Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Implementação de um Sistema de Visão Computacional Embarcado em um Robô Móvel para Realização de Atividades Agrícolas

Danielly Omori Antunes de Oliveira

CASCAVEL
2020

DANIELLY OMORI ANTUNES DE OLIVEIRA

IMPLEMENTAÇÃO DE UM SISTEMA DE VISÃO COMPUTACIONAL EMBARCADO EM UM ROBÔ MÓVEL PARA REALIZAÇÃO DE ATIVIDADES AGRÍCOLAS

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Josué Pereira de Castro

CASCAVEL
2020

DANIELLY OMORI ANTUNES DE OLIVEIRA

IMPLEMENTAÇÃO DE UM SISTEMA DE VISÃO COMPUTACIONAL EMBARCADO EM UM ROBÔ MÓVEL PARA REALIZAÇÃO DE ATIVIDADES AGRÍCOLAS

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Josué Pereira de Castro (Orientador)
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Adriana Postal
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Suzan Kelly Borges Piovesan
Colegiado de Ciência da Computação, UTFPR -
Santa Helena

Cascavel, 14 de julho de 2021

DEDICATÓRIA

Dedico este trabalho à minha irmã e minha mãe, que batalharam arduamente durante anos para tornar possível eu chegar até aqui. Amo vocês com todo meu coração!

EPÍGRAFE

“Sing with hope and the fear will be gone”
So Ist Es Immer- Hiroyuki Sawano

AGRADECIMENTOS

Primeiramente gostaria de agradecer com todo meu coração à minha mãe e a minha irmã, pessoas essas que tornaram possível eu chegar aqui, finalizando meu Trabalho de Conclusão de Curso.

Mãe, obrigada por lutar contra tudo e todos para que eu e a Katia pudéssemos concluir nossos estudos e nos tornarmos mulheres independentes. Obrigada por todos os lanches que você fez para que eu pudesse passar noites estudando, sem morrer de fome. Obrigada por me mostrar que nós mulheres precisamos ser bem duronas, para enfrentar todos os obstáculos que irão aparecer, de cabeça erguida e dando o nosso melhor sempre. Obrigada por ser essa mulher incrível, que me inspira todo dia. Espero um dia ser um décimo da pessoa incrível que você é.

Katia, lembra de todos os textos meus que você corrigiu e ficou irada pois eu não levava em consideração o que o leitor iria entender com as minhas palavras? Aqui vão dezenas de páginas escritas utilizando todos conselhos que você me deu durante a minha vida. Minha irmã, obrigada por estar ao meu lado sempre, por ter desbravado esse mundo acadêmico quando ninguém da nossa família conseguiu, obrigada por ser forte e corajosa e também me inspirar todo dia. E principalmente, obrigada por ter trazido ao mundo a coisa mais lindinha e fofa do mundo, Ruan, que tem alegrado todos meus dias desde quando ficamos sabendo da sua existência.

Também agradeço a todos outros membros da família por todo apoio e esperança que vocês tiveram em mim durante esses cinco anos de graduação. Espero que eu possa ter aberto mais portas para as gerações seguintes, assim como fizeram para mim. Espero que possamos evoluir cada vez mais!

Outra pessoa que não posso deixar de agradeçer é meu namorado Vinícius, que me aturou surtando durante toda a realização deste trabalho, desde códigos que não funcionavam, até prazos se aproximando e me deixando super ansiosa. Obrigada por sempre me incentivar e me dar forças para seguir em frente, independentemente do quão cansada eu estivesse. Obrigada por me fazer acreditar mais em mim e me mostrar que, apesar de lerda, eu sou capaz de fazer

muitas coisas.

Por fim, não tem como não agradecer a todas as pessoas que eu convivi durante esses cinco anos e tornaram essa jornada inesquecível. Obrigada a todos meus colegas de curso que surtaram comigo, me ajudaram a aprender as matérias, me fizeram rir, ficaram até tarde fazendo trabalhos. Obrigada por todas as noites indo no lanche, para ficar conversando sobre coisas aleatórias. Obrigada pelas viagens, projetos participados, obrigada por tudo! Agradeço também a todos professores que deram seu melhor para poder ensinar coisas que seriam úteis para nós e por compartilhar suas experiências de vida conosco, podem ter certeza que um pedacinho de vocês estará para sempre no meu coração e me farão lembrar de todos esses momentos que passamos juntos. Espero que nos vejamos em breve para que eu possa dar um abraço bem apertado em todos.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ABDI	Agência Brasileira de Desenvolvimento Industrial
CNN	Redes Neurais Convolucionais (*Convolutional Neural Networks*)
GB	Gigabyte
GPU	Unidade de Processamento Gráfico (*Graphics Processing Unit*)
ICRISAT	Instituto Internacional de Pesquisa de Culturas para o Trópico Semi-Árido
ID	Identificador
KNN	K Vizinhos mais Próximos (*k-nearest neighbor*)
MB	Megabyte
PC	Computador Pessoal
RAM	Memória de Acesso Aleatório (*Random Access Memory*)
RGB	Vermelho-Verde-Azul (*Red-Green-Blue*)
RNA	Redes Neurais Artificiais
SO	Sistema Operacional
SVM	Máquina de Vetores de Suporte (*Support Vector Machine*)
TI	Tecnologia da Informação
USB	Porta Serial Universal (*Universal Serial Bus*)
YOLO	*You Only Look Once*

Sumário

Resumo

Neste trabalho foi desenvolvido um sistema de visão computacional com objetivo de ser embarcado em um robô móvel para a detecção de objetos em ambiente agrícola. Foram desenvolvidos dois protótipos de Redes Neurais Convolucionais para a realização desta tarefa, o primeiro baseado na arquitetura *AlexNet* e o segundo no sistema YOLO. O segundo protótipo apresentou melhores resultados referentes a desempenho e por gerar uma rede menor. Por este motivo, o mesmo foi escolhido para ser embarcado no robô, sendo capaz de detectar objetos através de imagens, vídeos e em tempo real.

Palavras-chave: Redes Neurais Convolucionais, Visão Computacional, YOLO, AlexNet, RaspBerry Pi.

Capítulo 1

Introdução

Nos últimos trinta anos ocorreu um grande desenvolvimento na área de Tecnologia da Informação (TI) e a sua utilização em processos de produção trouxeram evoluções na forma como tecnologias poderiam aumentar a produtividade industrial. Graças a isso, soluções passaram a ser desenvolvidas de forma mais eficaz e barata, tendo um melhor custo/benefício do que era possível antes de toda essa mudança (SANTOS et al., 2018).

Toda essa "informatização" fez com que fosse discutido sobre um novo modelo de indústria, a Indústria 4.0. Este termo foi apresentado na Alemanha, no ano de 2011, fazendo referência ao que seria a Quarta Revolução Industrial (PEREIRA; SIMONETTO, 2018). Nesta revolução é previsto que ocorra a integração entre humanos e máquinas, compondo uma rede de entidades localizadas em posições geograficamente distribuídas, e que devem fornecer serviços e produtos de forma autônoma (SILVA; FILHO; MIYAGI, 2015). Diversas áreas vêm estudando a Indústria 4.0, como: Administração, Engenharia Elétrica e Ciência da Computação (PEREIRA; SIMONETTO, 2018) e é notável que segmentos das mais diversas áreas têm se interessado em utilizar tecnologias que aumentem a produtividade em seu meio.

Uma área que tem recebido muito incentivo para entrar no mundo tecnológico é a área agrícola. No Brasil, por ser um dos grandes alicerces da economia, é fundamental o estudo sobre o uso de tecnologias que possam ser usadas a seu favor pois é possível aumentar o rendimento de lavouras em até 67% (JARDIM, 2019). Segundo um relatório publicado pela *McKinsey* no ano de 2014, o uso de *big data* na agricultura brasileira, poderia gerar um ganho de até R$ 24 bilhões nos cinco anos posteriores ao estudo para as culturas de soja, milho e trigo (PIONEER, 2014).

Alguns projetos já vem sendo desenvolvidos com o intuito de estimular este mercado a in-

vestir cada vez mais no desenvolvimento de soluções voltadas para o agronegócio. Segundo a revista Globo Rural (2020), os ministérios da Agricultura, Economia e da Ciência, Tecnologia e Inovações, juntamente com a Agência Brasileira de Desenvolvimento Industrial (ABDI) lançaram o programa Agro 4.0, que tem como objetivo promover o aumento de eficiência e produtividade, além de reduzir custos no agronegócio brasileiro.

Tais necessidades, unidas com o fato que, na região oeste do Paraná a agricultura tem extrema importância, motivaram o desenvolvimento deste trabalho. Neste projeto, desenvolveu-se uma Rede Neural Convolucional que utiliza Visão Computacional para realizar a classificação de objetos do meio agrícola. A rede passou por um treinamento, utilizando bases de dados contendo imagens relacionadas ao meio agrícola e depois foi adaptada para que fosse capaz de classificar os objetos transmitidos em tempo real pela câmera embutida existente no robô.

1.1 Objetivo

Este trabalho teve como objetivo desenvolver um sistema de visão em um robô embarcado de pequeno porte, com uma câmera embutida, tornando-a capaz de detectar objetos em um meio-ambiente agrícola, localizando-os e identificando-os corretamente.

1.1.1 Objetivos Específicos

1. Realizar uma revisão bibliográfica sobre assuntos pertinentes, tais como:
    - Redes Neurais Convolucionais;
    - Visão Computacional em Robótica;
    - Aprendizagem de máquina.
2. Seleção dos dados que serão utilizados para a realização do treinamento da rede.
3. Desenvolvimento de uma rede neural que seja capaz de reconhecer objetos que serão transmitidos em tempo real através de uma câmera convencional.
4. Carregamento da rede neural desenvolvida no robô móvel, realizando as alterações necessárias para que a rede seja capaz de receber informações da câmera embarcada existente no dispositivo.

5. Integração do sistema de visão ao sistema de controle do robô, para que a visão possa ser usada como parte de seu sistema de localização.

1.2 Organização do Trabalho

Este trabalho abordará os seguintes temas:

- **Capítulo 1 - Introdução**: Apresentação do tema que será abordado, esclarecendo qual o objetivo geral que se deseja cumprir e quais são os objetivos específicos a serem realizados durante o desenvolvimento do trabalho;
- **Capítulo 2 - Visão Computacional**: Abordagem breve do histórico, principais etapas que compõem um sistema de visão computacional e apresentação de alguns trabalhos que utilizam esse tipo de sistema para a automatização de determinadas tarefas;
- **Capítulo 3 - Redes Neurais Convolucionais**: Introdução sobre Redes Neurais Artificiais e o surgimento das Redes Neurais Convolucionais (CNN). Também será realizada uma explicação sobre o funcionamento das CNN, quais seus principais tipos de camadas e arquiteturas criadas utilizando esse tipo de rede;
- **Capítulo 4 - Materiais**: Apresentação dos materiais utilizados para o desenvolvimento do projeto;
- **Capítulo 5 - Implementação**: Explicação detalhada sobre como foram implementados os protótipos criados nesse projeto, abordando aspectos como organização da arquitetura e parametrização utilizada;
- **Capítulo 6 - Resultados e Conclusões**: Apresentação dos resultados obtidos neste trabalho, apontando melhorias que poderão ser realizadas em projetos futuros.

Capítulo 2

Visão Computacional

2.1 Introdução

No reino animal a percepção visual possui grande importância em aspectos relacionados ao comportamento e à sobrevivência. A capacidade de visualizar objetos em um campo de visão e realizar o reconhecimento de forma tão rápida e precisa é um dos processos mais complexos já observados, o que torna a visão algo de difícil replicação em máquinas, as quais não possuem o poder de processamento que um cérebro dispõe (KRAGIC; VINCZE, 2009) (ACADEMY, 2018).

A área de Visão computacional é considerada uma ciência recente, tendo sua primeira menção no artigo *Eyes and ears for the computer*, escrito por David e Selfridge (1962). Ela pode ser considerada um subcampo da Inteligência Artificial e Aprendizagem de Máquina, e seu objetivo principal é a replicação da visão humana através da utilização de *softwares* e *hardwares* específicos para essa finalidade. Sua finalidade consiste em estudar formas de fazer com que máquinas sejam capazes de "enxengar" o que está a sua volta, extraindo informações significativas por meio de imagens capturadas por câmeras de vídeo, *scanners* e outros meios de entrada de dados (BROWNLEE, 2019) (ACADEMY, 2018) (MILANO; HONORATO, 2014) .

A proposta de ensinar a máquina a "enxergar" se desenvolveu inicialmente de forma lenta. Pesquisadores perceberam que primeiro seria necessário entender com exatidão o funcionamento de um sistema de visão humano para depois ser capaz de replicar isso em uma máquina (MILANO; HONORATO, 2014). Esse processo de entendimento perdurou por anos pois, até os mais "triviais" problemas apresentados se tornavam complicados conforme as soluções desenvolvidas eram testadas e não obtinham o resultado esperado (DAVIES, 2017).

Apesar desses impasses, com aumento do poder de processamento, da memória e da capacidade de armazenamento dos computadores, a área de Visão Computacional teve seu progresso acelerado de forma que no ano de 2020, quase sessenta anos após sua primeira menção, ela foi integrada aos mais variados tipos de aplicações. Seu uso, que parecia algo inalcançável, no ano de 2020 pode ser visto em veículos autônomos, mercado de marketing, segurança, serviços públicos, medicina e para auxílio militar (ALIGER, 2020), tendo extrema importância em diversos contextos, automatizando vários processos manuais, tidos como desgastantes.

2.2 Etapas de um Sistema de Visão Computacional Básico

Conforme a área de Visão Computacional foi se consolidando, inúmeros sistemas foram desenvolvidos e publicados. Segundo Zareiforoush et al. (2015), esses sistemas são compostos, em sua maioria, por dois estágios principais: aquisição e processamento das imagens. Também podem ser incluídas as etapas de iluminação e digitalização, com o objetivo de facilitar a aquisição dos dados e, consequentemente, o sistema de visão desenvolvido.

A finalidade de cada uma dessas etapas será descrita a seguir (ZAREIFOROUSH et al., 2015).

1. **Iluminação**: Como ocorre com o olho humano, o sistema de visão computacional também é influenciado pela intensidade e qualidade da iluminação. Isto posto, antes do processo de aquisição de imagens, é essencial assegurar que o objeto a ser analisado esteja localizado em um ambiente com boa iluminação;

2. **Aquisição**: Tida como primeira etapa de qualquer sistema de visão computacional, nesta etapa é quando ocorre a captura de sinais eletrônicos por meio de dispositivos em um formato numérico, tornando possível ao desenvolvedor obter os dados que serão utilizados em seu sistema de visão computacional. Essa aquisição pode ser realizada a partir de câmeras monocromáticas ou coloridas, *scanners*, ultrassons, raios X, entre outros dispositivos;

3. **Digitalização**: Essa etapa é necessária nos casos em que as imagens a serem utilizadas estão em um formato analógico. Esse processo é feito por meio de um digitalizador

que dividirá a imagem em um *grid* 2D formado por pequenas regiões contendo diversos pixels;

4. **Processamento**: Considerada a principal etapa de um sistema com visão computacional, nela são utilizadas diversas técnicas com objetivo de melhorar a qualidade da imagem, eliminando defeitos como complexidade geométrica, foco inapropriado, ruídos, iluminação não uniforme e "borrões" derivados de movimento na hora da captura. Para Zareiforoush et al. (2015) essa etapa pode ser dividida em 3 níveis denominados: baixo, intermediário e alto.

   O nível baixo se refere ao pré-processamento, incluindo o procedimento de captura da imagem bruta e modificações com intuito de corrigir distorções geométricas, eliminar ruídos, ajustar os níveis de cinza e corrigir certos desfoques (SHIRAI, 1987);

   O nível intermediário engloba operações de segmentação, representação e descrição da imagem. É uma das partes mais importantes em um sistema de visão computacional pois é na segmentação que ocorre a divisão da imagem para destacar regiões que possuam forte correlação com as áreas de interesse (SONKA; HLAVAC; BOYLE, 2014);

   O nível alto envolve o reconhecimento e interpretação das áreas de interesse obtidas na etapa anterior. Esse processo normalmente é realizado por classificadores estatísticos ou redes neurais de multi-camadas, os quais serão descritos na seção 2.3.

2.3 Visão Computacional e Aprendizagem de Máquina

A Aprendizagem de Máquina foi um dos fatores que possibilitou à área de Visão Computacional se desenvolver efetivamente (BROWNLEE, 2019). Seus estudos permitiram o desenvolvimento de métodos capazes de detectar automaticamente padrões em dados, conseguindo realizar previsões baseando-se em comportamentos observados em um conjunto de dados, ou até mesmo auxiliando em tomadas de decisões tidas como mais incertas (MURPHY, 2012).

Na área de Visão Computacional, a Aprendizagem de Máquina pode ser aplicada de duas formas (SEBE et al., 2005):

1. Aumentando a percepção de ambiente circundante, de forma a melhorar todos os sinais detectados em representações internas;

2. Fazendo um elo entre as representações internas de um ambiente e a representação de conhecimento necessária para que o sistema seja capaz de realizar a tarefa de reconhecimento.

A utilização de métodos da Aprendizagem de Máquina, em conjunto com Visão Computacional, tornou possível o desenvolvimento de sistemas cada vez mais robustos, com objetivo de automatizar tarefas que necessitariam obrigatoriamente da presença humana, sendo capazes de receber imagens como entrada e extrair informações relevantes delas.

Para melhor exemplificar a integração destas duas áreas, serão apresentados a seguir alguns trabalhos correlatos.

2.3.1 Trabalhos Correlatos

O trabalho de Åstrand e Baerveldt (2002) descreve um robô móvel agrícola capaz de realizar o controle de ervas daninhas em uma plantação de beterrabas. O sistema de visão do robô utilizava duas câmeras, localizadas na parte superior e inferior de sua estrutura. A primeira possibilitava ao sistema localizar "corredores" pela plantação, tornando possível ao robô locomover-se de forma autônoma pelas direções corretas, e a outra câmera apontava para baixo e tinha como finalidade analisar as plantas existentes em seu campo de visão. Para que o sistema fosse capaz de reconhecer as ervas daninhas em meio às beterrabas, utilizou-se um algoritmo de Aprendizagem de Máquina específico para problemas de classificação, no qual o objetivo é receber uma informação e atribuir um rótulo a ela. Neste algoritmo é determinado um classificador, "agente" responsável por analisar e classificar os dados solicitados. Nesse trabalho o classificador utilizado foi o K Vizinhos mais Próximos (*K-Nearest Neighbor* - KNN) e nos estágios desenvolvidos em laboratório ele obteve uma acurácia de 96%.

No artigo publicado por Jodas et al. (2013) também foi descrito um sistema que fosse capaz de fazer com que um robô móvel pudesse deslocar-se por plantações de forma autônoma, mas com o foco na realização de tarefas como detecção de pragas em plantações e controle da aplicação de agrotóxicos. A solução proposta também utilizou algoritmos para problemas de classificação, porém foi utilizado o classificador Máquina de Vetor de Suporte (*Support Vector*

Machine - SVM). Sua utilização tinha como foco reconhecimento de caminhos existentes nas imagens obtidas para o seu treinamento. Técnicas para correções das imagens foram utilizadas durante as etapas iniciais e ao fim do projeto o sistema foi capaz de atingir uma acurácia de 93,19%.

O trabalho desenvolvido por Mustaffa e Khairul (2017) possuía um objetivo diferente dos demais citados: a criação um sistema apto a identificar tamanho e maturidade de frutas. Neste trabalho utilizou-se a biblioteca OpenCV e linguagem Python para a criação do sistema. Para a identificação do nível de maturação das frutas, foi utilizado outro algoritmo derivado da aprendizagem de máquina para problemas de Agrupamento (*Clustering*). Seu objetivo era reunir amostras que possuem características semelhantes de forma a separar aquelas que são distintas. Foi utilizado método *K-means* para a realização desta tarefa. Em relação à identificação do tamanho das frutas, os autores desenvolveram um sistema que baseava-se em uma manga com medidas já conhecidas pelo sistema como referência, o que tornava-o capaz de mensurar as demais frutas. Os valores de altura e largura eram determinados através do uso da métrica de distância Euclidiana.

Paul et al. (2020) em seu artigo abordam duas soluções desenvolvidas por empresas na Índia com objetivo de facilitar a vida de camponeses. A primeira, criada pela companhia Intello Labs, fornecia uma avançada tecnologia para reconhecimento de imagens utilizando redes neurais profundas. Seu uso foi empregado em um aplicativo capaz de detectar a qualidade dos produtos agrícolas através de fotografias tiradas pelos agricultores e em um sistema que alerta sobre a infestações em safras, além de fornecer orientações sobre tratamentos e mecanismos de prevenção. A segunda solução tinha como foco auxiliar agricultores, que sofrem com o excesso de chuvas e seca existentes na Índia. Ela foi desenvolvida pela Microsoft em conjunto com o Instituto Internacional de Pesquisa de Culturas para o Trópico Semi-Árido (ICRISAT) e utilizou técnicas de Aprendizagem de Máquina e Inteligência de Negócios (*Business Intelligence* - BI). O aplicativo desenvolvido, além de determinar qual o período ideal para realizar plantações, também é capaz de alertar agricultores se seus produtos estão vuneráveis à ataques de pragas considerando as condições climáticas.

Capítulo 3

Redes Neurais Convolucionais

3.1 Introdução

O cérebro humano é uma máquina poderosa, hábil para processar um número expressivo de informações em uma fração de tempo. A capacidade de reconhecimento de um rosto familiar entre uma multidão, por exemplo, gerava muitas dúvidas entre pesquisadores, que não conseguiam compreender como tal tarefa era realizada de forma tão eficiente. Iniciaram-se então estudos para a criação de um sistema inteligente capaz de realizar tarefas como reconhecimento de padrões, classificação, processamento de imagens, entre outras, utilizando o aprendizado por experiência, como é feito por seres humanos (ACADEMY, 2019).

Esses estudos originaram as chamadas Redes Neurais Artificiais (RNA) e seu primeiro modelo foi publicado por McCulloch e Pitts (1943), utilizando conceitos matemáticos e unidades lógicas de limiar (*threshold logic unit*)[1] (ACADEMY, 2019). Essa rede era um classificador binário - capaz de reconhecer duas categorias diferentes, baseando-se no valor de entrada. No entanto, os pesos utilizados na rede para a execução da classificação eram inseridos manualmente, o que fazia com que o modelo não fosse tão "independente", pois precisava da intervenção humana (ROSEBROCK, 2017).

O neurobiologista Rosenblatt (1958) criou então o algoritmo *Perceptron* em uma pesquisa sobre o funcionamento do sistema de visão de uma mosca. Ele possuía duas camadas e era utilizado para classificar conjuntos de entradas com valor contínuo em somente duas classes, utilizando operações simples de adição e subtração. Seu diferencial, se comparado ao modelo

[1] Modelo de neurônio desenvolvido por McCulloch e Pitts. Ele é ativado quando recebe uma entrada excitatória suficiente que não é compensada por uma entrada inibitória igualmente forte (KRUSE et al., 2013).

proposto por McCulloch e Pitts (1943), era que ele era capaz de aprender os pesos necessários para realizar a classificação automaticamente, sem a necessidade de intervenção humana. Sua utilização, porém, era limitada a problemas linearmente separáveis, e essa restrição foi apresentada por Minsky e Papert (1969), fazendo com que as pesquisas sobre redes neurais ficassem estagnadas por décadas (ACADEMY, 2019) (ROSEBROCK, 2017).

Vários modelos foram propostos no decorrer desses anos, como o ADALINE e o MADALINE, criados por Widrow e Hoff no ano de 1959, porém, o assunto Redes Neurais só voltou a ser abordado após a criação do algoritmo de Retropropagação (*Backpropagation*) no ano de 1986. Além de utilizar múltiplas camadas, esse modelo também era capaz de empregar os erros obtidos durante o treinamento para ajustar os pesos existentes pela rede. Seu aspecto mais importante, todavia, era ser capaz de resolver problemas não linearmente separáveis, sanando assim as limitações dos modelos propostos anteriormente (ACADEMY, 2019).

Dessa forma, a área de redes neurais voltou a ter destaque, em razão das promissoras características apresentadas pelos modelos propostos e aumento do poder computacional, permitindo a criação de modelos cada vez mais complexos, que obtivessem ótimos resultados. Essa evolução culminou na criação da subárea Redes Neurais Profundas (*Deep Learning*), que ganhou popularidade após a publicação do artigo de Hinton e Salakhutdinov (2006), no qual era abordado como uma rede neural com várias camadas poderia ser pré-treinada, uma camada por vez (ACADEMY, 2019).

As redes neurais profundas empregam algoritmos para tratar dados e imitar o processamento feito pelo cérebro humano, sendo responsável por avanços recentes na área de visão computacional, reconhecimento de fala, processamento de linguagem natural e reconhecimento de áudio. Na Figura 3.1 é possível analisar que redes neurais profundas utilizam um número muito maior de camadas intermediárias (denominadas ocultas) do que uma rede simples, as tornando poderosas para a realização de tarefas como as descritas anteriormente (ACADEMY, 2019).

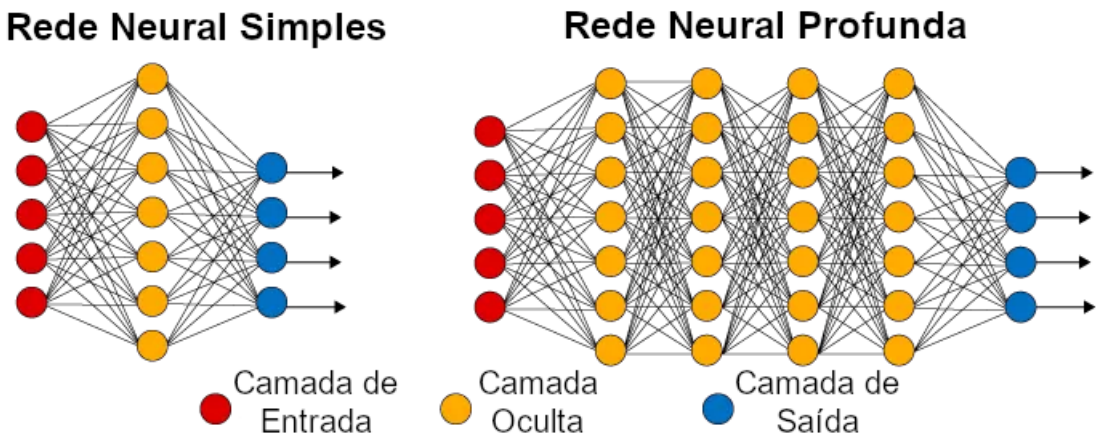


Figura 3.1: Estrutura de uma Rede Neural Normal x Estrutura de uma Rede Neural Profunda - Adaptada de Academy (2019)

Diversos modelos de redes neurais profundas foram desenvolvidas no decorrer dos anos. Neste trabalho, trataremos apenas das Redes Neurais Convolucionais (CNN - *Convolutional Neural Network*).

3.2 Redes Neurais Convolucionais

O artigo desenvolvido por LeCun et al. (1998) descrevia um reconhecedor capaz de resolver o problema de identificação de números manuscritos em imagens. Utilizando o algoritmo *Back-propagation* em uma rede *feedforward* [2] com diversas camadas ocultas e mapas de unidades replicadas em cada camada, LeCun foi capaz de criar uma poderosa rede, que não se limitava em ser somente um reconhecedor. Ele deu o nome ao seu trabalho de LeNet e mais tarde essa arquitetura foi formalizada sob o nome de Redes Neurais Convolucionais (ACADEMY, 2019).

Redes Neurais Convolucionais são redes neurais profundas utilizadas para problemas envolvendo classificação de imagens, agrupamento por similaridade e reconhecimento de objetos dentro de cena. Ela foi um dos motivos que tornou a aprendizagem profunda comprovada e atualmente podem ser vistas em carros autônomos, robótica, drone, diagnóstico médico, entre outras aplicações, juntamente com a área da visão computacional (ROSEBROCK, 2017) (ACADEMY, 2019).

[2]Nesse tipo de rede, cada camada se conecta à próxima camada, sem caminhos para voltar. Todas camadas, portanto, possuem a mesma direção, rumo a camada de saída (RUELA, 2012).

Segundo Rosebrock (2017), elas possuem dois benefícios:

1. **Invariância local**: permite que objetos possam ser classificados independente do ambiente em que ele está. Isso se deve ao uso de camadas de *pooling* (serão explicadas na seção 3.3.1), que utilizam filtros específicos para identificar em qual região o objeto está localizado.

2. **Composicionalidade**: Cada filtro compõe um grupo de recursos de nível inferior em uma representação de nível superior, o que permite que as redes neurais aprendam características mais ricas e profundas na rede. Sendo assim, a rede é capaz de utilizar conceitos menores para compor coisas maiores. Por exemplo, ela é capaz de utilizar pixels para criar bordas em uma imagem, utilizar essas bordas para criar diferentes formas, e por fim utilizar essas formas para construir objetos complexos.

3.2.1 Funcionamento

Redes neurais convolucionais possuem características em seu funcionamento que as tornam diferentes das demais redes neurais. A primeira delas é como é realizado o processamento das imagens. Para esse tipo de rede, uma imagem nada mais é do que uma matriz tridimensional, com eixos x e y referindo-se aos valores de largura e altura e o eixo z a profundidade (ACADEMY, 2019) (BONNER, 2019).

Quando se trata de imagens coloridas, os três canais de cores, vermelho-verde-azul (RGB - *Red Green Blue*), precisarão ser tratados e por isso o eixo de profundidade da matriz terá valor 3. Isso ocorre pois, como é possível ver na Figura 3.2, são utilizadas 3 camadas empilhadas, uma para cada canal, para que possa ser possível representar a cor daquele pixel. Como exemplo, a cor do primeiro pixel da Figura 3.2 será formada pelos valores: 35 para o tom vermelho, 9 para o tom verde e 4 para o tom azul (ACADEMY, 2019).

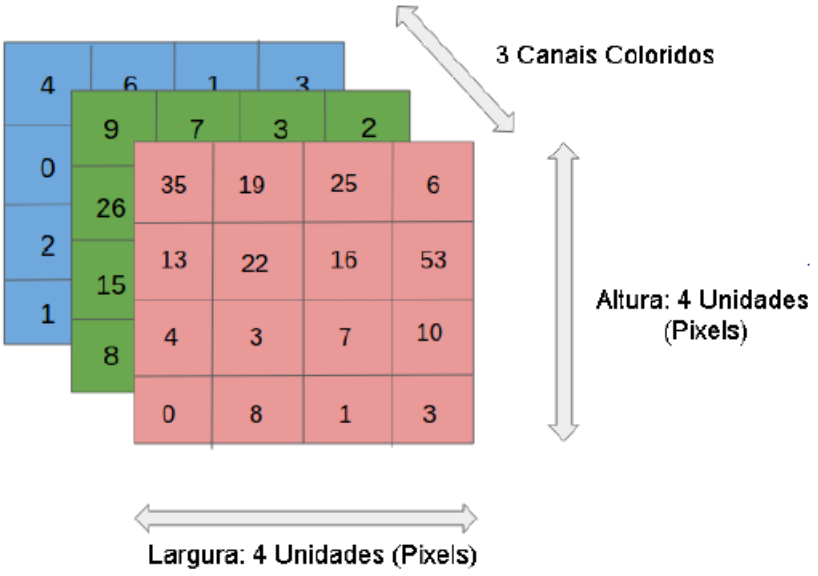


Figura 3.2: Representação de cores em uma Rede Neural Convolucional - Adaptada de Academy (2019)

Segundo o mesmo autor, a estrutura de uma rede neural convolucional também é um pouco diferente das demais. Conforme pode ser visto na Figura 3.3, a entrada da rede consiste em uma imagem colorida expressa de forma matemática por uma matriz $N \times M \times 3$ (caso seja uma imagem preto e branco o valor 3 será substituído por 1). Conforme essa matriz vai passando pelas camadas, suas dimensões vão sendo alteradas de forma que seja possível extrair o máximo de características possíveis sobre o objeto a ser analisado. Esses atributos vão ser analisados até as camadas finais, nas quais será gerado um conjunto de probabilidades que serão utilizadas para determinar a qual classe aquela imagem pertence (BONNER, 2019).

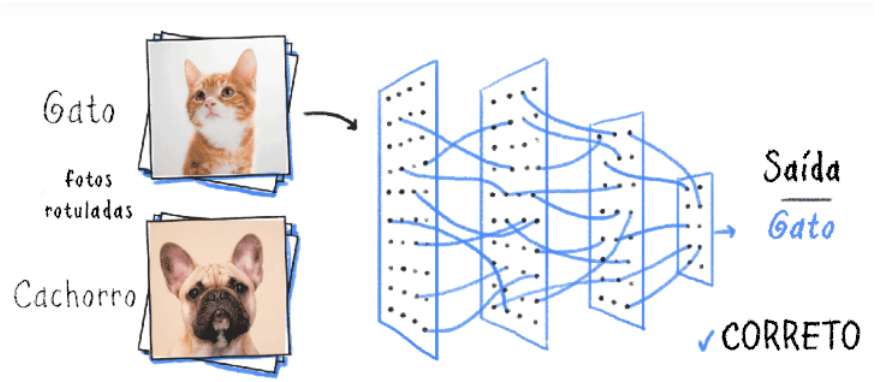


Figura 3.3: Modelo de uma Rede Neural Convolucional - Adaptada de Academy (2019)

Todo esse processo de extração de características e geração de probabilidades é executado por meio da utilização de tipos de camadas específicas. São várias camadas e algumas delas serão abordadas a seguir.

3.2.2 Tipos de Camadas de uma CNN

Segundo Academy (2019) e Rosebrock (2017), as principais camadas de uma rede neural convolucional são:

1. **Camada Convolucional** (*Convolutional Layer*): Esse tipo de camada é o principal elemento das redes neurais convolucionais. Nela são criados K filtros, geralmente em um formato quadrado pequeno, capazes de extrair características. Para isso, eles "deslizam" por toda a imagem de entrada, realizando as operações de convolução e armazenando os valores de saída em um mapa de ativação bidimensional. Esse processo pode ser melhor analisado na Figura 3.4.

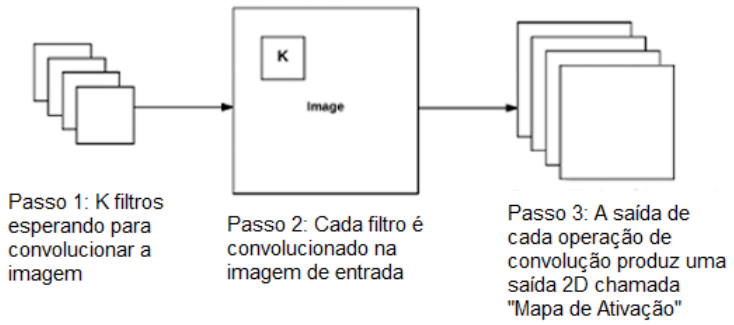


Figura 3.4: Exemplo de camada convolucional sendo aplicada em uma entrada - Adaptada de Rosebrock (2017)

Como todas as camadas posteriores a essa esperam uma matriz tridimensional de entrada e os mapas de ativação são bidimensionais, é necessários agregá-los para que eles possam ter profundidade, como pode ser visto na Figura 3.5. Esse mapa de ativação possuindo o tamanho correto, ele se torna capaz de transitar pela rede, sendo capaz de "aprender" quais características existem em determinadas localizações espaciais na imagem e ativar os neurônios corretamente. Em níveis mais profundos das camadas, esses filtros podem ativar na presença de características de mais alto nível como parte da face, uma pata do cachorro, etc.

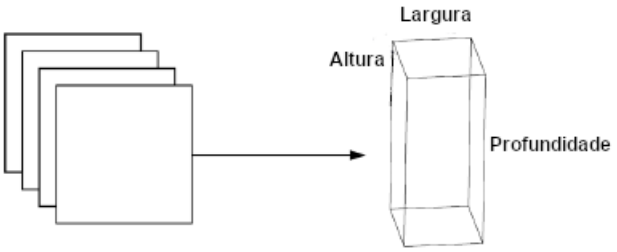


Figura 3.5: Camada convolucional após ser empilhada para formar o volume de entrada para a próxima camada - Adaptada de Rosebrock (2017)

Outro ponto que deve ser observado são as dimensões que o volume de saída terá ao ser gerado pelas convoluções realizadas. Para esse controle, são utilizados três parâmetros:

- **Profundidade (*Depth*)**: Esse parâmetro está relacionado ao número de canais de cores que a imagem possui. Conforme pode ser visto na Figura 3.6, a matriz possui

valor de profundidade igual à três, então o filtro deverá analisar esse número de camadas para gerar a saída para o mapa de ativação.

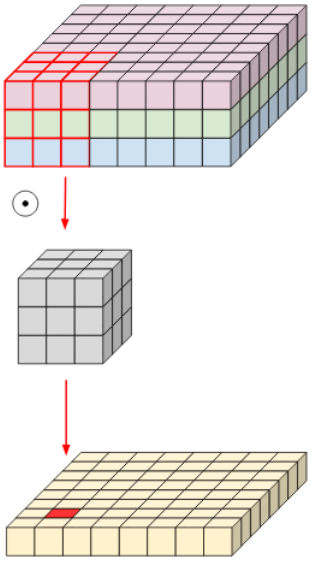
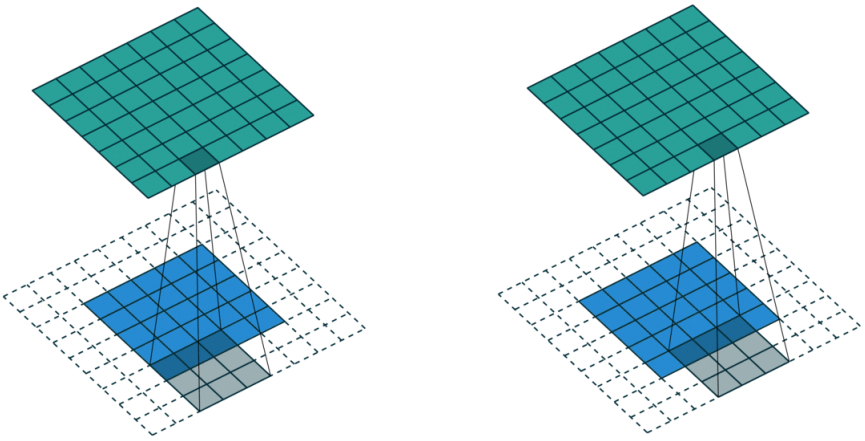
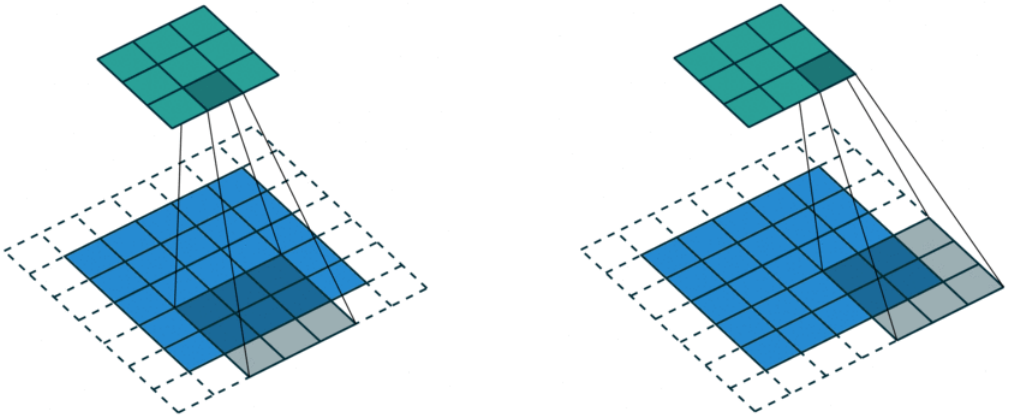


Figura 3.6: Parâmetro de profundidade com valor 3 (PANDEY, 2018)

- ***Stride***: É o parâmetro que define qual será o deslocamento dado pelo filtro durante seu processo de convolução, ou seja, quantos *pixels* o filtro irá caminhar para o lado e para baixo a cada iteração a ser realizada. A Figura 3.7(a) e 3.7(b) ilustram como é feito essa caminhada. No primeiro exemplo, Figura 3.7(a), o valor do parâmetro *stride* é igual à 1. Isso significa que quando o filtro iniciar uma nova iteração, ele irá caminhar um *pixel* para o lado e ao chegar ao fim da linha, ele será movido para a primeira coluna novamente, porém para linha abaixo. Já no segundo caso, Figura 3.7(b), o qual o valor do *stride* é 2, portanto, o filtro irá "pular" 2 pixels para o lado ou dois para baixo toda vez que iniciar uma nova iteração com a imagem. Esse recurso torna possível à camadas convolucionais, reduzir o tamanho dos mapas de ativação gerados, fazendo com que a matriz de entrada para a próxima camada possua dimensões menores.



(a) Parâmetro Stride igual à 1



(b) Parâmetro Stride igual à 2

Figura 3.7: Exemplo de convolução variando o valor do parâmetro stride - Adaptada de Wikimedia (2020)

- ***Zero Padding***: Esse parâmetro é utilizado para armazenar o tamanho da matriz de entrada, através da inserção de valores 0 nas bordas. Sem esse parâmetro as dimensões da imagem de entrada seriam reduzidas rapidamente conforme as convoluções fossem sendo aplicadas, o que tornaria impossível o aprendizado da rede. Na Figura 3.8 é possível analisar que ao receber uma matriz de tamanho $4x4$ como entrada e aplicar o *zero padding*, a saída gerada possui tamanho similar ao observado na matriz de entrada.

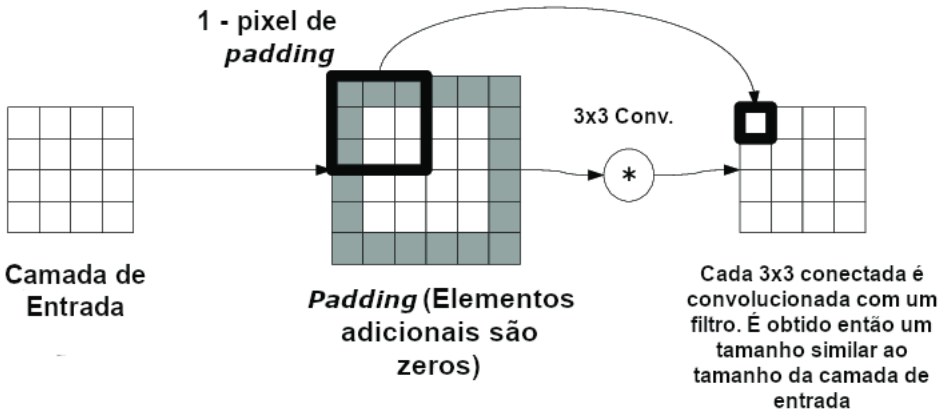


Figura 3.8: Exemplo da utilização do parâmetro *zero padding* - Adaptada de Tarasiuk, Tomczyk e Stasiak (2020)

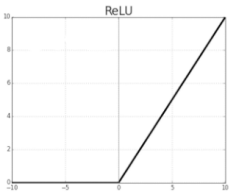
2. **Camada de Ativação** (*Activation Layers*): Após camadas convolucionais, geralmente é aplicado uma de ativação não linear como a *ReLu*[3], *Leaky ReLU*[4], *sigmoid*[5] ou *softmax*[6] para que a rede seja capaz de aprender qualquer tipo de funcionalidade. Elas definem se um neurônio deve ser ativado ou não, conforme as informações recebidas e seus comportamentos podem ser vistos nas Figuras 3.9(a), 3.9(b), 3.9(c) e 3.9(d).

[3]Unidade Linear Retificada. Por ser uma função não linear, torna possível copiar os erros para trás e ter várias camadas de neurônios ativados pela função ReLU. Possui a desvantagem de, caso a soma ponderada antes da função seja negativa, irá fazer com que a unidade produza zero e os pesos deixem de ser atualizados (COSTA, 2019) (ACADEMY, 2019).

[4]Unidade Linear Retificadora com Vazamento. Reduz o problema apresentado pela função ReLu. As alterações realizadas na função fazem com que o gradiente não seja mais 0, e os pesos continuem sendo atualizados (COSTA, 2019)

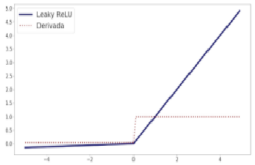
[5]Função suave e é continuamente diferenciável. Útil para gerar saídas não lineares, porém perde capacidade de ensino a rede quando o gradiente está se aproximando do valor zero (ACADEMY, 2019).

[6]Idealmente usada na camada de saída do classificador, onde realmente estamos tentando gerar as probabilidades para definir a classe de cada entrada (ACADEMY, 2019).



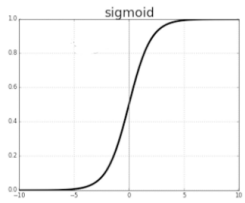
$$\boldsymbol{ReLU} = max(0, x)$$

(a) Função ReLU



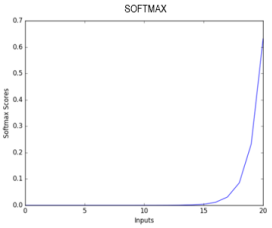
$$R_{leaky}(x) = \max(ax, x)$$

(b) Função Leaky ReLU



$$\phi(x) = \frac{1}{1 + e^{-x}}$$

(c) Função Sigmóide



$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

(d) Função *Softmax*

Figura 3.9: Funções de ativação utilizadas

3. **Camada de *Pooling*** (*Pooling Layer*): Outra forma de reduzir o tamanho de um volume de entrada e simplicar as informações geradas na camada anterior. Seu funcionamento é semelhante ao que é observado nas camadas convolucionais: é definido um tamanho para essa camada e ela vai "deslizar pela imagem", aplicando a função escolhida. Por conta disso, elas normalmente são posicionadas após as camadas convolucionais.

   Em relação as funções utilizadas, normalmente são dois tipos:

   - Máximo (Max): É escolhido o maior número dentre os valores analisados para ser utilizado no volume de saída. Normalmente são utilizadas na metade da rede, quando se quer reduzir o tamanho do volume de entrada.
   - Média (Avg): Calcula a média entre os valores analisados para ser utilizada no volume de saída. É posicionada ao fim da rede, quando se quer evitar utilizar camadas totalmente conectadas.

   Uma melhor compreensão desse processo pode ser obtida ao se analisar a Figura 3.10, que mostra uma camada de *Pooling* utilizando a função de máximo sendo aplicada em uma matriz.

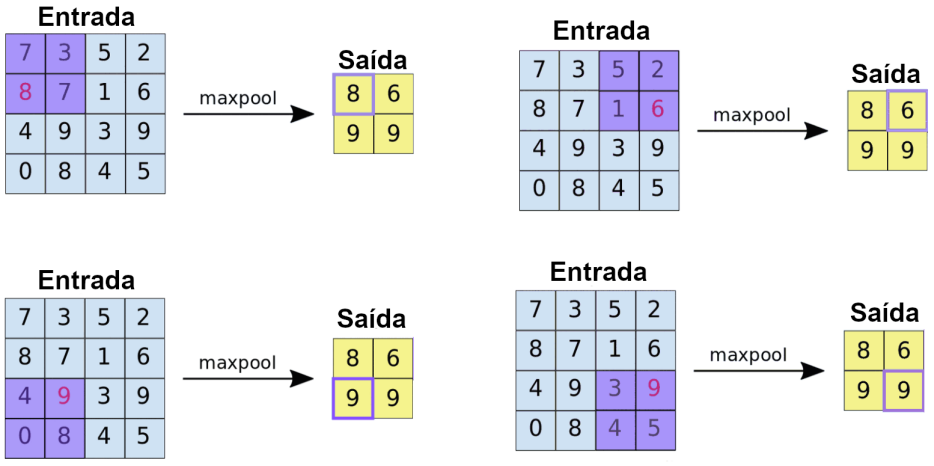


Figura 3.10: Exemplo de camada de *Pooling* utilizando a função de máximo - Adaptada de Alves (2018)

4. **Camada Totalmente Conectada** (*Fully Connected Layer*): Utilizadas normalmente ao fim da rede convolucional, essa camada terá N neurônios, sendo esse N o número de classes que o modelo sendo treinado possui. Ela realiza a finalização da classificação.

5. ***Dropout***: Essa camada modifica as conexões existentes pela rede, como forma de diminuir o problema de *overfitting*. Ela recebe um parâmetro que define a probabilidade de que uma área da rede neural seja desligada durante o processo de treinamento, fazendo com que a rede seja capaz de classificar corretamente, mesmo que alguns neurônios não estejam sendo ativados.

3.2.3 Exemplos de Redes Neurais Convolucionais

Uma rede neural convolucional utiliza diversas combinações das camadas acima citadas para poder compor a sua rede e resolver o problema no qual ele foi determinado. É possível ver na Figura 3.11, por exemplo, que para realizar a classificação daquela imagem em específico foram utilizadas 2 camadas convolucionais, 2 camadas de *pooling* e duas camadas totalmente conectadas para gerar a saída.

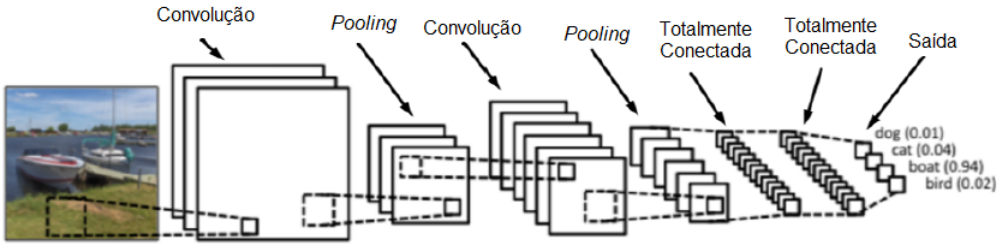


Figura 3.11: Arquitetura de uma Rede Neural Convolucional - Adaptada de Sampaio (2020)

Alguns pesquisadores, porém, criaram arquiteturas específicas de redes neurais convolucionais com os mais diversos objetivos. A seguir serão abordadas alguns exemplos dessas redes que foram desenvolvidas e foram bem sucedidas.

AlexNet

O trabalho desenvolvido por Krizhevsky, Sutskever e Hinton (2017) tinha como objetivo classificar a ImageNet (YANG et al., 2020), uma das maiores bases de dados conhecidas[7]. Essa rede possuía oito camadas, sendo cinco delas camadas convolucionais e três camadas totalmente conectadas. A saída da última camada totalmente conectada foi utilizada como entrada para uma camada de ativação que fez uso da função *softmax* para realizar a classificação entre as 1000 classes existentes. Todo esse processo foi realizado utilizando duas GPUs (*Graphics Processing Unit* - Unidade de Processamento Gráfico), o que facilitou o processamento da rede.

Sua organização pode ser vista na Figura 3.12, porém resumidamente é dada da seguinte forma:

- A primeira camada convolucional recebe uma imagem de entrada de tamanho $224 \times 224 \times 3$ com 96 filtros de tamanho $11 \times 11 \times 3$, com o parâmetro *stride* igual ao valor 4;
- A segunda camada recebe como entrada a saída da primeira camada, após ser tratada por uma camada de *Pooling*, com 256 filtros de tamanho $5 \times 5 \times 48$;
- A terceira possui 384 filtros de tamanho $3 \times 3 \times 256$. Não é conectada a nenhuma camada intermediária;

[7] A ImageNet possui mais de 14 milhões de imagens, divididas nas mais diversas classes, como objetos, frutos, plantas, animais, entre outros.

- A quarta possui 384 filtros de tamanho $3 \times 3 \times 192$;
- A última camada possui 256 filtros de tamanho $3 \times 3 \times 192$;
- As camadas totalmente conectadas possuem 4096 neurônios cada.

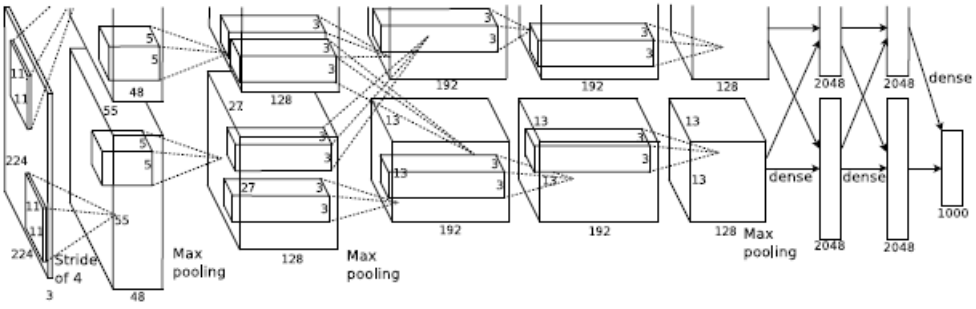


Figura 3.12: Arquitetura da AlexNet (KRIZHEVSKY; SUTSKEVER; HINTON, 2017)

GoogLeNet

O projeto desenvolvido por Szegedy et al. (2015) descreve a arquitetura de uma rede neural profunda utilizada na área de visão computacional, com vantagem de não demandar tanto poder computacional nem uso de memória.

Sua composição é baseada na utilização de módulos chamados *Inception*, desenvolvidos pelos mesmos autores. Sua principal função na rede *GoogLeNet* é justamente fazer com que seja possível treinar a rede sem que seja necessário tanto processamento da máquina utilizada. A arquitetura desse módulo pode ser vista na Figura 3.13 e seu funcionamento é mostrado na Figura 3.14. É possível ver que após a camada de *pooling*, é utilizado o módulo *Inception* para executar as convoluções e unir as informações em uma só camada concatenada.

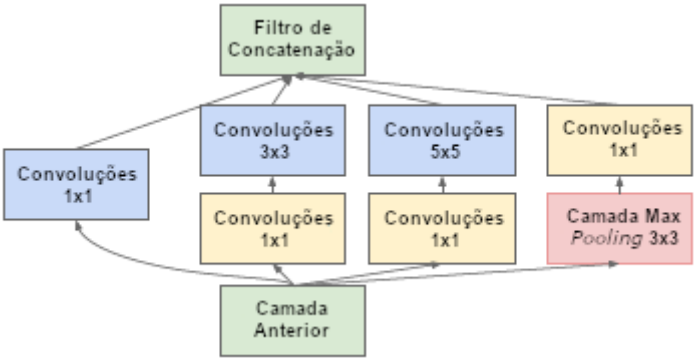


Figura 3.13: Módulo *Inception* - Adaptada de França e Soares (2016)

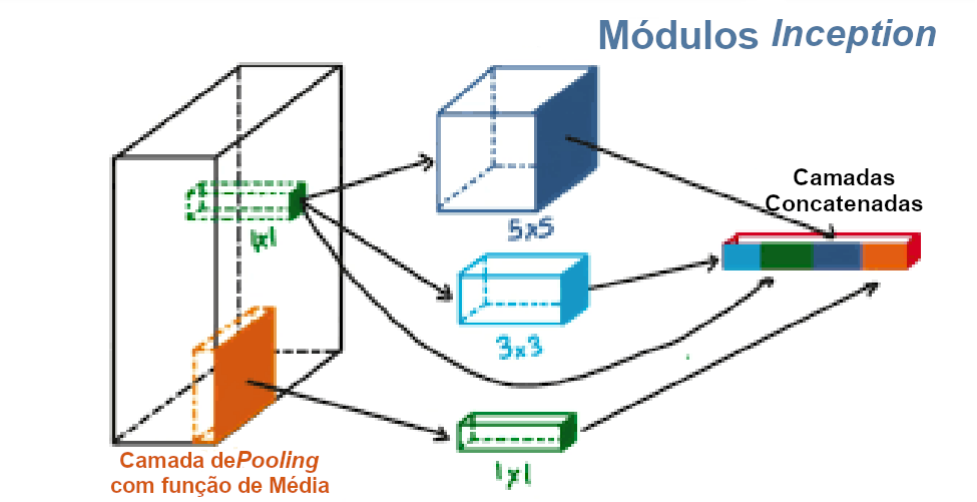


Figura 3.14: Utilização módulo *Inception* - Adaptada de França e Soares (2016)

Em sua arquitetura final são utilizadas 22 camadas de profundidade (27 considerando as camadas de *pooling*). Esse valor, porém, refere-se apenas à profundidade da rede, pois sua composição utiliza em torno de cem camadas. Além disso, são utilizados 9 módulos *Inception*, uma camada de *pooling* com função de média de tamanho 5×5 e parâmetro *stride* igual à 3. Também foram utilizadas: uma camada de *Dropout* com 70% de abandono[8], uma camada totalmente conectada com 1024 saídas, uma camada de ativação com a função ReLU e uma camada linear com a função *softmax* para realizar a classificação.

[8]Chance de "desligar" determinada área da rede neural.

A Figura 3.15 apresenta uma parte de como os módulos Inceptions são utilizados pela *GoogLeNet*. As camadas azuis são camadas convolucionais, as vermelhas as camadas de *pooling* e as verdes são outros tipos de camadas. Além disso, os quadrados vermelhos indicam a localização dos módulos *Inception*.

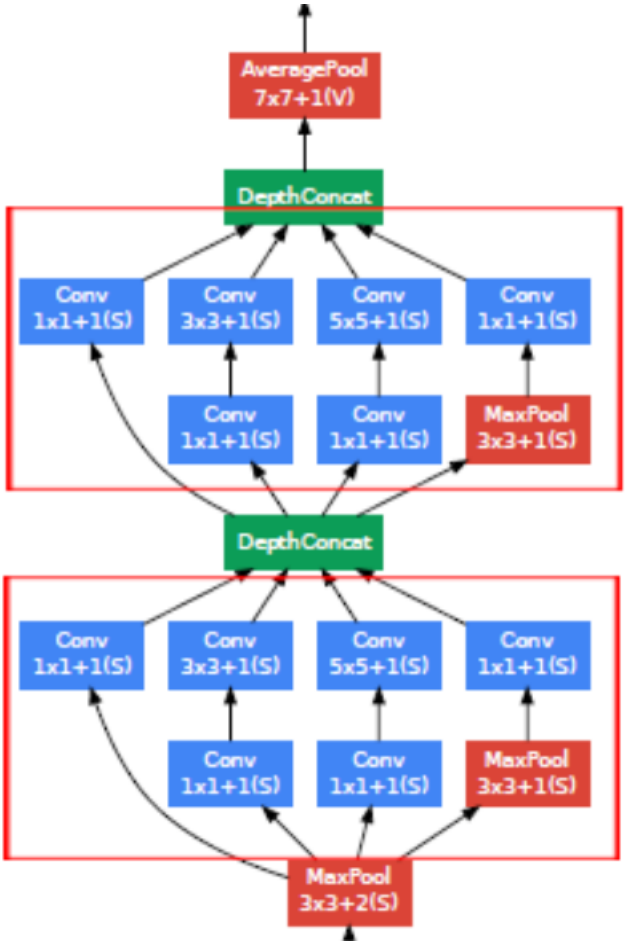


Figura 3.15: Fragmento da arquitetura da GoogLeNet - Adaptada de França e Soares (2016)

ResNet

Criada por He et al. (2016), sua organização é baseada nas células piramidais, existentes no córtex cerebral. A ideia dos autores foi introduzir a chamada "conexão de atalho de identidade" (*Identity shortcut connection*) que tem como objetivo "pular" algumas camadas na rede. Esses pulos foram sugeridos pois os autores notaram que em determinadas redes neurais, caso fosse aumentada a profundidade da rede, ela começava a se deteriorar, obtendo taxa de erros cada vez maiores. Este problema já havia sido abordado por He e Sun (2015) em uma publicação

anterior.

Sua estrutura é formada por camadas convolucionais de tamanho 3×3, com parâmetro *stride* igual à 2, camadas de *pooling* utilizando função de média global, uma camada totalmente conectada com mil entradas e uma camada de ativação utilizando a função *softmax* no fim. Um exemplo de organização de *ResNet* pode ser vista na Figura 3.16.

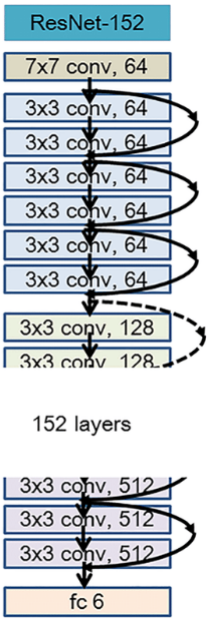


Figura 3.16: Arquitetura da ResNet - Adaptada de Han et al. (2018)

SqueezeNet

A rede proposta por Iandola et al. (2016) possuía como meta obter resultados semelhantes aos que a rede *AlexNet* conseguiu, utilizando, porém, metade dos parâmetros. Para que isso fosse possível, os autores abordaram 3 estratégias:

- Trocar todos as camadas convolucionais de tamanho 3×3 por de tamanho 1×1;
- Diminuir o número de canais de entrada por filtros 3×3, através da utilização de camadas "*squeeze*";
- Realizar a diminuição da dimensão das matrizes de entrada apenas em etapas mais avançadas, de forma que as camadas convolucionais fossem capazes de gerar maiores mapas de ativação.

Assim como na *GoogLeNet*, os autores também criaram módulos, chamados "*fire*", compostos por uma camada *Squeeze* (que possui apenas filtros convolucionais de tamanho 1×1) que alimenta uma camada Expandida formada por filtros de tamanho 1×1 e 3×3. Essa organização pode ser melhor visualizada na Figura 3.17. Ao fim desse módulo é utilizada uma camada de ativação com a função ReLu, além de ter sido utilizado o parâmetro *zero-padding* com valor de 1 pixel e uma camada *dropout* com abandono de 50% no módulo final.

Um exemplo de arquitetura da *SqueezeNet* pode ser visto na Figura 3.18.

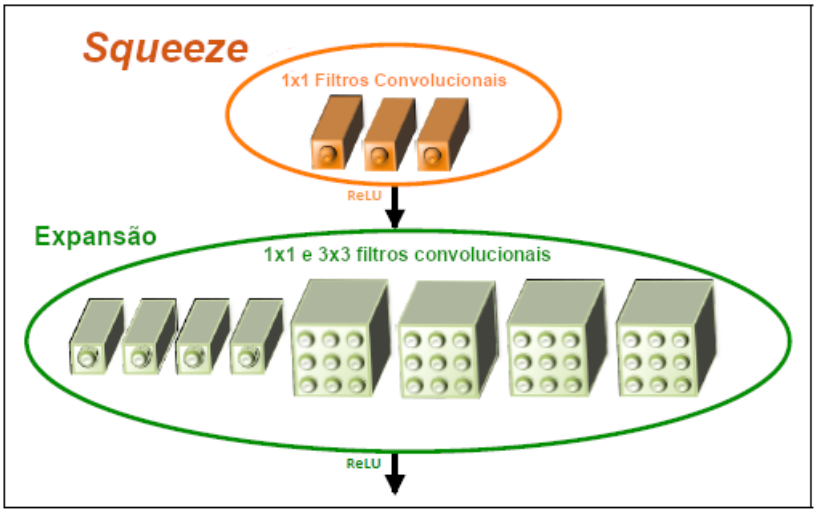


Figura 3.17: Módulos *Fire* - Adaptada de Iandola et al. (2016)

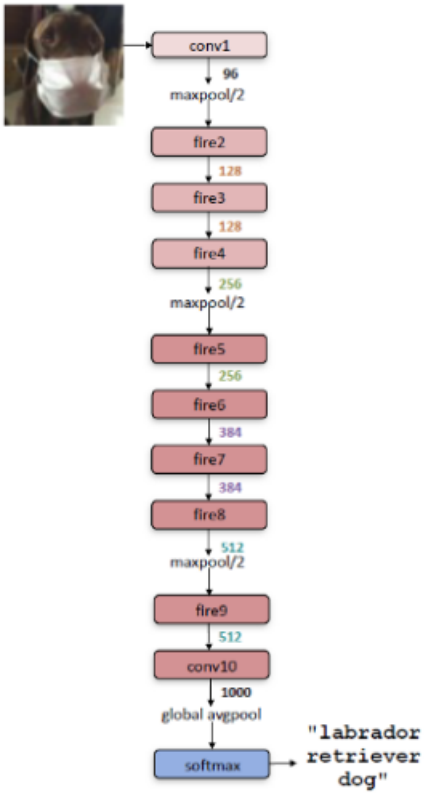


Figura 3.18: Arquitetura da *SqueezeNet* (IANDOLA et al., 2016)

YOLO (*You Only Look Once*)

YOLO (*You Only Look Once*) é um sistema de reconhecimento em tempo real, desenvolvido com o objetivo de detectar vários objetos em um único frame, além de gerar caixas delimitadoras (*Bounding Box Predictions*) que indicam em que região da imagem os objetos estão localizados e a qual classe eles pertencem (ORAC, 2020).

Sua primeira versão foi desenvolvida por Joseph Redmon, no ano de 2015, durante seu doutorado. Em uma palestra, foi apresentado o sistema detectando até 80 categorias diferentes simultaneamente, em uma taxa média de 30 FPS, com uma ótima precisão. O sistema ficou tão famoso que, até o ano de 2021, foram desenvolvidas outras quatro versões, nos anos 2016, 2018, 2020 e 2021 respectivamente (ALVES, 2020).

A versão YOLO-V4, última lançada durante o desenvolvimento deste trabalho, foi publicada por Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao em abril de 2020. Essa

versão se mostrou eficiente, de acordo com métricas do dataset MSCOCO[9] para a detecção de objetos (ALVES, 2020).

Seu funcionamento se difere das arquiteturas anteriormente abordadas pois, ele não somente irá realizar o treinamento da rede, como também a divisão da imagem em áreas de interesse, permitindo que o algoritmo também seja capaz de delimitar a área em que os objetos serão encontrados (TECHZIZOU, 2020).

Para que isso seja possível, a primeira etapa do algoritmo consiste em dividir a imagem em um *grid* de $S \times S$ células (Figura 3.19); cada uma dessas células pode analisar até cinco quadrados de distância ao seu redor com objetivo de verificar a existência de um objeto de interesse, retornando uma pontuação de confiança que indica se existe ou não um objeto naquela determinada área. Esta pontuação será utilizada para o desenho das caixas delimitadoras (TECHZIZOU, 2020) (ALVES, 2020).

Em paralelo, o algoritmo também irá realizar outra etapa, com objetivo de prever a qual classe cada um dos objetos detectados pertence. Nesta etapa não são analisadas células ao redor pois o objetivo é apenas a realização da classificação. Como resultado dessa análise, são retornados um conjunto de valores contendo a probabilidade de o objeto detectado pertencer a cada uma das classes (TECHZIZOU, 2020) (ALVES, 2020).

Ao fim dessas etapas, são combinados os valores de confiança e as probabilidades, resultando em uma pontuação final, que irá determinar a área em que o objeto detectado está localizado e a qual classe ele pertence (TECHZIZOU, 2020) (ALVES, 2020).

[9]MSCOCO, abreviação para *Microsoft Common Objects in Context*, é um grande conjunto de dados de reconhecimento/classificação de imagens, detecção de objetos, segmentação e legendagem. Mais informações podem ser encontradas em: https://cocodataset.org/#home

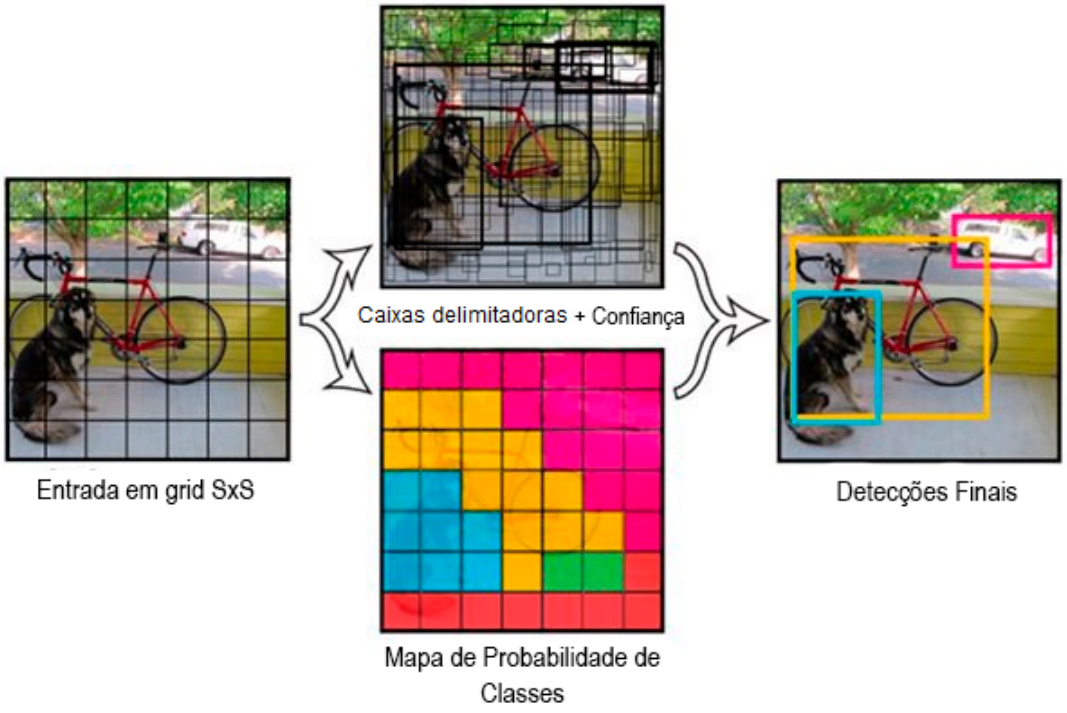


Figura 3.19: Funcionamento do YOLO - Adaptada de Techzizou (2020)

Para que todo esse processo de detecção de objeto e previsão de classes ocorra, os autores utilizaram a arquitetura padrão de um sistema de detecção de objetos de um estágio, que pode ser visto na Figura 3.20. Ele é composto por: *backbone*, que é previamente treinado utilizando a base de dados ImageNet e é onde são extraídas as características essenciais de cada classe e o *head* que irá realizar a predição de classes e desenhar as caixas delimitadores em cada imagem. Além desses dois componentes, os autores também inseriram algumas camadas entre eles, com objetivo de coletar mapas de características e transportá-los nos sentidos *top-down* e *botton-up*. Esse grupo de camadas extras é chamado *neck* (BOCHKOVSKIY; WANG; LIAO, 2020) (RUGERY, 2020).

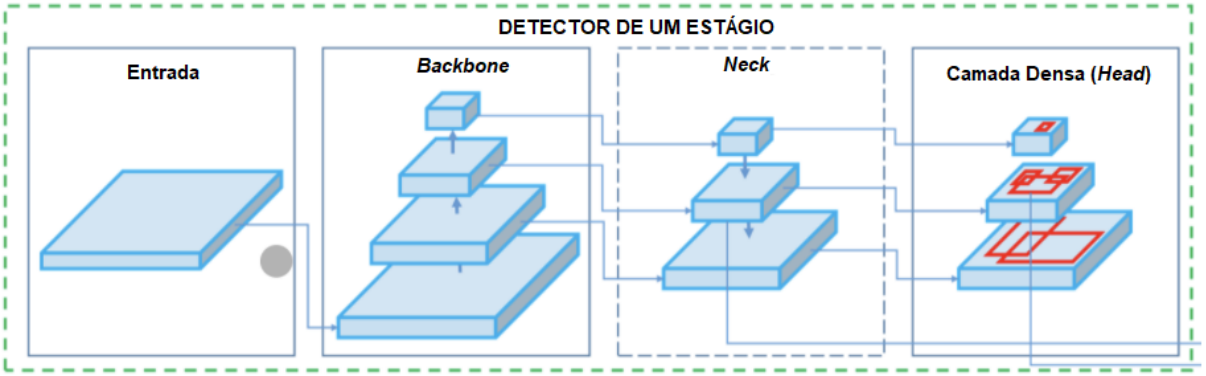


Figura 3.20: Arquitetura de detector de objetos com um estágio - Adaptada de Rugery (2020)

Para cada uma dessas etapas, Bochkovskiy, Wang e Liao (2020) apresentam em seu artigo arquiteturas de redes neurais convolucionais e métodos que podem ser utilizados para melhorar o desempenho do treinamento da rede. Esses "melhorias" são divididas em novas etapas pelo autor. O *backbone*, por exemplo, é composto por três etapas (BOCHKOVSKIY; WANG; LIAO, 2020) (RUGERY, 2020):

- *Bag of freebies*[10]: Métodos que aumentam o custo do treinamento ou mudam a estratégia enquanto deixam o custo de inferência baixo. Alguns exemplos de métodos são:

    1. **Aumento de dados**: aumenta a variabilidade de uma imagem para melhorar a generalização do modelo treinado;
    2. **Distorção fotométrica**: cria novas versões de uma imagem, ajustando brilho, saturação, matiz, contraste e ruído;
    3. **Distorção Geométrica**: Modifica a imagem utilizando operações de rotação, inversão, mudança de escala ou corte;
    4. ***MixUp***: Combina duas imagens em uma nova utilizando interpolação linear com média ponderada;
    5. ***CutMix***: Recorta um pedaço da imagem e insere esse fragmento no meio de outra imagem, melhorando a robustez do modelo contra corrupções de entrada e seus desempenhos em detecções desbalanceadas;

[10] Saco de Brindes.

6. **Perda Focal**: A perda focal é usada para resolver o problema do desequilíbrio de classes, aumentando o "foco" nas classes que possuem menos instâncias. Ela calcula a perda de um modelo, reduzindo a penalização aplicada aos erros;
7. **Suavização de Rótulos**: Ajusta o limite superior alvo da previsão para um valor inferior, para casos em que o modelo esteja memorizando os dados ao invés de aprender, reduzindo assim o *overfitting*. Caso a precisão de uma classe esteja chegando em 100%, por exemplo, ela é reduzida para 90% de forma que o modelo continue aprendendo características;
8. **Perda *IoU***: Utilizada para verificar a diferença entre os quadrados delimitadores criados pelo algoritmo e seus valores reais.

- *Bag of specials*: Métodos que aumentam o custo de inferência, mas podem melhorar significativamente a precisão da detecção de objetos. Um exemplo de função que realiza isso é a função de ativação *Mish* (função de ativação neural não monotônica auto regularizada) (MISRA, 2019), que além de melhorar a expressividade e o fluxo de informação, evita a saturação do treinamento que ocorre em casos de gradientes próximos de zero.

- *CSPDarknet53*: Rede neural convolucional baseada na rede *DenseNet* (Redes Convolucionais Densamente Conectadas) (HUANG et al., 2017) que utiliza a entrada anterior e a concatena com a entrada atual antes de passar para a camada densa. Na CSP, a entrada é dividida em duas partes, na qual uma parte passará por uma camada densa e a outra será conectada no final com o resultado da camada densa, resultando em diferentes camadas densas aprendendo repetidamente as informações de gradiente copiadas.

Já para o estágio *Neck*, segundo Bochkovskiy, Wang e Liao (2020), os métodos que podem ser utilizados são:

- Camada SPP (*Spatial Pyramid Pooling*): Utilizada para evitar que as variações que a imagem sofre em seu tamanho afete o desempenho da rede. Esse tipo de camada é capaz de detectar formas geométricas circulares, produzindo um mapa de características que destaque essas formas e mantenha a localização da mesma. Dessa forma, mesmo que a imagem vá sendo reduzida ao decorrer da rede, ainda será possível saber onde estão localizados os objetos;

- PaNet: Esse tipo de camada é utilizada para melhorar a propagação de informações entre toda a rede. Na versão utilizada no YOLO-V4, são concatenados vetores contendo a entrada da camada e o vetor da camada anterior.

Por fim, na etapa *head* Bochkovskiy, Wang e Liao (2020) apresentam os seguintes métodos, sendo dois deles, os mesmo apresentados na etapa do *backbone*:

- *Bag of freebies (BoF)*:

    1. **Perda *CIoU***: Nova versão da perda IoU, que inclui duas modificações. A primeira é a melhoria do cálculo da distância do ponto central e a segunda a melhoria no cálculo da proporção da imagem. Ambos irão calcular a diferença entre o resultado obtido durante a fase de predição e os valores verdadeiros;
    2. ***CmBN (Cross mini Batch Normalization)***: A Camada *Batch Normalization* (Normalização em Lote) é utilizada para aplicar uma transformação que mantém os valores de saída próximos ao valor 0 e o desvio padrão próximo à 1, fazendo com que a rede se torne mais rápida e estável (KERAS, 2021). Porém, esse tipo de camada tem a performance reduzida quando a imagem começa a ter suas dimensões reduzidas. Para resolver esse problema, foi criada a Normalização Cruzada de Mini Lote, que melhora a qualidade da estimativa das imagens mesmo que elas possuam tamanho reduzido;
    3. **Regularização *DropBlock***: Camadas de *Dropout* podem não funcionar bem em camadas convolucionais pois elas descartam recursos aleatórios e esse tipo de camada utiliza os recursos de forma espacialmente correlatos. Na camada *DropBlock* esses recursos são descartados em blocos, fazendo com que a rede procure por evidências em outros neurônios para ajustar os dados;
    4. **Aumento de dados do mosaico**: Combina quatro imagens do treinamento em determinadas proporções, permitindo que o modelo aprenda a reconhecer objetos em escala menor. Além disso, o incentiva a localizar diferentes tipos de imagens em um mesmo quadro;
    5. **Treinamento de autodefesa**: Técnica de aumento de dados que consiste em dois estágios: no primeiro a rede neural sofre um "ataque", que altera a imagem original

de entrada, de forma que é criada a ilusão que não há objeto na imagem. No segundo momento, a rede é apresentada para a mesma imagem, mas sem que ela seja modicada;

- *Bag of specials (BoS)*:

    1. ***SAM-block***: Camada utilizada para chamar atenção das características mais importantes de cada classe. Ela aplica duas transformações separadas ao mapa de características, a primeira utilizando uma camada convolucional, e a segunda usando uma camada de *Max Pooling* e uma de *Avg Pooling*, que geram dois conjuntos de mapa de recursos. Esses mapas então são concatenaos e passados para uma nova camada de convolução, seguido por uma função sigmóide que destaca onde os recursos mais importantes estão. Na versão utilizada no *YOLO-V4*, não são utilizadas as camadas de *pooling*;

- *DIoU-NMS*: NMS (Supressão não máxima) são utilizadas para remover caixas que representem o mesmo objeto, mantendo a mais precisa em comparação com o quadrado delimitador real.

Uma melhor representação da arquitetura utilizada pelo sistema YOLO-V4 pode ser vista na Figura 3.21

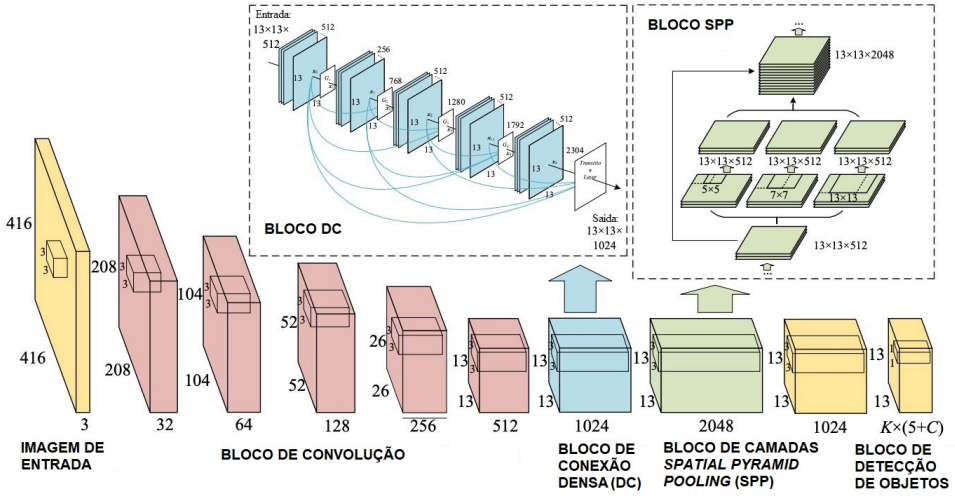


Figura 3.21: Arquitetura de camadas YOLO-V4 - Adaptada de Shah (2020)

Além dessas camadas específicas para o melhoramento do desempenho da rede, o sistema YOLO também apresenta outros quatro tipos de camadas que são utilizadas em sua estrutura. São elas:

- Camada de Rota (*route*): Conceito hierárquico proposto no YOLO. Sua função é utilizar os valores de convoluções de algum ponto da rede para a camada em que se está operando (PROGRAMMERSOUGHT, 2019). Segundo Bochkovskiy em seu fórum no Github ®[11], o funcionamento dessa camada ocorre da seguinte forma:

    1. Caso seja passado um valor único, será utilizado como entrada o resultado da camada com índice igual ao valor da camada atual menos o valor passado como parâmetro, sem nenhum processamento. Utilizando o valor -2 como parâmetro, por exemplo e esteja sendo processada a camada 26, será feito o cálculo $26 - 2$, que resulta no valor 24. Será buscado então o valor da saída da camada 24 para ser atribuído à camada 26.
    2. Caso sejam passados dois valores como parâmetro, será feito a subtração entre o valor da camada atual e ambos os parâmetros passados e os resultados serão concatenados. Por exemplo, caso sejam passados os valores -1 e -3 e se esteja processando a camada 26, o resultado dos cálculos serão 25 e 23, então em seguida será buscado na rede quais os valores de saída dessas duas camadas. Utilizando o exemplo dado pelo autor, com os valores das camadas 23 e 25 iguais à $13 \times 13 \times 1024$ e $13 \times 13 \times 2048$, esses números serão somados, resultando no valor $13 \times 13 \times 3072$.

- Camada de *Upsampling*: Uma camada de *Upsampling* funciona de forma oposta à uma camada de *pooling*, duplicando as dimensões das imagens recebidas (BOCHKOVSKIY, 2020a) (ZEILER; FERGUS, 2014).

- Camada YOLO: Camada responsável pela detecção realizada pela rede nas versões YOLO-V3/V4. Contém parâmetros como:

    1. **Âncoras *(Anchors)***: Tamanhos iniciais dos quadrados delimitadores. Poderá ser ajustado posteriormente;

[11] https://github.com/AlexeyAB/darknet/issues/279#issuecomment-397248821

2. ***Mask***: índice das âncoras utilizas nessa camada;
3. ***Class*** (Classes): Número de classes do problema;
4. ***Num***: Número total de âncoras utilizadas;
5. ***Jitter***: Corta e redimensiona imagens aleatoriamente, mudando a proporção de aspecto de $x(1 - 2 * jitter)$ para $x(1 + 2 * jitter)$ (o parâmetro de aumento de dados é usado apenas a partir da última camada);
6. ***scale_x_y***: Utilizado para eliminar a sensibilidade do grid, melhorando as previsões em torno dos limites das regiões da caixa de âncora;
7. ***cls_normalizer***: Normaliza o *Objectness*-delta [12];
8. ***IoU_normalizer***: Normaliza o IoU-delta [13];
9. ***IoU_loss***: Métrica utilizada, pode ser *mse* (*Mean Squared Error* - Erro Quadrado Médio), *giou* (*Generalized Intersection over Union* - Intercessão Generalizada antes de União), *DioU* (*Distance-IoU* - Distância-IoU) ou *CIoU* (*Complete-IoU* - Completo-IoU) (PROGRAMMERSOUGHT, 2021);
10. ***ignore_thresh***: Utilizado em conjunto com a camada NMS (*Non-maximum Suppression* - Supressão não máxima), verifica se o valor de perda IoU calculado é maior que o valor desse parâmetro. Caso seja, irá manter a detecção duplicada;
11. ***truth_thresh***: Utilizado em conjunto com a camada NMS. Verifica se o valor de perda IoU é maior que esse parâmetro e caso seja irá ajustar as detecções duplicantes;
12. ***Random***: Redimensiona a rede aleatoriamente a cada 10 iterações;
13. ***nms_kind***: Tipo de métrica utilizado na camada NMS. Pode ter valores *greedynms* ou *diounms*;
14. ***beta_nms***: Tipo de *threshold* utilizado quando o parâmetro *nms_kind* é igual à *greedynms*;

[12] A pontuação de objetividade (*Objectness score*) é definida para medir o quão bem o detector identifica os locais e classes de objetos durante a navegação. O *Objectness*-delta representa a diferença entre os valores reais e o que foi calculado pela rede (CHOI et al., 2019)

[13] IoU (Intercessão antes de União (Intersection over Union)) é uma métrica utilizada para calcular a precisão média que o desenho da Caixa Delimitadora obteve. O IoU-delta é a diferença entre o valor original e o que foi calculado (HOFESMANN, 2020)

- Camada NET: Camada responsável por parâmetros gerais da rede. Podem ser eles (BOCHKOVSKIY, 2020b):

    1. ***Batch***: Número de imagens que serão processadas por lote;
    2. ***Subdivisions***: Número de mini lotes em um lote. A GPU irá processar todos os mini lotes dentro de um lote para então atualizar os pesos da rede;
    3. ***Width***: Largura das imagens de entrada;
    4. ***Height***: Altura das imagens de entrada;
    5. ***Channels***: Número de canais da rede;
    6. ***Momentum***: Otimizador utilizado para definir o quanto o histórico irá afetar a posterior mudança de pesos;
    7. ***Decay***: Otimizador que define uma atualização mais fraca dos pesos de forma que elimine o desequilíbrio no conjunto de dados;
    8. ***Angle***: Gira aleatoriamente as imagens durante o treinamento;
    9. ***Saturation***: Muda aleatoriamente a saturação das imagens durante o treinamento;
    10. ***Exposure***: Muda aleatoriamente o brilho das imagens durante o treinamento;
    11. ***Hue***: Muda aleatoriamente a matiz das imagens durante o treinamento;
    12. ***Learning_rate***: Taxa de aprendizado inicial;
    13. ***Burn_in***: *Burnin* inicial que será processado para as primeiras 1000 iterações. Esse parâmetro define quantas iterações serão descartadas no processo inicial do treinamento;
    14. ***Max_batches***: Define o número de iterações que serão realizas durante o treinamento;
    15. ***Policy***: Define a política de mudança da taxa de aprendizado. Pode ter os valores *constant*, *sgdr*, *steps*, *step*, *sig*, *exp*, *poly*, *random*;
    16. ***Steps***: Caso o parâmetro *Policy* seja igual à *steps*, o número de iterações será multiplicado pelo parâmetro *Scales*;
    17. ***Scales***: Irá definir a escala que multiplicará o número de iterações.

Capítulo 4

Materiais

4.1 Materiais Utilizados

Esta seção tem como objetivo apresentar brevemente os recursos que foram utilizados para o desenvolvimento deste trabalho.

4.1.1 Raspberry PI

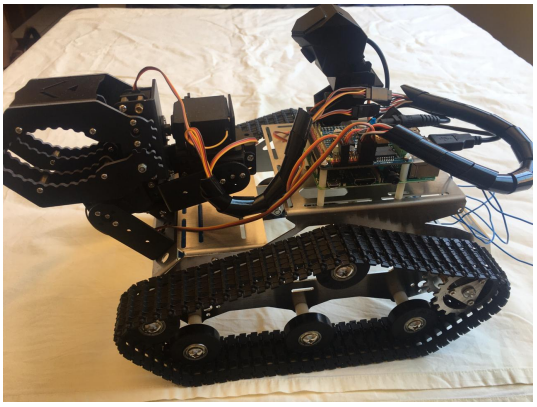
Raspberry Pi é uma série de computadores de baixo custo, pequeno tamanho e placa única, desenvolvido pela *Raspberry Pi Foundation* no Reino Unido, com objetivo de promover o ensino de informática básica nas escolas. Sua popularidade, no entanto, se expandiu além do propósito pretendido, alcançando também o mercado de sistemas embarcados e pesquisa científica. Sua arquitetura inclui processador(es), Memória de Acesso Aleatório (*Random Access Memory* - RAM), entradas USB, HDMI, áudio e vídeo composto, para câmera e telas LCD e uma série de pinos GPIO (*General Purpose Input/Output* - Entrada/saída de propósito geral), (GOGONI, 2019). No entanto, por se tratarem de dispositivos de placa única, eles não são modulares e seu *hardware* não pode ser melhorado pois está integrado na própria placa (PAJANKAR, 2015).

Algumas linguagens de programação acompanham o Raspberry como Scratch, Python e Pygames. Suas utilizações são focadas em, respectivamente: ensinar lógica de programação de forma lúdica, desenvolvimento de projetos mais avançados como robôs e *clusters* e facilitar a criação de jogos. Além disso, linguagens como C, Ruby, Java e Pearl também podem ser utilizadas.

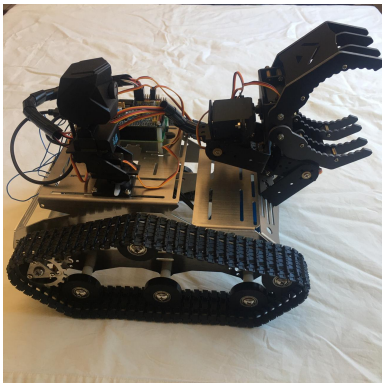
O sistema operacional (SO) que o Raspberry utiliza é o Raspbian, uma variação de uma

distribuição Debian do Linux. Essa escolha se deu pois sistemas Linux possuem código aberto e são gratuitos, o que trás vantagens como tornar a placa mais barata, induzir à programação e melhorias no SO pelos colaboradores, além de incitar os alunos a terem um maior envolvimento com *hardware* e *software*. Apesar disto, ele também pode ser utilizado em outros SOs, tais como Arch Linux, Xbian e QtonPi (EBERMAM et al., 2017).

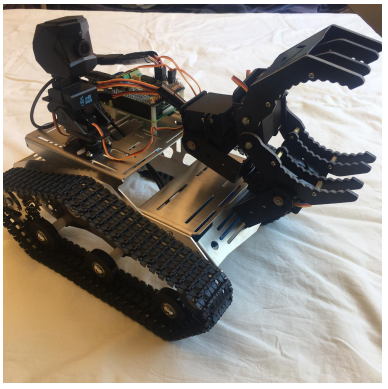
Neste projeto foi utilizada uma placa Raspberry Pi modelo 3B integrada a um robô wi-fi com chassi TH *Robot Tank*. Ele é produzido pela *Xiao R Geek Techonology* e possui uma câmera USB (*Universal Serial Bus* - Porta Serial Universal) em um suporte com dois graus de liberdade, além de um braço manipulador com 4 graus de liberdade. O modelo do robô pode ser visto na Figuras 4.1(a), 4.1(b) e 4.1(c).



(a) Exemplo 1



(b) Exemplo 2



(c) Exemplo 3

Figura 4.1: Modelo de robô utilizado

4.1.2 OpenCV

OpenCV é uma biblioteca *open source* utilizada para o desenvolvimento de aplicações que utilizam Visão Computacional, criada pela Intel no ano de 2000. Ela está disponível para a maioria das plataformas, incluindo Linux, macOS, Windows, Android e iOS. Seu objetivo é prover uma biblioteca de visão computacional de uso simples, auxiliando pessoas a desenvolverem aplicações com visão computacional rapidamente (KAEHLER; BRADSKI, 2016). Ela possui uma comunidade de mais de 47 mil usuários e pode ser utilizada em diversas linguagens como C++, Python, Ruby e Java.

Suas centenas de algoritmos disponíveis tornam possíveis o desenvolvimento de aplicações voltadas para as mais diversas áreas da visão, tais como: inspeção de produtos de fábrica, imagens médicas, segurança, interface de usuário, calibração de câmera, visão estéreo e robótica.

Outro ponto forte da biblioteca OpenCV é o seu módulo de funções para aplicações que utilizam algoritmos de Aprendizagem de Máquina. Elas são divididas em cinco grupos e são eles (MARENGONI; STRINGHINI, 2009):

1. Processamento de imagens;
2. Análise estrutural;
3. Análise de movimento e rastreamento de objetos;
4. Reconhecimento de padrões;
5. Calibração de câmera e reconstrução 3D.

4.1.3 Python

Python é uma linguagem de programação de alto nível, interpretada, orientada a objetos, de tipagem dinâmica e forte, de código livre, publicada no ano de 1991, por Guido van Rossum. Sendo uma das linguagens mais utilizadas entre programadores segundo a IEEE *Spectrum* no ano de 2018 (CASS, 2018), ela possui mais de 125.000 bibliotecas criadas por terceiros, o que a torna popular em diversos campos na programação, com destaque para a área de ciência de dados e inteligência artificial (ITMÍDIA, 2019).

4.1.4 Tensor Flow

Tensor Flow é um sistema de aprendizado para máquinas que opera em grande escala e em ambientes heterogêneos. Ele utiliza gráficos de fluxo de dados para representar computação, estado compartilhado e operações que alteram esse estado, mapeando os nós desse gráficos por várias máquinas, em vários dispositivos computacionais, incluindo GPUs e os chamados TPUs (*Tensor Processing Units* - Unidade de Processamento de Tensor). Isso faz com que a arquitetura seja flexível ao programador, que pode utilizar otimizações e algortimo de treinamento e inferências de redes neurais que ele disponibiliza (ABADI et al., 2016).

4.2 Ambientes Utilizados

O treinamento dos protótipos criados para esse projeto foram executados no ambiente Google Colaboratory[1], que fornece GPU para acelerar o processo de treinamento das redes neurais. As configurações utilizadas no ambiente para a execução dos modelos foram as padrões gratuitas do Colab, consistindo em:

- Processador: Intel(R) Xeon(R), 2.30GHz, ;
- RAM: 12GB;
- Espaço Disponíve: 25GB;
- GPU: Nvidia K80, 12GB.

Já para a realização dos testes foram utilizadas duas máquinas diferentes. A primeira era um computador pessoal de mesa, e nele foram realizados os testes iniciais, verificando o desempenho dos protótipos criados fora do ambiente de treinamento, e se eles iriam necessitar de novos treinamentos para melhorar sua performance. Suas configurações eram:

- Sistema Operacional: Microsoft Windows 10 Pro x64 Compilação 18362.719;
- Processador: Intel Core i5-5200 CPU 2.20GHz, 2 Núcleos e 4 processadores lógicos;
- RAM: 16 GB;

[1] https://colab.research.google.com

- GPU: NVIDIA GeForce 920M.

A outra máquina utilizada para a realização dos testes foi a placa Raspberry Pi, modelo 3B, embutida no robô. Suas configurações consistiam em:

- Sistema Operacional: Raspberry PI OS 32-bits. *Released*: 2021-03-04;
- Processador: Quad Core 1.2GHz Broadcom BCM2837 64bit CPU;
- RAM: 1GB.

4.3 Bases de Dados

Com objetivo de abranger objetos existentes no ambiente rural, criou-se uma base de dados contendo seis classes: animais, flores, gotejador, veículos, pessoas e plantas. Elas foram escolhidas após um estudo sobre quais objetos eram mais encontrados em ambiente agrícola local.

Para construí-la, foram reunidas imagens de bases de dados já existentes no site Kaggle[2] e em alguns casos necessitou-se buscar imagens em outras fontes. Maiores informações referentes à composição das bases podem ser vistas à seguir:

- **Animais**: Foram utilizadas duas bases de imagens para o desenvolvimento dessa classe. A base *256-Objects*, conta com mais de 30 mil imagens, divididas entre 256 classes, mas para esse projeto foram utilizadas apenas as classes *dog*, *horse* e *duck*. Ela pode ser encontrada no link *https://www.kaggle.com/rohitupadhya/256objects*. A segunda base utilizada foi a base *Animal-10*, composta por mais de 26 mil imagens, divididas em 10 classes. Foram utilizadas as classes *cat*, *chiken*, *cow*, *dog*, *horse* e *sheep* e ela está disponível no site *https://www.kaggle.com/viratkothari/animal10*;
- **Flores**: A Base *Hackathon Blossom (Flower Classification)* possui 7577 imagens divididas em 102 classes. Utilizou-se todas as classes dessa base e ela está disponível no link *https://www.kaggle.com/spaics/hackathon-blossom-flower-classification*;

[2]https://www.kaggle.com

- **Gotejador**: Para a construção dessa classe, foram utilizadas imagens disponíveis no Google, utilizando a palavra chave “gotejador”;

- **Veículos**: Foi utilizada a base *Vehicle Data Set*, selecionando apenas as classes *Car*, *Truck*, *Van*, *Taxi*, *Ambulance* e *Bus*. Ela pode ser encontrada em: *https://www.kaggle.com/iamsandeepprasad/vehicle-data-set*. Além disso, se utilizou a classe *bulldozer* da base *256-Objects* para que fosse possível abranger veículos de maior porte;

- **Pessoas**: A base *1 Million Fake Faces* possui 160 mil imagens de rostos de pessoas falsas. Ela está disponível em *https://www.kaggle.com/tunguz/1-million-fake-faces*. Também foram utilizadas imagens disponíveis no Google (utilizando as palavras chaves “fotos paparazzi famosos”) para poder completar a base de dados, pois não foi possível encontrar base de dados que representassem o corpo inteiro de pessoas;

- **Plantas**: Inicialmente utilizou-se as bases *V2 Plant Seedlings Dataset* e *Agriculture crop images* para fazerem parte da classe planta. A primeira base contém 5539 imagens divididas em 12 classes, e está disponível no link *https://www.kaggle.com/vbookshelf/v2-plant-seedlings-dataset*. Já a segunda possui 1107 imagens divididas em 5 classes e pode ser encontrada em *https://www.kaggle.com/aman2000jaiswal/agriculture-crop-images*. Notou-se, porém, que utilizar essas bases apenas não seriam suficientes, pois elas não eram capazes de representar tão bem a vegetação brasileira. Por este motivo, foi realizada uma viista ao Núcleo Experimental de Engenharia Agrícola (NEEA) da UNIOESTE, para realizar a coleta de imagens de outros tipos de plantações como soja e milho.

O processo de preparação da base de dados ocorreu da seguinte forma: primeiro foram reunidas todas instâncias das bases utilizadas, separando-as por suas classes e em seguida, foram escolhidas 170 instâncias para as classes planta, veículo, animal, flor e pessoa e 150 para a classe gotejador, que por ser mais característica, não necessitava de tantas imagens. A base final, composta por um total de mil imagens, foi dividida em 70% para treinamento, 15% para validação e 15% para testes.

Capítulo 5

Implementação

5.1 Experimentos Realizados

Foram desenvolvidos dois modelos de redes neurais convolucionais para a execução do projeto. O primeiro protótipo foi construído baseando-se na estrutura da rede neural *AlexNet*. Por se tratar de um modelo de rede que foi construída com objetivo de realizar classificação da ImageNet, uma base de dados que contém diversos tipos de objetos, optou-se por sua utilização como modelo inicial. Este modelo, porém, mostrou-se não tão efetivo, já que gerou arquivos muito grandes, que seriam difíceis para serem transferidos para o robô, que possui pouca memória. Por conta disso, optou-se por desenvenvolver um segundo modelo de rede neural convolucional, que tivesse como foco a utilização em ambientes embarcados, com baixo poder de processamento. Este modelo utilizou o sistema YOLO-V4 *Tiny*, uma versão que utiliza menos parâmetros que o YOLO convencional.

5.1.1 Protótipo I

O primeiro protótipo criado baseou-se na arquitetura *AlexNet*, utilizando o modelo desenvolvido por Alake (2020), redimensionando as imagens para o tamanho igual à 200×200. Para o treinamento, foram utilizadas 1.000 épocas, com 7 iterações cada, totalizando 7.000 iterações. Esse número de épocas foi escolhido após testes preliminares que verificaram que caso fossem utilizados valores maiores, a rede entraria em processo de *overfitting*. Além disso, optou-se por usar uma etapa de validação após cada iteração.

Os parâmetros utilizados foram taxa de aprendizado de $0,00261$ e *Batch Size*[1] igual à 100.

[1] Refere-se ao número de exemplos de treinamento utilizados em cada iteração (ACADEMY, 2019)

Também foram utilizados parâmetros disponíveis no *TensorFlow* que possuem o objetivo de melhorar a precisão do modelo criado. Foram eles:

- **Métricas de perda**: Em problemas de Redes Neurais profundas, funções de custo e perda são usadas para otimizar o modelo durante seu treinamento, visando diminuir o valor de perda, pois, quanto menos o modelo errar, melhor ele é (KOECH, 2020).

  A Entropia Cruzada (*Crossentropy*) é uma das possíveis métricas utilizadas para descrever a perda entre duas distribuições de probabilidades (ACADEMY, 2019). Mais informações podem ser encontradas em Academy (2019) e em Boer et al. (2005).

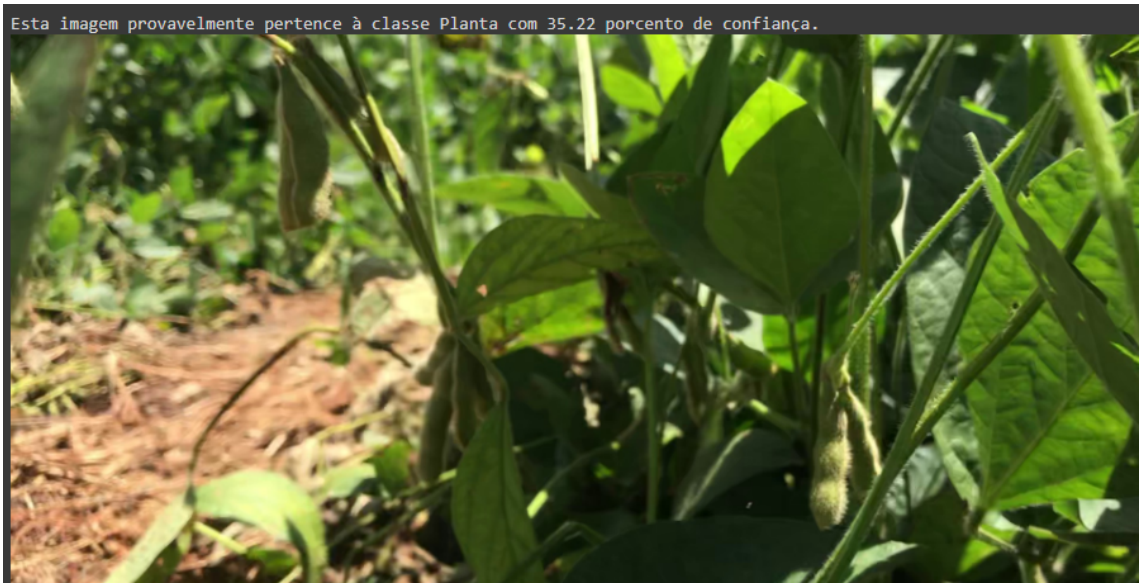
  No modelo criado, foi utilizada a métrica *Sparse Categorical Crossentropy*, variação da Entropia Cruzada para modelos que estão sendo treinados com duas ou mais classes (TENSORFLOW, 2021).

- **Métodos de otimização**: Taqi et al. (2018) afirmam que, para que uma rede neural possa ser capaz de aprender, é necessário que os pesos entre as camadas possam ser alterados, visando melhorar o desempenho da rede. Para isso, é necessário analisar continuamente o resultado que é obtido do processo de classificação e o que era esperado.

  O otimizador utilizado neste modelo foi o Adam, que é baseado no gradiente de primeira ordem de funções objetivas estocáticas, baseado em estimativas adaptativas de momentos de ordem inferior. Mais detalhes referentes ao seu funcionamento podem ser encontrados em Kingma e Ba (2014).

O código completo pode ser encontrado no Google Colaboratory[2] e a estrutura da rede desenvolvida pode ser vista na Tabela 5.1. Além disso, a Figura 5.1 mostra a classificação realizada pela rede em relação à duas imagens.

[2]https://colab.research.google.com/drive/1vqYAJEXrIi3dtMOgSA4mqeCQwO6lpyBU?usp=sharing



(a) Exemplo I Classificação *AlexNet*

(b) Exemplo II Classificação *AlexNet*

Figura 5.1: Exemplos de classificação utilizando o Modelo I

Tabela 5.1: Arquitetura da rede *AlexNet* desenvolvida

MODELO I - ALEXNET	
TIPO DE CAMADA	PARÂMETROS UTILIZADOS
Camada de Entrada	- Imagem com dimensões $200 \times 200 \times 3$
1ª Camada Convolucional	- Camada de convolução com 96 filtros, tamanho do *kernel* 11×11, parâmetro *stride* igual à 4; - Camada de Normalização; - Camada de Ativação utilizando função ReLu; - Camada de *MaxPooling*, com parâmetro *stride* e *pool size* igual à 2.
2ª Camada Convolucional	- Camada de convolução com 256 filtros, tamanho do *kernel* 5×5, parâmetro *stride* igual à 1; - Camada de Normalização; - Camada de Ativação utilizando função ReLu.
3ª Camada Convolucional	- Camada de convolução com 384 filtros, tamanho do *kernel* 3×3, parâmetro *stride* igual à 1; - Camada de Normalização; - Camada de Ativação utilizando função ReLu.
4ª Camada Convolucional	- Camada de convolução com 384 filtros, tamanho do *kernel* 3×3, parâmetro *stride* igual à 1; - Camada de Normalização; - Camada de Ativação utilizando função ReLu.
5ª Camada Convolucional	- Camada de convolução com 256 filtros, tamanho do *kernel* 3×3, parâmetro *stride* igual à 1; - Camada de Normalização; - Camada de Ativação utilizando a função ReLu; - Camada de *MaxPooling*, com *stride* e *pool size* igual à 2×2.
1ª Camada Totalmente Conectada	- Camada *Flatten*; - Camada *Dense*, com entrada de 4096 valores e função de ativação ReLu.
2ª Camada Totalmente Conectada	- Camada *Dense*, com entrada de 4096 valores e função de ativação ReLu.
3ª Camada Totalmente Conectada	- Camada *Dense*, com entrada igual à 5, número de classes utilizada no problema, e função de ativação softmax.

5.1.2 Protótipo II

Ao analisar o primeiro modelo gerado, notou-se que ele não seria adequado para ser embarcado no robô, principamente por demandar muita memória. Portanto, para o segundo protótipo, buscou-se uma alternativa voltada para a utilização em ambiente embarcado e mobile, como

é o caso do robô. A solução encontrada para contornar esse problema a foi o YOLO-V4*Tiny*, versão reduzida do sistema YOLO, capaz de criar um modelo de rede neural mais simples, com número de parâmetros reduzidos, visando utilização de redes neurais em ambientes que não possuem tanto poder computacional disponível.

O número de frames por segundo (FPS) no YOLOV4-*tiny* é aproximadamente oito vezes maior do que no YOLO-V4. No entanto, a precisão do YOLO-V4 *tiny* é de aproximadamente dois terços de sua versão normal, quando testado no conjunto de dados MS COCO (TECHZIZOU, 2020).

Conforme solicitado nas documentações do YOLO-V4 *tiny*, foram utilizadas configurações específicas para a realização do treinamento, tais como imagens redimensionadas para o tamanho 416×416, taxa de aprendizado igual à $0,00261$, e número máximo de *batches* igual ao número de classes multiplicados por 2.000, o que totalizou em 12.000 lotes.

Outro tratamento realizado pelo YOLO é a criação das caixas delimitadoras nas imagens de treinamento e utilização do rótulo nas mesmas. Foi utilizado o site RoboFlow®[3], para a realização dessa tratativa e um exemplo desse processo pode ser visto na Figura 5.2.

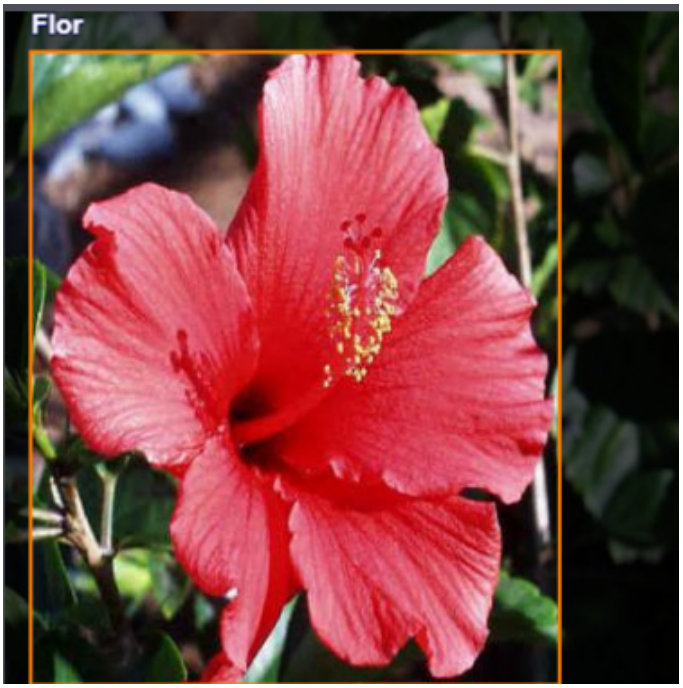


Figura 5.2: Exemplo de imagem rotulada

[3]https://app.roboflow.com

A organização das camadas seguiu o modelo desenvolvido por Jacob e Samrat (2020) e está disponível no Google Colaboratory[4]. Sua estrutura pode ser vista na Tabela 5.2.

5.1.3 Análise de Desempenho

Durante a realização dos experimentos, seguiu-se algumas etapas para análise do desempenho dos protótipos criados. A primeira delas consistiu em realizar o treinamento das redes, obtendo informações relacionadas ao desempenho apresentado. Foram coletados dados referentes ao consumo máximo de memória, tempo gasto para a realização do treinamento da rede e precisão obtida durante a etapa de testes.

Ao fim do treinamento, foram exportados os pesos gerados pela rede para um arquivo que pudesse ser utilizado em um ambiente externo ao que foi utilizado nesta etapa. O tamanho deste arquivo também foi utilizado para análise pois como ele seria utilizado no robô, seu tamanho também era algo a ser considerado.

Tabela 5.2: Estrutura da rede *YOLO*

MODELO II - YOLO-V4 TINY	
TIPO DE CAMADA	PARÂMETROS UTILIZADOS
Camada NET	- Batch = 64; - Subdivisions = 24; - Width = 416; - Height = 416; - Channels = 3; - Momentum = $0,9$; - Decay = $0,0005$; - Angles = 0; - Saturation = $1,5$; - Exposure = $1,5$; - Hue = $,1$; - Learning_rate= $0,00261$; - Burn_in = 1000; - Max_batches = 10.000; - Policy = 'steps'; - Steps = 8000 - 9000; - Scales = $,1$ - $,1$;
Continua na próxima página	

[4]https://colab.research.google.com/drive/1JIHClGomsXoLw86aWzK5T05u6uoh0cxJ?usp=sharing

Tabela 5.2 – Continuação

MODELO II - YOLO-V4 TINY	
TIPO DE CAMADA	PARÂMETROS UTILIZADOS
1ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 32 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 2 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
2ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 64 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 2 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
3ª Camada Convolucional	- Camada de Normalização;- Camada de convolução com 64 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
Camada de Rota	- Parâmetro *layer* com valor igual à -1, parâmetro grupo igual à 2 e id do grupo igual à 1.
4ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 32 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
5ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 32 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
Camada de Rota	- Parâmetro *layer* com valor igual à -1 e -2.
6ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 64 filtros, tamanho de *kernel* igual a 1×1, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
Camada de Rota	- Parâmetro *layer* com valor igual à -6 e -1.
Camada de *MaxPooling*	- Parâmetro *stride* igual a 2×2 e *pool size* igual à 2×2.
7ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 128 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
Camada de Rota	- Parâmetro *layer* com valor igual à -1, parâmetro grupo igual à 2 e id do grupo igual à 1.

Continua na próxima página

Tabela 5.2 – Continuação

MODELO II - YOLO-V4 TINY	
TIPO DE CAMADA	PARÂMETROS UTILIZADOS
8ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 64 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
9ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 64 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
Camada de Rota	- Parâmetro *layer* com valor igual à -1 e -2.
10ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 128 filtros, tamanho de *kernel* igual a 1×1, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
Camada de Rota	- Parâmetro *layer* com valor igual à -6 e -1.
Camada *MaxPolling*	- Parâmetro *stride* igual a 2×2 e *pool size* igual à 2×2.
11ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 256 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
Camada de Rota	- Parâmetro *layer* com valor igual à -1, parâmetro grupo igual à 2 e id do grupo igual à 1.
12ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 128 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
13ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 128 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
Camada de Rota	- Parâmetro *layer* com valor igual à -1 e -2.
14ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 256 filtros, tamanho de *kernel* igual a 1×1, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.

Continua na próxima página

Tabela 5.2 – Continuação

MODELO II - YOLO-V4 TINY	
TIPO DE CAMADA	PARÂMETROS UTILIZADOS
Camada de Rota	- Parâmetro *layer* com valor igual à -6 e -1.
Camada *MaxPooling*	- Parâmetro *stride* igual a 2×2 e *pool size* igual à 2×2.
15ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 512 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
16ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 256 filtros, tamanho de *kernel* igual a 1×1, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
17ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 512 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
18ª Camada Convolucional	- Camada de convolução com 30 filtros, tamanho de *kernel* igual a 1×1, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; Camada de Ativação utilizando a função Linear.
Camada YOLO	- *Mask* = 3 - 4 - 5; - *Anchors* = $10{,}14$ - $23{,}27$ - $37{,}58$ - $81{,}82$ - $135{,}169$ - $344{,}319$; - *Classes* = 5; - *Num* = 6; - *Jitter* = $,3$; - *Scale_x_y* = $1,05$; - *Cls_Normalizer* = $1,0$; - *Iou_Normalizer* = $0,07$; - *Iou_loss* = $ciou$; - *Ignore_thresh* = $,7$; - *Truth_thresh* = 1; - *Random* = 0; - *Nms_kind* = *'greedynms'*; - *Beta_nms* = 0.6;
Camada de Rota	- Parâmetro *layer* com valor igual à -4.
19ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 128 filtros, tamanho de *kernel* igual a $1X1$, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.

Continua na próxima página

Tabela 5.2 – Continuação

MODELO II - YOLO-V4 TINY	
TIPO DE CAMADA	PARÂMETROS UTILIZADOS
Camada de *Upsampling*	- Parâmetro *stride* igual à 2.
Camada de Rota	- Parâmetro *layer* com valor igual à -1 e 23.
20ª Camada Convolucional	- Camada de Normalização; - Camada de convolução com 256 filtros, tamanho de *kernel* igual a 3×3, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Leaky.
21ª Camada Convolucional	- Camada de convolução com 30 filtros, tamanho de *kernel* igual a 1×1, parâmetro *stride* igual à 1 e parâmetro *padding* igual à 1; - Camada de Ativação utilizando a função Linear.
Camada Yolo	- *Mask* = 1 - 2 - 3; - *Anchors* = $10, 14$ - $23, 27$ - $37, 58$ - $81, 82$ - $135, 169$ - $344, 319$; - *Classes* = 5; - *Num* = 6; - *Jitter* = $, 3$; - *Scale_x_y* = $1, 05$; - *Cls_normalize* = $1, 0$; - *Iou_normalize* = $0, 07$; - *Iou_loss* = *'ciou'*; - *Ignore_thresh* = $, 7$; - *Truth_thresh* = 1; - *Random* = 0; - *Nms_kind* = *'greedynms'*; - *Beta_nms* = $0, 6$;
Fim da tabela	

Capítulo 6

Resultados e Conclusões

6.1 Protótipo I

O protótipo I obteve uma precisão média de 72% durante a etapa de testes, atingindo um uso máximo de memória RAM de 2,56 GB e levando um total de 36 minutos e 22 segundos para finalizar seu treinamento.

Apesar de ter bons resultados referentes ao desempenho, a *AlexNet* se mostrou uma rede "pesada", gerando um total de 171.557.638 parâmetros quando compilada utilizando o *TensorFlow*. Isto influenciou diretamente no tamanho do arquivo contendo os pesos finais da rede, que ao ser exportado, totalizou 670 MB[1].

6.2 Protótipo II

O protótipo II apresentou uma precisão média de 77,40% durante a etapa de testes, utilizando no máximo 1,28 GB de memória RAM do ambiente de treinamento, porém, levando um total de 3 horas, 56 minutos e 41 segundos para a realização dessa etapa.

Apesar da demora na etapa de treinamento, o modelo YOLO apresentou uma rede menor que a do protótipo anterior, gerando um total de 5.891.874 parâmetros em seu modelo quando convertido para o formato *Tensorflow*. O arquivo contendo os pesos finais da rede totalizou o tamanho de 23 MB.

[1] O cálculo realizado para determinar o número de parâmetros pode ser visto em: https://www.ti-enxame.com/pt/neural-network/como-calcular-o-numero-de-parametros-para-rede-neural-convolucional/831562385/

6.3 Comparações

Como pode ser observado na Tabela 6.1, o protótipo I apresentou uma única vantagem em relação ao concorrente, referente ao tempo de treinamento.

Tabela 6.1: Comparação entre os protótipos

COMPARAÇÃO PROTÓTIPOS		
Parâmetros de Análise	**Protótipo I**	**Protótipo II**
Número de Iterações	7.000	12.000
Tempo de Execução	36m 22s	3h 56m 41s
Gasto Máximo de Memória	2,6 GB	1,8 GB
Precisão	72,00%	77,40%
Número de Parâmetros Gerados	171.557.638	5.891.874
Tamanho do Arquivo de Pesos Gerados	670MB	23MB
Possui algum sistema para marcar a área que os objetos estão?	Não	Sim

Já o protótipo II, além de apresentar melhor precisão, obteve destaque em dois pontos. O primeiro no que diz respeito ao tamanho da rede gerada e o segundo referente ao gasto de memória RAM máximo durante o treinamento. Como visto no Capítulo 5, a placa Raspberry Pi possui limitações referente ao poder de processamento, tendo disponível apenas 1 GB de memória RAM. Por conta disso, o modelo a ser embarcado nele deveria ser algo mais simples, que não demandasse tanto poder computacional.

Outro ponto positivo do protótipo II em relação ao seu componente é que o sistema YOLO já possui integrado em seu algoritmo de treinamento o processo de criação de caixas delimitadoras, que indicam a área em que o objeto classificado se encontra na imagem de entrada. Para que a *AlexNet* seja capaz de realizar essa tarefa, seria necessário utilizar algoritmos próprios para o tratamento e segmentação de imagem e para o desenho dessas caixas delimitadoras. Esse foi o caso do trabalho realizado por Wang, Xu e Han (2019), que utilizou a *AlexNet* para realizar o treinamento da rede, mas também apresentou em seu projeto algoritmos de tratamento de imagens para a realização da segmentação da imagem em objeto de interesse e fundo. Técnicas como Curva ROC (Característica de Operação do Receptor - *Receiver Operating Characteristics*), *F-Measure*, Coeficiente *Dice* (PRABHA; KUMAR, 2016) poderiam ser utilizadas nesta etapa, porém ocasionariam em mais gasto da memória do robô.

Por esses motivos escolheu-se o protótipo II para ser utilizado no processo de desenvolvi-

mento da visão computacional do robô. Além de apresentar um melhor uso da memória RAM que o outro protótipo e maior acurácia, o protótipo também possui integrado ao seu processo de treinamento toda a parte de segmentação da imagem em área de interesse.

6.4 Embarque do Protótipo II no robô

Para o embarque do Protótipo II no robô, utilizou-se os códigos fontes disponibilizados pelo repositório *tensorflow-yolov4-tflite*[2], no site GitHub®. Nele, encontram-se diversos *scripts* que recebem pesos de redes YOLO e os convertem para um formato aceito pelo *TensorFlow*. Esses *scripts* podem então ser utilizados para a deteção de objetos em vídeos, imagens ou em tempo real, através do uso da câmera.

Uma primeira etapa de testes foi realizada em uma máquina local, visando analisar o funcionamento dos códigos disponibilizados. Para isso, clonou-se o repositório citado anteriormente, que possui a estrutura de diretórios mostrada na Figura 6.1.

Figura 6.1: Estrutura de pastas do repositório

[2]https://github.com/theAIGuysCode/tensorflow-yolov4-tflite

Tabela 6.2: Comandos utilizados

COMANDOS UTILIZADOS	
ID	**Comando**
1	python save_model.py –weights ./data/custom-yolov4-tiny-detector_final.weights –output ./checkpoints/yolov4-tiny-416 –input_size 416 –model yolov4 –tiny
2	python detect.py –weights ./checkpoints/yolov4-tiny-416 –size 416 –model yolov4 –images ./data/images/imagem265.jpg –tiny
3	python detect_video.py –weights ./checkpoints/yolov4-tiny-416 –size 416 –model yolov4 –video ./data/video/IMG_2404.mp4 –output ./detections/results.mp4 –tiny
4	python detect_video.py –weights ./checkpoints/yolov4-tiny-416 –size 416 –model yolov4 –video 0 –output ./detections/results.avi –tiny

Para o funcionamento correto do sistema, alguns passos devem ser seguidos. Primeiramente é necessário criar um arquivo chamado *custom.names*, contendo o nome de todas as classes utilizadas no modelo, separadas por uma quebra de linha. Esse arquivo deverá ser salvo no caminho /data/classes.

Em seguida, no arquivo config.py, localizado na pasta /core, deve ser alterado o valor da variável *__C.YOLO.CLASSES* para *"./data/classes/custom.names"*. Dessa forma, os scripts de detecção serão capazes de reconhecer com quais classes o modelo foi treinado.

Para finalizar essa etapa de configurações, foi necessário mover o arquivo contendo os pesos finais da rede do protótipo II para a pasta /data e e rodar o Comando de ID 1, apresentado na Tabela 6.2, no terminal. Esse comando é o responsável por converter os pesos exportados para o formato *TensorFlow*.

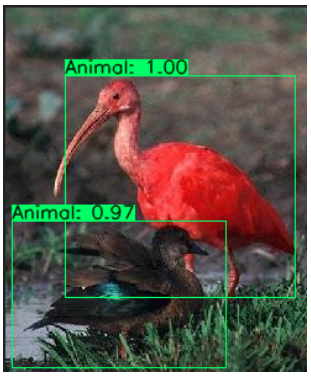
Ao fim desta conversão, foi possível rodar o sistema de detecção para reconhecer objetos em três diferentes tipos de entrada: imagem, vídeos e utilizando a câmera. Os comandos utilizados para cada uma dessas entradas foram apresentados na Tabela 6.2 com os ID 2, 3 e 4, respectivamente.

Em relação aos parâmetros utilizados nos comandos apresentados na Tabela 6.2, eles possuem as seguintes funções:

- *weights*: caminho para o arquivo contendo os pesos do modelo treinado;
- *output*: Caminho para onde serão salvos os arquivos contendo o modelo da rede convertido para o formato *TensorFlow*;
- *input_size*: Tamanho das imagens de entrada utilizadas no treinamento;

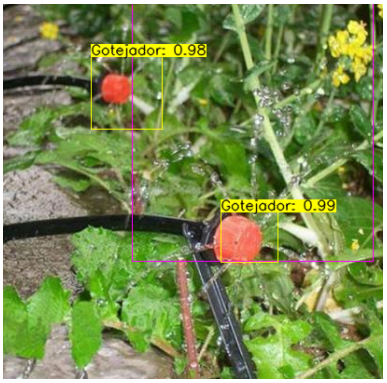
- *model*: Tipo do modelo de entrada. Neste projeto, por exemplo, o valor deve ser igual à YOLOv4;
- *tiny*: Indica se o modelo treinado segue o formato *tiny* ou normal;
- *size*: Tamanho das imagens de entrada utilizadas no treinamento;
- *images*: Caminho para a imagem que será utilizada no processo de detecção;
- *video*: Caminho para o vídeo que será utilizado no processo de detecção. Caso esse valor seja um número inteiro, como 0, significa que será utilizada como entrada uma câmera capturando imagens em tempo real, ao invés de um arquivo;
- *output*: Caminho no qual serão salvos os vídeos após o processo de detecção de objetos.

A Figura 6.2 apresenta os resultados obtidos após rodar o sistema YOLO em imagens de diferentes classes.



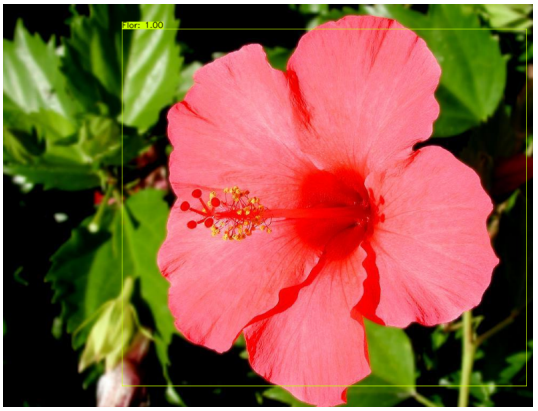
(a) Detecção da classe Animal

(b) Detecção da classe Veículo



(c) Detecção da classe Gotejador

(d) Detecção da classe Pessoa



(e) Detecção da classe Flor

(f) Detecção da classe Planta

Figura 6.2: Exemplos de Detecção em Imagens

6.5 Considerações Finais e Recomendações para Trabalhos Futuros

Conforme definido no Capítulo 1, na seção de Objetivos, foi desenvolvido um sistema de visão computacional embarcado em um robô móvel de pequeno porte e que possui uma câmera embutida. Este sistema se mostrou capaz de detectar e identificar objetos presentes em ambiente agrícola, tais como plantas, veículos, pessoas, entre outros.

O primeiro objetivo definido consistia em realizar uma revisão bibliográfica sobre assuntos pertinentes. Para isso, foram realizas pesquisas referentes aos assuntos: visão computacional, aprendizagem de máquina, redes neurais convolucionais. Além disso, também discorreu-se sobre arquiteturas de redes neurais convolucionais disponíveis atualmente e que poderiam ser utilizadas para o desenvolvimento do trabalho. Nesta etapa também estudou-se as principais ferramentas disponíveis para a criação do sistema de visão computacional, como a linguagem Python e a biblioteca OpenCV e *Tensorflow*.

A etapa seguinte consistiu na criação de uma base de dados, capaz de representar uma gama de objetos que poderiam ser encontrados ambientes rurais locais. Para isso, foram reunidas 1.000 imagens de diversas fontes, contendo 170 instâncias das classes Animal, Flores, Plantas, Pessoas e Veículos e 150 da classe Gotejador.

O terceiro objetivo tinha como foco a criação de uma rede neural capaz de reconhecer os objetos em uma câmera tradicional. Uma pequena alteração foi realizada neste objetivo, no qual ambos protótipos criados utilizaram como entrada de dados para o treinamento a base de dados criada anteriormente, ao invés de utilizar uma câmera. Isso foi feito pois era mais simples obter as imagens de ambiente rural utilizando imagens existentes na internet. Além disso, a pandemia também dificultou a coleta de imagens. Portanto, primeiro foram realizados os treinamentos utilizando a base de dados criada e após obter-se resultados satisfatórios, foi alterado o método de entrada para a câmera tradicional, transmitindo dados em tempo real.

Em relação aos modelos criados, o primeiro protótipo foi construído baseando-se na estrutura da *AlexNet*. Este protótipo, apesar de ter apresentado um bom tempo de treinamento, gerou uma rede pesada, impossibilitando sua utilização pelo robô, que possui pouca memória disponível. Em virtude desta dificuldade, construiu-se o segundo protótipo, baseado no sistema YOLO, que além de apresentar acurácia superior, também gerou uma rede com menos

parâmetros, podendo ser embarcada no robô móvel.

Após essa etapa de treinamento e escolha do modelo que seria embarcado no robô, foram utilizados os scripts disponibilizados no repositório *tensorflow-yolov4-tflite* para realização da detecção das classes em diferentes entradas. Nesta etapa, foi possível, por exemplo, utilizar a câmera convencional para a realização da detecção.

Como quarto objetivo, foi definido que a rede neural desenvolvida seria embarcada ao robô de forma que fossem recebidas como entradas as imagens capturadas pela câmera embarcada. Para isso, foram passados todos os arquivos do repositório baixado e os pesos obtidos durante a etapa de treinamento.

Por fim, o objetivo final do trabalho era integrar o sistema de visão ao controle remoto do robô para que fosse possível criar um sistema de localização para o mesmo. Esta não foi concluída com exatidão por dois motivos. O primeiro foi que apesar de ter sido capaz de realizar a tarefa de classificação corretamente, o sistema de visão apresentou um baixo desempenho quanto ao processamento em tempo real de imagens, não sendo capaz de capturar mais de um frame por segundo. Isso acaba dificultando a utilização do sistema de visão em um controle remoto de movimento do robô pois, as imagens recebidas estariam atrasadas se comparadas à localização do robô. O outro motivo foi referente à não completude do sistema de controle remoto (o qual não está incluído nos objetivos deste trabalho), que impossibilitou a incorporação do sistema de visão. Portanto, tendo em vista esta pendência referente à integração do sistema de visão ao sistema de controle remoto do robô, ficam como recomendação para trabalhos futuros:

- Otimização do script de detecção por vídeo, buscando melhorar a taxa de frames por segundos obtidos durante a captura de imagens em tempo real;
- Melhora no *hardware* do robô, tornando seu poder de processamento melhor, e comparando os resultados obtidos com os desse trabalho;
- Criação do sistema de controle remoto para o robô e integração dele ao sistema de visão desenvolvido neste trabalho, tornando possível realizar a movimentação do robô;
- Desenvolvimento de novos protótipos, com arquiteturas não contempladas neste projeto, para fins de comparação.

Apêndice A

Códigos Desenvolvidos

Neste capítulo serão descritos os algoritmos utilizados para a realização deste trabalho. O Algoritmo 1 foi baseado no código desenvolvido por Alake (2020), enquanto o Algoritmo 2 se espelhou no protótipo desenvolvido por Jacob e Samrat (2020).

Por fim, os Algoritmos 3, 4 e 5 podem ser encontrados no repositório *tensorflow-yolov4-tflite*[1], no site GitHub®.

[1] https://github.com/theAIGuysCode/tensorflow-yolov4-tflite

Algoritmo 1 Protótipo I

```
import numpy as np
import random
import cv2
import os
import csv
import time
import random
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import matplotlib.image as mpimg
import tensorflow as tf
from keras.models import model_from_json
from google.colab import files
from tensorflow import keras
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten,
    Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization
from sklearn import metrics
from PIL import Image
from scipy.stats import mode
from os import listdir

Directory='/content/BaseTCC/BaseNova'
subfolders = listdir(Directory)
print(subfolders)

Classes = len(subfolders)
Classes

training_data =[]

for n in range(Classes):
    path =  Directory+'/'+subfolders[n]
    for img in os.listdir(path):
      try:
        img_array = cv2.imread(os.path.join(path,img))
        img_array = cv2.resize(img_array,(200,200),interpolation = cv2.
    INTER_AREA)
        training_data.append([img_array, n])
      except Exception as e:
         pass
random.shuffle(training_data)

X = []
Y = []
for features, label in training\_data:
  X.append(features)
  Y.append(label)
```

```
x\_treino, x\_teste\_, y\_treino, y\_teste_ = train\_test\_split(X,Y, test\
    _size=0.30, stratify = Y)
x\_validacao, x\_teste, y\_validacao, y\_teste = train\_test\_split(x\
    _teste\_,y\_teste\_, test\_size=0.50, stratify = y\_teste\_)

x_treino = np.array(x_treino)
x_validacao = np.array(x_validacao)
x_teste = np.array(x_teste)
y_treino = np.array(y_treino)
y_validacao = np.array(y_validacao)
y_teste = np.array(y_teste)

model = Sequential()

# CAMADAS DE CONVOLUCAO
    ---------------------------------------------------------------------
# 1 Camada de Convolucao
model.add(Conv2D(filters = 96, kernel_size= (11,11), strides= 4,
    input_shape = (200,200,3)))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

# 2 Camada de Convolucao
model.add(Conv2D(filters = 256, kernel_size= (5,5), strides= 1,padding='
    same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
#model.add(MaxPooling2D(pool_size=(3,3), strides=2, padding='valid'))

# 3 Camada de Convolucao
model.add(Conv2D(filters = 384, kernel_size= (3,3), strides = 1, padding='
    same'))
model.add(BatchNormalization())
model.add(Activation('relu'))

# 4 Camada de Convolucao
model.add(Conv2D(filters = 384, kernel_size=(3,3), strides= 1, padding='
    same'))
model.add(BatchNormalization())
model.add(Activation('relu'))

# 5 Camada de Convolucao
model.add(Conv2D(filters = 256, kernel_size=(3,3), strides= 1, padding='
    same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides= (2,2)))
```

```
#CAMADAS TOTALMENTE CONECTADAS ----------------------------------------

# 1 Camada totalmente conectada
model.add(Flatten())
model.add(Dense(4096, activation='relu')) # Camada Conectada

# 2 Camada totalmente conectada
model.add(Dense(4096, activation='relu'))

# 3 Camada totalmente conectada
model.add(Dense(Classes, activation='softmax'))

model.compile(optimizer='adam',loss=tf.losses.SparseCategoricalCrossentropy
    (from_logits=True),metrics=['sparse_categorical_accuracy'])

model.summary()

keras.optimizers.Adam(learning_rate=0.00261)

history = model.fit(x_treino, y_treino, epochs = 1000, validation_data=(
    x_validacao, y_validacao),validation_freq=1,batch_size=100)

plt.plot(history.history['sparse_categorical_accuracy'], label='Acuracia
    Treino')
plt.plot(history.history['val_sparse_categorical_accuracy'], label = '
    Acuracia Validacao')
plt.xlabel('Epoca')
plt.ylabel('Acuracia')
plt.ylim([0.0, 1.1])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(x_teste,y_teste, verbose=2)
print(test_acc)

model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)

model.save_weights("model_weights.h5")
model.save("model.h5")
print("Modelo salvo!")

!mkdir -p saved_model
model.save("saved_model/my_model")

files.download('model_weights.h5')

files.download('model.h5')
```

Algoritmo 2 Protótipo II

```
#Clonando o repositorio do darknet
!git clone https://github.com/roboflow-ai/darknet.git

 %cd /content/darknet/
!sed -i 's/OPENCV=0/OPENCV=1/g' Makefile
!sed -i 's/GPU=0/GPU=1/g' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/g' Makefile
!sed -i "s/ARCH= -gencode arch=compute_60,code=sm_60/ARCH= ${ARCH_VALUE}/g"
    Makefile
!make

# Baixa os ultimos pesos yolov4-tiny lancados
%cd /content/darknet
!wget https://github.com/AlexeyAB/darknet/releases/download/
   darknet_yolo_v4_pre/yolov4-tiny.weights
!wget https://github.com/AlexeyAB/darknet/releases/download/
   darknet_yolo_v4_pre/yolov4-tiny.conv.29

# Copia os arquivos da base de dados hospedada no site Roboflow
%cd /content/darknet
!curl -L "https://app.roboflow.com/ds/a0qehqK3wu?key=3jcR2vy9kM" > roboflow
   .zip; unzip roboflow.zip; rm roboflow.zip

# Configura os diretorios de arquivos de treinamento para cada conjunto de
   dados
%cd /content/darknet/
%cp train/_darknet.labels data/obj.names
%mkdir data/obj

# Copia as imagens e rotulos
%cp train/*.jpg data/obj/
%cp valid/*.jpg data/obj/

%cp train/*.txt data/obj/
%cp valid/*.txt data/obj/

with open('data/obj.data', 'w') as out:
  out.write('classes = 3\n')
  out.write('train = data/train.txt\n')
  out.write('valid = data/valid.txt\n')
  out.write('names = data/obj.names\n')
  out.write('backup = backup/')

# Escreve uma lista contendo imagens de treinamento
import os

with open('data/train.txt', 'w') as out:
  for img in [f for f in os.listdir('train') if f.endswith('jpg')]:
    out.write('data/obj/' + img + '\n')
```

```
# Escreve uma lista contendo imagens de validacao
import os

with open('data/valid.txt', 'w') as out:
  for img in [f for f in os.listdir('valid') if f.endswith('jpg')]:
    out.write('data/obj/' + img + '\n')


# Organizacao do treinamento

def file_len(fname):
  with open(fname) as f:
    for i, l in enumerate(f):
      pass
  return i + 1

num_classes = file_len('train/_darknet.labels')
max_batches = num_classes*2000
steps1 = .8 * max_batches
steps2 = .9 * max_batches
steps_str = str(steps1)+','+str(steps2)
num_filters = (num_classes + 5) * 3

if os.path.exists('./cfg/custom-yolov4-tiny-detector.cfg'): os.remove('./
    cfg/custom-yolov4-tiny-detector.cfg')

from IPython.core.magic import register_line_cell_magic

@register_line_cell_magic
def writetemplate(line, cell):
    with open(line, 'w') as f:
        f.write(cell.format(**globals()))

%%writetemplate ./cfg/custom-yolov4-tiny-detector.cfg

[net]
batch=64
subdivisions=24
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1
```

```
1
2 learning_rate=0.00261
3 burn_in=1000
4 max_batches = {max_batches}
5 policy=steps
6 steps={steps_str}
7 scales=.1,.1
8
9 [convolutional]
10 batch_normalize=1
11 filters=32
12 size=3
13 stride=2
14 pad=1
15 activation=leaky
16
17 [convolutional]
18 batch_normalize=1
19 filters=64
20 size=3
21 stride=2
22 pad=1
23 activation=leaky
24
25 [convolutional]
26 batch_normalize=1
27 filters=64
28 size=3
29 stride=1
30 pad=1
31 activation=leaky
32
33 [route]
34 layers=-1
35 groups=2
36 group_id=1
37
38 [convolutional]
39 batch_normalize=1
40 filters=32
41 size=3
42 stride=1
43 pad=1
44 activation=leaky
45
46 [convolutional]
47 batch_normalize=1
48 filters=32
49 size=3
50 stride=1
51 pad=1
52 activation=leaky
```

```
1
2  [route]
3  layers = -1,-2
4
5  [convolutional]
6  batch_normalize=1
7  filters=64
8  size=1
9  stride=1
10 pad=1
11 activation=leaky
12
13 [route]
14 layers = -6,-1
15
16 [maxpool]
17 size=2
18 stride=2
19
20 [convolutional]
21 batch_normalize=1
22 filters=128
23 size=3
24 stride=1
25 pad=1
26 activation=leaky
27
28 [route]
29 layers=-1
30 groups=2
31 group_id=1
32
33 [convolutional]
34 batch_normalize=1
35 filters=64
36 size=3
37 stride=1
38 pad=1
39 activation=leaky
40
41 [convolutional]
42 batch_normalize=1
43 filters=64
44 size=3
45 stride=1
46 pad=1
47 activation=leaky
48
49 [route]
50 layers = -1,-2
```

```
1
2  [convolutional]
3  batch_normalize=1
4  filters=128
5  size=1
6  stride=1
7  pad=1
8  activation=leaky
9
10 [route]
11 layers = -6,-1
12
13 [maxpool]
14 size=2
15 stride=2
16
17 [convolutional]
18 batch_normalize=1
19 filters=256
20 size=3
21 stride=1
22 pad=1
23 activation=leaky
24
25 [route]
26 layers=-1
27 groups=2
28 group_id=1
29
30 [convolutional]
31 batch_normalize=1
32 filters=128
33 size=3
34 stride=1
35 pad=1
36 activation=leaky
37
38 [convolutional]
39 batch_normalize=1
40 filters=128
41 size=3
42 stride=1
43 pad=1
44 activation=leaky
45
46 [route]
47 layers = -1,-2
```

```
[convolutional]
batch_normalize=1
filters=256
size=1
stride=1
pad=1
activation=leaky

[route]
layers = -6,-1

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=256
size=1
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters={num_filters}
activation=linear
```

```
1
2 [yolo]
3 mask = 3,4,5
4 anchors = 10,14,  23,27,  37,58,  81,82,  135,169,  344,319
5 classes={num_classes}
6 num=6
7 jitter=.3
8 scale_x_y = 1.05
9 cls_normalizer=1.0
10 iou_normalizer=0.07
11 iou_loss=ciou
12 ignore_thresh = .7
13 truth_thresh = 1
14 random=0
15 nms_kind=greedynms
16 beta_nms=0.6
17
18 [route]
19 layers = -4
20
21 [convolutional]
22 batch_normalize=1
23 filters=128
24 size=1
25 stride=1
26 pad=1
27 activation=leaky
28
29 [upsample]
30 stride=2
31
32 [route]
33 layers = -1, 23
34
35 [convolutional]
36 batch_normalize=1
37 filters=256
38 size=3
39 stride=1
40 pad=1
41 activation=leaky
42
43 [convolutional]
44 size=1
45 stride=1
46 pad=1
47 filters={num_filters}
48 activation=linear
```

```
1
2 [yolo]
3 mask = 1,2,3
4 anchors = 10,14,  23,27,  37,58,  81,82,  135,169,  344,319
5 classes={num_classes}
6 num=6
7 jitter=.3
8 scale_x_y = 1.05
9 cls_normalizer=1.0
10 iou_normalizer=0.07
11 iou_loss=ciou
12 ignore_thresh = .7
13 truth_thresh = 1
14 random=0
15 nms_kind=greedynms
16 beta_nms=0.6
17
18  %cat cfg/custom-yolov4-tiny-detector.cfg
19
20 #TREINAMENTO A BASE DE DADOS"
21
22 !./darknet detector train data/obj.data cfg/custom-yolov4-tiny-detector.cfg
       yolov4-tiny.conv.29 -dont_show -map
23
24 #CONVERTENDO PESOS PARA MODELO TENSORFLOW
25
26 #Primeiro clonar o repositorio com o codigo para fazer a conversao
27 # %cd /content
28 !git clone https://github.com/hunglc007/tensorflow-yolov4-tflite.git
29 # %cd /content/tensorflow-yolov4-tflite
30
31 #Converter as classes do modelo COCO para ser usada no tf
32 !cp /content/darknet/data/obj.names /content/tensorflow-yolov4-tflite/data/
      classes/
33 !ls /content/tensorflow-yolov4-tflite/data/classes
34
35 !sed -i "s/coco.names/obj.names/g" /content/tensorflow-yolov4-tflite/core/
      config.py
36
37 #Convertendo os pesos para tensorflow e tensorflow lite
38 %cd /content/tensorflow-yolov4-tflite
39 !python save_model.py \
40   --weights /content/darknet/backup/custom-yolov4-tiny-detector_final.
      weights \
41   --output ./checkpoints/yolov4-tiny-416 \
42   --input_size 416 \
43   --model yolov4 \
44   --tiny \
```

Algoritmo 3 Script para a conversão dos pesos Yolov4 para o formato *TensorFlow*

```
import tensorflow as tf
from absl import app, flags, logging
from absl.flags import FLAGS
from core.yolov4 import YOLO, decode, filter_boxes
import core.utils as utils
from core.config import cfg

flags.DEFINE_string('weights', './data/yolov4.weights', 'path to weights file')
flags.DEFINE_string('output', './checkpoints/yolov4-416', 'path to output')
flags.DEFINE_boolean('tiny', False, 'is yolo-tiny or not')
flags.DEFINE_integer('input_size', 416, 'define input size of export model')
flags.DEFINE_float('score_thres', 0.2, 'define score threshold')
flags.DEFINE_string('framework', 'tf', 'define what framework do you want to convert (tf, trt, tflite)')
flags.DEFINE_string('model', 'yolov4', 'yolov3 or yolov4')

def save_tf():
  STRIDES, ANCHORS, NUM_CLASS, XYSCALE = utils.load_config(FLAGS)

  input_layer = tf.keras.layers.Input([FLAGS.input_size, FLAGS.input_size, 3])
  feature_maps = YOLO(input_layer, NUM_CLASS, FLAGS.model, FLAGS.tiny)
  bbox_tensors = []
  prob_tensors = []
  if FLAGS.tiny:
    for i, fm in enumerate(feature_maps):
      if i == 0:
        output_tensors = decode(fm, FLAGS.input_size // 16, NUM_CLASS, STRIDES, ANCHORS, i, XYSCALE, FLAGS.framework)
      else:
        output_tensors = decode(fm, FLAGS.input_size // 32, NUM_CLASS, STRIDES, ANCHORS, i, XYSCALE, FLAGS.framework)
      bbox_tensors.append(output_tensors[0])
      prob_tensors.append(output_tensors[1])
```

```
else:
    for i, fm in enumerate(feature_maps):
      if i == 0:
        output_tensors = decode(fm, FLAGS.input_size // 8, NUM_CLASS,
    STRIDES, ANCHORS, i, XYSCALE, FLAGS.framework)
      elif i == 1:
        output_tensors = decode(fm, FLAGS.input_size // 16, NUM_CLASS,
    STRIDES, ANCHORS, i, XYSCALE, FLAGS.framework)
      else:
        output_tensors = decode(fm, FLAGS.input_size // 32, NUM_CLASS,
    STRIDES, ANCHORS, i, XYSCALE, FLAGS.framework)
      bbox_tensors.append(output_tensors[0])
      prob_tensors.append(output_tensors[1])
  pred_bbox = tf.concat(bbox_tensors, axis=1)
  pred_prob = tf.concat(prob_tensors, axis=1)
  if FLAGS.framework == 'tflite':
    pred = (pred_bbox, pred_prob)
  else:
    boxes, pred_conf = filter_boxes(pred_bbox, pred_prob, score_threshold=
    FLAGS.score_thres, input_shape=tf.constant([FLAGS.input_size, FLAGS.
    input_size]))
    pred = tf.concat([boxes, pred_conf], axis=-1)
  model = tf.keras.Model(input_layer, pred)
  utils.load_weights(model, FLAGS.weights, FLAGS.model, FLAGS.tiny)
  model.summary()
  model.save(FLAGS.output)

def main(_argv):
  save_tf()

if __name__ == '__main__':
    try:
        app.run(main)
    except SystemExit:
        pass
```

Algoritmo 4 Script de detecção em Imagens

```
import tensorflow as tf
physical_devices = tf.config.experimental.list_physical_devices('GPU')
if len(physical_devices) > 0:
    tf.config.experimental.set_memory_growth(physical_devices[0], True)
from absl import app, flags, logging
from absl.flags import FLAGS
import core.utils as utils
from core.config import cfg
from core.yolov4 import filter_boxes
from tensorflow.python.saved_model import tag_constants
from PIL import Image
import cv2
import numpy as np
from tensorflow.compat.v1 import ConfigProto
from tensorflow.compat.v1 import InteractiveSession

flags.DEFINE_string('framework', 'tf', '(tf, tflite, trt')
flags.DEFINE_string('weights', './checkpoints/yolov4-416',
                    'path to weights file')
flags.DEFINE_integer('size', 416, 'resize images to')
flags.DEFINE_boolean('tiny', False, 'yolo or yolo-tiny')
flags.DEFINE_string('model', 'yolov4', 'yolov3 or yolov4')
flags.DEFINE_list('images', './data/images/kite.jpg', 'path to input image'
    )
flags.DEFINE_string('output', './detections/', 'path to output folder')
flags.DEFINE_float('iou', 0.45, 'iou threshold')
flags.DEFINE_float('score', 0.25, 'score threshold')
flags.DEFINE_boolean('dont_show', False, 'dont show image output')

def main(_argv):
    config = ConfigProto()
    config.gpu_options.allow_growth = True
    session = InteractiveSession(config=config)
    STRIDES, ANCHORS, NUM_CLASS, XYSCALE = utils.load_config(FLAGS)
    input_size = FLAGS.size
    images = FLAGS.images

    # Carrega modelo
    if FLAGS.framework == 'tflite':
            interpreter = tf.lite.Interpreter(model_path=FLAGS.weights)
    else:
            saved_model_loaded = tf.saved_model.load(FLAGS.weights, tags=[
    tag_constants.SERVING])

    # Busca as imagens e roda o Yolov4 para cada uma delas
    for count, image_path in enumerate(images, 1):
        original_image = cv2.imread(image_path)
        original_image = cv2.cvtColor(original_image, cv2.COLOR_BGR2RGB)

        image_data = cv2.resize(original_image, (input_size, input_size))
        image_data = image_data / 255.
```

```
images_data = []
        for i in range(1):
            images_data.append(image_data)
        images_data = np.asarray(images_data).astype(np.float32)

        if FLAGS.framework == 'tflite':
            interpreter.allocate_tensors()
            input_details = interpreter.get_input_details()
            output_details = interpreter.get_output_details()
            print(input_details)
            print(output_details)
            interpreter.set_tensor(input_details[0]['index'], images_data)
            interpreter.invoke()
            pred = [interpreter.get_tensor(output_details[i]['index']) for
    i in range(len(output_details))]
            if FLAGS.model == 'yolov3' and FLAGS.tiny == True:
                boxes, pred_conf = filter_boxes(pred[1], pred[0],
    score_threshold=0.25, input_shape=tf.constant([input_size, input_size]))
            else:
                boxes, pred_conf = filter_boxes(pred[0], pred[1],
    score_threshold=0.25, input_shape=tf.constant([input_size, input_size]))
        else:
            infer = saved_model_loaded.signatures['serving_default']
            batch_data = tf.constant(images_data)
            pred_bbox = infer(batch_data)
            for key, value in pred_bbox.items():
                boxes = value[:, :, 0:4]
                pred_conf = value[:, :, 4:]

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

        # Le as classes do arquivo config
        class_names = utils.read_class_names(cfg.YOLO.CLASSES)

        # por padrao permite todas classes no arquivo .names
        allowed_classes = list(class_names.values())

        image = utils.draw_bbox(original_image, pred_bbox, allowed_classes
    = allowed_classes)

        image = Image.fromarray(image.astype(np.uint8))
```

```
 if not FLAGS.dont_show:
            image.show()
        image = cv2.cvtColor(np.array(image), cv2.COLOR_BGR2RGB)
        cv2.imwrite(FLAGS.output + 'detection' + str(count) + '.png', image
    )

if __name__ == '__main__':
    try:
        app.run(main)
    except SystemExit:
        pass
```

Algoritmo 5 Script para detecção em vídeos

```
import time
import tensorflow as tf
physical_devices = tf.config.experimental.list_physical_devices('GPU')
if len(physical_devices) > 0:
    tf.config.experimental.set_memory_growth(physical_devices[0], True)
from absl import app, flags, logging
from absl.flags import FLAGS
import core.utils as utils
from core.yolov4 import filter_boxes
from tensorflow.python.saved_model import tag_constants
from PIL import Image
import cv2
import numpy as np
from tensorflow.compat.v1 import ConfigProto
from tensorflow.compat.v1 import InteractiveSession

flags.DEFINE_string('framework', 'tf', '(tf, tflite, trt')
flags.DEFINE_string('weights', './checkpoints/yolov4-416',
                    'path to weights file')
flags.DEFINE_integer('size', 416, 'resize images to')
flags.DEFINE_boolean('tiny', False, 'yolo or yolo-tiny')
flags.DEFINE_string('model', 'yolov4', 'yolov3 or yolov4')
flags.DEFINE_string('video', './data/video/video.mp4', 'path to input video
    or set to 0 for webcam')
flags.DEFINE_string('output', None, 'path to output video')
flags.DEFINE_string('output_format', 'XVID', 'codec used in VideoWriter
   when saving video to file')
flags.DEFINE_float('iou', 0.45, 'iou threshold')
flags.DEFINE_float('score', 0.25, 'score threshold')
flags.DEFINE_boolean('dont_show', False, 'dont show video output')

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

    if FLAGS.framework == 'tflite':
        interpreter = tf.lite.Interpreter(model_path=FLAGS.weights)
        interpreter.allocate_tensors()
        input_details = interpreter.get_input_details()
        output_details = interpreter.get_output_details()
        print(input_details)
        print(output_details)
    else:
        saved_model_loaded = tf.saved_model.load(FLAGS.weights, tags=[
   tag_constants.SERVING])
        infer = saved_model_loaded.signatures['serving_default']
```

```
# Inicia a captura de video
    try:
        vid = cv2.VideoCapture(int(video_path))
    except:
        vid = cv2.VideoCapture(video_path)

    out = None

    if FLAGS.output:
        # por padrao VideoCapture retorna um float ao inves de int
        width = int(vid.get(cv2.CAP_PROP_FRAME_WIDTH))
        height = int(vid.get(cv2.CAP_PROP_FRAME_HEIGHT))
        fps = int(vid.get(cv2.CAP_PROP_FPS))
        codec = cv2.VideoWriter_fourcc(*FLAGS.output_format)
        out = cv2.VideoWriter(FLAGS.output, codec, fps, (width, height))

    while True:
        return_value, frame = vid.read()
        if return_value:
            frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
            image = Image.fromarray(frame)
        else:
            print('Video has ended or failed, try a different video format!
    ')
            break

        frame_size = frame.shape[:2]
        image_data = cv2.resize(frame, (input_size,input_size))
        image_data = image_data / 255.
        image_data = image_data[np.newaxis,...].astype(np.float32)
        start_time = time.time()

        if FLAGS.framework == 'tflite':
            interpreter.set_tensor(input_details[0]['index'], image_data)
            interpreter.invoke()
            pred = [interpreter.get_tensor(output_details[i]['index'])
            for i in range(len(output_details))]
            if FLAGS.model == 'yolov3' and FLAGS.tiny == True:
                boxes, pred_conf = filter_boxes(pred[1], pred[0],
    score_threshold=0.25, input_shape=tf.constant([input_size, input_size]))
            else:
                boxes, pred_conf = filter_boxes(pred[0], pred[1],
    score_threshold=0.25,input_shape=tf.constant([input_size, input_size]))
        else:
            batch_data = tf.constant(image_data)
            pred_bbox = infer(batch_data)
            for key, value in pred_bbox.items():
                boxes = value[:, :, 0:4]
                pred_conf = value[:, :, 4:]
```

```
boxes, scores, classes, valid_detections = tf.image.combined_non_max_suppression(
            boxes=tf.reshape(boxes, (tf.shape(boxes)[0], -1, 1, 4)),
            scores=tf.reshape(pred_conf, (tf.shape(pred_conf)[0], -1, tf.shape(pred_conf)[-1])), max_output_size_per_class=50,
            max_total_size=50,
            iou_threshold=FLAGS.iou,
            score_threshold=FLAGS.score
        )
        pred_bbox = [boxes.numpy(), scores.numpy(), classes.numpy(), valid_detections.numpy()]
        image = utils.draw_bbox(frame, pred_bbox)
        fps = 1.0 / (time.time() - start_time)
        print("FPS: %.2f" % fps)
        result = np.asarray(image)
        cv2.namedWindow("result", cv2.WINDOW_AUTOSIZE)
        result = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)

        if not FLAGS.dont_show:
            cv2.imshow("result", result)

        if FLAGS.output:
            out.write(result)
        if cv2.waitKey(1) & 0xFF == ord('q'): break
    cv2.destroyAllWindows()

if __name__ == '__main__':
    try:
        app.run(main)
    except SystemExit:
        pass
```

Referências Bibliográficas

ABADI, M. et al. Tensorflow: A system for large-scale machine learning. In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. [S.l.: s.n.], 2016. p. 265–283.

ACADEMY, D. S. *Deep Learning Book*. 2019. Disponível em: ⟨https://www.deeplearningbook.com.br⟩. Acesso em: 08 jul 2021.

ACADEMY, E. D. S. *O que é visão computacional?* 2018. Disponível em: ⟨https://blog.dsacademy.com.br/o-que-e-visao-computacional/⟩. Acesso em: 08 jul 2020.

ALAKE, R. *Implementing AlexNet CNN Architecture Using TensorFlow 2.0+ and Keras*. 2020. Disponível em: ⟨https://towardsdatascience.com/implementing-alexnet-cnn-architecture-using-tensorflow-2-0-and-keras-2113e090ad98⟩. Acesso em: 08 jul 2021.

ALIGER, E. *Visão computacional e suas aplicações*. 2020. Disponível em: ⟨https://www.aliger.com.br/blog/saiba-o-que-e-visao-computacional⟩. Acesso em: 08 jul 2021.

ALVES, G. *Entendendo Redes Convolucionais (CNNs)*. 2018. Disponível em: ⟨https://medium.com/neuronio-br/entendendo-redes-convolucionais-cnns-d10359f21184⟩. Acesso em: 08 jul 2021.

ALVES, G. *Detecção de Objetos com YOLO – Uma abordagem moderna*. 2020. Disponível em: ⟨https://iaexpert.academy/2020/10/13/deteccao-de-objetos-com-yolo-uma-abordagem-moderna/⟩. Acesso em: 08 jul 2021.

ÅSTRAND, B.; BAERVELDT, A.-J. An agricultural mobile robot with vision-based perception for mechanical weed control. *Autonomous robots*, Springer, v. 13, n. 1, p. 21–35, 2002.

BOCHKOVSKIY, A. *CFG Parameters in the different layers*. 2020. Disponível em: ⟨https://github.com/AlexeyAB/darknet/wiki/CFG-Parameters-in-the-different-layers⟩. Acesso em: 08 jul 2021.

BOCHKOVSKIY, A. *CFG Parameters in the [net] section*. 2020. Disponível em: ⟨https://github.com/AlexeyAB/darknet/wiki/CFG-Parameters-in-the-\\%5Bnet\\%5D-section⟩. Acesso em: 08 jul 2021.

BOCHKOVSKIY, A.; WANG, C.-Y.; LIAO, H.-Y. M. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

BOER, P.-T. D. et al. A tutorial on the cross-entropy method. *Annals of operations research*, Springer, v. 134, n. 1, p. 19–67, 2005.

BONNER, A. *The Complete Beginner's Guide to Deep Learning: Convolutional Neural Networks and Image Classification*. 2019. Disponível em: ⟨https://towardsdatascience.com/wtf-is-image-classification-8e78a8235acb⟩. Acesso em: 08 jul 2021.

BROWNLEE, J. *Deep Learning for Computer Vision: Image Classification, Object Detection, and Face Recognition in Python*. [S.l.]: Machine Learning Mastery, 2019.

CASS, S. *The 2018 Top Programming Languages*. 2018. Disponível em: ⟨https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages⟩. Acesso em: 08 jul 2021.

CHOI, H. et al. An objectness score for accurate and fast detection during navigation. *arXiv preprint arXiv:1909.05626*, 2019.

COSTA, R. *Conceitos Básicos de Redes Neurais Artificiais*. 2019. Disponível em: ⟨https://www.linkedin.com/pulse/conceitos-bsicos-de-redes-neurais-artificiais-ruan-costa⟩. Acesso em: 08 jul 2021.

DAVID, E.; SELFRIDGE, O. Eyes and ears for computers. *Proceedings of the IRE*, IEEE, v. 50, n. 5, p. 1093–1101, 1962.

DAVIES, E. R. *Computer vision: principles, algorithms, applications, learning*. [S.l.]: Academic Press, 2017.

EBERMAM, E. et al. *Programação para leigos com Raspberry Pi*. [S.l.]: Edifes, Vitória, 2017.

FRANÇA, H. F. do C.; SOARES, A. Googlenet-going deeper with convolutions. Computer Vision Foundation, 2016.

GOGONI, R. *O que é o Raspberry Pi*. 2019. Disponível em: ⟨https://tecnoblog.net/282739/o-que-e-o-raspberry-pi⟩. Acesso em: 08 jul 2021.

HAN, S. et al. Deep neural networks show an equivalent and often superior performance to dermatologists in onychomycosis diagnosis: Automatic construction of onychomycosis datasets by region-based convolutional deep neural network. *PLOS ONE*, v. 13, p. e0191493, 01 2018.

HE, K.; SUN, J. Convolutional neural networks at constrained time cost. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2015. p. 5353–5360.

HE, K. et al. Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2016. p. 770–778.

HINTON, G. E.; SALAKHUTDINOV, R. R. Reducing the dimensionality of data with neural networks. *science*, American Association for the Advancement of Science, v. 313, n. 5786, p. 504–507, 2006.

HOFESMANN, E. *IoU a better detection evaluation metric*. 2020. Disponível em: ⟨https://towardsdatascience.com/iou-a-better-detection-evaluation-metric-45a511185be1⟩. Acesso em: 11 jul 2021.

HUANG, G. et al. Densely connected convolutional networks. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2017. p. 4700–4708.

IANDOLA, F. N. et al. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and < 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

ITMÍDIA. *Python: 10 motivos para aprender a linguagem em 2019*. 2019. Disponível em: ⟨https://computerworld.com.br/carreira/python-10-motivos-para-aprender-a-linguagem-em-2019/⟩. Acesso em: 08 jul 2021.

JACOB, S.; SAMRAT, S. *Train YOLOv4-tiny on Custom Data - Lightning Fast Object Detection*. 2020. Disponível em: ⟨https://blog.roboflow.com/train-yolov4-tiny-on-custom-data-lighting-fast-detection/⟩. Acesso em: 08 jul 2021.

JARDIM, A. Inovação para o agro 4.0. *AgroANALYSIS*, v. 38, n. 4, p. 48, 2019.

JODAS, D. S. et al. Desenvolvimento de um sistema para navegaçao de robôs móveis por caminhos em plantaçoes. *Interciência & Sociedade*, v. 2, n. 1, 2013.

KAEHLER, A.; BRADSKI, G. *Learning OpenCV 3: computer vision in C++ with the OpenCV library*. [S.l.]: "O'Reilly Media, Inc.", 2016.

KERAS. *BatchNormalization layer*. 2021. Disponível em: ⟨https://keras.io/api/layers/normalization_layers/batch_normalization/⟩. Acesso em: 08 jul 2021.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

KOECH, K. E. *Cross-Entropy Loss Function*. 2020. Disponível em: ⟨https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e⟩. Acesso em: 08 jul 2021.

KRAGIC, D.; VINCZE, M. *Vision for robotics*. [S.l.]: Now Publishers Inc, 2009.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, ACM New York, NY, USA, v. 60, n. 6, p. 84–90, 2017.

KRUSE, R. et al. Threshold logic units. In: *Computational Intelligence*. [S.l.]: Springer, 2013. p. 15–35.

LECUN, Y. et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, Ieee, v. 86, n. 11, p. 2278–2324, 1998.

MARENGONI, M.; STRINGHINI, S. Tutorial: Introdução à visão computacional usando opencv. *Revista de Informática Teórica e Aplicada*, v. 16, n. 1, p. 125–160, 2009.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943.

MILANO, D. de; HONORATO, L. B. *Visão computacional*. 2014. Disponível em: ⟨https://d1wqtxts1xzle7.cloudfront.net/35825905/2010_IA_FT_UNICAMP_visaoComputacional.pdf?1417697059=&response-content-disposition=inline\\%3B+filename\\%3DVISAO_COMPUTACIONAL_Palavras_Chaves.pdf&Expires=1625796335&Signature=eSrFKdj0rewoi~DCnSAldgeSmmikiXo9BslPEGrBIk9oW~uU67zw3uwGLAXOhR-vkmNWWM9xH5Kj__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA⟩. Acesso em: 08 jul 2021.

MINSKY, M.; PAPERT, A. S. *Perceptrons: An introduction to computational geometry*. [S.l.]: MIT press, 1969.

MISRA, D. Mish: A self regularized non-monotonic neural activation function. *arXiv preprint arXiv:1908.08681*, CoRR, v. 4, 2019.

MURPHY, K. P. *Machine learning: a probabilistic perspective*. [S.l.]: MIT press, 2012.

MUSTAFFA, I. B.; KHAIRUL, S. F. B. M. Identification of fruit size and maturity through fruit images using opencv-python and rasberry pi. In: IEEE. *2017 International Conference on Robotics, Automation and Sciences (ICORAS)*. [S.l.], 2017. p. 1–3.

ORAC, R. *What's new in YOLOv4?* 2020. Disponível em: ⟨https://towardsdatascience.com/whats-new-in-yolov4-323364bb3ad3⟩. Acesso em: 08 jul 2021.

PAJANKAR, A. *Raspberry Pi computer vision programming*. [S.l.]: Packt Publishing Ltd, 2015.

PANDEY, A. *Depth-wise Convolution and Depth-wise Separable Convolution*. 2018. Disponível em: ⟨https://medium.com/@zurister/depth-wise-convolution-and-depth-wise-separable-convolution-37346565d4ec⟩. Acesso em: 08 jul 2021.

PAUL, A. et al. A review on agricultural advancement based on computer vision and machine learning. In: *Emerging Technology in Modelling and Graphics*. [S.l.]: Springer, 2020. p. 567–581.

PEREIRA, A.; SIMONETTO, E. de O. Indústria 4.0: conceitos e perspectivas para o brasil. *Revista da Universidade Vale do Rio Verde*, v. 16, n. 1, 2018.

PIONEER. *McKinsey vê ganho de R$24 bi para agricultura do Brasil em 5 anos com "big data"*. 2014. Disponível em: ⟨http://www.pioneersementes.com.br/media-center/noticias/2229/mckinsey-ve-ganho-de-r24-bi-para-agricultura-do-brasil-em-5-anos-com-big-data⟩. Acesso em: 08 jul 2021.

PRABHA, D. S.; KUMAR, J. S. Performance evaluation of image segmentation using objective methods. *Indian J. Sci. Technol*, v. 9, n. 8, p. 1–8, 2016.

PROGRAMMERSOUGHT. *Understanding the route layer in yolov4*. 2019. Disponível em: ⟨https://www.programmersought.com/article/38284717937/⟩. Acesso em: 08 jul 2021.

PROGRAMMERSOUGHT. *IoU, GIoU, DIoU, CIoU loss function*. 2021. Disponível em: ⟨https://www.programmersought.com/article/27474159794/⟩. Acesso em: 11 jul 2021.

ROSEBROCK, A. *Deep learning for computer vision with python*. [S.l.]: PyImageSearch, 2017. v. 1.

ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, American Psychological Association, v. 65, n. 6, p. 386, 1958.

RUELA, A. S. Redes neurais feedforward e backpropagation. *UFOP (http://www.decom.ufop.br/imobilis/wpcontent/uploads/2012/06/03_Feedforward-e-Backpropagation.pdf)(Acedido em 3 05 2014)*, 2012.

RUGERY, P. *Explanation of YOLO V4 a one stage detector*. 2020. Disponível em: ⟨https://becominghuman.ai/explaining-yolov4-a-one-stage-detector-cdac0826cbd7/⟩. Acesso em: 08 jul 2021.

RURAL, R. G. *Governo lança edital de quase R$ 5 milhões para estimular tecnologias 4.0 no agro*. 2020. Disponível em: ⟨https://revistagloborural.globo.com/Noticias/Pesquisa-e-Tecnologia/noticia/2020/09/governo-lanca-edital-de-quase-r-5-milhoes-para-estimular-tecnologias-40-no-agro.html⟩. Acesso em: 08 jul 2021.

SAMPAIO, C. *CNN - Convolutional Neural Network*. 2020. Disponível em: ⟨http://olharcomputacional.com/CNN/⟩. Acesso em: 11 jul 2021.

SANTOS, B. P. et al. Industry 4.0: challenges and opportunities. *Revista Produção E Desenvolvimento*, Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, 2018.

SEBE, N. et al. *Machine learning in computer vision*. [S.l.]: Springer Science & Business Media, 2005. v. 29.

SHAH, D. *YOLOv4 — Version 3: Proposed Workflow*. 2020. Disponível em: ⟨https://medium.com/visionwizard/yolov4-version-3-proposed-workflow-e4fa175b902⟩. Acesso em: 08 jul 2021.

SHIRAI, Y. *Three-Dimensional Computer Vision*. Springer Berlin Heidelberg, 1987. (Symbolic Computation). ISBN 9783540151197. Disponível em: ⟨https://books.google.com.br/books?id=cv5RAAAAMAAJ⟩. Acesso em: 21 jan 2021.

SILVA, R. d.; FILHO, D. J. S.; MIYAGI, P. E. Modelagem de sistema de controle da indústria 4.0 baseada em holon, agente, rede de petri e arquitetura orientada a servicos. *XII Simpósio Brasileiro de Automação Inteligente. Natal*, 2015.

SONKA, M.; HLAVAC, V.; BOYLE, R. *Image Processing, Analysis, and Machine Vision*. [S.l.]: Cengage Learning, 2014. ISBN 9781133593607.

SZEGEDY, C. et al. Going deeper with convolutions. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2015. p. 1–9.

TAQI, A. M. et al. The impact of multi-optimizers and data augmentation on tensorflow convolutional neural network performance. In: IEEE. *2018 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*. [S.l.], 2018. p. 140–145.

TARASIUK, P.; TOMCZYK, A.; STASIAK, B. Automatic identification of local features representing image content with the use of convolutional neural networks. *Applied Sciences*, v. 10, p. 5186, 07 2020.

TECHZIZOU. *YOLOv4 VS YOLOv4-tiny*. 2020. Disponível em: ⟨https://medium.com/analytics-vidhya/yolov4-vs-yolov4-tiny-97932b6ec8ec⟩. Acesso em: 08 jul 2021.

TENSORFLOW. *tf.keras.losses.SparseCategoricalCrossentropy*. 2021. Disponível em: ⟨https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy⟩. Acesso em: 08 jul 2021.

WANG, R.; XU, J.; HAN, T. X. Object instance detection with pruned alexnet and extended training data. *Signal Processing: Image Communication*, Elsevier, v. 70, p. 145–156, 2019.

WIKIMEDIA. *Convolution arithmetic - Padding strides*. 2020. Disponível em: ⟨https://commons.wikimedia.org/wiki/File:Convolution_arithmetic_-_Padding_strides.gif⟩. Acesso em: 08 jul 2021.

YANG, K. et al. Towards fairer datasets: Filtering and balancing the distribution of the people subtree in the imagenet hierarchy. In: *Conference on Fairness, Accountability, and Transparency*. [S.l.: s.n.], 2020.

ZAREIFOROUSH, H. et al. Potential applications of computer vision in quality inspection of rice: A review. *Springer Science+Business Media*, Springer Science+Business Media, 2015.

ZEILER, M. D.; FERGUS, R. Visualizing and understanding convolutional networks. In: SPRINGER. *European conference on computer vision*. [S.l.], 2014. p. 818–833.