UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA MECÂNICA - FEMEC
ENGENHARIA MECATRÔNICA

ARTHUR MASSARU NONAKA
Orientador: Prof. Dr. Rogério Sales Gonçalves

ANÁLISE DO TREINAMENTO EM NUVEM DE REDES NEURAIS CONVOLUCIONAIS PARA LOCALIZAÇÃO DE TRAVES EM JOGO DE FUTEBOL DE ROBÔS HUMANOIDES

Uberlândia, MG
2022

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA MECÂNICA - FEMEC
ENGENHARIA MECATRÔNICA

ARTHUR MASSARU NONAKA

ANÁLISE DO TREINAMENTO EM NUVEM DE REDES NEURAIS CONVOLUCIONAIS PARA LOCALIZAÇÃO DE TRAVES EM JOGO DE FUTEBOL DE ROBÔS HUMANOIDES

Trabalho de Conclusão de Curso apresentada ao Curso de Engenharia Mecatrônica da Universidade Federal de Uberlândia como parte dos requisitos necessários para a obtenção do grau de Bacharel em Engenharia Mecatrônica.

Orientador: Prof. Dr. Rogério Sales Gonçalves

Uberlândia, MG
2022

Nonaka, Arthur.

Análise do treinamento em nuvem de Redes Neurais Convolucionais para localização de traves em jogo de futebol de robôs humanoides/ Arthur Massaru Nonaka. –, 2022-
54 p. 1 :il. (colors; grafs; tabs).

Orientador: Prof. Dr. Rogério Sales Gonçalves

Trabalho de Conclusão de Curso – Universidade Federal de Uberlândia,
Faculdade de Engenharia Mecânica - FEMEC, Engenharia Mecatrônica, 2022.

1. Redes Neurais Convolucionais. 2. Aprendizado profundo. 2. Treinamento em nuvem. 3. Futebol de robôs humanoides. I. Prof. Dr. Rogério Sales Gonçalves. II. Universidade Federal de Uberlândia. III. Análise do treinamento em nuvem de Redes Neurais Convolucionais para localização de traves em jogo de futebol de robôs humanoides

Arthur Massaru Nonaka

ANÁLISE DO TREINAMENTO EM NUVEM DE REDES NEURAIS CONVOLUCIONAIS PARA LOCALIZAÇÃO DE TRAVES EM JOGO DE FUTEBOL DE ROBÔS HUMANOIDES

Trabalho de Conclusão de Curso apresentada ao Curso de Engenharia Mecatrônica da Universidade Federal de Uberlândia como parte dos requisitos necessários para a obtenção do grau em Bacharel em Engenharia Mecatrônica.

Aprovada em Uberlândia, 17 de Janeiro de 2022.

Prof. Dr. Rogério Sales Gonçalves
Universidade Federal de Uberlândia
Orientador

Prof. Dr. Mauricio Cunha Escarpinati
Universidade Federal de Uberlândia - UFU
Examinador

Prof. Dr. Pedro Augusto Queiroz de Assis
Universidade Federal de Uberlândia - UFU
Examinador

Dedico esse trabalho à minha família que sempre me apoiou

Agradecimentos

Gostaria de agradecer primeiramente à Deus por essa oportunidade e pela superação das dificuldades.

Agradecer também à minha família e minha namorada pelo amor, incentivo, companhia e pelo suporte durante minha caminhada.

À todos os meus amigos, que fiz durante a graduação por proporcionarem tantos bons momentos e aprendizados que levarei pra vida.

A esta universidade, seu corpo docente, direção e administração que oportunizaram a janela que hoje vislumbro um horizonte superior, eivado pela acendrada confiança no mérito e ética aqui presentes.

A EDROM que me abriu portas para aprender sobre o tema e me ajudou a crescer pessoal e profissionalmente.

Ao Prof. Dr. Rogério Sales Gonçalves pela oportunidade e apoio na elaboração deste trabalho.

À todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

“É difícil pensar em uma grande indústria que não será transformada pela inteligência artificial. Isso inclui saúde, educação, meios de transporte, varejo, comunicações e agricultura. Existem caminhos surpreendentemente claros para a IA fazer uma grande diferença em todas essas indústrias” - Andrew Ng

Resumo

Esse trabalho é destinado à pesquisa de métodos de treinamento em nuvem de Redes Neurais Convolucionais para localização de traves em jogos de futebol de robôs humanoides. Foi observado em projetos a necessidade de desenvolvimento desses algorítmos barrada por questões de hardwares com baixo processamento, por isso, o objetivo é avaliar soluções já existentes na literatura, porém pouco difundidas, para que tais estruturas possam ser treinadas sem depender de equipamentos de *hardware*, o que muitas vezes limitam usuários por não terem as capacidades de processamento necessárias. A pesquisa foi feita treinando duas redes neurais em um banco de dados real e avaliando sua performance através de vídeos de teste. Embora a qualidade e eficiência dos modelos tenham se mostrado abaixo do necessário é observado a possibilidade de se otimizar e alcançar um resultado melhor.

Palavras-chave: Redes Neurais Convolucionais. Reconhecimento de objetos. Aprendizado profundo. Treinamento em nuvem.

Abstract

Work aimed at researching cloud training methods in Convolutional Neural Networks for locating goalposts in humanoid robot soccer games. The need to develop these algorithms was observed in projects, barred by hardware issues with low processing power, so the objective is to evaluate solutions that already exist in the literature, but little disseminated, so that such structures can be trained without depending on hardware equipment, which often limit users by not having the necessary processing capabilities, the research was done by training two neural networks in a real database and evaluating their performance through test videos. Although the models quality and efficiency has been shown to be below the desired level, the possibility of optimizing and achieving a better result is observed.

Keywords: Convolutional Neural Networks. Object Detection. Deep learning. Cloud training.

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas e Siglas

EDROM	Equipe de Desenvolvimento em Robótica Móvel
DL	*Deep Learning*
FEMEC	Faculdade de Engenharia Mecânica
GPU	*Graphics Processing Unit*
HSV	*Hue, Saturation, Value*
IA	Inteligência Artificial
ML	*Machine Learning*
RGB	*Red, Green, Blue*
RN	Rede Neural
RNA	Rede Neural Artificial
RNC	Rede Neural Convolucional
RNR	Rede Neural Recorrente
UFU	Universidade Federal de Uberlândia

Lista de Símbolos

α	Letra grega Alfa

Sumário

1 Introdução

Desde o começo dos estudos na área envolvendo IA (Inteligência Artificial) observam-se tecnologias cada vez maiores e mais complexas. Por exemplo, o ramo que trata as *Deep Neural Networks* (redes neurais profundas, traduzindo do inglês) vem dando origem a arquiteturas diferentes e mais eficientes, sendo que em geral quanto maior e mais profunda a rede for, mais variáveis para calibrar demandando um maior poder de processamento. (Szegedy et al. (2014)).

Hoje as maiores empresas de tecnologia (Google, Microsoft, Facebook por exemplo) tem a sua disposição supercomputadores capazes de processar milhões de cálculos por segundo possibilitando treinar uma Rede Neural (RN) de forma relativamente rápida. Tais algoritmos são utilizados para as mais diversas funções, tais como reconhecimento de faces, fala, escrita, dentre outros.

Olhando para o treinamento tradicional da Redes Neurais Convolucionais (RNC), que é a categoria dentro das Neural Networks (Redes Neurais) utilizada para tratamento de imagens, nota-se que é um processo que requer tempo e um imenso poder de processamento. Tornando o processo inviável para muitos que querem se aprofundar nessa área mas são barrados por questões de *hardware*. (Szegedy et al. (2014)).

Portanto fez-se necessário encontrar alternativas que supram essas dificuldades seja na forma de treinamento ou na metodologia utilizada. Algumas das soluções aparecem na forma de evoluções de *hardware*, adquirir por exemplo uma undade GPU (*Graphics Processing Unit*, unidade de processamento gráfico) que por sua vez demanda dinheiro.

Nesse trabalho vamos demonstrar algumas soluções encontradas para realizar o treinamento de uma rede capaz de identificar objetos sem a necessidade de *hardwares* potentes, se utilizando de duas ferramentas, treinamento em nuvem e transferência de aprendizado. Conceitos que serão melhor explicados ao longo desse relatório.

1.1 Justificativa

O projeto realizado na categoria Humanoide da EDROM (Equipe de Desenvolvimento em Robótica Móvel) que é a equipe de robótica da Faculdade de Engenharia Mecânica (FEMEC) da Universidade Federal de Uberlândia (UFU), consiste em desenvolver robôs humanoides para participar de campeonatos de futebol, nas quais, de forma autônoma, os robôs devem entrar em campo, e percebendo o ambiente a sua volta devem marcar gols e se defender de gols adversários. Fazendo o uso apenas de sensores humanamente possíveis, ou seja, percepções que um ser humano consegue desenvolver, visão, audição, olfato, senso direção são permitidos, enquanto que sensores infravermelhos, magnéticos por sua vez são proibidos.

Para desempenhar as funções necessárias para marcar gols, é necessário que o robô seja capaz de perceber o ambiente ao redor, bola, traves, outros robos. E o principal método utilizado atuamente para desempenhar tal tarefa, é a visão computacional, mais precisamente utilizando redes neurais convolucionais. Para implementar tais algoritmos, até certo tempo, dependeu-se muito da capacidade de processamento dos *notebooks* pessoais dos membros da equipe, assim, quando a pessoa saísse não era possível melhorar ou até desenvolver novos projetos.

Portanto, esse trabalho visa desenvolver uma ferramenta para treinar e testar novos algoritmos em nuvem, sem depender de melhorias de *hardware*, podendo ser executada por qualquer computador com acesso à internet.

O treinamento de algorítmos de IA em nuvem já é amplamente difundido dentro da comunidade envolvida nos estudos dessa área. Já é possível encontrar diversos tutoriais de como realizar esse treinamento, portanto a pesquisa em questão não se trata de uma inovação na área de visão computacional ou aprendizado profundo, mas sim uma melhoria para o projeto da equipe, uma vez que não existia nenhuma forma de realizar o feito e nem conhecimento a respeito do tema.

O estudo nesse caso foi direcionado para a localização de traves, mas deseja-se a flexibilização desse projeto de forma que seja possível utilizá-lo em outras aplicações, tais como, reconhecimento de bolas, robôs e outros objetos.

1.2 Objetivos

O objetivo principal desse trabalho é desenvover uma ferramenta que possibilite o treinamento de RNCs para o reconhecimento de traves de futebol em um jogo de robôs humanoide a partir da nuvem. Os objetivos específicos são:

1. Estudar os problemas de reconhecimento de imagens usando redes neurais;
2. Estudar os diferentes parâmetros de treinamento que compõem uma rede neural artificial e seus efeitos;
3. Comparar diferentes modelos presentes na literatura;
4. Buscar compreender os resultados e se aprofundar no tema de inteligência artificial;
5. Ajudar o desenvolvimento dos projetosa acadêmicos desenvolvidos dentro da faculdade.

As características buscadas no projeto são:

1. Um projeto flexível de forma que seja possivel utilizá-lo para outros fins, treinar a mesma estrutura para outros objetos por exemplo;

2. Treinar uma rede rápida e eficiente para ser utilizada em problemas em tempo real com certa confiabilidade;

3. Praticidade de manuseio e fácil entendimento, não sendo necessária uma experiência muito avançada em desenvolvimento de códigos de programação,

4. Que seja *plug and play*, sendo necessária somente a entrada dos dados (fotos e rótulos) por parte do usuário.

2 Fundamentação Teórica

Neste capítulo é apresentado uma contextualização da pesquisa com um resumo das discussões já feitas por outros autores sobre o assunto abordado e os conceitos principais relativos ao tema.

2.1 Imagem

Antes de abordar todos os conceitos básicos de uma rede neural em si, deve-se entender como as imagens são tradas e como a RN a enxerga. Vamos considerar uma imagem qualquer com 100 pixels de altura e 100 pixels de largura, conforme a Figura 2.1.

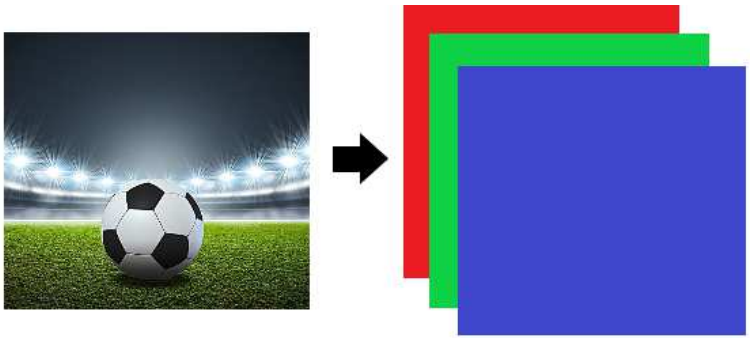
Figura 2.1 – Representação do número de pixels em uma imagem

Fonte: Compilação do autor, edição a partir da imagem disponível na internet. Disponível em: https://br.pinterest.com/pin/748793875525826934. (Acesso em 25 de Setembro de 2021).

A imagem acima pode ser considerada uma matriz bidimensional de ordem 100x100. Porém para representarmos uma imagem colorida, deve-se observar que todo o espectro de cores pode ser expressado por uma combinação de vermelho, verde e azul. Temos então que a rede não recebe somente uma única matriz, mas três matrizes, de mesmas dimensões que a original (100 pixels de altura e 100 de largura) porém monocromáticas na visualização RGB (do inglês, *Red/Green/Blue*) cada uma contendo uma cor. Dessa forma, se concaternarmos todas elas a imagem original é formada. Como visto na Figura 2.2.

Existem outros tipos de visualização muito utilizados na área de visão computacional, como por exemplo o HSV (do ingles, *Hue* - matiz, *Saturation* - saturação e *Value* - valor) (Pedrini (2021)), mas por ora vamos nos concentrar no RGB para melhor compreensão.

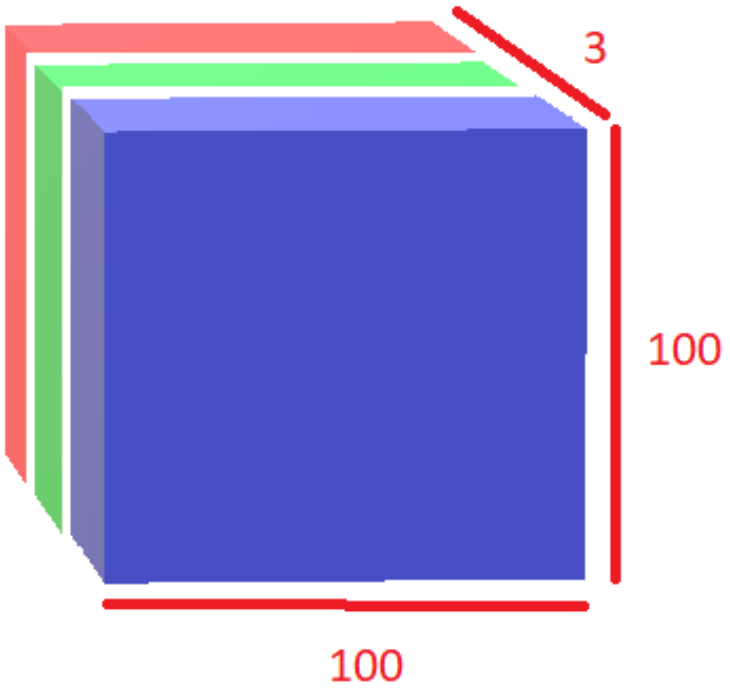
Figura 2.2 – Representação RGB de uma imagem



Fonte: Compilação do autor, edição a partir da imagem disponível na internet. Disponível em: https://br.pinterest.com/pin/748793875525826934/ (acesso em 25 de Setembro de 2021)

Esse conjunto de matrizes no fim pode ser representada como uma única matriz tridimensional de ordem 100x100x3, conforme ilustrado na Figura 2.3.

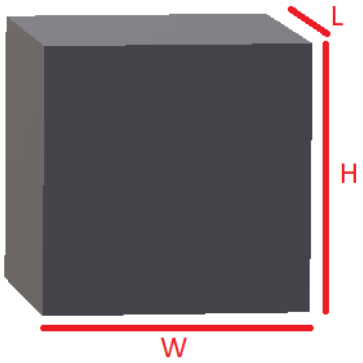
Figura 2.3 – Representação RGB de uma imagem



Fonte: Própria

Para efeitos de visualização vamos retratá-la como um cubo de dimensões H (altura), W (largura), L (comprimento ou profundidade), conforme exemplificado na Figura 2.4.

Figura 2.4 – Representação de uma imagem na forma de matriz tridimensional



Fonte: Própria

2.2 Aprendizado profundo

Redes neurais dependendo de sua profundidade (número de camadas ocultas) e da quantidade de parâmetros são classificadas como redes neurais profundas.

Por sua vez, aprendizado profundo (ou DL, do inglês *Deep Learning*) é o nome dado a um ramo dentro do aprendizado de máquina que busca treinar uma rede neural profunda de forma que ela consiga reconhecer padrões escondidos dentro de um determinado grupo de dados. (Pacheco e Pereira (2018)).

Segundo Goodfellow, Bengio e Courville (2016) os algoritmos de DL devem ser capazes de aprender sem nenhuma interferência ou supervisão, segundo suas próprias regras o que considerar e o que ignorar para chegar ao resultado esperado. Esse método de treinamento vêm se tornando cada vez mais útil para as grandes empresas de tecnologia (Google, Facebook, Amazon, etc) devido à imensa quantidade de dados disponível, inviabilizando a análise por um ser humano. Além de serem auxiliadas pela crescente capacidade computacional dos processadores Goodfellow, Bengio e Courville (2016).

2.3 Redes neurais

As redes neurais artificiais (RNA) ou somente redes neurais, como são chamadas, são um ramo de estudo dentro da área da IA, que aborda o funcionamento de estruturas computacionais, composta por nós ou neurônios interconectados que imitam o funcionamento do cérebro humano. Esses neurônios são capazes de a partir do dado bruto reconhecer padrões, guardar valores na memória e aprimorar sua capacidade em um processo de aprendizagem devido a um processo de treinamento. (Data Science Academy (2019)).

Segundo (HAYKIN, 1999 apud SAGE, 1990) um sistema de inteligência artificial deve apresentar três características: deve ser capaz de armazenar conhecimento; utilizar esse conhe-

cimento para resolver algum problema específico; e aprender mais conhecimentos através do treinamento. Portanto uma RN deve apresentar essas três características básicas, e irá utilizar dos neurônios e suas interligações para isso.

O desenvolvimento das RNs começa no ano de 1943 com o trabalho de McCulloch e Pitts. ((HAYKIN, 1999)) E de lá pra cá vem ganhando força devido à grande capacidade computacional dos processadores atuais, alimentados pela imensa quantidade de dados a serem analisados. Atualmente, existem redes neurais trabalhando em diversas tarefas do dia a dia, tais como reconhecimento e tradução de fala nos celulares, reconhecimento de padrões de consumo nas redes sociais, reconhecimento de imagens nos carros autônomos e tantos outros.

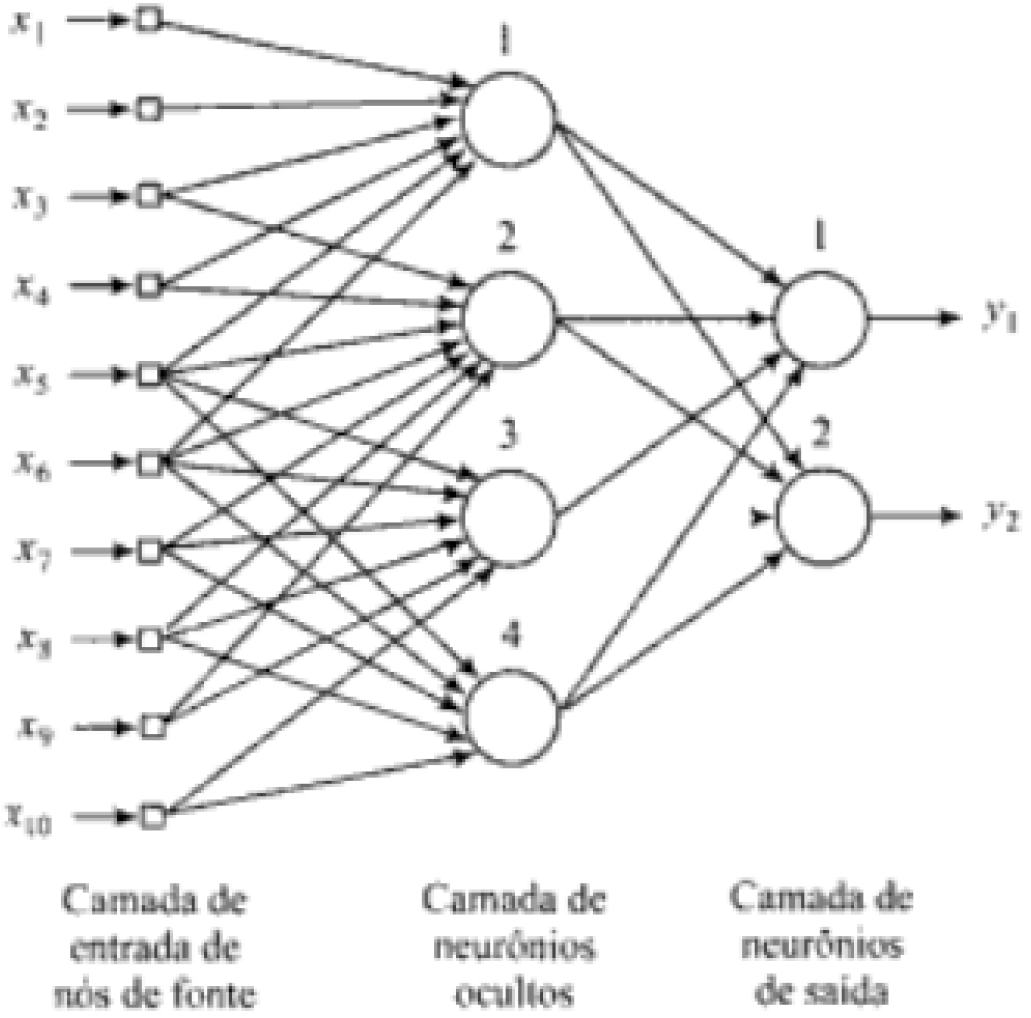
Antes de explicar a estrutura de uma rede neural cabe explicar que não é o propósito desse trabalho entrar em detalhes específicos e detalhados sobre todas as variáveis e cálculos envolvidos na construção de uma rede e seu treinamento, apenas uma explicação básica sobre os fundamentos mais importantes para que o leitor entenda a idéia principal e o desenvolvimento do projeto proposto.

Em resumo, a estrutura de uma rede neural (conforme visto na Figura 2.5), é formada por nós ou neurônios, que é a menor unidade de uma RN. Dentro desses neurônios temos funções específicas que permtirão que a rede analise os dados, fazeendo cálculos, armazenando valores e previsões. Essas unidades estão interligadas umas às outras de forma a compartilhar informações e dados entre elas, podendo estar sozinhas como no caso de um perceptron, ou organizadas em camadas.

Camadas são conjuntos de nós a desempenharem uma função, sendo a primeira a camada de entrada dos dados, podendo ser um vetor, uma imagem, um trecho de áudio, por exemplo. A última se refere à de saída, que é a resposta que deseja-se obter da rede. E por fim tem-se as camadas ocultas, ou intermediárias, que vão fazer todo o processamento dos dados de entrada até chegar na camada de saída. Assim, podemos ter uma rede com N camadas, sendo que em geral, quanto maior o valor de N mais profunda a rede, maior o poder de análise mas por outro lado maior a demanda de tempo e processamento.

A Figura 2.5 mostra a representação clássica de uma rede neural onde cada camada é conectada com a seguinte formando uma rede, cujos parâmetros são passados da camada anterior e tratadas antes de serem repassadas para a seguinte em um processo de alimentação direta. Existem vários tipo dessas estruturas, com funções e tamanhos variados. Algumas delas serão apresentadas mais pra frente. Na figura, os parâmetros x1, x2 ... x10, são as variáveis de entrada, enquanto que y1 e y2 são as saídas da rede, ou seja o que deseja-se prever.

Figura 2.5 – Representação clássica de uma rede neural



Fonte: HAYKIN, 2000, p.55.

A estrutura original das redes neurais passou por várias transformações e funções diferentes, cada uma com suas peculiaridades e desenvolvidas para desempenhar papéis específicos no mundo das IAs. Dentre elas cabe citar as Redes Neurais Recorrentes (RNR) utilizados para problemas com características de correlação temporal, como previsão do tempo, Redes Neurais Convolucionais (RNC) que são utilizadas amplamente para reconhecimento de imagens (estrutura abordada mais profundamente nesse trabalho) e as autoencoders que podem ser utilizados para a compactação de dados. Vale citar também que as pesquisas envolvendo as diversas estruturas de RNs evoluem a cada dia, e sempre estão surgindo variantes dessas estruturas para os mais diversos fins.

2.3.1 Redes Neurais Convolucionais

As RNCs são uma classe dentro do mundo das redes neurais muito utilizadas no reconhecimento de imagens, ela recebe esse nome devido à movimentos de convolução que acontecem nas camadas ocultas dentro da estrutura da rede. (Data Science Academy (2019)).

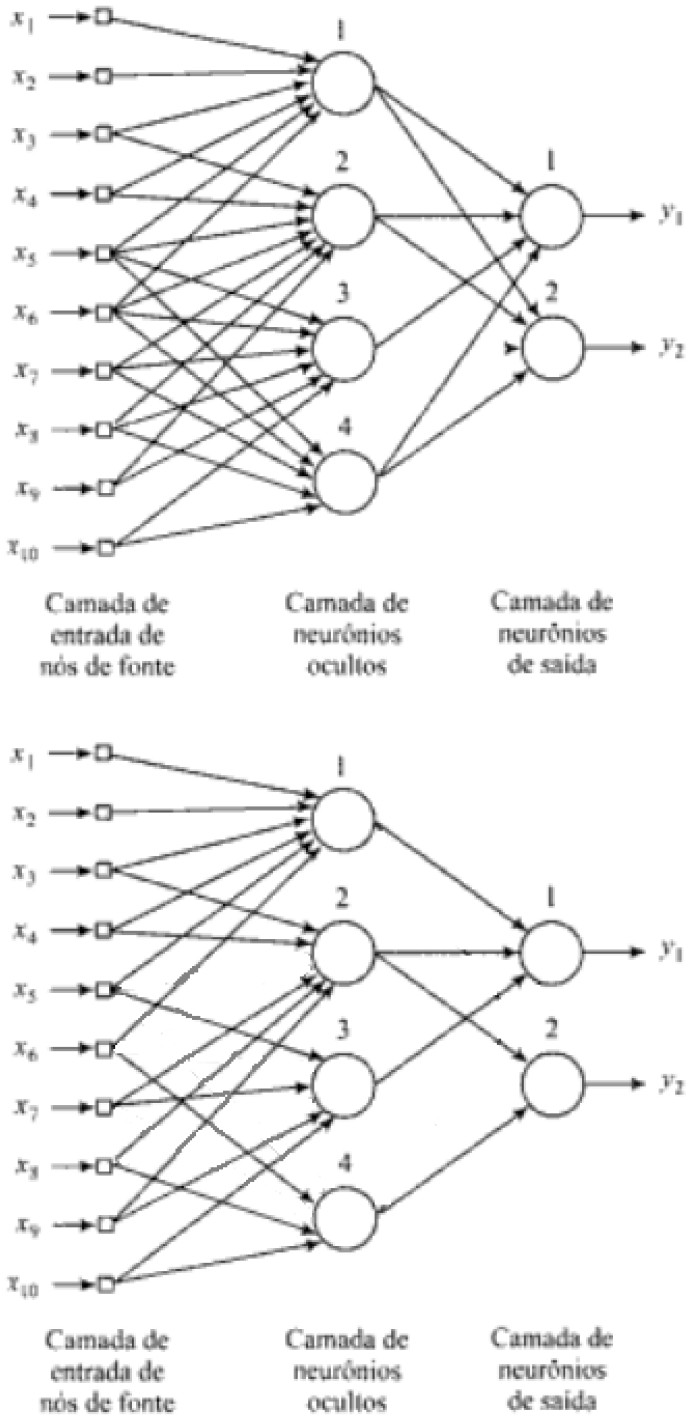
A estrutura das redes convolucionais segue o modelo apresentado anteriormente, assim como vários outros modelos de RN. Podem apresentar diversos tipos de camadas, sendo as mais comuns:

1. Camadas de convolução: onde ocorrem as convoluções em si, as imagens passam por filtros para identificar características básicas como formas, traços, padrões de cores. O resultado dessa "filtragem"vai compor a camada seguinte, e assim por diante.

2. Camadas de *pooling*: utilizado para ressaltar as caractarísticas principais de uma camada, de forma a passar para a camada seguinte somente as partes consideradas importantes economizando tempo e poder de processamento.

Outra diferença importante a ser citada é que ao contrário das redes neurais tradicionais (como o perceptron por exemplo), as RNCs não são totalmente conectadas, ou seja, os neurônios de uma camada não são ligados a todos os neurônios da camada seguinte, e isso é importante pois reduz de forma considerável a quantidade bruta de parâmetros e variáveis a serem processadas. A figura 2.6 abaixo mostra a diferença de redes completamente conectadas e parcialmente conectadas.

Figura 2.6 – Diferença entre redes totalmente (1ª figura) e parcialmente conectadas (2ª figura)



Fonte: Compilação do autor a partir de: (HAYKIN, 2000, p.55.)

Atualmente na literatura já é possível encontrar RNCs profundas com centenas de camadas e funções diferentes não se limitando as duas supracitadas. (Shaikh (2018)).

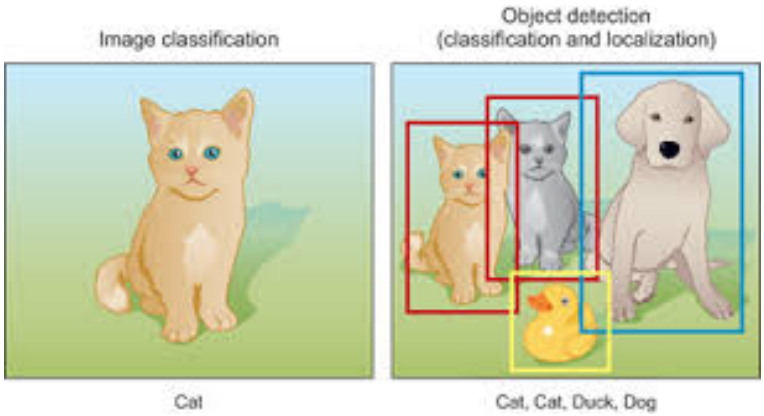
As redes convolucionais podem ser utilizadas para vários fins dentro da área de visão computacional, e a determinação de qual a função desejada interfere em toda a estrutura e no treinamento da rede, as principais são:

1. Classificação: o problema mais básico dentro da visão computacional, consiste em tentar classificar uma imagem segundo uma gama de saídas esperadas, como exemplo clássico dentro do universo da IA temos a classificação de cães e gatos.

2. Detecção: o mais complexo, consiste em não só dizer quais objetos estão na imagem mas também onde estão, e qual a localização de cada um dentro da matriz imagem.

Uma representação de cada uma das funções supracitadas se encontra na figura 2.7.

Figura 2.7 – Diferença entre classificação e detecção de objetos



Fonte: Disponível em https://livebook.manning.com/book/deep-learning-for-vision-systems/chapter-7/v-8/ (acesso em 25 de setembro de 2021)

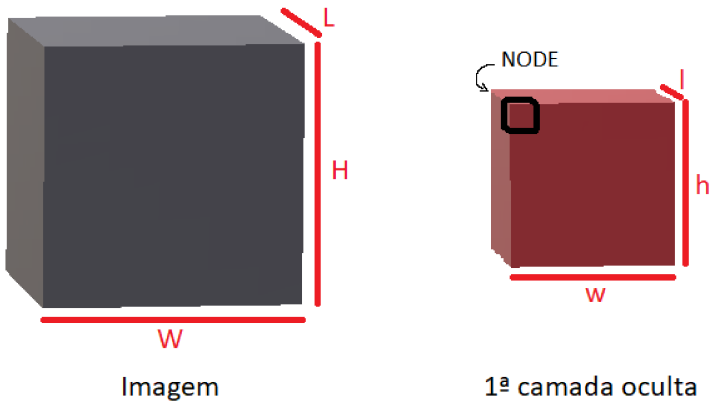
O objetivo desse trabalho é treinar uma rede de forma remota para fazer a detecção de traves de futebol em um jogo de robôs humanoides.

2.3.1.1 Camadas de convolução

Explicando um pouco como se dão as convoluções e o por quê esse tipo de processo é tão útil para o tratamento de imagens.

Como visto anteriormente, uma imagem, pode ser representada como uma matriz de ordem (H x W x L), sendo L o número de canais, no caso da visualização RGB são 3. Cada camada oculta (por onde a imagem vai ser passada) também é uma matriz de ordem (h-altura x w-profundidade x l-comprimento). Vide Figura 2.8.

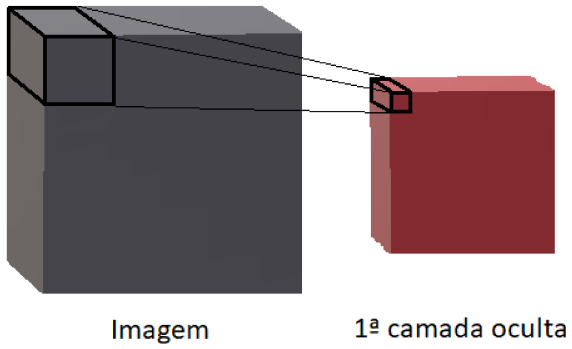
Figura 2.8 – Representação das camadas ocultas



Fonte: Própria

Cada um desses neurônios "enxerga"uma parcela da imagem (demonstrado na Figura 2.9) de forma que o conjunto dos neurônios irá varrer toda a matriz de entrada. Ou seja, se pegarmos um neurônio da primeira camada oculta, veremos que ele está ligado somente a uma vizinhança de pixels. Vide Figura 2.9

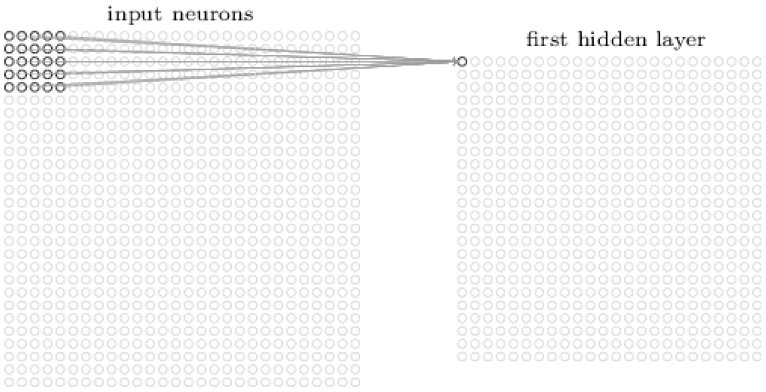
Figura 2.9 – Camadas ocultas: ligações



Fonte: Própria

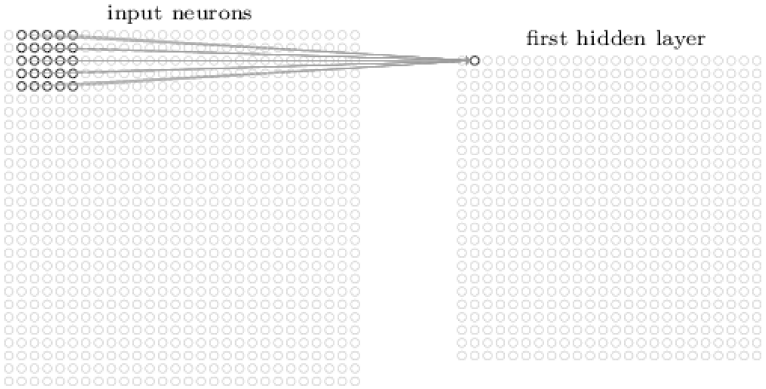
Essa "vizinhança"a qual o neurônio oculto se conecta é chamado de campo receptivo local (Data Science Academy (2019)), cada *node* de uma camada oculta é totalmente conectado a todos os *nodes* das camadas anteriores dentro do seu respectivo campo receptivo, determinando o número de ligações entre camadas. Para facilitar o entendimento vamos voltar a camada para uma matriz bidimensional (demonstrado nas Figuras 2.10 e 2.11), mas lembre-se que esse processo ocorre tridimensionalmente.

Figura 2.10 – Campo receptivo local: primeiro *(*node) da primeira camada oculta



Fonte: Disponível em https://www.deeplearningbook.com.br/campos-receptivos-locais-em-redes-neurais-convolucionais/ (acesso em 25 de setembro de 2021)

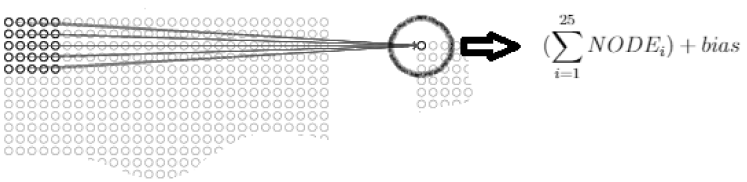
Figura 2.11 – Campo receptivo local: segundo *(*node) da primeira camada oculta



Fonte: Disponível em https://www.deeplearningbook.com.br/campos-receptivos-locais-em-redes-neurais-convolucionais/ (acesso em 25 de setembro de 2021)

Por sua vez cada ligação possui um peso, e o neurônio oculto vai ter um parâmetro geral chamado *bias* (traduzindo para o português, viés). Assim o *(*node) oculto sofre influência de todos os neurônios no seu campo receptivo conectado na camada anterior, e de uma outra variável chamada viés. Demonstrado na Figura 2.12.

Figura 2.12 – Representação do node



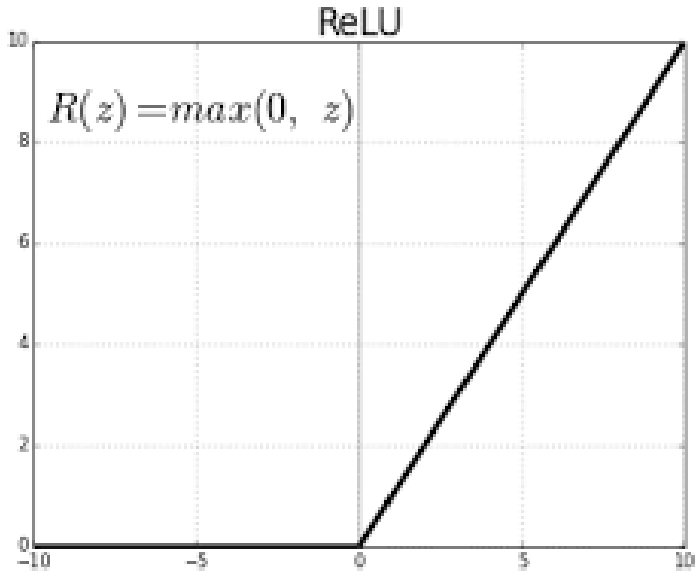
Fonte: Própria

Todos esses valores foram passados por uma função de ativação, que se trata de uma função não linear, para que a rede saia da linearidade e consiga aprender outras funcionalidades, existem diversos tipos de funções de ativação, tais como Relu (Figura 2.13), Leaky ReLU (Figura 2.14), softmax, tanh dependendo da necessidade da rede, (Data Science Academy (2019)). Para esse projeto utilizamos o ReLU,cuja vantagem sobre outras funções de ativação é de não ativar todos os neurônios ao mesmo tempo. Assim quando uma entrada é negativa ela será convertida em zero e o neurônio não será ativado, (Data Science Academy, 2019). Isso é demonstrado na Figura 2.13:

1. ReLU:

$$y = max(0, z) \tag{2.1}$$

Figura 2.13 – Gráfico ReLu



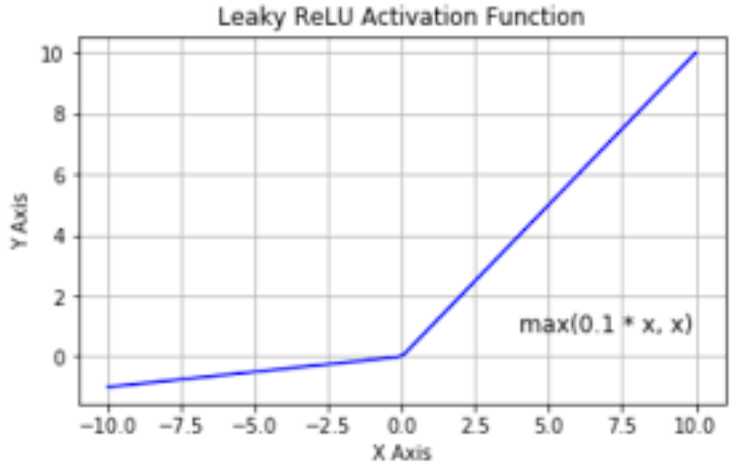
Fonte: Disponível em https://www.researchgate.net/figure/Grafico-da-funcao-de-ativacao-ReLU-Conforme-os-valores-se-aproximam-de-zero-menos_fig4_342369912 (Acesso: 25 de Setembro de 2021)

2. Leaky ReLU:

$$y = max(\alpha x, x) \tag{2.2}$$

Onde α é um número muito pequeno tendendo a zero.

Figura 2.14 – Gráfico Leaky ReLu



Fonte: Disponível em https://medium.com/escueladeinteligenciaartificial/funciones-de-activaci%C3%B3n-para-redes-neuronales-de00fefb7150 (Acesso: 25 de Setembro de 2021)

O processo de convolução propriamente dito acontece como se o campo receptivo local "deslizasse"horizontalmente na imagem, de forma que cada região será analisada por um único neurônio até que se chegasse ao limite da imagem tocando a borda. Para exemplificar esse processo imagine que o campo receptivo vai saltar uma casa para a direita, e acionar o *(*node) seguinte na camada oculta. Esse processo acontece até que toda a imagem seja varrida pelo campo receptivo e cada neurônio oculto tenha sido acionado (Data Science Academy (2019)).

Durante o treinamento existem alguns valores que podem determinar a qualidade e a velocidade do resultado final, são chamados hiperparâmetros, os principais são o *stride* (passo) e o *padding* (preenchimento) esses são abordados em estudos mais aprofundados sobre o assunto e existem maneiras de determinar quais são os melhores valores para treinar a rede. Todavia, não vamos focar nesses parâmetros nessa pesquisa bastando saber que eles existem.

Outra ferramenta muito útil para as RNCs é o compartilhamento de pesos entre neurônios ocultos, ou seja, no exemplo dado, todos os neurônios de uma camada oculta terão os mesmos pesos e o mesmo *bias*, valores que serão atualizados igualmente conforme o treinamento avança. Isso é importante para reduzir drasticamente o número de valores e de operações a serem realizadas, economizando energia e tempo. Essa característica determina que a imagem irá passar por um filtro, formando um mapa de características ou *feature map* traduzido do inglês (Data Science Academy (2019)).

Um questionamento muito comum em relação ao processo de treinamento é por que utilizar essa abordagem para executar o processo de reconhecimento de imagens? Não seria mais correto se cada neurônio da rede se conectasse a todos os pixels? Assim poderia processar a imagem como um todo e não somente uma parte.

A resposta para essa pergunta se dá devido a dois motivos principais, primeiro pode ser

explicado no processo de reconhecimento de um objeto por uma pessoa. Quando uma pessoa vê a foto de um cachorro, por exemplo, por mais trivial que aparente ser, ela não olha a imagem como um todo e conclui automaticamente que se trata de uma espécie canina. O processo de reconhecimento se dá por partes, ela busca características predefinidas e aprendidas em experiências anteriores que definem o animal, quatro patas, rabo, focinho, orelhas, pelos. Todas essas observações são feitas em um piscar de olhos para que o observador conclua que se trata de um cachorro. É dessa forma que a rede neural irá aprender, ela deve reconhecer características básicas, *feature maps*, e depois formar a figura do todo para poder dar resposta correta. São os mapas característicos que vão "aprender"a desempenhar essa função. Primeiro será treinado a encontrar características básicas, um traço, uma sombra, uma curva. Depois com a associação desses parâmetros básicos serão formados características mais complexas, até que a imagem como um todo seja reconhecida. Ou seja, não interessa à rede em primeiro momento como é o objeto como um todo, mas sim as características básicas que levam a figura a ser classificada como o objeto desejado.

No fim uma camada oculta de convolução é o conjunto de "N"*feature maps*, cada um com seus pesos, parâmetros e *bias* próprios. O valor de N varia e depende da rede desejada, mas de forma geral quanto maior o número de mapas maior a capacidade da rede em identificar características úteis para o objetivo proposto e mais tempo e poder de processamento é necessário para treiná-la.

Por isso as RNCs são em geral redes profundas, com dezenas de camadas ocultas e cada camada contendo dezenas de mapas de características, pois assim cada camada deve aprender a reconhecer padrões básicos, começando com um traço, ou um sombreamento, até formar uma figura, e no fim o objeto como um todo.

Já o segundo motivo se dá devido à economia computacional, digamos que em uma imagem de 100x100, colorida (3 canais), esteja conectada a todos os neurônios. Para cada unidade de processamento teríamos trezentos mil (100x100x3) parâmetros a serem analisados, e se tivermos 10 neurônios na primeira camada, o tamanho dos dados passa para três milhões! Agora imaginem o gasto de poder computacional utilizado para imagens em 4k, com uma resolução de 3840 x 2160 pixels, além de uma primeira camada com centenas de neurônios.

Por isso ao se conectar somente com uma vizinhança a rede ao mesmo tempo que consegue aprender de forma mais eficiente, economiza uma quantidade considerável de energia e tempo no processo.

2.3.1.2 Camadas de *pooling*

As camadas de *pooling*, ou camadas de agrupamento, são camadas ocultas dentro de uma RNC que geralmente vêm depois de uma camada de convolução. O funcionamento dela é mais simples se comparado à primeira. O processo de *pooling* tem a função de simplificar um *feature map*, de forma a destacar as características mais fortes, e ao mesmo tempo reduzir o número de

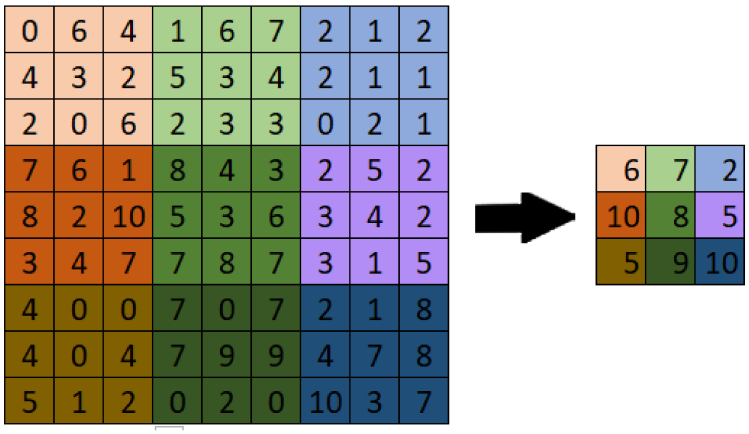
parâmetros a serem tratados nas próximas camadas (Data Science Academy (2019)).

Por exemplo, vamos supor que um filtro dentro da primeira camada oculta aprendeu a localizar traços verticais, o que a camada de *pooling* vai fazer é olhar o feature map e reforçar somente as características desejadas e reduzir a complexidade e o tamanho da camada seguinte.

Como ela faz isso? Como o *pooling* funciona propriamente falando? Bom, existem diversos tipos de agrupamento, vamos passar pelo mais comum para exemplificar. Digamos que o projetista da rede neural escolheu o *pooling* de ordem 3x3, nesse exemplo vamos usar o chamado *MAX-pooling*.

Assim como nas camadas de convolução, cada *node* da camada de agrupamento vai se conectar somente a um grupo de pixels, no nosso exemplo cada neurônio enxerga um grupo de 3 por 3, ou seja 9 *nodes* da camada anterior. O que esse neurônio vai fazer é olhar a entrada e identificar valor mais alto dessa vizinhança, em outras palavras, ele vai observar se nessa área existe a característica desejada. No fim, somente os valores mais representativos, vão ser passados para as próximas camadas, formando um mapa de agrupamento. Além de reforçar as características desejadas, o *pooling* também simplifica, e reduz, o número dos parâmetros processados, no exemplo supracitado, tivemos uma entrada de 81 parâmetros e 9 saídas. Exemplo mostrado na Figura 2.15.

Figura 2.15 – Max Pooling



Fonte: Própria

Um ponto importante a ser citado, é que, para cada *feature map* existe uma mapa de agrupamento, ou seja, se na primeira camada de convolução utilizamos 50 mapas característicos, na camada seguinte de *pooling* teremos 50 mapas de agrupamento.

Essas duas camadas estudadas trabalham geralmente em duplas, primeiro uma camada de convolução para determinar características importantes na imagem formando um *feature map*, em seguida uma camada de *pooling* para agrupar e reforçar as principais características para formar um mapa de agrupamento. Depois o processo se repete de forma a encontrar atributos mais complexos.

2.4 Processo de treinamento

Vimos os principais tipos de camadas ocultas, suas funções e uma explicação breve de como elas funcionam, agora vamos abordar o processo de treinamento da rede em si, uma das etapas mais importantes para o projeto de uma rede neural eficiente.

O treinamento de um algorítmo de inteligência artificial pode ser classificado de acordo com os dados de entrada, podendo ser supervisionados ou não supervisionados (Haykin (1999)). A diferença entre eles é que no treinamento supervisionado o banco de dados de entrada deve ser previamente rotulado, ou seja, apresentar alguma saída esperada para comparação da eficiência da rede. Enquanto o não supervisionado trata somente dos dados brutos a fim de classificar em grupos dependendo das características observadas sem nenhuma interferência externa.

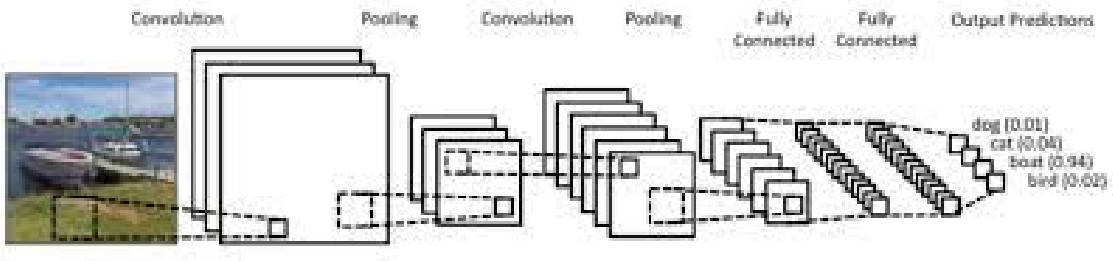
Para o treinamento de redes neurais convolucionais utilizaremos o aprendizado supervisionado. Isso quer dizer que deve-se classificar os dados de entrada de forma que a rede consiga identificar os objetos necessários e comparar com as respostas esperadas. Esse processo varia dependendo da função da rede e isso interfere diretamente na última camada da RNC. Por exemplo, já vimos que as RNCs podem ser treinadas para desempenhar diferentes atividades: classificação, localização. Os rótulos devem ser compatíveis com o problema previsto, por exemplo, para o clássico problema de classificação entre cães e gatos, nossa entrada seriam as imagens dos animais e os rótulos seriam a classificação simplesmente, se adotarmos uma classificação binária, poderíamos ter dois *nodes* de saída, um para cada animal, de 0 a 1, sendo o maior a resposta da rede, o mesmo raciocínio se dá para mais de duas saídas.

Já para o problema de localização, o algorítmo não deve apenas reconhecer o objeto como também saber onde ele se encontra dentro da imagem, mas como fazer isso? Imaginando que a imagem tem coordenadas (x, y) a saída da rede deve ter as coordenadas da região de interesse mais um *(*node) pra cada classe dentro do treinamento, assim um dos possíveis formatos de saída seria:

1. Classe - determina a classe observada, para um problema de multiclasses cada uma recebe uma ID, por exemplo, para detectar objetos presentes no trânsito, nossa rede precisaria reconhecer pedestres, carros, sinalização, semáforos, etc.
2. Coordenada X inicial - refere-se ao X inicial do objeto detectado.
3. Coordenada Y inicial - refere-se ao Y inicial do objeto detectado.
4. Coordenada altura - refere-se à altura do objeto detectado.
5. Coordenada largura - refere-se à largura do objeto detectado.

Um esquema simplificado demonstrando a estrutura de uma rede neural convolucional, desde a entrada até a saída, Figura 2.16

Figura 2.16 – Saída das RNC para detecção



Fonte: Disponível em https://matheusfacure.github.io/2017/03/12/cnn-captcha/ (acesso em 25 de setembro de 2021)

As representações de coordenadas podem mudar dependendo do projeto, pode-se utilizar tanto coordenadas iniciais e finais, altura e largura, como a posição do centro, enfim, basta que seja possível localizar o objeto dentro da imagem dada.

Assim como a saída, a entrada de dados para treinamento de uma rede voltada para localização deve apresentar todas essas informações para guiar o aprendizado do algoritmo, e essas informações estão presentes nos rótulos, mais para frente vamos tratar sobre como rotular nossos dados de treinamento e quais as ferramentas disponíveis que auxiliam nesse processo.

Vimos que para um problema de localização a saída fica um pouco mais complexa, sendo necessária seis *nodes* de saída no mínimo para performar uma processo com qualidade. Para projetos cujo objetivo é localizar mais de um objeto ao mesmo tempo, a abordagem deve mudar um pouco mas em geral segue essa lógica.

Já vimos do que se trata o treinamento supervisionado, e como rotular as imagens para que nossa rede consiga identificar os objetos propostos. Agora vamos falar um pouco sobre o processo de aprendizado e como a RNC vai tratar os dados rotulados inseridos.

2.4.1 Aprendizado por retropropagação

Nessa parte vamos ver como se dá o processo de aprendizagem de uma rede, quais são os caminhos que levam ela sair do zero e aprender a localizar objetos.

Vimos que para treinar uma RNC precisamos de um banco de dados rotulado. A partir desses dados a rede irá rodar dois processos principais diversas vezes até conseguir desempenhar uma localização eficiente. Esses dois processos são a propagação direta e a retropropagação.

2.4.1.1 Propagação direta

O processo de propagação direta se dá, como o próprio nome diz, de forma que as imagens do banco de dados passem por toda a rede, através das camadas de convolução e de *pooling* até chegar no final, onde apresentará uma saída.

Dentro de cada neurônio e suas conexões existem pesos, variáveis que devem ser calibradas durante o treinamento, esses valores serão inicializados aleatoriamente muito próximos de zero.

Então, no processo de propagação direta os pesos são iniciados, e recebem as imagens para fazer o processamento, os mapas de característica e de agrupamento são gerados, até chegar nos *nodes* de saída. O valor resultante é comparado aos rótulos carregados previamente, obtendo-se o erro, que é a comparação da resposta obtida com a resposta esperada.

Existem diversas funções para calcular o erro, dependendo de cada projeto e funcionalidade. Em geral é importante que os erros sejam diferenciáveis, para ser possível calcular o gradiente da função. O gradiente permite descobrir qual o sentido para diminuir o erro, por isso a função deve ser diferenciável.

2.4.1.2 Retropropagação

No processo de retropropagação ou propagação inversa, o erro calculado na propagação direta é utilizado para fazer a calibração dos pesos de forma inversa, partindo das camadas de saída, até as primeiras camadas.

Como é feita a calibração dos pesos? Depois de se calcular o valor da função erro, o objetivo é diminuir esse valor, para isso calcula-se o gradiente da função (por isso é necessário que seja diferenciável). A partir do gradiente a rede usa o valor do gradiente do erro para atualizar os parâmetros, até a primeira camada.

Esses dois processos de propagação direta e propagação inversa são repetidos diversas vezes de forma iterativa, até que o erro fique abaixo de um limite, mostrando que todos os valores foram calibrados e a rede foi treinada.

2.4.2 Transferência de aprendizado

O processo de treinamento de uma rede neural profunda requer uma quantidade enorme de tempo, processamento e dados. (Data Science Academy (2019)). As RNCs mais profundas atualmente possuem milhões de parâmetros distribuídos entre dezenas de camadas ocultas, sendo necessário um banco de dados com milhões de imagens para alcançar um estado da arte. Além disso os grandes *players* responsáveis pelo seu desenvolvimento e treinamento contam com GPUs e servidores potentes dedicados ao processo de aprimoramento dessas estruturas.

Mas e se uma pessoa quiser treinar uma rede dessa com boa qualidade a partir do zero no computador em casa? Ela provávelmente vai esbarrar em dificuldades relacionados a poder computacional, que eleva em muito o tempo necessário para adiquirir um bom resultado, ou então no tamanho do banco de dados necessário para tal.

Outro problema é a falta de flexibilidade da rede. Digamos que alguém por algum motivo decidiu treinar uma rede neural profunda para reconhecimento de carros, concluindo o projeto

depois de muito tempo e dedicação. Porém agora essa mesma pessoa quer reconhecer caminhões, mas a sua rede não foi treinada para isso. Ela deveria recomeçar o processo de treinamento do zero para esse novo projeto, além de precisar levantar todo o banco de dados novamente, demandando mais tempo e energia?

Para resolver esses pontos temos à nossa disposição o método de *Transfer Learning* (transferência de aprendizado em português), que consiste em aproveitar o conhecimento prévio de uma rede, e adaptá-lo para uma nova funcionalidade. Nesse processo as camadas iniciais da rede pré treinada são congeladas, de forma a não terem mais seus pesos calibrados; Assim apenas as ultimas camadas seriam realmente alteradas, economizando tempo, energia e demandando um banco de dados muito menor.

Sobretudo, existem alguns pontos importantes a serem considerados antes de adotar essa abordagem para aplicar em um projeto. Esses pontos definem a semelhança da rede previamente treinada com a arquitetura desejada.

2.4.2.1 Funcionalidade

Vimos que dentro do universo das redes neurais convolucionais existem diversas funcionalidades. Elas determinam como o projeto vai ser treinado, quais são as entradas (banco de dados rotulado), e quais são as saídas esperadas. Assim, para que uma rede seja treinada a partir de uma outra com sucesso, é necessário que ambas tenham as mesmas funcionalidades. Por exemplo, não adianta tentar reaproveitar uma rede de classificação para um projeto de localização e vice versa.

2.4.2.2 Classe de objetos

Outro aspecto muito importante a ser observado são as classes de objetos, ou seja, quais objetos se deseja detectar. Atualmente redes pré treinadas possuem um vasto leque de classes cujo treinamento foi realizado, sobretudo deve-se verificar se o objeto a ser detectado está dentro, ou pelo menos se assemelha com a classe já treinada.

Por exemplo, nesse projeto o objetivo é se treinar uma rede para identificação de traves de futebol, Por isso, escolhemos uma rede pré treinada em cima de objetos, e não uma outra desenvolvida para identificar plantas ou animais.

Antes de escolhermos um modelo, é necessario escolher o banco de dados que no caso seriam o conjunto de imagens na qual o modelo foi treinado, a internet atualmente conta com diversas opções que vão desde objetos até imagens de tomografias. Para o projeto, que consiste em treinar uma rede neural artificial para reconhecimento de traves de futebol em um jogo de robôs humanoides, foram utilizados dois: o primeiro se chama ImageNet (um banco de dados que hoje conta com mais de 14 milhões de imagens e mais de 20 mil classes catalogadas. Que consta em seu material diversos materiais de uso cotidiano). (Imagenet (2021)). Já o segundo é o MS

COCO (*Common Objects in COntext*) que assim como o primeiro conta com uma quantidade enorme de imagens e diversos objetos classificados. (COCO Dataset (2021))

2.4.3 Modelos pré-treinados

Uma vez escolhido o *(*dataset) vamos passar para os modelos, que será a estrutura da rede neural que vamos nos basear para treinar nosso próprio projeto. Existem diversos modelos pré-treinados disponíveis na comunidade, e com a evolução das pesquisas na área de redes neurais e visão computacional muitos outros estão surgindo.

Não vamos nos aprofundar em quais são os melhores modelos, e nem nas características específicas sobre a arquitetura das redes, pois nosso objetivo é avaliar se é possível e viável treinar uma rede neural, utilizando o método de transferência de aprendizado em uma rede neural sem precisar investir em *hardwares* potentes, além disso, hoje em dia já existem diversos trabalhos que comparam esses algoritmos.

Por isso, para efeito de comprovação vamos treinar dois modelos separados, utilizando processamento em nuvem e verificar se essas arquiteturas podem ser treinadas de maneira simples, funcional e de qualquer computador com acesso à internet. O primeiro modelo se chama Inception, uma estrutura desenvolvida pelo Google também conhecida como GoogLeNet, treinada em cima da ImageNet, modelo que ficou em primeiro lugar na *ImageNet Large-Scale Visual Recognition Challenge 2014* (ILSVRC14), uma competição que reúne os principais desenvolvedores de vanguarda na área de reconhecimento de objetos utilizando redes neurais (Shaikh (2018)).

O segundo modelo é o YOLO (*You Only Look Once*), uma arquitetura voltada para velocidade, e muito utilizada em aplicações que necessitam de reconhecimento em tempo real, ela foi treinada em cima do *(*dataset) COCO (Redmon e Farhadi (2018)).

2.4.4 Parâmetros

Durante o processo de treinamento existem alguns parâmetros que moldam o desenvolvimento em si, e eles podem influenciar no resultado final. Apesar de existirem, várias dessas variáveis vamos citar somente os principais.

2.4.4.1 Número de iterações

O número de iterações, ou *epochs* (do inglês, épocas) trata do número de vezes que o algoritmo vai processar o ciclo de propagação direta e retropropagação. Em outras palavras, esse termo define quanto a rede vai ser treinada.

Existem vários problemas em não definir um número adequado de *epochs*, tanto em excesso quanto em falta de treinamento. Em caso de um número insuficiente de iterações, a rede não irá aprender as características básicas do objeto em questão, o que recebe o nome de

underfitting, não apresentando um bom resultado final, com baixa taxa de acertos. Por outro lado em caso de treinamento em excesso o modelo vai apresentar o que chamamos de *overfitting* que significa um sobreajuste da rede em relação ao banco de dados. Em outras palavras, nossa rede vai decorar cada exemplo do banco de dados e não generalizá-lo, apresentando um bom resultado para o banco de dados de treinamento porém falhando em aplicações reais.

O ideal é que o treinamento fique entre esses dois pontos, de forma que a rede consiga aprender as características necessárias para um bom reconhecimento e ao mesmo tempo seja capaz de generalizar essas características e não apenas decorar as imagens do banco de dados. Por isso o número de iterações é muito importante no processo de treinamento de um rede. Na prática, para encontrar o número coerente de *epochs* podemos observar o erro, de forma que um aumento no número de épocas traga pouca ou nenhuma melhora na qualidade do resultado e em um processo de tentativa e erro encontrar o melhor valor para esse parâmetro (Data Science Academy (2019)).

2.4.4.2 Tamanho do lote

O tamanho do lote, *batch size*, se refere a quantas imagens vão ser passadas em uma iteração pela rede, isso ajuda no processo de generalização, pois o algoritmo terá que descobrir os detalhes em comum entre todas as imagens.

Novamente, existem prós e contras na escolha de um número muito grande ou muito pequeno, o tamanho de lote está diretamente relacionada o tempo de cada iteração. Então, em caso de um número muito pequeno, embora o treinamento seja executado mais rapidamente, pode ser que haja uma redução na qualidade. Já para um *batch* muito elevado, em geral a rede irá generalizar bastante, porém o tempo gasto vai ser muito grande. Devendo o projetista encontrar o meio termo ideal para o processo.

Alguns autores recomendam que o processo seja iniciado com tamanhos menores de lote, para que a rede inicie o aprendizado mais rapidamente, aumentando o tamanho durante o processo para melhorar a qualidade se aproximando do ótimo global. (Data Science Academy (2019)).

2.5 Treinamento em nuvem

Vimos que o processo de *transfer learning* pode agilizar em muito o processo de treinamento, economizando tempo e sem precisar de um hardware muito potente. Mas existem alguns problemas para aplicar a transferência de aprendizado e retreinar uma rede pode ser necessário ainda um considerável poder computacional dependendo dos parâmetros escolhidos (*epochs* e *batch size*), o que inviabiliza a atividade em computadores de uso pessoal. Outro ponto importante é a necessidade de se configurar o ambiente de desenvolvimento para execução do projeto, instalação de bibliotecas, escrever códigos de programação, configurar a rede.

Para contornar os problemas utilizou-se o treinamento em nuvem. Hoje em dia já existem diversos serviços que disponibilizam GPUs com alto poder de processamento em ambiente já preparados para receber as bibliotecas necessárias para esse processo. De forma totalmente grátis, é possível configurar um *script* com todos as linhas de código necessitando somente da execução, sem gastar um tempo enorme configurando seu computador e instalando softwares. Além de permitir o treinamento de forma dinâmica, por qualquer computador com acesso a internet.

O objetivo desse projeto, além de treinar uma rede de forma remota, é verificar a possibilidade de se desenvolver uma aplicação simples e prática, de forma que seja necessário apenas carregar o banco de dados com as imagens devidamente rotuladas, rodar o *script* e verificar o resultado em uma aplicação real, *plug and play*.

2.6 Ferramentas

Nessa seção vamos ver quais ferramentas vão ser utilizadas para realizar esse projeto.

2.6.1 Banco de dados

Para começar deve-se selecionar as imagens que vão ser utilizadas para treinar a rede neural. Esse passo é importante pois pode definir o quão bem seu algoritmo vai reconhecer os objetos na sua aplicação. Deve-se checar quão próximas em questões de qualidade, tamanho, e contexto as imagens de treinamento são das imagens reais, a rede vai aprender aquilo que mostrarmos para ela.

Por exemplo, para esse projeto, vamos desenvolver uma rede capaz de identificar traves. As imagens são captadas através de câmeras conectadas aos robôs, e não apresentam resoluções nem qualidade muito altas. Por isso, não adianta reunir uma série de imagens em alta resolução pois isso pode prejudicar o resultado final.

Além disso, deve-se atentar para a orientação e o tamanho dos objetos desejados. Traves de futebol são barras geralmente posicionadas próximo à vertical, dependendo do ângulo horizontal da câmera (o robô pode estar deitado, mas nesse caso não faria sentido localizar traves), por isso foram retiradas as imagens com traves em ângulos não convencionais e fora da aplicação do projeto. É importante pois as redes neurais convolucionais não são invariantes quanto a rotação das imagens, ou seja, caso forem repassadas imagens com traves dispostas na horizontal, ou em algum ângulo não convencional, a rede iria aprender podendo prejudicar a aplicação.(Rosebrock (2021)).

Outro aspecto que devemos ficar atentos é a origem das imagens, de onde elas vieram. No exemplo acima a rede vai ser utilizada para jogos de futebol de robôs humanoides, esses eventos ocorrem em ambientes padronizados portanto não é vantajoso utilizar imagens de traves de futebol de jogos da copa do mundo ou outras competições do tipo. É recomedável que o banco

de dados contenha imagens próximas de mesmo contexto às de uso visado.

Onde buscar o banco de dados? Uma das soluções é buscar imagens na internet, e reuní-las em um diretório, porém isso demanda muito tempo. Hoje já existem algumas plataformas que disponibilizam essas imagens gratuitamente. Uma das mais conhecidas é o Kaggle um site voltado para a comunidade de IA que reúne bancos de imagens com os mais variados temas, alguns já rotulados, outros por rotular.

Depois de reunido e filtrado, vem o processo de organização, costuma-se organizar o banco de dados em três partes, a parte de treinamento (*train set*), a parte de validação (*validation set*) e a parte de teste (*test set*).

O *train set* se refere ao banco de dados cuja rede vai ser exposta, geralmente esse é o conjunto com o maior número de imagens dentre os três. Ele é utilizado para fazer a calibração de parâmetros, e todo o treinamento já previamente explicado. A nossa rede vai primeiro passar por essa etapa. Depois do *train set* a rede vai ser exposta ao *validation set*, que é utilizado para checar o quão bem nosso algoritmo conseguiu aprender, etapa onde é possível verificar possíveis correções nos parâmetros da rede (número de iterações, tamanho de lote), ou correções no banco de dados (aumentar as imagens, filtrar melhor, etc).

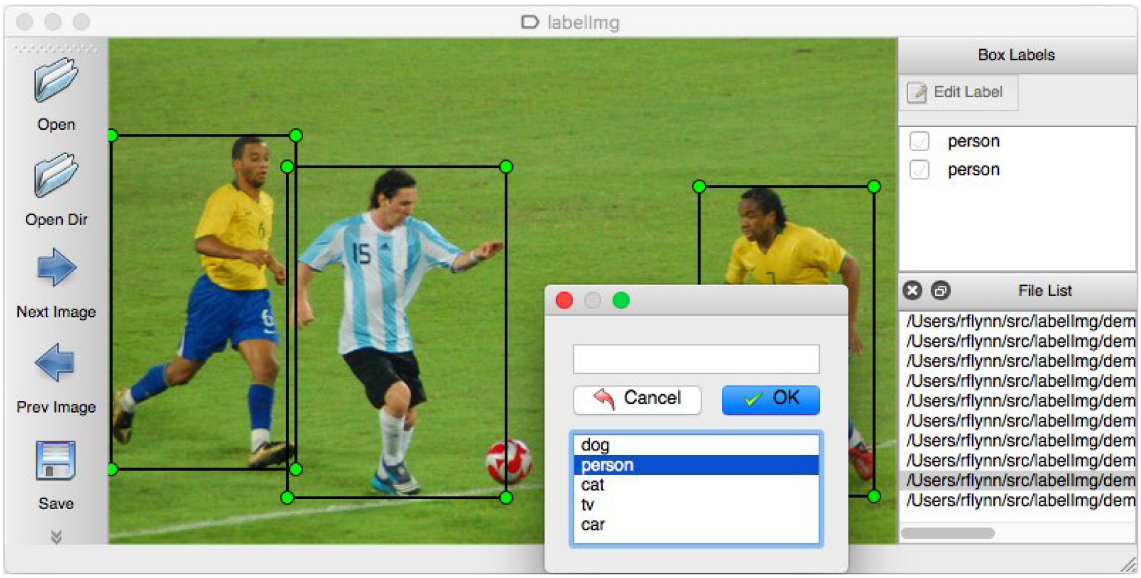
Por último, passamos para a fase final de verificação, o *test set* que reúne imagens da aplicação real, e busca verificar quão valido o nosso projeto é. Para esse conjunto, é importante que ele seja totalmente isolado dos demais, as imagens não podem estar presentes nos outros conjuntos, para que seja verificado se a rede realmente aprendeu e não só "decorou"as respostas.

2.6.2 LabelImg

Com o *dataset* em mãos chegou a hora de rotular os dados, o software que vamos utilizar chama *LabelImg*, mas existem vários outros disponíveis na internet gratuitos.

O que queremos é que todas as imagens do *train set* apresentem o que chamamos de *bounding box*. Como na imagem abaixo. Encontra-se uma prévia do *software* na Figura 4.4)

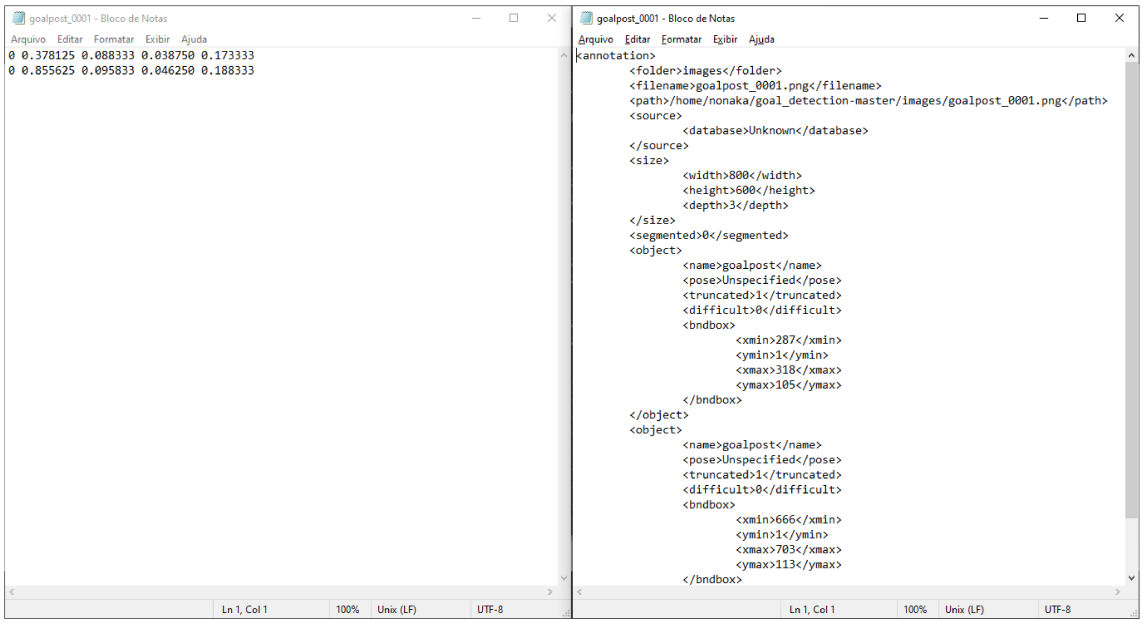
Figura 2.17 – Software LabelImg e Bounding Boxes



Fonte: Disponível na página oficial do *software* (acesso em 25 de setembro de 2021)

O problema é que, para os dois modelos escolhidos (Inception e Yolo), utiliza-se dois tipos de *label* em diferentes formatos .xml (Inception) e .txt (YOLO), vide imagem abaixo.

Figura 2.18 – Rótulo Inception x Rótulo YOLO



Fonte: Própria

Felizmente não temos que rotular as imagens duas vezes, podemos programar códigos, ou até encontrá-los prontos na internet para converter um modelo no outro de forma simples.

2.6.3 Python

Para o desenvolvimento do trabalho foi utilizado Python para programação dos notebooks, pois é uma linguagem de alto nível, de fácil entendimento, e ainda por cima apresenta um grande suporte nessa área de aprendizado de máquina, contando com bibliotecas e APIs que simplificam bastante.

2.6.4 Google Colaboratory

Para o desenvolvimento em nuvem utilizaremos o *Google Colaboratory*, uma plataforma para desenvolvimento de algoritmos em Python que pode ser rodada no navegador, sem precisar instalar nenhuma biblioteca no computador. Essa plataforma disponibiliza para os usuários alto poder de processamento com CPUs e GPUs dependendo da aplicação, o armazenamento de imagens, e de todos os arquivos necessários para configurar a rede pode ser feita no *Google Drive*. Ou seja, não é preciso instalar nenhum software.

2.6.5 Google Drive

Utilizaremos o Google Drive para armazenamento dos arquivos necessários para realizar o treinamento, desde as imagens e rótulos, até arquivos de configuração dos parâmetros da rede.

2.6.6 Tensorflow

Tensorflow é uma biblioteca voltada para o desenvolvimento de códigos de ML, de código aberto, oferece diversas ferramentas e API's (Application Programming Interface, traduzindo Interface de Programação de Aplicativos) para a comunidade desenvolver projetos nas diversas áreas do aprendizado de máquina Abadi et al. (2015). Essa biblioteca traz diversas funções para facilitar o treinamento, sem precisar que o código seja escrito no nível mais complexo.

3 Desenvolvimento

As ferramentas (descritas no capítulo anterior) utilizadas foram:

1. Banco de imagens;
2. LabelImg;
3. Google Drive;
4. Google Colaboratory;
5. Python.

O desenvolvimento do projeto foi dividido nas seguintes etapas:

1. Organização do *dataset*
2. Rotular as imagens
3. Organização do ambiente virtual (Google Drive)
4. Desenvolvimento do notebook Python para treinamento em nuvem
5. Treinamento e validação
6. Teste em vídeos reais

É importante salientar que como queremos testar a rede em dois modelos pré treinados, vamos montar dois notebooks Python, um para cada modelo. Consequentemente, realizou-se as etapas de treinamento, validação e testes duas vezes também.

Como visto no capítulo anterior o primeiro passo para treinar nossa rede neural é escolher o *dataset*, isto é, as imagens que vamos dar ao algoritmo para que ele aprenda a identificar o objeto proposto.

Para isso vamos utilizar o site Kaggle, que apresenta diversas imagens de traves de futebol utilizadas durante as competições de robótica ao redor do mundo. Além disso, outra fonte de dados é o repositório da própria EDROM, que durante suas competições gravou vídeos que utilizaremos para testar a rede.

No final, juntando essas duas fontes de dados, têm-se mais de duas mil imagens em seu estado bruto. O próximo passo é filtrar essas imagens pois nem todas elas apresentam as características necessárias para serem utilizadas no treinamento. Para fazer a filtragem do banco

de imagens levamos em conta a qualidade, muitas delas são tremidas, ou com baixa qualidade, não sendo vantajoso incluí-las no treinamento. Deve-se retirar as que não mostram as traves, muitas imagens acompanham a bola, outros robôs, como o objetivo é reconhecer traves, deixamos somente elas pra ser mais eficiente. Por fim, buscamos imagens distintas entre si, e, como elas são retiradas de filmagens muitas, são muito semelhantes.

Depois de filtradas, o próximo passo seria separar o *dataset* em *train set*, *validate set* e *test set*. Porém, dado que o objetivo é o reconhecimento de traves em vídeo, o *test set* será uma das filmagens da EDROM, cujas imagens não entraram no banco de treinamento. Inicialmente, era desejado realizar o teste rodando o próprio algoritmo da câmera dos robôs e fazer a identificação em tempo real, porém, devido à pandemia, não foi possível fazer presencialmente.

No fim depois de filtradas e separadas a contagem foi a seguinte:

1. *Train Set* - 429 imagens;
2. *Validate Set* - 10 imagens.

Foi escolhido um número pequeno de imagens de validação, pois elas iriam servir apenas como uma estimativa se o treinamento seguiu o curso esperado, de forma que o verdadeiro teste seria o vídeo. Além disso, como depois de filtradas o número de imagens foi bastante reduzido, optou-se por não utilizar muitos exemplos de validação para não prejudicar o treinamento.

Em caso de um resultado de treinamento fraco é possível adicionar mais exemplos de treinamento e assim aumentar também a quantidade de exemplares para validação.

É importante citar uma boa prática de organização do banco de dados que é renomear as imagens seguindo um padrão, muitas vezes depois de baixar no Kaggle, ou então separar os *frames* de vídeos as imagens ficam com nomes diferentes e bagunçados dificultando a correção de erros, e o processamento delas, por isso todas elas foram renomeadas seguindo o padrão: ”*goalpost_nnn.jpg*” onde ”nnn” é a numeração da imagem.

3.1 Rotulando as imagens

Depois de escolhido o banco de dados, é necessário rotular as imagens. Para isso foi utilizado o software *LabelImg*, previamente explicado no Capítulo 2.

Durante a rotulação dos dados podemos escolher se as *labels* serão em formato .xml (utilizado para o modelo Inception) ou .txt (para o YOLO). Felizmente não é necessário colocar rótulos em todas as imagens duas vezes, com um código simples em Python é possível fazer a conversão de um tipo para outro.

Para rotular as imagens basta abrir o local onde elas estejam armazenadas, é importante que os nomes já tenham sido alterados e padronizados pois as *labels* terão o mesmo nome

das imagens originais alterando o formato, depois de abertas basta selecionar a ferramenta de marcação e desenhar as *bounding boxes* na região desejada, escrever o nome da classe e salvar em outro diretório para manter a organização dos arquivos, assim teremos dois diretórios, o das imagens e o das anotações.

Feito isso o software irá criar um arquivo com as coordenadas da classe em um arquivo, esse arquivo é a *label* que será utilizada no treinamento.

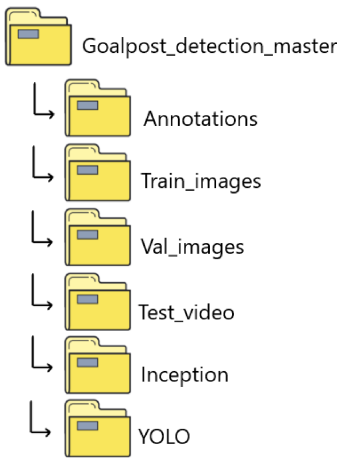
No fim desse processo vamos ter um arquivo rótulo para cada imagem.

3.2 Organização do ambiente virtual (Google Drive)

Depois de feitas as anotações necessárias, devemos organizar o nosso ambiente de desenvolvimento, que no caso é o Google Drive. O treinamento irá utilizar o espaço disponível para leitura e escrita de arquivos, além de armazenar as imagens, por isso é recomendado que pelo menos metade do armazenamento total esteja disponível.

O diretório será organizado na seguinte maneira:

Figura 3.1 – Organização do Drive



Fonte: Própria

Uma breve explicação de todas essas pastas:

1. Goalpost_detection_master - é a pasta mãe do projeto como um todo, todos os arquivos, imagens e dados serão salvos aqui;
2. Annotations - é a pasta onde ficarão os arquivos de rótulo, tanto para o modelo *Inception* quanto para o *YOLO*;
3. Train_images - para as imagens de treino;

4. Val_images - para as imagens de validação;
5. Test_video - para o vídeo de teste;
6. Inception - aqui é onde todos os dados de configuração, pesos, e resultados do modelo *Inception* ficarão armazenados;
7. YOLO - a mesma explicação do tópico acima porém para o modelo *YOLO*.

Mais a frente será necessário clonar o diretório Git para os respectivos modelos, que serão armazenados nas subpastas acima nomeadas.

3.3 Inception

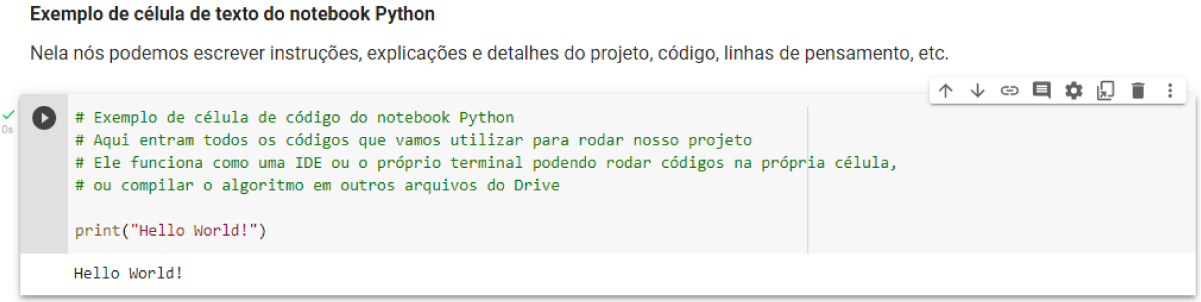
3.3.1 Preparo do notebook Python para treinamento

O notebook consiste em um arquivo de extensão .ipynb que permite escrever algoritmos em Python, textos, importar imagens, compilar os códigos e plotar gráficos em um único lugar. Nesse espaço é possível criar células e escrever desde um texto com imagens, até códigos e linhas de programação. Para isso existem dois tipos de célula:

1. Texto: célula não executável, aqui são inseridas informações, instruções, imagens, etc. Utilizadas para explicar o funcionamento do projeto em si, e a finalidade do código;
2. Código: aqui entra a programação em si, é uma célula executável que funciona como um IDE (*Integrated Development Environment*, que significa ambiente de desenvolvimento integrado). É basicamente o código de programação que se quer executar.

As células estão representadas na Figura 3.2.

Figura 3.2 – Células do notebook Python



Fonte: Própria

Atualmente já é possível encontrar diversos materiais de como construir e treinar uma rede neural convolucional utilizando o Google Colab, embora nenhuma delas seja voltado para reconhecimento de traves de futebol podemos utilizar a base de como é realizado o treinamento.

Outro material bastante util também é o treinamento de RNC's utilizando a GPU do próprio computador (Verdone (2017)). Nesse caso basta adaptá-lo ao formato de leitura do Notebook Python. Então foram constuídos notebooks baseados em alguns materiais encontrados na comunidade.

Para cada modelo pré-treinado, vamos ter alguns passos de preparação antes de iniciar o treinamento. É importante citar que não é necessário criar do zero os arquivos, modelos e códigos de treinamento, podemos utilizar o Git para clonar e utilizar essas ferramentas dos repositórios oficiais e da própria comunidade.

Para o modelo Inception utilizamos como referência o tutorial de Verdone (2017), seguindo os seguintes passos de preparação:

1. Conectar ao Drive, onde ficarão armazenados o banco de imagens, o vídeo de teste e todos os arquivos necessários para o treinamento;
2. Clonar o Git do link https://github.com/dodandeniya/TFmodels.git;
3. Compilar e testar a framework que vai ser utilizada no treinamento;
4. Importar as bibliotecas necessárias;
5. Criar arquivos de referência e gravação.

3.3.2 Alteração dos arquivos de configuração - Inception

Depois de clonar os repositórios é possível notar que vários arquivos surgiram nos diretórios do seu Drive, eles foram copiados padronizados, então para que funcionem corretamente é necessário fazer algumas alterações.

Para o modelo Inception o arquivo de configuração se encontra no caminho:

MyDrive/goalpost_detection_master/object_detection/data/ssd_inception_v2_coco.config

Nesse código se encontram hyperparâmetros, funções de ativação, camadas, número de iterações, tamanho de lote e diversos outras variáveis que influenciam no treinamento da rede (recomenda-se um estudo mais aprofundado caso o objetivo seja otimizar a qualidade).

Após alterar as linhas devem ficar como mostradas abaixo:

1. line 158 - num_steps: "Número de iterações desejado"
2. line 171 - input_path: "./object_detection/data/goalpost_train.record"
3. line 173 - label_map_path: "./object_detection/data/goalpost_label_map.pbtxt"
4. line 185 - input_path: "./object_detection/data/goalpost_train.record"

5. line 173 - label_map_path: "./object_detection/data/goalpost_label_map.pbtxt"

O arquivo completo .config se encontra nos anexos.

3.3.3 Treinamento e validação - Inception

Após a preparação dos notebooks e arquivos de configuração proceder-se para o treinamento. Para treinar basta compilar a célula que chamará o código e passar todos os parâmetros necessários em cada modelo.

O processo de validação foi feito aplicando-se os pesos da rede em 10 imagens, sendo 9 delas imagens semelhantes às encontradas no *dataset* e uma em um ambiente diferente, com algumas interferências e outros objetos.

Já o processo de teste, como mencionado anteriormente, era ideal que se realizasse ao vivo com a câmera da robô fazendo a localização do gol. Porém devido ao contexto foi utilizado um vídeo gravado durante a competição de 2019 pela equipe.

Para o modelo Inception, adotamos um tamanho de lote de 24 imagens, que é o padrão utilizado no material. No processo de treinamento adotou-se a seguinte metodologia:

1. Foram escolhidos diferentes pontos de análise, onde observou-se como o aprendizado ocorreu, se houve *overfitting* ou *underfitting*, adotou-se arbitrariamente os seguintes pontos: 5 mil, 10 mil, 15 mil, 20 mil, 25 mil, 30 mil e 40 mil iterações;

2. Em cada um desses pontos a rede gerou os pesos, que foram aplicados nas imagens de validação e no vídeo de teste;

3. Entre os *checkpoints* foram coletados diversos valores de erro, bem como do tempo médio de cada iteração para gerar um gráfico de aprendizado e mensurar o tempo aproximado para o treinamento.

O processo completo de treinamento para 40 mil iterações demorou cerca de 8 horas, sendo proporcional aos demais treinamentos com menos iterações.

Para que o treinamento salvasse os pesos nesses exatos pontos, foi necessário alterar o arquivo de configuração da rede, exportar os valores e aplicá-los nas imagens e no vídeo. Assim, depois de obtidos a validação e o teste alterava-se o arquivo de configuração para o *checkpoint* seguinte e repetia se o processo.

Com os pesos aplicados, as imagens de validação foram avaliadas para verificar a qualidade de treinamento em cada *checkpoint*. Utilizando duas metodologias.

Calculou-se a média das predições, para cada trave a rede deu um valor de confinça, variando de 0 a 100%. Em valores maiores de 50%, desenhou-se o *bounding box*, com esses dados calculou-se a média, em caso de não identificação a trave iria com valor 0. Equação abaixo:

$$\frac{\sum_{i=1}^{N} conf}{N} \tag{3.1}$$

Onde N é o número de traves, e *conf* é o valor de confiança (de 0 a 100% para cada *bounding box*.

Calculou-se também a média de acertos, nesse caso não importa o nível de confiança e sim se a rede identificou corretamente a trave. Ou seja:

$$\frac{N_{acertos}}{N_{total}} \tag{3.2}$$

A coleta dos valores de erro e tempo das iterações entre *checkpoints* foi feita apartir do log no próprio notebook. Foi observado que o erro diminuiu drasticamente no começo do treinamento onde foi coletados mais dados, porém próximo a 7 mil iterações ficou estável.

Essa abordagem foi adotada devido ao fato de que o treinamento utilizando esse modelo pré-treinado é mais demorado, possibilitando um estudo aprofundado com muitas iterações.

Os valores coletados, as imagens de validação e o vídeo de teste estão no próximo capítulo.

3.3.4 Teste em vídeos reais - Inception

Depois de realizado o treinamento e a validação do modelo é necessário testá-lo.

A avaliação desse processo se dá na maior parte de forma empírica, observando se a localização está boa ou não. Alguns fatores que podemos observar na qualidade do vídeo de teste é a qualidade do reconhecimento propriamente dito, uma rede bem treinada é capaz de reconhecer os objetos propostos com confiança, de forma precisa e contínua. Outro ponto importante a ser observado é a presença de falsos positivos, ou seja, identificação em objetos que não estão dentro do proposto, um treinamento de qualidade a presença de falsos positivos é minimizada ou até mitigada.

É importante avaliar os dois modelos utilizando os mesmos parâmetros para uma base comparativa que permita estudar ambos com um olhar crítico, fazer rastreio de erros e por fim melhorar o treinamento como um todo.

3.4 YOLO

3.4.1 Preparo do notebook Python para treinamento em nuvem

Para o desenvolvimento desse notebook seguimos como referência o material disponível no próprio site do Redmon e Farhadi (2018), os passos de preparação são:

1. Conectar ao Drive, onde ficarão armazenados o banco de imagens, o vídeo de teste e todos os arquivos necessários para o treinamento;
2. Clonar o repositório Yolo do link https://github.com/ultralytics/yolov3;
3. Importar as bibliotecas necessárias.

O código dos notebooks para os dois modelos estão disponíveis nos anexos.

3.4.2 Alteração dos arquivos de configuração - YOLO

Para o modelo YOLO o arquivo de configuração se encontra no caminho:

"My Drive/goalpost_detection_master/yolov3/cfg/yolov3.cfg"Nesse código se encontram hyperparâmetros, funções de ativação, camadas, número de iterações, tamanho de lote e diversos outras variáveis que influenciam no treinamento da rede (recomenda-se um estudo mais aprofundado caso o objetivo seja otimizar a qualidade). Nesse caso é necessário alterar somente uma linha no código, que deve ficar como mostrado abaixo:

1. line 783 - classes=1

É importante para a qualidade dos resultados que o arquiteto da rede tenha conhecimento dos parâmetros e suas funções, por isso recomenda-se estudar e explorar tais arquivos para descobrir o que cada código ou variável faz.

3.4.3 Treinamento e validação - YOLO

Para o modelo YOLO, adotamos um tamanho de lote de 16 imagens, que é o padrão utilizado no material. Ao contrário do primeiro modelo, o treinamento nesse caso é mais rápido. Por isso não foram coletados os valores de erro e de tempo por iteração, além disso, o próprio modelo oferece os gráficos para estudarmos o avanço do treinamento. Observou-se somente o tempo total de treinamento e os gráficos de erro e acurácia.

Para um estudo o treinamento foi dividido e realizado com diferentes números de iteração (50, 150 e 300 iterações), e assim é possível analisar o avanço do treinamento.

No processo utilizando 300 iterações, demorou em torno de 2h35 com a gravação de checkpoints intermediários, esse recurso permite que o treinamento seja salvo caso encerrado

antes da hora, e também em caso de necessidade de treinamento além do proposto permite que retorne no ponto que parou. Por outro lado aumenta a duração de processamento.

A validação foi feita da mesma forma que no modelo Inception, avaliando-se o desempenho da rede nas imagens de validação e dividindo em duas categorias.

Os valores coletados, as imagens de validação e o vídeo de teste estão no próximo capítulo.

3.4.4 Teste em vídeos reais - YOLO

Nessa parte o processo seguiu da mesma forma que a desenvolvida no Subseção 3.3.4.

É importante avaliar os dois modelos utilizando os mesmos parâmetros para uma base comparativa que permita estudar ambos com um olhar crítico, fazer rastreio de erros e por fim melhorar o treinamento como um todo.

4 Resultados

Neste capítulo vamos analisar os resultados obtidos a partir do treinamento dos modelos para nossa rede neural. Nos dois modelos utilizados, adotou-se abordagens diferentes devido ao tempo de treinamento demandado, sobretudo os resultados devem cooperar para demonstrar o mesmo objetivo.

4.1 Inception

4.1.1 Erro e tempo médio por iteração

Para o modelo Inception adotou-se a metodologia de aplicar o aprendizado (obtido no *Train Set*) da rede nas imagens de validação e no vídeo teste, para verificar a qualidade do treinamento, além de coletar valores de erro, e tempo por iteração durante o treinamento.

A tabela com os valores de erro e tempo encontra-se a seguir:

Tabela 4.1 – Erro e tempo médio durante o treinamento para o modelo Inception

Iteração	Erro (%)	Tempo/iteração(s)
1	31,81	1,3
100	7,7	0,7
200	7,34	0,72
300	7,05	0,72
400	6,85	0,709
500	6,02	0,698
600	6,19	0,825
700	5,85	0,717
800	6,41	0,986
900	5,28	0,701
1000	5,61	0,749
1500	4,42	0,679
2000	4,87	0,712
2500	4,74	0,793
3000	3,5	0,692
3500	3,34	0,667
4000	3,71	0,799
4500	3,22	0,733
5000	3,01	0,681
6000	3,2	0,734
7000	2,94	0,712
8000	2,85	0,702
9000	2,97	0,716
10000	2,71	0,724
11000	2,68	0,732
12000	2,85	0,761
13000	2,39	0,749
14000	2,83	0,71
15000	2,32	0,727
17500	2,14	0,716
20000	2,77	0,669
22500	2,5	0,771
25000	2,75	0,696
27500	1,88	0,735
30000	1,92	0,711
40000	2,81	0,748

Podemos observar que o houve uma redução considerável do erro observado, principalmente no início do treinamento, posteriormente, esse valor estabilizou. Enquanto isso é possível notar também que o tempo médio de cada iteração se manteve mais ou menos constante, independente do treinamento ter avançado ou não, exceto na primeira iteração.

4.1.2 Imagens de validação

Olhado agora para as 10 imagens de validação, que continham ao todo 13 ocorrências de trave (em algumas delas as duas traves estavam à vista, em outras somente uma) obteve-se os seguintes resultados, utilizando a métrica de avaliação explicada no Capítuo 3.

Tabela 4.2 – Desempenho do modelo Inception nas imagens de validação

Trave #	5k	10k	15k	20k	25k	30k	40k
1	0%	90%	97%	96%	99%	96%	97%
2	0%	94%	52%	57%	0%	61%	84%
3	0%	94%	99%	97%	99%	97%	97%
4	0%	50%	50%	50%	50%	0%	0%
5	0%	0%	0%	63%	67%	0%	69%
6	0%	63%	83%	84%	50%	80%	89%
7	0%	86%	83%	72%	86%	86%	79%
8	0%	0%	0%	50%	50%	0%	0%
9	0%	57%	66%	0%	63%	51%	0%
10	0%	0%	0%	50%	50%	0%	0%
11	0%	93%	97%	79%	88%	90%	98%
12	0%	0%	0%	54%	0%	0%	0%
13	0%	72%	86%	95%	84%	82%	50%
Média	0%	54%	55%	65%	60%	49%	51%
Acertos	0%	69%	69%	92%	85%	62%	62%

Observa-se que, o melhor desempenho em cima das imagens de validação se deu no *checkpoint* com 20 mil iterações seguido pelo de 25 mil. Isso pode ser explicado olhando para o mecanismo utilizado para aprendizado, uma das possibilidades é que nesse ponto os pesos ficaram melhor calibrados, e ao aplicar o fator de correção para recalibrá-los acabou-se por piorar o resultado. Outra possível explicação é a ocorrência de *overfitting* onde a rede "decora"as imagens de treino, ficando muito rígida para abstrair e identificar traves em situações diferentes.

Essas hipóteses podem ser verificadas treinando-se a rede durante um tempo maior, e estudando os resultados obtidos. Além de alterar os hiperparâmetros da rede e buscar o ótimo global, processo que demandaria bastante tempo e fica como possibilidade para trabalhos futuros.

Em todos os *checkpoints* a rede conseguiu localizar corretamente a imagem de validação com o ambiente diferente do padrão, incluindo nos pesos referentes ao treinamento de 5 mil iterações.

4.1.3 Vídeo de teste

Analisar os resultados de erro, e as imagens de validação ajudam a ter uma noção se o treinamento está evoluindo ou não, porém para saber realmente se o processo é util para o

projeto em questão é necessário avaliar o desempenho da rede em cima do vídeo de teste. Por isso utilizaremos três métricas para avaliar a qualidade do aprendizado:

1. Capacidade de identificação de traves, observando a eficiência;
    a) Inexistente: a rede não foi capaz de identificar traves;
    b) Fraca: é observado uma pequena capacidade de identificação;
    c) Média: a rede demonstra capacidade para identificação, porém muito abaixo do desejável;
    d) Boa: a rede demonstra boa capacidade de reconhecimento porém ainda abaixo do desejável.
    e) Consistente: consegue identificar traves de forma consistente.
2. Habilidade de localizar corretamente duas traves ao mesmo tempo;
    a) Inexistente: a rede não localiza duas traves ao mesmo tempo;
    b) Fraca: é observado em poucos momentos a localização dupla;
    c) Média: a rede consegue de forma considerável porém aquém do desejável fazer a localização;
    d) Boa: observado uma boa capacidade de dupla localização.
    e) Consistente: consegue identificar duas traves de forma consistente.
3. Existência de falsos positivos, que são os casos em que a rede identifica a trave mesmo não havendo nenhuma.

Para essas métricas obteve-se os seguintes resultados:

Tabela 4.3 – Desempenho do modelo Inception no vídeo teste

Métricas	5k	10k	15k	20k	25k	30k	40k
Identificação de traves	Inexistente	Fraca	Fraca	Média	Média	Fraca	Fraca
Localização de duas traves	Inexistente	Fraca	Fraca	Média	Boa	Fraca	Inexistente
Falsos positivos	–	Sim	Sim	Sim	Não	Sim	Sim

Idealmente busca-se uma rede que consiga identificar consistentemente as duas traves, para que o algorítimo na robô indique a direção que a bola deve ser chutada, além de minimizar a presença de falsos positivos que podem atrapalhar nesse momento de decisão. Por isso observando os valores coletados nota-se que os melhores resultados são dos pesos de 25 mil iterações. Concordante com os resultados obtidos nas imagens de validação.

Sobretudo ainda não foi possível chegar a um resultado satisfatório que possibilite a utilização durante um jogo de futebol. Podendo ser justificados pelo *overfitting* da rede ou pela

descalibração local dos pesos durante a aprendizagem. O que pode ser resolvido treinando um número mais de iterações.

As imagens abaixo mostram alguns exemplos retirados dos vídeos de teste, falsos positivos, dupla identificação.

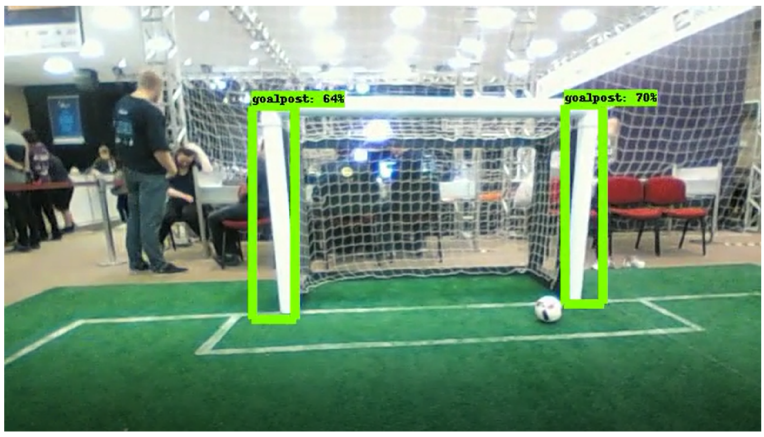
Figura 4.1 – Falsos positivos do modelo Inception

Fonte: Própria

Figura 4.2 – Dupla identificação não conquistada

Fonte: Própria

Figura 4.3 – Dupla identificação conquistada



Fonte: Própria

4.1.4 Viabilidade

Apesar de não ter reduzido ao longo do treinamento, o tempo é uma métrica importante para analisar a viabilidade de se treinar uma rede neural convolucional utilizando a rede. Considerando a média dos valores coletados (0,747 s/iteração), o tempo total para treinamento das 40 mil iterações foi de 8 horas e 18 minutos, todavia, o treinamento teve de ser dividido em dois dias, pois o Google Colab limita o uso diário de seus núcleos GPU (Unidade de Processamento Gráfico, do inglês *Graphics Processing Units*).

Já o tempo de treinamento em um computador pessoal, irá variar dependendo da velocidade do processador, memória disponível, uso de GPU ou CPU.

Utilizando o notebook pessoal (processador Intel i5, 8GB de RAM, SSD de 500GB) e tentando treinar a mesma estrutura de rede observou-se que era inviável devido a memória (por não haver GPU, utilizou-se somente a CPU). Posteriormente reduziu-se o *batch size* de 24, que é o tamanho utilizado na nuvem, para 1, o que prejudica bastante o aprendizado pois em vez de olhar para 24 imagens ao mesmo tempo para calibrar os pesos irá olhar uma imagem por vez. Como resultado, observou-se que o tempo entre cada iteração era de aproximadamente 5 segundos.

Adotando esse valor, para treinar os mesmos 40 mil passos, demandaria 55 horas e 33 minutos, e não necessariamente alcançaria o mesmo resultado. Portanto, embora o treinamento da rede utilizando o modelo pré treinado Inception não tenha retornado resultados satisfatórios, no quesito de projeto é possível observar que o aprendizado existe, e olhando para a velocidade de treinamento, nota-se que o treinamento em nuvem pode ser até seis vezes mais rápido, dependendo do computador utilizado. Mostrando, por fim, a viabilidade de se realizar o treinamento utilizando esse método.

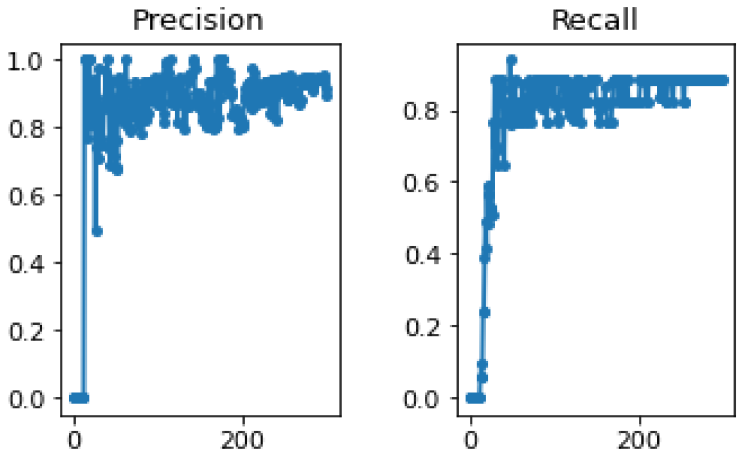
4.2 YOLO

Para o modelo Yolo, o próprio *Notebook* traz métricas de validação, que possibilitam avaliar se o modelo desempenhou bem ou não o processo de aprendizagem, as seguintes métricas estão disponíveis:

1. Precision - Mede a acurácia do modelo em classificar um objeto em positivo.
2. Recall - Mede a capacidade de detectar positivos reais dentro das imagens.

Obtiveram-se os seguintes gráficos:

Figura 4.4 – Gráficos de desempenho do treinamento no modelo YOLO



Fonte: Notebook Python goalpost_detection_yolo_v3.ipynb no Google Colaboratory (acesso em 25 de Setembro de 2021)

Observando-se os gráficos acima é possível observar que o desempenho da rede aumentou bastante, a partir do valor de precisão, alcançando um valor entre 80% e 100%.

4.2.1 Imagens de validação

Agora analisando o desempenho da rede sobre as imagens de validação adotando a metodologia explicada no capítulo anterior, obtendo-se os valores observados na Tabela 4.4.

Observando esses resultados é possível notar que nos treinamentos de 50 e 150 iterações a rede não foi capaz de aprender e reconhecer traves nas imagens de validação. Por outro lado utilizando-se 300 iterações foi possível ampliar bastante o desempenho da rede.

O tempo total de treinamento para 300 iterações foi de aproximadamente 2 horas e 30 minutos, sendo proporcional para os treiamentos com menos iterações.

NOTA: Em muitas imagens não é possível saber a taxa de precisão da identificação pois a *bounding box* extrapola os limites da imagem como no exemplo abaixo. Por isso foi adotado

Tabela 4.4 – Desempenho do modelo YOLO nas imagens de validação

Trave ID	50	150	300
1	0%	0%	50%
2	0%	0%	50%
3	0%	0%	50%
4	0%	0%	50%
5	0%	0%	50%
6	0%	0%	50%
7	0%	0%	95%
8	0%	0%	50%
9	0%	0%	36%
10	0%	0%	50%
11	0%	0%	90%
12	0%	0%	78%
13	0%	0%	68%
Nota	0%	0%	59%
Acertos	0%	0%	100%

um valor padrão de 50% quando isso ocorre, o que explica a nota da rede ter sido baixa embora o acerto tenha sido de 100%.

4.2.2 Vídeo de teste

Adotando-se as mesmas métricas utilizadas na avaliação dos vídeos de teste:

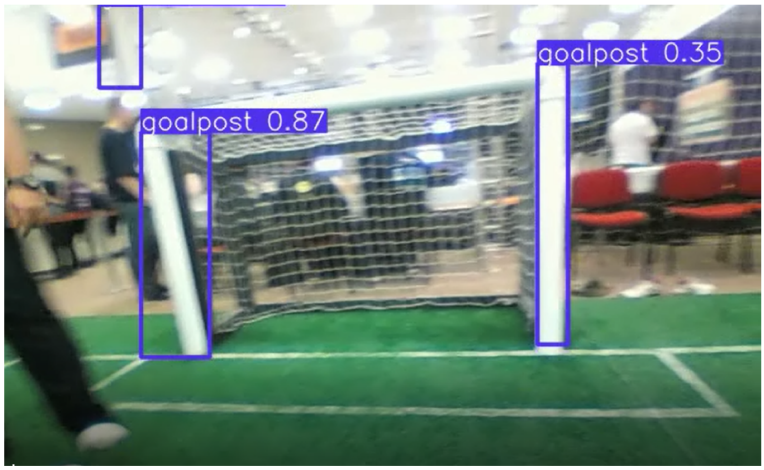
Tabela 4.5 – Desempenho do modelo YOLO no vídeo teste

Métricas	50	150	300
Identificação de traves	Inexistente	Inexistente	Consistente
Localização de duas traves	Inexistente	Inexistente	Consistente
Falsos positivos	–	–	Sim

Observa-se um bom resultado no vídeo de teste para o treinamento de 300 iterações, embora seja possível observar a presença de falsos positivos, enquanto para os demais os resultados foram ruins.

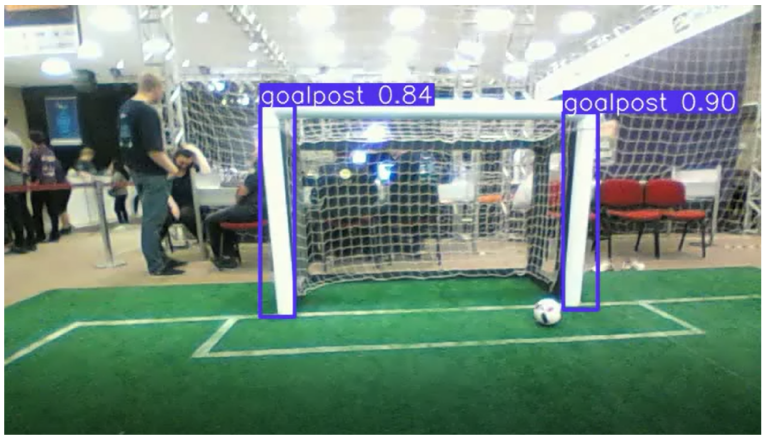
As imagens abaixo mostram alguns exemplos retirados dos vídeos de teste, falsos positivos, dupla identificação.

Figura 4.5 – Falsos positivos do modelo YOLO



Fonte: Própria

Figura 4.6 – Dupla identificação YOLO



Fonte: Própria

4.2.3 Viabilidade

O método de treinamento para esse modelo apresentou melhores resultados se comparados ao modelo anterior, com alta taxa de acerto e confiabilidade. Embora ainda seja necessário testar o modelo em situações reais de identificação de traves, acredita-se que o treinamento em nuvem traga beneficios em relação ao treinamento local.

5 Considerações Finais

Com a chegada do *Big Data*, aumento da capacidade de processamento, e novas tecnologias surgindo os algoritmos de ML ganham força. Com eles vêm também as inteligências utilizando redes neurais e aprendizado profundo. Nota-se que tais tecnologias se tornam cada vez maiores e mais complexas devido ao grande leque de possibilidades de soluções.

Um ponto a se observar nesse avanço é a necessidade de um crescente poder de processamento que muitas das vezes não é acompanhado por indivíduos que têm vontade de aprender ou já trabalham na área. As vezes o treinamento de uma arquitetura dessas se torna inviável devido a falta de processamento. Problema enfrentado pela EDROM, onde muitas vezes ficavam limitados ao notebook dos membros da equipe serem capazes de rodar tais processos.

Para tentar solucionar isso, foram propostas algumas soluções: o método de treinamento utilizando a transferência de aprendizado, que reduz drasticamente a quantidade de variáveis a serem treinadas por utilizar de um modelo previamente treinado, o treinamento em nuvem, que utiliza de processadores disponibilizados remotamente para aplicações como essa.

A partir do desenvolvimento desse projeto foi possível aprender bastante sobre o mundo da Inteligência Artificial, mais especificamente redes neurais convolucionais e os fatores que influenciam tais arquiteturas, comparando modelos e o processo de treinamento em si. Trazendo um grande aprendizado na área.

5.1 Conclusão

Para que os metodos de treinamento se apresentassem viáveis era necessário que se treinasse uma rede com certo nivel de confiança, de forma simples, que pudesse ser utilizada em aplicações em projetos reais e sem que fosse necessário hardwares muito potentes com alto poder de processamento.

Observando os resultados obtidos notou-se uma rede treinável uma fez que foi possível perceber a evolução da qualidade conforme o treinamento ia avançando.

Se apresentando viável também em relação ao tempo e a capacidade de processamento, chegando a treinar ate 6 vezes mais rápido, sem contar a melhora na qualidade do processo, uma vez que computadores mais antigos nao conseguem realizar a atividade com os mesmos parâmetros.

Outra vantagem observada é a possibilidade de se executar o treinamento de qualquer computador com a acesso à internet.

E embora a qualidade da rede, principalmente a *Inceptio* não ter apresentado uma quali-

dade adequada, é possível continuar os estudos e descobrir processos de otimizaçao da rede.

5.2 Trabalhos Futuros

Observando o tema trabalhado e os resultados obtidos, pode-se elencar possíveis temas de melhorias para o projeto atual, sendo eles:

1. Melhoria dos hiperparâmetros, uma vez treinada, é possível tentar calibrar os hiperparâmetros (tamanho de lote, função de ativação, camadas congeladas, número de steps, etc.) para encontrar melhores resultados de treinamento;
2. Tentar o treinamento utilizando diferentes versões dos modelos pré-treinados, existem diversas variações do Inception e também do YOLO, e é possível treinar a rede utilizando o mesmo notebook e descobrir qual o melhor para o objetivo proposto;
3. Adicionar ao treinamento a identificação de outros elementos presentes nos jogos de futebol da competição, bolas, robôs por exemplo;
4. Tentar treinar uma rede capaz de se localizar dentro do campo a partir de parâmetros capturados pela câmera.

Referências

ABADI, M.; AGARWAL, A.; BARHAM, P.; BREVDO, E.; CHEN, Z.; CITRO, C.; CORRADO, G. S.; DAVIS, A.; DEAN, J.; DEVIN, M.; GHEMAWAT, S.; GOODFELLOW, I.; HARP, A.; IRVING, G.; ISARD, M.; JIA, Y.; JOZEFOWICZ, R.; KAISER, L.; KUDLUR, M.; LEVENBERG, J.; MANé, D.; MONGA, R.; MOORE, S.; MURRAY, D.; OLAH, C.; SCHUSTER, M.; SHLENS, J.; STEINER, B.; SUTSKEVER, I.; TALWAR, K.; TUCKER, P.; VANHOUCKE, V.; VASUDEVAN, V.; VIéGAS, F.; VINYALS, O.; WARDEN, P.; WATTENBERG, M.; WICKE, M.; YU, Y.; ZHENG, X. TensorFlow: Large-scale machine learning on heterogeneous systems. 2015. Software available from tensorflow.org. Disponível em: <https://www.tensorflow.org/>.

COCO Dataset. *COCO Dataset*. 2021. <https://cocodataset.org>. Acessado em 28/03/2021.

Data Science Academy. *Deep Learning Book*. 2019. Disponível em: <http://deeplearningbook.com.br/campos-receptivos-locais-em-redes-neurais-convolucionais/>. Acesso em: 13 de março 2021.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>.

HAYKIN, S. *Neural networks: a comprehensive foundation, 2/E*. [S.l.]: Artmed Editora S.A., 1999. ISBN 01327333501.

IMAGENET. 2021. <http://www.image-net.org>. Acessado em 28/03/2021.

PACHECO, C. A. R.; PEREIRA, N. S. Deep learning conceitos e utilização nas diversas Áreas do conhecimento. *Ada Lovelace*, Fábrica de tecnologias Turing, 2018. <http://anais.unievangelica.edu.br/index.php/adalovelace/article/view/4132>.

PEDRINI, H. *Introdu¸c~ao ao Processamento Digital de Imagem*. 2021. Slide-Aula Disponível em: <https://www.ic.unicamp.br/~helio/disciplinas/MC920/aula_cores.pdf>. Acesso em: 10 de março 2022.

REDMON, J.; FARHADI, A. Yolov3: An incremental improvement. *arXiv*, 2018.

ROSEBROCK, A. *Are CNNs invariant to translation, rotation, and scaling?* 2021. Disponível em: <https://pyimagesearch.com/2021/05/14/are-cnns-invariant-to-translation-rotation-and-scaling/>. Acesso em: 14 de março 2022.

SAGE, A. Neural controller based on back-propagation algorithm. *Concise Encyclopedia of Information Processing in Systems & Organizations*, Artmed Editora S.A., p. 510, 1990.

SHAIKH, F. *Inception Neural Network*. 2018. Disponível em: <https://www.analyticsvidhya.com/blog/2018/10/understanding-inception-network-from-scratch/>. Acesso em: 19 de outubro 2021.

SZEGEDY, C.; LIU, W.; JIA, Y.; SERMANET, P.; REED, S. E.; ANGUELOV, D.; ERHAN, D.; VANHOUCKE, V.; RABINOVICH, A. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. Disponível em: <http://arxiv.org/abs/1409.4842>.

VERDONE, A. *Transfer Learning using Tensorflow's Object Detection API: detecting R2-D2 and BB-8*. 2017. Disponível em: <https://averdones.github.io/tensorflow-object-detection-star-wars/>. Acesso em: 19 de outubro 2021.

Apêndices

APÊNDICE A – Notebook Python para o modelo Inception V2

O *Notebook* está disponível para acesso no repositório do GitHub, seguindo o link:

<github.com/arthurnonaka/goalpost_detection/blob/master/goalpost_detection_V1.ipynb>

APÊNDICE B – Notebook Python para o modelo YOLO V3

O *Notebook* está disponível para acesso no repositório do GitHub, seguindo o link:

<github.com/arthurnonaka/goalpost_detection/blob/master/goalpost_detection_yolo_v3.ipynb>

Anexos

ANEXO A – Arquivo de configuração do modelo inception_v2_coco

```
# SSD with Inception v2 configuration for MSCOCO Dataset.
# Users should configure the fine_tune_checkpoint field in the train config as
# well as the label_map_path and input_path fields in the train_input_reader and
# eval_input_reader. Search for "PATH_TO_BE_CONFIGURED" to find the fields that
# should be configured.

model {
  ssd {
    num_classes: 1
    box_coder {
      faster_rcnn_box_coder {
        y_scale: 10.0
        x_scale: 10.0
        height_scale: 5.0
        width_scale: 5.0
      }
    }
train_config: {
  batch_size: 24
  optimizer {
    rms_prop_optimizer: {
      learning_rate: {
        exponential_decay_learning_rate {
          initial_learning_rate: 0.004
          decay_steps: 800720
          decay_factor: 0.95
        }
      }
      momentum_optimizer_value: 0.9
      decay: 0.9
      epsilon: 1.0
    }
  }
  #fine_tune_checkpoint: "./object_detection/data/model.ckpt"
  from_detection_checkpoint: true
  # Note: The below line limits the training process to 200K steps, which we
  # empirically found to be sufficient enough to train the pets dataset. This
  # effectively bypasses the learning rate schedule (the learning rate will
  # never decay). Remove the below line to train indefinitely.
  num_steps: 40000
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  data_augmentation_options {
    ssd_random_crop {
    }
  }
}
  tf_record_input_reader {
    input_path: "./object_detection/data/goalpost_train.record"
  }
  label_map_path: "./object_detection/data/goalpost_label_map.pbtxt"
}

eval_config: {
  num_examples: 8000
  # Note: The below line limits the evaluation process to 10 evaluations.
  # Remove the below line to evaluate indefinitely.
  max_evals: 10
}

eval_input_reader: {
  tf_record_input_reader {
    input_path: "./object_detection/data/goalpost_val.record"
  }
  label_map_path: "./object_detection/data/goalpost_label_map.pbtxt"
```