe-se que a precisão do primeiro método cai bastante, enquanto o segundo método continua bastante preciso.

Ademais, testou-se o resultado com imagens obtidas no laboratório do Grupo de Pesquisa em Automação, controle e Robótica. O resultado obtido foi parecido com as imagens do banco de dados, porém o primeiro método apresentou resultados mais consistentes.

Figura 40 – Comparação entre os dois algoritmos com imagens no GPAR



Fonte: o próprio autor

De modo geral, observando a comparação gráfica dos resultados computados em 8 imagens aleatórias do banco COCO, percebe-se que além de mais rápido o *Mobilenet-SSD* é mais preciso. Além disso, percebe-se que o PeopleDetector funciona melhor para detectar humanos em posição completamente vertical e em posições que mostram com clareza suas pernas e braços.

5.2.3 Comparação de Resultados

Os dois algoritmos foram testados em imagens do banco CoCO, a comparação dos resultados encontra-se na Tabela 02.

Tabela 02– Comparação de Resultados

Método	mAP	Tempo Médio de Processamento (s)
PeopleDetector	48%	1,522
Mobilenet-SSD	73%	0,332

Fonte: elaborada pelo autor.

Como trata-se de um problema de identificação e não classificação, foi utilizado como parâmetro comparativo a *mean average precision*, já para o parâmetro de tempo foi calculado a média dos tempos e processamento de cada imagem. Utilizou-se uma amostra de 46 imagens sendo 40 delas obtidas do banco de dados COCO e o restante proveniente das imagens obtidas no GPAR.

Percebe-se que o segundo método foi superior tanto em precisão quanto em tempo de processamento e, além disso, por ser implementado utilizando bibliotecas também presentes no sistema operacional do *Raspberry PI*, se mostrou mais adaptado para aplicações em plataformas embarcadas.

CAPÍTULO 06: CONCLUSÃO E TRABALHOS FUTUROS

O capítulo final desse trabalho é dividido em dois tópicos, o primeiro apresenta as conclusões obtidas e o segundo apresenta possíveis melhorias e trabalhos futuros a serem desenvolvidas.

6.1. Conclusão

No desenvolvimento desse trabalho, foram implementados dois métodos diferentes para a detecção de humanos, um baseado numa arquitetura de aprendizado profunda e outro numa arquitetura rasa de aprendizado supervisionado.

O primeiro método de detecção possui um processo de extração de características da imagem bastante específico que não muda durante o processo, características essas que são inseridas num classificador SVM, e conforme visto anteriormente, possui poucos parâmetros a serem aprendido, por conta disso o seu treinamento requer menos imagens.

Já o segundo método possui um método de extração de características que é alterado durante o próprio treinamento por meio da descida de gradiente com o intuito de minimizar o erro da rede, mudando os pesos das dezenas de camadas de convolução existentes no modelo, porém em compensação possui milhões de parâmetros o que requer um treinamento muito mais intenso.

Ao comparar o resultado dos dois métodos é notável a superioridade da rede neural profunda, que realizou a detecção cerca de 4,6 vezes mais rápido que o primeiro método e, além disso, apresentou-se mais de 25% mais preciso, mostrando que existe, nesse caso, um trade-off entre a simplicidade do modelo e sua precisão.

É importante frisar também que, embora os resultados do primeiro método não tenham sido tão bons, é possível que a limitação da configuração da ferramenta *PeopleDetect* tenham influenciado negativamente o resultado de modo que se o algoritmo tivesse sido implementado utilizando uma plataforma com mais liberdade de alteração dos parâmetros ou tivesse sido treinado com o banco de imagens próprio, o método poderia ter obtido resultados mais consistentes e precisos.

É possível argumentar que a utilização de uma rede que não é especializada em detectar humanos e sim treinada para detectar 20 classes diferentes pode comprometer a precisão e robustez do algoritmo, porém a precisão e fluidez do método já são de ótima qualidade para inúmeras aplicações.

Ademais, é possível ressaltar que, como as bibliotecas de *OpenCV* e *Imutils* estão presentes em outras plataformas, como o *Raspberry PI*, o método *Mobilenet-SSD* pode ser aplicável para robôs móveis que utilizem o *Raspberry* como unidade de processamento.

6.2. Trabalhos Futuros

- *Treinamento de Rede própria:* Apesar dos resultados do Mobilenet-SSD já terem apresentado um bom grau de confiabilidade, talvez seja possível atingir um resultado ainda melhor se fosse treinada uma rede convolucional profunda projetada especificamente para detecção de pessoas, utilizando técnicas de transferência de conhecimento no modelo utilizado, visto que ele possui uma licença MIT.
- *Aplicações em planejamento de trajetória*: Aliar a qualidade da detecção de pessoas, rapidez e compatibilidade com o *Raspberry* do método com técnicas de fusão sensorial com laser com o intuito de utilizar a detecção de pessoas como influencia na trajetória de robôs móveis. Não seria factível a utilização da detecção pura do método visto a ausência de noção de profundidade da visão mono.

REFERÊNCIAS

A BEGINNER'S Guide to Neural Networks and Deep Learning. Disponível em: <https://skymind.ai/wiki/neural-network>. Acesso em: 08 nov. 2018.

BECKER, Dans. **Rectified Linear Units (ReLU) in Deep Learning**. 2018. Disponível em: <https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning?scriptVersionId=3528657>. Acesso em: 05 nov. 2018.

COMMON Objects in Context. Disponível em: <http://cocodataset.org/#home>. Acesso em: 09 nov. 2018.

CONVOLUTIONAL Neural Networks for Visual Recognition. Disponível em: <http://cs231n.github.io/convolutional-networks/>. Acesso em: 08 nov. 2018.

DALAL, Navneet; TRIGGS, Bill. **Histograms of Oriented Gradients for Human Detection**. 2005, San Diego, CA, USA, USA. IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) ... [S.l.]: IEEE, 2005. p. 886-893. Disponível em: <https://ieeexplore.ieee.org/document/1467360>. Acesso em: 03 nov. 2018.

Data Science Academy. **Deep Learning Book**, 2018. Disponível em: <http://www.deeplearningbook.com.br/>. Acesso em: 04 nov. 2018.

DOLLAR, Piotr et al. **Pedestrian Detection: An Evaluation of the State of the Art**. 2011. Disponível em: <https://ieeexplore.ieee.org/abstract/document/5975165>. Acesso em: 07 dez. 2018.

FUNÇÕES de Ativação. Disponível em: <https://matheusfacure.github.io/2017/07/12/activ-func/#relu>. Acesso em: 08 nov. 2018.

GANDHI, Rohith. **Support Vector Machine: Introduction to Machine Learning Algorithms**. 2018. Disponível em: <https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47>. Acesso em: 14 nov. 2018.

HAYKIN, S. **Redes neurais: princípios e prática**. Porto Alegre: Bookman, 2001.

HOLLEMANS, Matthijs. **MOBILENET version 2**. 2018. Disponível em: <http://machinethink.net/blog/mobilenet-v2/>. Acesso em: 08 nov. 2018.

HOWARD, Andrew G. et al (2017). **Mobilenets: Efficient convolutional neural networks for mobile vision applications**. CoRR, abs/1704.04861.

JIA, Yangqing et al (2014). **Caffe: Convolutional architecture for fast feature embedding**. CoRR, abs/1408.5093.

JIA, Yangqing et al. **CAFFE Tutorial**. 2014. Disponível em: <http://caffe.berkeleyvision.org/tutorial/>. Acesso em: 05 nov. 2018.

KLEIN, Bruno. **DEEP Learning Notes**: Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization. 2013. Disponível em:

<https://github.com/brunoklein99/deep-learning-notes/blob/master/README.md>. Acesso em: 05 nov. 2018.

KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. **ImageNet classification with deep convolutional neural networks**. 2012, Lake Tahoe, Nevada, USA. NIPS'12 Proceedings of the 25th International Conference on Neural Information Processing Systems... [S.l.: s.n.], 2012. p. 1097-1105. v. 1. Disponível em: <http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf>. Acesso em: 05 nov. 2018.

LIU, Wei et al. **SSD: Single Shot MultiBox Detector**. 14., 2016, Amsterdam, The Netherlands. Computer Vision – ECCV 2016... [S.l.]: Springer, 2016. p. 21-37. Disponível em: <https://arxiv.org/pdf/1512.02325.pdf>. Acesso em: 08 nov. 2018.

Machine Learning Guru. **Image Filtering: A comprehensive tutorial towards 2D convolution and image filtering** (The first step to understand Convolutional Neural Networks (CNNs)). 2018. Disponível em: <https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning?scriptVersionId=3528657>. Acesso em: 03 nov. 2018.

MALLICK, Satya. **Histogram of Oriented Gradients**. 2016. Disponível em: <https://www.learnopencv.com/histogram-of-oriented-gradients/>. Acesso em: 03 nov. 2018.

MJOLNES, Eric; DECOSTE, Dennis. **Machine Learning for Science: State of the Art and Future Prospects**. 293. 2001. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=89F0D5C026F43F9A3598733EC4E54411?doi=10.1.1.18.7659&rep=rep1&type=pdf>. Acesso em: 07 dez. 2018.

NG, Andrew; KATANFOROOSH, Kian; MOURRI, Younes Bensouda. Improving Deep **Neural Networks: Hyperparameter tuning, Regularization and Optimization: RMSprop**. Disponível em: <https://www.coursera.org/lecture/deep-neural-network/rmsprop-BhJlm>. Acesso em: 08 nov. 2018.

NIELSEN, Michael. **Using Neural Nets to Recognize Handwritten Digits**. 2018. Disponível em: <https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning?scriptVersionId=3528657>. Acesso em: 05 nov. 2018.

OVERVIEW OF mini-batch gradient descent. Disponível em: <http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf>. Acesso em: 08 nov. 2018.

RICCO, Jay. **MAX-POOLING / Pooling**. 2017. Disponível em: <https://computersciencewiki.org/index.php/Max-pooling_/_Pooling>. Acesso em: 05 nov. 2018.

RILEY, Alex. **DIFFERENCE between contiguous and non continuous array**. 2014. Disponível em: <https://stackoverflow.com/questions/26998223/what-is-the-difference-between-contiguous-and-non-contiguous-arrays/26999092>. Acesso em: 05 nov. 2018.

RIZWAN, Muhammad. **RMSprop**. Disponível em: <https://engmrk.com/rmsprop/?utm_campaign=News&utm_medium=Community&utm_source=DataCamp.com>. Acesso em: 08 nov. 2018.

ROSEBROCK, Adrian. **Intersection over Union (IoU) for object detection**. 2016. Disponível em: <https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/>. Acesso em: 08 nov. 2018.

SINHA, Utkarsh. **CONVOLUTIONS**. 2017. Disponível em: <http://aishack.in/tutorials/image-convolution-examples/>. Acesso em: 03 nov. 2018.
THE MIT License (MIT). Disponível em: <https://mit-license.org/>. Acesso em: 09 nov. 2018.

UNDERSTAND the Softmax Function in Minutes. Disponível em: <https://medium.com/data-science-bootcamp/understand-the-softmax-function-in-minutes-f3a59641e86d>. Acesso em: 06 nov. 2018.

VISION.PEOPLEDETECTOR System object: Detect upright people using HOG features. Disponível em: <https://www.mathworks.com/help/vision/ref/vision.peopledetector-system-object.html>. Acesso em: 03 nov. 2018.

VISUAL Object Classes Challenge 2012 (VOC2012). Disponível em: <http://host.robots.ox.ac.uk/pascal/VOC/voc2012/#voc2012vs2011>. Acesso em: 09 nov. 2018.

VOULODIMOS, Athanasios et al. **Deep Learning for Computer Vision: A Brief Review**. 2017. Disponível em: <https://www.hindawi.com/journals/cin/2018/7068349/>. Acesso em: 07 dez. 2018.

APÊNDICE A – CÓDIGO PRIMEIRO MÉTODO

```
%%
clear all;
close all;
%% Definição dos parâmetros da Câmera
cam=webcam;
cam.Contrast=30
cam.Resolution='320x240'
%% Definição dos parâmetros do PeopleDetector
detector=vision.PeopleDetector;
detector.ClassificationModel='UprightPeople_96x48';
detector.release;
detector.ScaleFactor=1.001;
detector.ClassificationThreshold=1.4;
detector.WindowStride=[8 8];
detector.MinSize=[96 48]
detector.MaxSize=[300 300]
go=1;
set(gcf,'doublebuffer','off');
%% Loop Principal
while go==1
      % Obtenção da imagem.

      I = snapshot(cam);



      I=imresize(I,[300 300]);
      % Detecção.

      [bboxes, scores] = step(detector, I);
      I = insertObjectAnnotation(I, 'rectangle', bboxes, scores);


      % Mostra a imagem

      image(I)
      drawnow;


end
```

APÊNDICE B – CÓDIGO SEGUNDO MÉTODO

```
from imutils.video import VideoStream
from imutils.video import FPS
import numpy as np
import imutils
import time
import cv2

# Inicialização da lista de classes treinadas na rede mobilenet-ssd

CLASSES = ["nada", "avião", "bicicleta", "pássaro", "barco",
                "garrafa", "onibus", "carro", "gato", "cadeira", "vaca", "mesa",
                "cachorro", "cavalo", "motocicleta", "humano", "vaso", "ovelha",
                "sofa", "trem", "monitor"]

IGNORE = ["nada", "aviao", "bicicleta", "pássaro", "barco", "garrafa", "onibus", "carro",
"gato", "cadeira", "vaca",
              "mesa", "cachorro", "cavalo", "motocicleta", "vaso", "ovelha", "sofa",
"trem", "monitor"]

CORES = np.random.uniform(0, 255, size=(len(CLASSES), 3))

# Carrega modelo da rede


net = cv2.dnn.readNetFromCaffe("MobileNetSSD_deploy.prototxt.txt",
"MobileNetSSD_deploy.caffemodel")

# Inicizalização da câmera

vs = VideoStream(src=0).start()
time.sleep(2.0)
```

```
# Loop principal
while True:

	# carrega frame da camera e define largura maxima para 400
	frame = vs.read()
	frame = imutils.resize(frame, width=400)

	# Conversão da imagem em blob
	(h, w) = frame.shape[:2]
	blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)),
                                        0.007843, (300, 300), 127.5)

	# Insere blob como entrada na rede e obtem-se saída
	net.setInput(blob)
	detections = net.forward()

	# Loop nas detecções
	for i in np.arange(0, detections.shape[2]):
		# Extração da confiança e localização das caixas
		confidence = detections[0, 0, i, 2]

		# Remoção das detecções com confiança menor que 60%
		if confidence > 0.6:
			# Extrai as classes detectadas na imagem
			idx = int(detections[0, 0, i, 1])

			if CLASSES[idx] in IGNORE:
				continue

			box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])

			(startX, startY, endX, endY) = box.astype("int")

			label = f'{CLASSES[idx]}: {confidence * 100:.2f}%'
			cv2.rectangle(frame, (startX, startY), (endX, endY),
                                CORES[idx], 2)
			y = startY - 15 if startY - 15 > 15 else startY + 15
			cv2.putText(frame, label, (startX, y),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.5, CORES[idx], 2)
			# desenha frame de saida
	cv2.imshow("Frame", frame)
	key = cv2.waitKey(1) & 0xFF

# quebra de loop
	if key == ord("q"):
		break
cv2.destroyAllWindows()
vs.stop()
```

APÊNDICE C – LINCENÇA DO MOBILENET-SSD

APÊNDICE D – LICENÇA DO IMUTILS

APÊNDICE E – LICENÇA DO OPENCV

APÊNDICE F – LICENÇA DO CAFFE

COPYRIGHT
All contributions by the University of California:
Copyright (c) 2014-2017 The Regents of the University of California (Regents)
All rights reserved.
All other contributions:
Copyright (c) 2014-2017, the respective contributors
All rights reserved.
Caffe uses a shared copyright model: each contributor holds copyright over
their contributions to Caffe. The project versioning records all such
contribution and copyright details. If a contributor wants to further mark
their specific copyright on a particular contribution, they should indicate
their copyright solely in the commit message of the change when it is
committed.
LICENSE
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF
THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
CONTRIBUTION AGREEMENT
By contributing to the BVLC/caffe repository through pull-request, comment,
or otherwise, the contributor releases their content to the
license and copyright terms herein.