

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ENGENHARIA ELETRÔNICA**

ANA CLÁUDIA BANDERCHUK

TÍTULO DO TCC

Florianópolis, 2021

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ENGENHARIA ELETRÔNICA**

ANA CLÁUDIA BANDERCHUK

TÍTULO DO TCC

Trabalho de conclusão de curso submetido
ao Instituto Federal de Educação, Ciência
e Tecnologia de Santa Catarina como parte
dos requisitos para obtenção do título de
Engenheiro Eletrônico

Orientador:
Prof. Dr. Fernando Santana Pacheco

Florianópolis, 2021

TÍTULO DO TCC

ANA CLÁUDIA BANDERCHUK

Este Trabalho foi julgado adequado para obtenção do Título de Engenheiro Eletrônico em abril de 2021 e aprovado na sua forma final pela banca examinadora do Curso de Engenharia Eletrônica do instituto Federal de Educação Ciência, e Tecnologia de Santa Catarina.

Florianópolis, 12 de março, 2021.

Banca Examinadora:

Fernando Santana Pacheco, Dr.

RESUMO

Este trabalho apresenta

Palavras-chave: um. dois.

ABSTRACT

This papper presents

Keywords:

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplos de problemas e seus níveis de complexidade para computadores ou seres humanos resolvê-los.	13
Figura 2 – Diagrama de Venn da Inteligência Artificial e suas áreas de estudo.	14
Figura 3 – Fluxograma que diferencia as áreas de estudo da Inteligência Artificial e suas etapas de processamento de dados.	15
Figura 4 – Ilustração das camadas de um modelo de Aprendizado Profundo.	16
Figura 5 – Comparação entre os modelos Aprendizado de Máquina e Aprendizado Profundo.	16
Figura 6 – Estrutura básica de uma Rede Neural com duas camadas.	17
Figura 7 – Diagrama de um Neurônio de uma Rede Neural.	18
Figura 8 – Exemplos de funções de ativação de um neurônio para uma Rede Neural.	19
Figura 9 – Estapas da atualização dos pesos de um nó em uma Rede Neural.	21
Figura 10 – Método do gradiente para uma variável.	21
Figura 11 – Matriz de Confusão para classificação binária.	23
Figura 12 – Ilustração das medidas de Precisão e Sensibilidade.	24
Figura 13 – Curva da Precisão em função da Sensibilidade para quatro classes distintas.	24
Figura 14 – Exemplo de caixa delimitadora prevista por uma rede neural em comparação com a localização real do objeto para o cálculo da IoU.	26
Figura 15 – Componentes de uma Rede Neural Convolucional típica.	27
Figura 16 – Exemplo de convolução entre dois vetores bidimensionais.	28
Figura 17 – Operações de convolução em uma imagem.	29
Figura 18 – Exemplos de funções utilizadas para operação de <i>pooling</i>	30
Figura 19 – Arquitetura da Rede Neural YOLO.	31
Figura 20 – Comparação da precisão média em função do processamento em frames por segundo para diferentes redes neurais utilizadas em detecção de objetos.	31
Figura 21 – Processo de detecção e classificação na rede neural YOLO.	32
Figura 22 – Diferença entre as tarefas de classificação e detecção de objetos em uma imagem.	33
Figura 23 – Tipos de defeito de fabricação em placas de circuito impresso.	33
Figura 24 – Extração de defeitos em placas de circuito impresso utilizando métodos comparativos.	35
Figura 25 – Fluxograma das etapas para classificação e detecção de defeitos utilizadas na abordagem híbrida RBCNN.	36
Figura 26 – Anotações das caixas delimitadoras do dataset HRIPCB.	40

Figura 27 – Estrutura de arquivos do <i>dataset</i> HRIPCB.	41
Figura 28 – Desempenho do início do treinamento da rede neural.	44
Figura 29 – Curvas de precisão em função da sensibilidade para cada classe treinada.	46
Figura 30 – Curva de precisão em função da sensibilidade para o conjunto de classes treinado.	46
Figura 31 – Resultado de predição para imagem com defeito, teste 1.	47
Figura 32 – Resultado de predição para imagem com defeito, teste 2.	48
Figura 33 – Resultado de predição para imagem com defeito, teste 3.	48
Figura 34 – Resultado de predição para imagem com defeito, teste 4.	49
Figura 35 – Resultado de predição para imagem com defeito, teste 5.	49
Figura 36 – Estrutura da interface de aplicação proposta.	50
Figura 37 – Fluxograma da detecção de defeitos de placas de circuito impresso dentro da interface de aplicação proposta.	52
Figura 38 – Tela Inicial 1 da Interface de Aplicação.	54
Figura 39 – Tela Inicial 2 da Interface de Aplicação.	55
Figura 40 – Tela da Interface de Aplicação para imagem com detecções.	56
Figura 41 – Tela da Interface de Aplicação para imagem sem detecções.	57
Figura 42 – PCI Número 01 do conjunto de dados HRIPCB.	70
Figura 43 – PCI Número 04 do conjunto de dados HRIPCB.	70
Figura 44 – PCI Número 05 do conjunto de dados HRIPCB.	71
Figura 45 – PCI Número 06 do conjunto de dados HRIPCB.	71
Figura 46 – PCI Número 07 do conjunto de dados HRIPCB.	72
Figura 47 – PCI Número 08 do conjunto de dados HRIPCB.	72
Figura 48 – PCI Número 09 do conjunto de dados HRIPCB.	73
Figura 49 – PCI Número 10 do conjunto de dados HRIPCB.	73
Figura 50 – PCI Número 11 do conjunto de dados HRIPCB.	74
Figura 51 – PCI Número 12 do conjunto de dados HRIPCB.	74

LISTA DE TABELAS

Tabela 1 – Distribuição dos defeitos no conjunto de dados HRIPCB.	40
Tabela 2 – Representação dos defeitos do conjunto de dados HRIPCB para o treinamento da Rede Neural.	42
Tabela 3 – Resultados de Precisão Média para as classes treinadas de defeitos em placas de circuito impresso.	45
Tabela 4 – Métricas de Avaliação para o conjunto das classes treinadas de defeitos em placas de circuito impresso, considerando um <i>threshold</i> de confiabilidade da predição de 75%.	45
Tabela 5 – Comparação da utilização do conjunto de dados HRIPCB para o reconhecimento de defeitos em placas de circuito impresso para diferentes modelos de Redes Neurais.	58

LISTA DE CÓDIGOS

Código 4.1 – Exemplo de arquivo de anotação para treinamento com YOLO.	42
Código 4.2 – Arquivo de configuração <i>obj.data</i>	42
Código 4.3 – Arquivo de configuração <i>obj.names</i>	43
Código 4.4 – Comando Darknet para o cálculo das métricas de avaliação da rede neural.	44
Código D.1 – Arquivo de Anotação do <i>Dataset HRIPCB</i>	75
Código E.1 – <i>Script para conversão de arquivos de anotação de xml para txt</i>	76
Código F.1 – <i>Script para divisão do dataset</i>	78
Código G.1 – Comandos executados no ambiente Google Colab para o treinamento da Rede Neural.	80
Código H.1 – Compilando o Darknet	81
Código H.2 – Executando a Interface de Aplicação	81

SUMÁRIO

1	INTRODUÇÃO	11
1.1	Justificativa	12
1.2	Descrição do problema	12
1.3	Objetivo geral	12
1.4	Objetivos específicos	12
2	FUNDAMENTAÇÃO TEÓRICA	13
2.1	Inteligência Artificial	13
2.2	Redes Neurais	15
2.2.1	Estrutura de Uma Rede Neural	17
2.2.2	Funções de Ativação	18
2.2.3	Treinamento de Uma Rede Neural	20
2.2.4	Métricas de Avaliação	22
2.2.4.1	Matriz de Confusão	22
2.2.4.2	Acurácia	22
2.2.4.3	Precisão e Sensibilidade	23
2.2.4.4	Precisão Média	23
2.2.4.5	<i>F-Score</i>	25
2.2.4.6	Intersecção sobre União	25
2.2.5	Redes Neurais Convolucionais	25
2.2.5.1	Operação de Convolução	26
2.2.5.2	Operação de <i>Pooling</i>	28
2.2.5.3	<i>You Only Look Once</i>	29
2.3	Classificação e Detecção de Defeitos em Placas de Circuito Impresso	32
2.3.1	Métodos Automatizados de Extração de Defeitos	33
2.3.1.1	Métodos Comparativos	34
2.3.1.2	Métodos Não-Referenciais	34
2.3.1.3	Métodos Híbridos	35
2.4	Frameworks e Bibliotecas para Detecção e Classificação de Objetos	36
2.4.1	Darknet	36
2.4.2	Flask	37
3	METODOLOGIA	38

4	TREINAMENTO DA REDE NEURAL PARA DETECÇÃO E LOCALIZAÇÃO DE DEFEITOS EM PLACAS DE CIRCUITO IMPRESSO	39
4.1	Seleção da Rede Neural e do <i>Framework</i> para o Treinamento	39
4.2	Seleção do Conjunto de Dados	39
4.3	Configuração dos Arquivos para o Treinamento	41
4.4	Treinamento	43
4.5	Resultados	43
5	INTERFACE DE APLICAÇÃO	50
6	DISCUSSÃO	58
7	CONSIDERAÇÕES FINAIS	59
	REFERÊNCIAS	60
	APÊNDICES	65
	APÊNDICE A – NOTAÇÕES UTILIZADAS EM REDES NEURAIS	66
	APÊNDICE B – ETAPAS DE TREINAMENTO UTILIZANDO A REDE NEURAL YOLO	68
	APÊNDICE C – PLACAS DE CIRCUITO IMPRESSO UTILIZADAS PARA O CONJUNTO DE DADOS HRIPCB	70
	APÊNDICE D – EXEMPLO DE ARQUIVO DE ANOTAÇÃO DO CONJUNTO DE DADOS HRIPCB EM XML	75
	APÊNDICE E – <i>SCRIPT</i> UTILIZADO NA CONVERSÃO DOS ARQUIVOS DE ANOTAÇÃO	76
	APÊNDICE F – <i>SCRIPT</i> UTILIZADO NA DIVISÃO DO <i>DATASET</i> PARA TREINAMENTO DA REDE NEURAL	78
	APÊNDICE G – CÓDIGO UTILIZADO PARA O TREINAMENTO DA REDE NEURAL NO AMBIENTE GOOGLE COLAB	80
	APÊNDICE H – CONFIGURAÇÕES PARA A EXECUÇÃO DA INTERFACE DE APLICAÇÃO DE DETECÇÃO DE DEFEITOS EM PLACAS DE CIRCUITO IMPRESSO	81

1 INTRODUÇÃO

Placas de circuito impresso são de longe o método mais utilizado para o projeto de eletrônicos modernos. Elas consistem em um sanduíche de uma ou mais camadas de cobre intercaladas com uma ou mais camadas de material isolante (ZUM-BAHLEN, 2008) e servem de suporte para os componentes eletrônicos responsáveis pelo funcionamento de um circuito eletrônico.

À medida em novas tecnologias são desenvolvidas, as placas de circuito impresso estão se tornando cada vez mais sofisticadas e delicadas (HU; WANG, 2020), de modo que a detecção de incertezas, tolerâncias, defeitos e erros de posição relativa associados ao processo de fabricação (LETA; FELICIANO; MARTINS, 2008) deve ser feita com mais cautela a fim de garantir o funcionamento do produto final.

Dessa forma, automatizar a inspeção de defeitos se tornou essencial para aprimorar a qualidade do processo de fabricação, já que técnicas de medição por visão computacional apresentam melhor regularidade, precisão e repetibilidade quando comparada a inspeção humana que, além da imprecisão e não-repetibilidade, está sujeita a subjetividade, fadiga e lentidão e está, ainda, associada a um alto custo (LETA; FELICIANO; MARTINS, 2008).

A inspeção ótica automatizada tem sido amplamente utilizada para detectar defeitos durante o processo de fabricação de uma PCB (CHIN; HARLOW, 1982). Conforme a evolução dessa tecnologia, três principais métodos de detecção tem se destacado: métodos comparativos, não-referenciais e híbridos (WU; WANG; LIU, 1996). O método comparativo, mais utilizado entre eles, está suscetível à interferência da iluminação e ruídos externos, além de necessitar mecanismos de alinhamento precisos para realização da comparação (HU; WANG, 2020). Com o avanço nos estudos da área de Aprendizado Profundo, o método não-referencial utilizando Redes Neurais tem mostrado ótimo desempenho para a detecção de objetos, podendo ser utilizado para a detecção de defeitos em placas de circuito impresso com resultados de alta confiabilidade.

Dessa forma, esse trabalho propõe o uso de técnicas de Aprendizado Profundo para a detecção de defeitos em placas de circuito impresso, como maneira de aprimorar o processo de fabricação e melhorar a qualidade das mesmas.

1.1 Justificativa

1.2 Descrição do problema

1.3 Objetivo geral

1.4 Objetivos específicos

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Inteligência Artificial

Apesar de ter chamado atenção nos últimos anos com a quantidade enorme de dados adquiridos e processados por grandes corporações como Google, Facebook, Amazon, e Apple (LAWLESS; MITTU; SOFGE, 2020), o termo ‘Inteligência Artificial’ não é tão atual assim. Ele foi utilizado pela primeira vez em 1955 por John McCarthy (ABRAHAM et al., 2021), que conduziu no ano seguinte um workshop cuja premissa central da proposta considera que o comportamento humano inteligente consiste em processos que podem ser formalizados e reproduzidos por uma máquina (DICK, 2019).

Um dos objetivos da Inteligência Artificial, de acordo com Mitchell, Michalski e Carbonell (2013), é fazer com que computadores realizem tarefas mais inteligentes de forma com que não haja necessidade dos seres humanos executá-las. Entretanto, um dos grandes desafios da área de IA atualmente é a execução de atividades consideradas simples e corriqueiras para pessoas, como reconhecimento de objetos e fala (GOODFELLOW; BENGIO; COURVILLE, 2016). Para ilustrar isso, a tabela da Figura 1 apresenta a diferença entre o nível de dificuldade que um computador ou ser humano tem para resolver determinado problema.

Figura 1 – Exemplos de problemas e seus níveis de complexidade para computadores ou seres humanos resolvê-los.

Problema	Computador	Humano
Multiplicar milhares de números grandes rapidamente	FÁCIL	DIFÍCIL
Identificar rostos em uma foto de uma multidão de pessoas	DIFÍCIL	FÁCIL

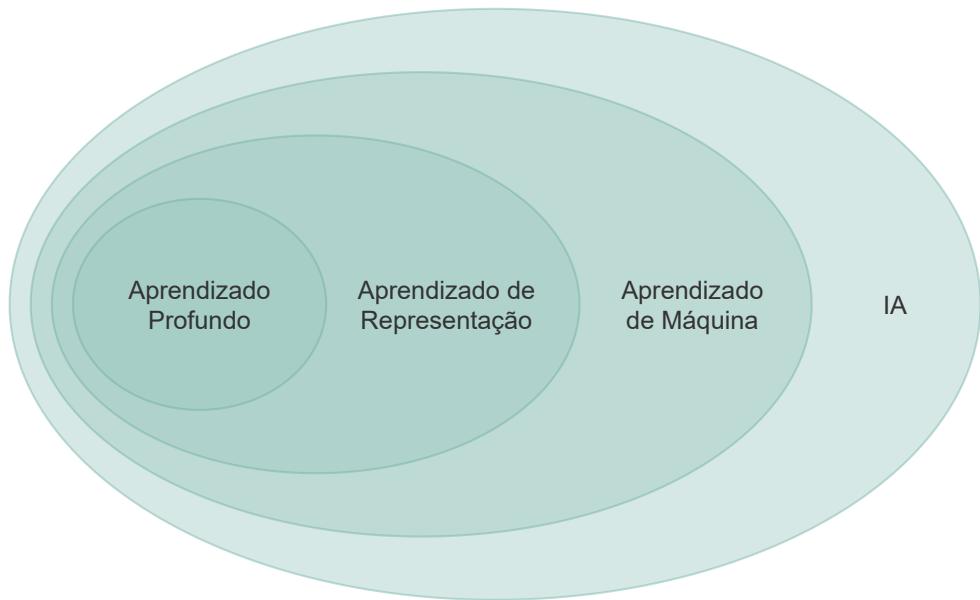
Fonte: Adaptado de Rashid (2016).

A Inteligência Artificial engloba a área de estudo do Aprendizado de Máquina, que por sua vez englobam as áreas do Aprendizado de Representação e Aprendizado Profundo, conforme o diagrama de Venn apresentado na Figura 2.

Para evidenciar as diferenças entre essas áreas de estudo, em comparação com simples Sistemas Baseados em Regras, Goodfellow, Bengio e Courville (2016) apresenta um fluxograma com as etapas de processamento entre a entrada e a saída de dados para cada uma delas, presente na Figura 3.

Sistemas Baseados em Regras não possuem componentes capazes de realizar qualquer aprendizado a partir de dados (GOODFELLOW; BENGIO; COUR-

Figura 2 – Diagrama de Venn da Inteligência Artificial e suas áreas de estudo.

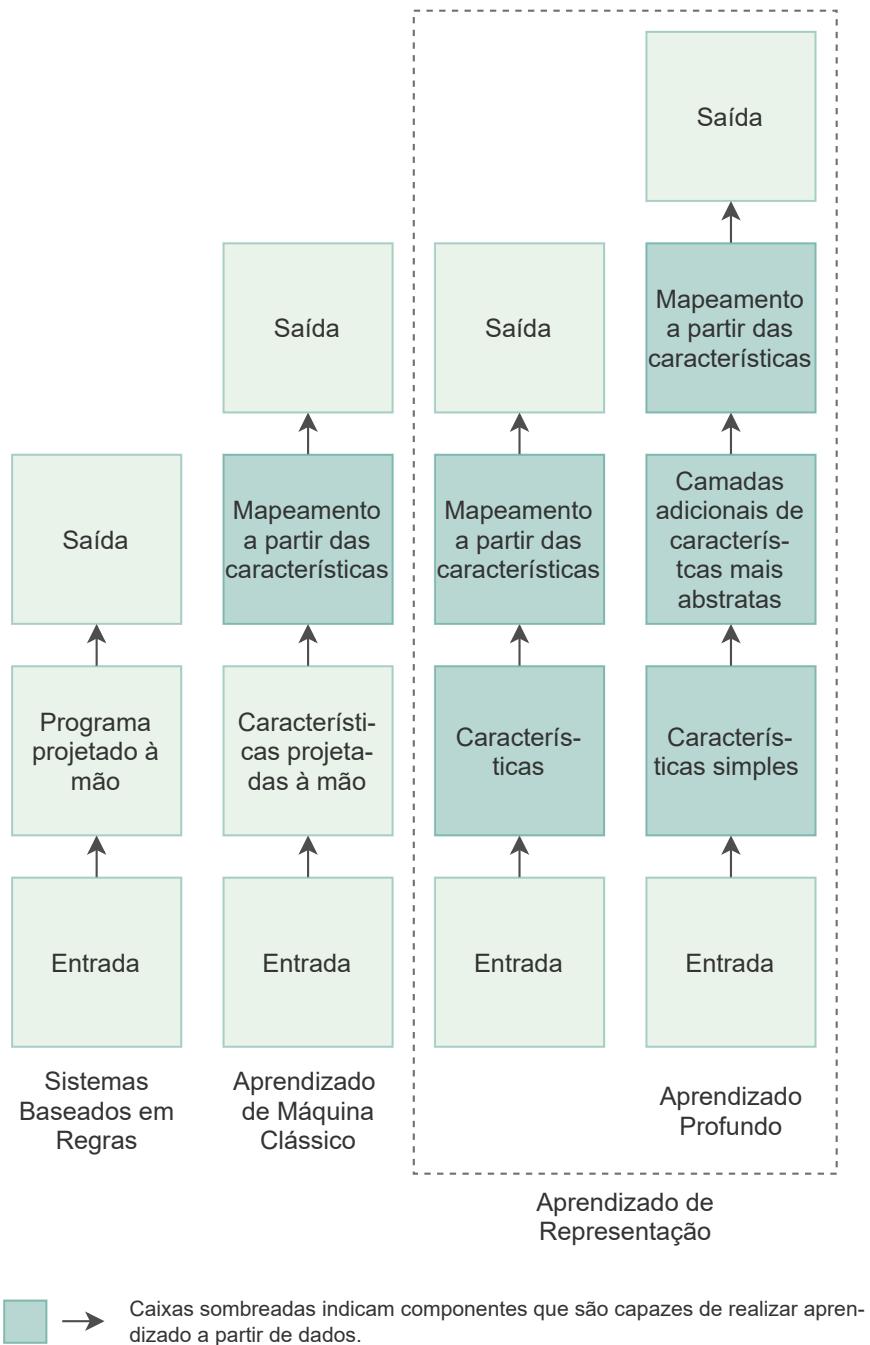


Fonte: Adaptado de Goodfellow, Bengio e Courville (2016).

VILLE, 2016) e são utilizados para resolução de problemas e/ou execução de tarefas que podem ser descritos por uma lista de regras formais, como por exemplo jogar Xadrez. Ao contrários desses sistemas, o Aprendizado de Máquina (do inglês *Machine Learning*) utiliza algoritmos computacionais para transformar características reunidas empiricamente em modelos utilizáveis (EDGAR; MANZ, 2017) possuindo a capacidade “de se aprimorar [...], aprendendo novos conhecimentos ao invés de serem programado com eles” (WOOLF, 2009).

No Aprendizado de Representação (do inglês *Feature Learning* ou *Representation Learning*) entretanto, não há necessidade de mapear manualmente essas características (GOODFELLOW; BENGIO; COURVILLE, 2016). Ou seja, por conta própria e de forma abstrata, os algoritmos são capazes de extrair as características importantes para a construção dos modelos utilizando redes neurais (ROBINS, 2020) (LESORT et al., 2018). O Aprendizado Profundo, por sua vez, resolve a dificuldade que o Aprendizado de Representação possui de extrair características abstratas de alto nível, tais como sotaques de um locutor (GOODFELLOW; BENGIO; COURVILLE, 2016). Segundo Mao et al. (2019), o Aprendizado Profundo “imita a função que o cérebro humano possui de interpretar dados usando redes neurais de várias camadas”. Um exemplo dessas variadas camadas é apresentado na Figura 4, onde características distintas são extraídas por cada camada. Uma comparação com o modelo de Aprendizado de Máquina é ilustrado na Figura 5.

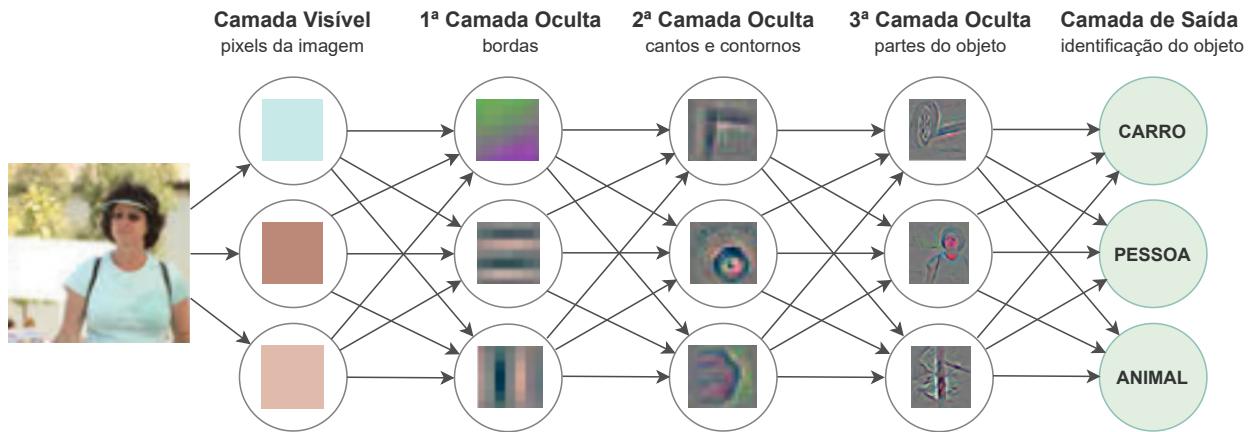
Figura 3 – Fluxograma que diferencia as áreas de estudo da Inteligência Artificial e suas etapas de processamento de dados.



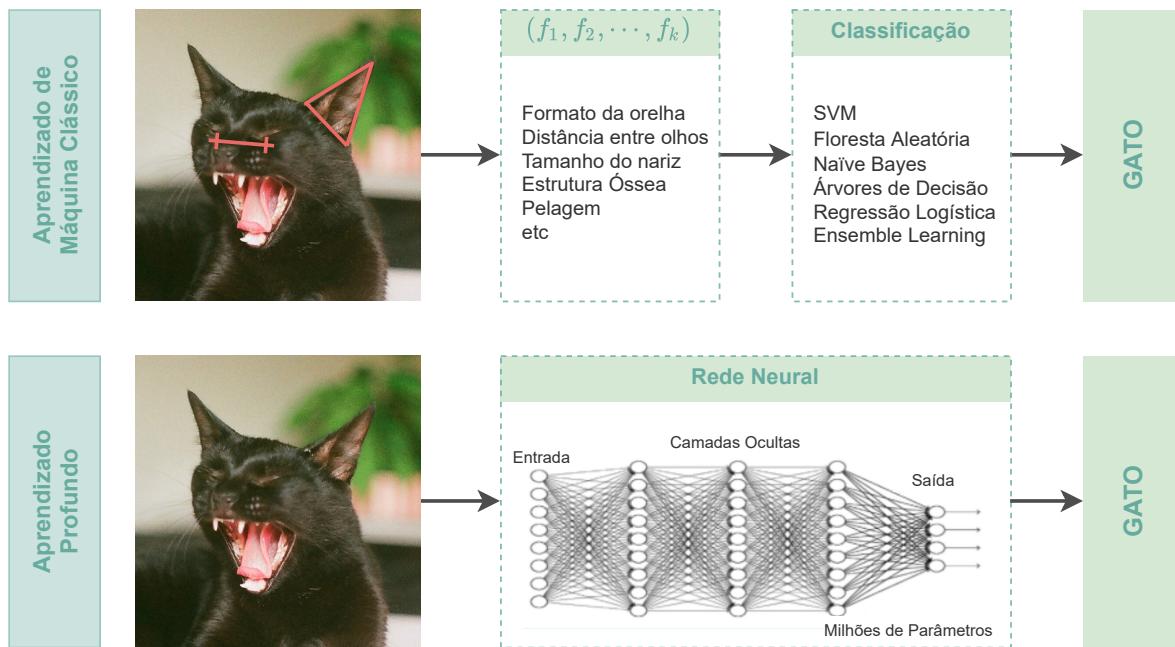
Fonte: Adaptado de Goodfellow, Bengio e Courville (2016).

2.2 Redes Neurais

Ao contrário da abordagem convencional de programação, onde dizemos a um computador o que deve ser feito ao dividir um problema em pequenas tarefas para que ele execute, uma Rede Neural (do inglês *Neural Network*) utiliza dados

Figura 4 – Ilustração das camadas de um modelo de Aprendizado Profundo.

Fonte: Adaptado de Goodfellow, Bengio e Courville (2016).

Figura 5 – Comparação entre os modelos Aprendizado de Máquina e Aprendizado Profundo.

Fonte: Adaptado de Robins (2020).

observacionais para aprender como resolver o problema (NIELSEN, 2018).

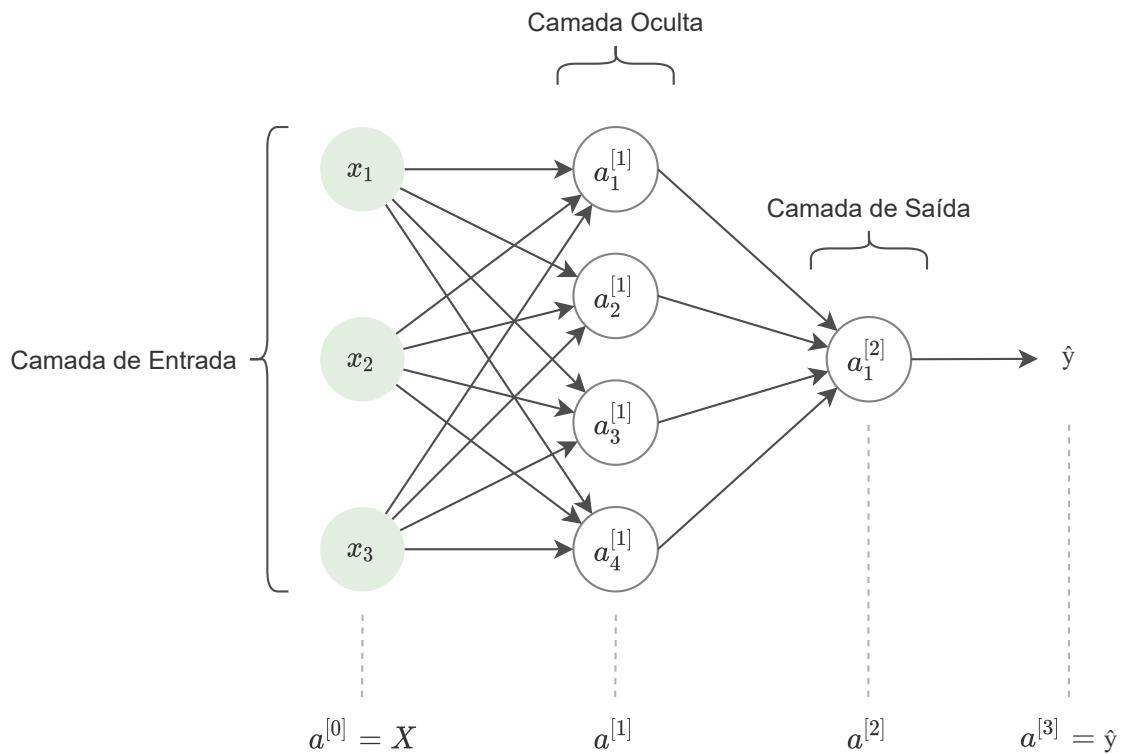
Walczak e Cerpa (2003) define Redes Neurais Artificiais como modelos que “simulam a atividade elétrica do cérebro e do sistema nervoso”. Porém enquanto alguns tipos de redes neurais tem sido utilizadas para entender o funcionamento do cérebro, na perspectiva de Aprendizado Profundo elas não são projetadas para serem modelos

realistas da função biológica (GOODFELLOW; BENGIO; COURVILLE, 2016).

2.2.1 Estrutura de Uma Rede Neural

Uma Rede Neural é composta por camadas de nós, conhecidos como neurônios, e conexões que interligam as saídas e entradas desses nós. A estrutura básica de uma Rede Neural é ilustrada na Figura 6, onde a primeira camada de neurônios é a camada de entrada, a última camada é a camada de saída e as camadas entre elas são chamadas de camadas ocultas. Na Figura 6, vetor X contém os valores de entrada, os vetores $a^{[l]}$ representam as funções de ativação referentes à l -ésima camada, e \hat{y} é o vetor de saída com os valores preditos. As notações utilizadas nesse capítulo respeitam as propostas por Ng (2020) presentes no Apêndice A.

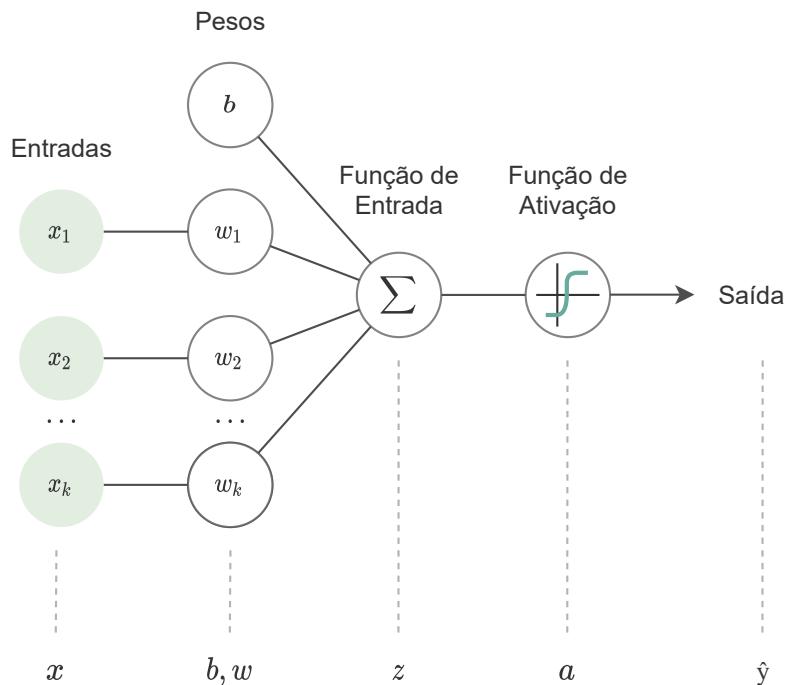
Figura 6 – Estrutura básica de uma Rede Neural com duas camadas.



Fonte: Elaboração própria (2021).

São nos neurônios que ocorrem as computações. A Figura 7 mostra um diagrama de um nó de uma Rede Neural, onde os pesos representados por b e w_k são responsáveis por atribuir significância às entradas com relação à tarefa que o algoritmo está tentando aprender (NICHOLSON, 2020). Esses produtos são então somados e passam por uma função de ativação.

Figura 7 – Diagrama de um Neurônio de uma Rede Neural.



Fonte: Adaptado de Nicholson (2020).

2.2.2 Funções de Ativação

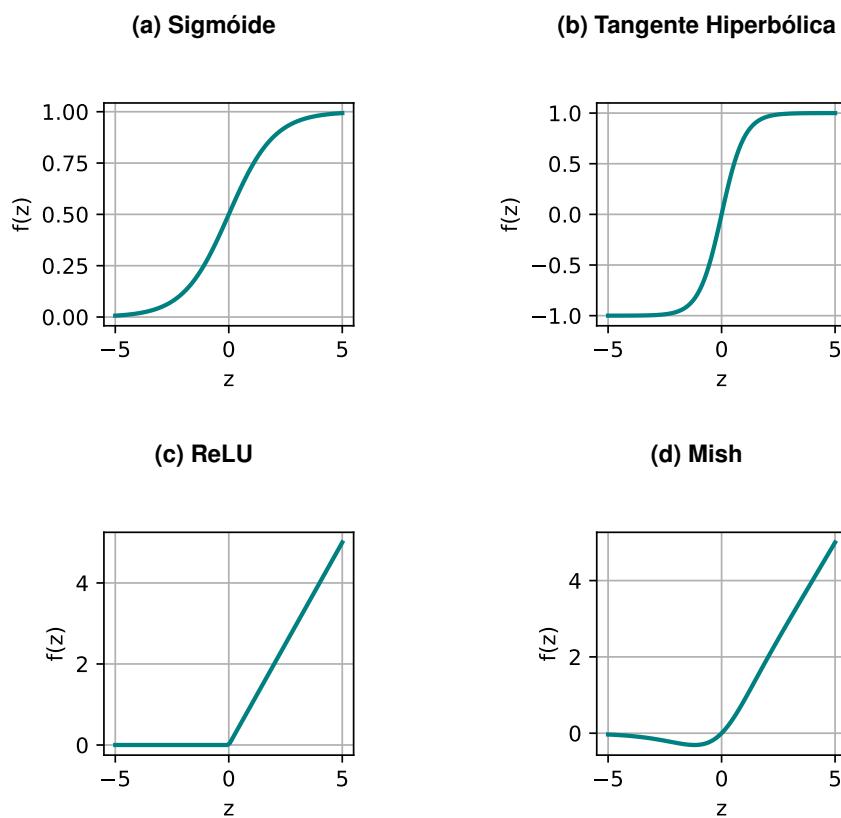
As funções de ativação desempenham um papel crucial na dinâmica de desempenho e treinamento em redes neurais (MISRA, 2019), determinando se um sinal deve progredir e em que medida ele deve progredir através da rede (NICHOLSON, 2020). Alguns exemplos de função de ativação estão na Figura 8.

A função Sigmoide definida pela Equação 2.1 Sharma (2017) é geralmente utilizada em classificações binárias, pois o resultado está sempre no intervalo entre zero e um (NG, 2020). A Figura 8a mostra um gráfico da função Sigmoide.

$$f(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \quad (2.1)$$

Para a otimização dos pesos durante o treinamento de uma rede neural, em geral é necessário calcular o gradiente das funções de ativação. Por isso, de acordo com Sharma (2017), o uso da tangente hiperbólica como função de ativação é preferível pois tem gradientes que não estão restritos a variar em uma certa direção e são mais acentuados quando comparado à sigmoide. A definição da tangente hiperbólica da

Figura 8 – Exemplos de funções de ativação de um neurônio para uma Rede Neural.



Fonte: Elaboração própria (2021).

Figura 8b está na Equação 2.2.

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (2.2)$$

Porém, para valores muito grandes ou muito pequenos de z , a derivada resultante, utilizada no gradiente, tende a ser nula tanto para a sigmoide quanto para a tangente hiperbólica (NG, 2020) o que geralmente causa lentidão no treinamento (MISRA, 2019). Para que isso não ocorra, utiliza-se a função ReLU, do inglês *Rectified Linear Unit*, mostrada na Figura 8c e definida na Equação 2.3. A ReLU é a função de ativação mais utilizada (NG, 2020) e é a função padrão recomendada para uso com a maioria das redes neurais (GOODFELLOW; BENGIO; COURVILLE, 2016).

$$f(z) = \text{ReLU}(z) = \max(0, z) \quad (2.3)$$

Com a evolução das Redes Neurais na última década, diversas funções de ativação vem sendo propostas com o objetivo de melhorar o desempenho do

treinamento. Um exemplo delas é a Mish (Figura 8d), proposta por Misra (2019) cuja função é definida pela Equação 2.4.

$$f(z) = z \cdot \tanh(\ln(1 + e^z)) \quad (2.4)$$

Segundo Misra (2019), ao contrário da ReLU, a função Mish é continuamente diferenciável, evitando efeitos colaterais indesejados quando utiliza-se a otimização baseada no método do gradiente.

2.2.3 Treinamento de Uma Rede Neural

Para o treinamento de uma Rede Neural, utiliza-se um conjunto de exemplos de treinamento $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, onde $x^{(i)}, y^{(i)}$ são, respectivamente, a entrada e a saída real do i -ésimo exemplo de treinamento de um conjunto contendo m exemplos de treinamento (NG, 2020). Para cada um desses exemplos, o objetivo é fazer o cálculo dos pesos w e b de forma que a saída estimada $\hat{y}^{(i)}$ seja próxima da saída real $y^{(i)}$.

O peso b é chamado de *bias* e não é multiplicado a nenhum valor da entrada para caso todos os valores de x sejam nulos, a saída $\hat{y}^{(i)}$ seja diferente de zero. O cálculo desses pesos é feito de forma iterativa e geralmente são iniciados de forma aleatória.

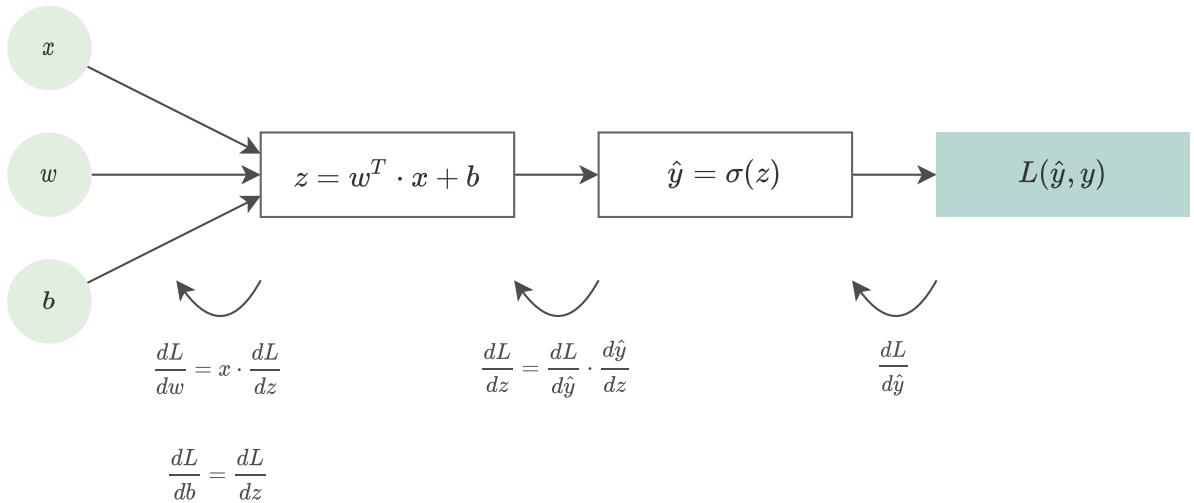
O processo de cálculo considerando um nó de uma rede neural está ilustrado na Figura 9, onde a função de ativação utilizada é a sigmoide e todos os valores calculados e os de entrada são representados na forma vetorial.

A primeira etapa a ser computada após a passagem das entradas pela função de ativação é função de perda, que mede quão boa é a saída prevista \hat{y} em comparação aos valores verdadeiros de y para apenas um conjunto de treinamento (NG, 2020).

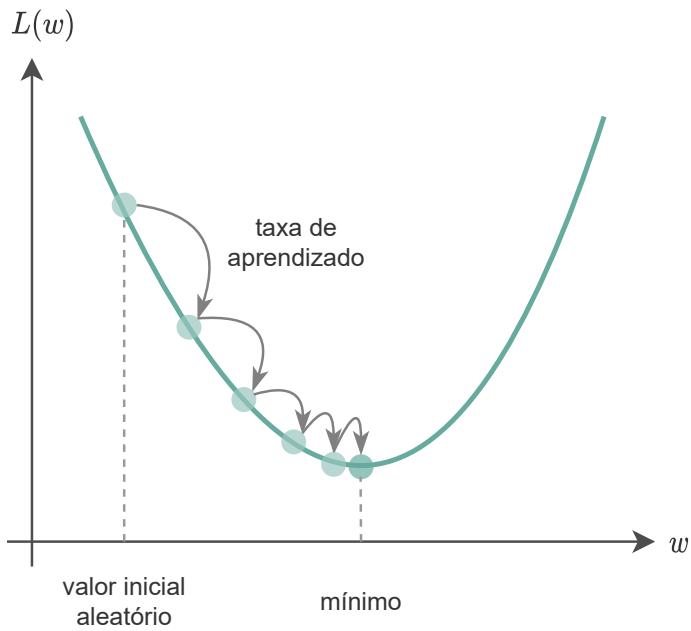
Um exemplo de função de perda que compara \hat{y} com y pode ser definida pela Equação 2.5 (NG, 2020), onde o objetivo do treinamento é obter $L(y, \hat{y}) = 0$.

$$L(y, \hat{y}) = -(y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})) \quad (2.5)$$

Dessa forma, como $\hat{y} = f(w, b)$, é necessário encontrar valores para os pesos w e b que minimizem a função de perda $L(y, \hat{y})$. Para isso, utiliza-se o método do gradiente, um algoritmo de primeira ordem iterativo utilizado em otimização para encontrar o mínimo local de uma função (YAN, 2021), exemplificado na Figura 10 para uma função de perda considerando apenas a variável w .

Figura 9 – Estapas da atualização dos pesos de um nó em uma Rede Neural.

Fonte: Adaptado de Ng (2020).

Figura 10 – Método do gradiente para uma variável.

Fonte: Adaptado de Sauer (2020).

Após serem computadas as derivadas da Figura 9, os pesos w e b são atualizados conforme a Equação 2.6 e a Equação 2.7 onde α representa uma taxa de

aprendizado (NG, 2020).

$$w = w - \alpha \cdot \frac{dL}{dw} \quad (2.6)$$

$$b = b - \alpha \cdot \frac{dL}{db} \quad (2.7)$$

A atualização desses pesos é feita até que a função de perda resulte valores muito próximos de zero ou quando o erro está dentro de uma faixa aceitável de acordo com a métrica de avaliação utilizada.

2.2.4 Métricas de Avaliação

Determinar os objetivos de treinamento em uma rede neural em termos de qual métrica de erro será utilizada é uma etapa necessária, já que para a maioria das aplicações é impossível obter erro zero absoluto. (GOODFELLOW; BENGIO; COURVILLE, 2016).

2.2.4.1 Matriz de Confusão

A Matriz de Confusão é uma técnica muito popular usada em problemas de classificação binária, representando a contagem dos resultados da predição feitos pela rede neural em comparação com os valores reais (KULKARNI; CHONG; BATARSEH, 2020), na forma de verdadeiros positivos, verdadeiros negativos, falsos positivos e falsos negativos. Um exemplo de matriz de confusão utilizada em classificações binárias está na Figura 11.

2.2.4.2 Acurácia

Uma das métricas mais comuns utilizadas durante a classificação é a Acurácia (KULKARNI; CHONG; BATARSEH, 2020), que pode ser calculada a partir da matriz de confusão utilizando a Equação 2.8, definida pela razão entre o número de previsões corretas e o número total de previsões.

$$\text{Acurácia} = \frac{VN + VP}{VN + FP + FN + VP} \quad (2.8)$$

A Acurácia pode causar uma impressão incorreta se utilizada como métrica no treinamento de um conjunto de dados desequilibrados, portanto outros métodos baseados na matriz de confusão podem ser utilizados (KULKARNI; CHONG; BATARSEH, 2020), como os métodos de Precisão e Sensibilidade.

Figura 11 – Matriz de Confusão para classificação binária.

		Classe Prevista	
		A	B
Classe Observada	A	Verdadeiros Negativos (VN)	Falsos Positivos (FP)
	B	Falsos Negativos (FN)	Verdadeiros Positivos (VP)

Fonte: Adaptado de Kulkarni, Chong e Batarseh (2020).

2.2.4.3 Precisão e Sensibilidade

A medida de Precisão (do inglês *Precision*), definida pela Equação 2.9, representa a fração de detecções feitas de forma correta; enquanto a Sensibilidade, conhecida do inglês como *Sensitivity* ou *Recall*, representa apenas a fração das previsões verdadeiras (GOODFELLOW; BENGIO; COURVILLE, 2016), como mostra a Equação 2.10. Ambas métricas fornecem informações importantes a respeito do treinamento, porém o objetivo é melhorar a Sensibilidade sem que a Precisão seja afetada (CHAWLA, 2005 apud KULKARNI; CHONG; BATARSEH, 2020). Uma ilustração das medidas de Precisão e Sensibilidade está na Figura 12.

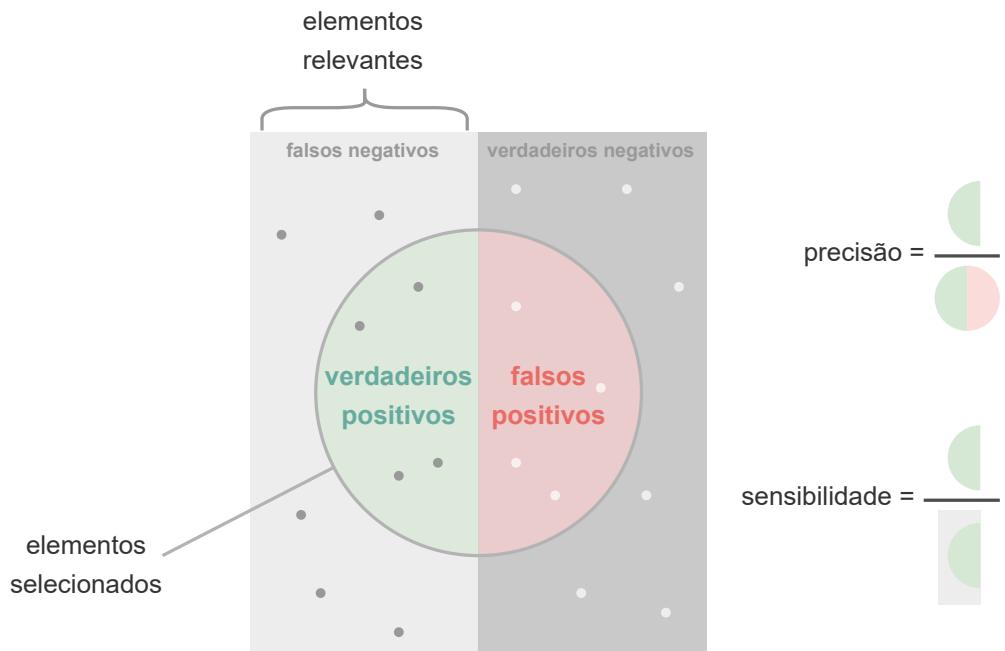
$$\text{Precisão} = \frac{\text{VP}}{\text{VP} + \text{FP}} \quad (2.9)$$

$$\text{Sensibilidade} = \frac{\text{VP}}{\text{VP} + \text{FN}} \quad (2.10)$$

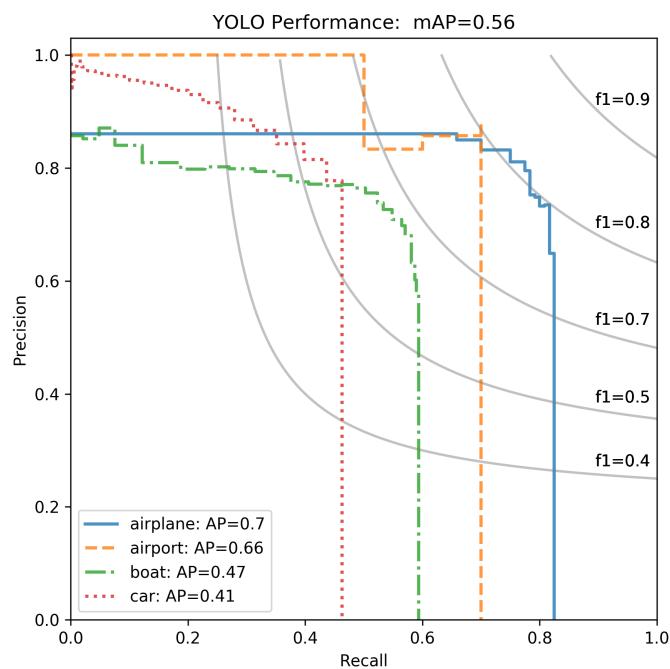
Uma curva de Precisão em função da Sensibilidade pode ser construída ao variar o *threshold* da probabilidade de um resultado pertencer a determinada classe (STEEN, 2020). Um exemplo dessa curva está na Figura 13, onde o eixo horizontal indica a Sensibilidade e o eixo vertical, a Precisão para quatro classes distintas.

2.2.4.4 Precisão Média

O valor da Precisão Média de cada classe, do inglês Average Precision (AP), é encontrado ao calcular a área sob a curva da Precisão em função da Sensibilidade. Já a média desses valores, conhecida no inglês como Mean Average Precision (mAP) é

Figura 12 – Ilustração das medidas de Precisão e Sensibilidade.

Fonte: Adaptado de Tan (2019).

Figura 13 – Curva da Precisão em função da Sensibilidade para quatro classes distintas.

Fonte: Etten (2018).

feita como maneira de avaliar um conjunto de treino completo, levando em conta todas as classes treinadas (TAN, 2019).

2.2.4.5 F-Score

A medida de *F-Score* procura encontrar o equilíbrio entre a Precisão e a Sensibilidade (MISHRA, 2018) e é feita por meio de uma média harmônica ponderada entre essas medidas (KULKARNI; CHONG; BATARSEH, 2020), conforme a Equação 2.11. O resultado está sempre dentro do intervalo entre zero e um e quanto maior o valor de *F-Score*, melhor é a performance do modelo (MISHRA, 2018).

$$F - Score = 2 \cdot \frac{\text{Precisão} \cdot \text{Sensibilidade}}{\text{Precisão} + \text{Sensibilidade}} \quad (2.11)$$

2.2.4.6 Intersecção sobre União

Para avaliar o treinamento da rede neural para detecção de objetos e encontrar os valores das métricas baseadas na matriz de confusão, utiliza-se o conceito de Intersecção sobre União.

A Intersecção sobre União, do inglês *Intersection over Union* (IoU), também conhecida como índice de Jaccard, é a métrica mais usada para comparar a similaridade entre duas formas arbitrárias (REZATOFIGHI et al., 2019). De forma geral, o IoU pode ser calculado pela Equação 2.12 (REZATOFIGHI et al., 2019), onde a área das formas arbitrárias são representadas pelas letras A e B.

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} \quad (2.12)$$

No caso da detecção de objetos em imagens, as formas utilizadas são as caixas delimitadoras desenhadas ao redor dos objetos na imagem, de forma que a IoU é calculada por meio da comparação entre a caixa delimitadora detectada pela rede neural e a localização real do objeto, como ilustra a Figura 14.

As métricas de AP e mAP discutidas na subseção 2.2.4.4 são obtidas variando a IoU para cada classe em um conjunto de imagens treinadas, onde uma detecção é considerada como um verdadeiro positivo se a IoU é maior que certo limiar (SIVARAJKUMAR, 2019) .

2.2.5 Redes Neurais Convolucionais

Redes Neurais Profundas são caracterizadas por sua grande quantidade de camadas ocultas. Cada uma das camadas é responsável pelo treinamento de um

Figura 14 – Exemplo de caixa delimitadora prevista por uma rede neural em comparação com a localização real do objeto para o cálculo da IoU.



Fonte: Adaptado de Sivarajkumar (2019).

conjunto distinto de dados com base na saída da camada anterior, de forma que quanto mais há avanço pela rede neural, mais complexos são as características que os nós podem reconhecer (NICHOLSON, 2020), como ilustrado na Figura 4.

Redes Neurais Convolucionais, do inglês *convolutional neural networks* (CNN), são um tipo de rede neural profunda, originalmente projetadas para análise de imagens (ZHU et al., 2018) e tem sido empregadas para esse tipo de processamento desde 1995 (YAN, 2021). O uso de CNNs reduz os requisitos de memória e possui uma melhor eficiência estática quando comparada a redes neurais tradicionais (GOODFELLOW; BENGIO; COURVILLE, 2016).

CNNs sempre contém dois tipos de operações básicas: as operações de convolução e *pooling* (ZHU et al., 2018), além de utilizarem funções de ativação. Os componentes de uma CNN típica estão ilustrados na Figura 15.

2.2.5.1 Operação de Convolução

Convolução é uma operação matemática onde duas funções produzem uma terceira função que expressa como o formato de uma é modificado ou filtrado pela outra (YAN, 2021). A operação de convolução em uma rede neural utiliza múltiplos tipos de *kernel* para extrair as características do conjunto de dados da entrada (ZHU et al., 2018), geralmente uma imagem, e cria mapas de características a partir dela

Figura 15 – Componentes de uma Rede Neural Convolucional típica.



Fonte: Adaptado de Goodfellow, Bengio e Courville (2016).

(GHOLAMALINEZHAD; KHOSRAVI, 2020). O *kernel* usualmente é um vetor de parâmetros multidimensional que é adaptado pelo algoritmo de aprendizado (GOODFELLOW; BENGIO; COURVILLE, 2016), assim como ocorre a atualização dos pesos w e b vistos na subseção 2.2.3.

Na sua forma discreta, a operação de convolução pode ser definida pela Equação 2.13 (GOODFELLOW; BENGIO; COURVILLE, 2016), onde x representa a entrada e w representa o *kernel* utilizado.

$$y[n] = (x * w)[n] = \sum_{a=-\infty}^{\infty} x[a] \cdot w[n-a] \quad (2.13)$$

Para imagens, as convoluções ocorrem em mais que um eixo ao mesmo tempo. Se a entrada é uma imagem bidimensional I , o *kernel* K provavelmente também será bidimensional, como define a Equação 2.14 (GOODFELLOW; BENGIO; COURVILLE, 2016).

$$Y[i, j] = (I * K)[i, j] = \sum_m \sum_n I[m, n] \cdot K[i - m, j - n] \quad (2.14)$$

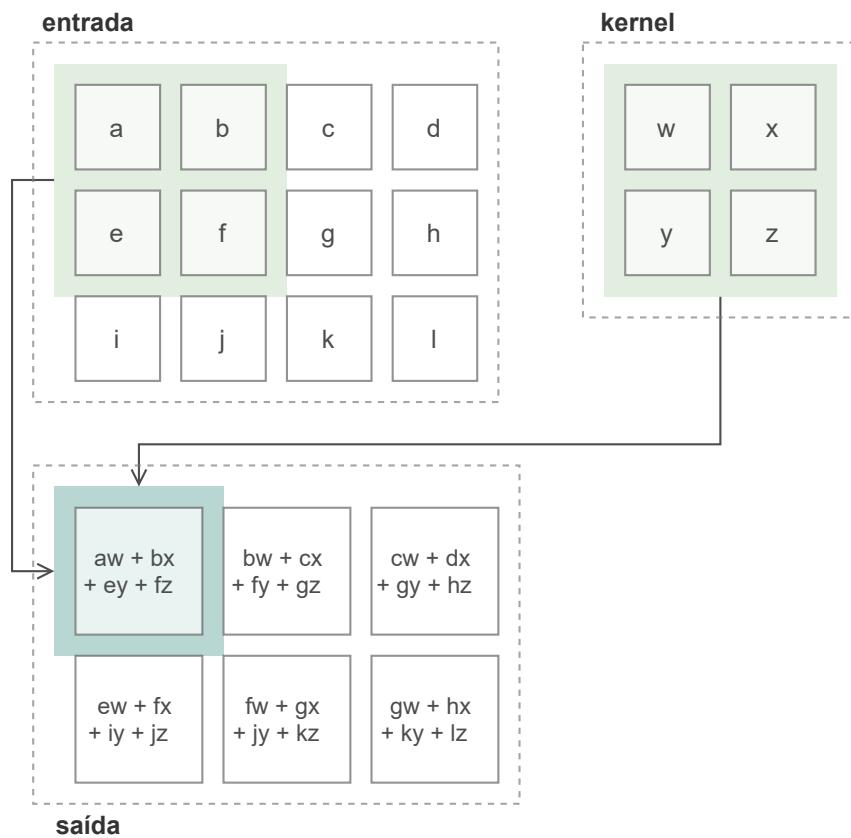
Porém, utilizando a propriedade comutativa da convolução ao inverter a ordem dos fatores I e K , a implementação se torna mais direta, já que há menos variação

no intervalo válido de valores de m e n (GOODFELLOW; BENGIO; COURVILLE, 2016), pois o *kernel* é menor. Além disso, para desinverter o *kernel* em relação à imagem, utiliza-se a operação chamada correlação cruzada (GOODFELLOW; BENGIO; COURVILLE, 2016), como mostra a Equação 2.15.

$$Y[i, j] = (I * K)[i, j] = \sum_m \sum_n I[i + m, j + n] \cdot K[m, n] \quad (2.15)$$

Um exemplo de uma convolução descrita pela Equação 2.15 utilizando um *kernel* de 2×2 está na figura Figura 16.

Figura 16 – Exemplo de convolução entre dois vetores bidimensionais.



Fonte: Adaptado de Goodfellow, Bengio e Courville (2016).

Exemplos de convolução em imagem utilizando distintos tipos de *kernel* está na Figura 17.

2.2.5.2 Operação de Pooling

As operações de *pooling* geralmente são utilizadas após as camadas de convolução para simplificar as informações das saídas dessas camadas (NIELSEN,

Figura 17 – Operações de convolução em uma imagem.

Operação	Kernel	Imagen Resultante
identidade	$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$	
detecção de borda	$\begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix}$ $\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$	 

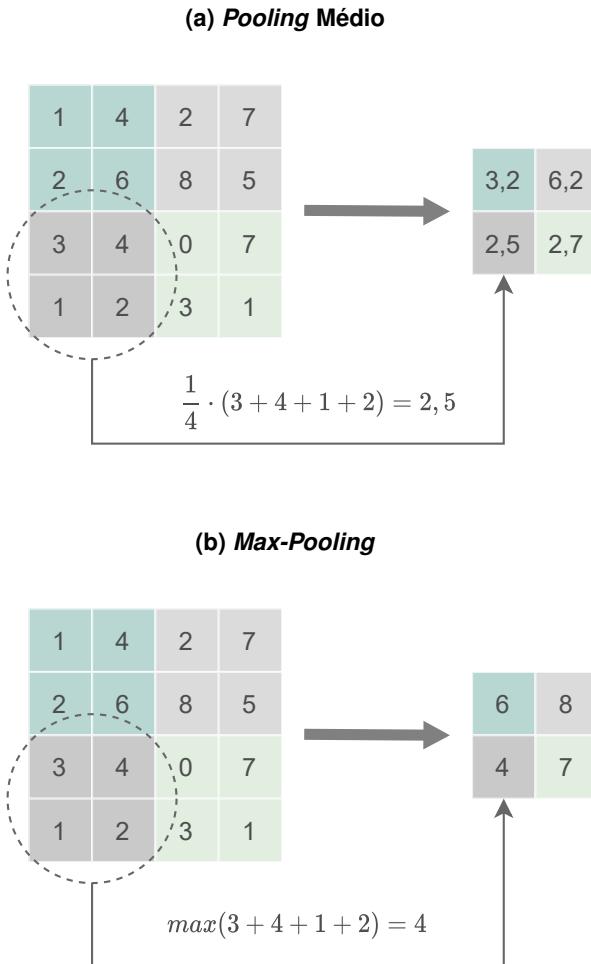
Fonte: Adaptado de Albawi, Mohammed e Al-Zawi (2017).

2018) ao reduzir o tamanho do mapa de características (GHOLAMALINEZHAD; KHOSRAVI, 2020) e ajudam a tornar as saídas aproximadamente invariantes a pequenas modificações da entrada (GOODFELLOW; BENGIO; COURVILLE, 2016).

Espera-se que uma camada de *pooling* ideal extraia somente as informações úteis e descarte detalhes irrelevantes (GHOLAMALINEZHAD; KHOSRAVI, 2020). A Figura 18 apresenta algumas das funções de *pooling*. A operação de *Max-Pooling* seleciona o pixel com maior valor dentre os píxeis de uma região (Figura 18b) e a operação de *Pooling* médio (Figura 18a) faz a média entre esses píxeis.

2.2.5.3 You Only Look Once

You Only Look Once (YOLO), cuja tradução direta do inglês é “você olha apenas uma vez”, é uma rede neural de passagem única (YAN, 2021) onde, ao contrário de outras abordagens, uma única rede neural é aplicada na imagem para fazer a

Figura 18 – Exemplos de funções utilizadas para operação de pooling.

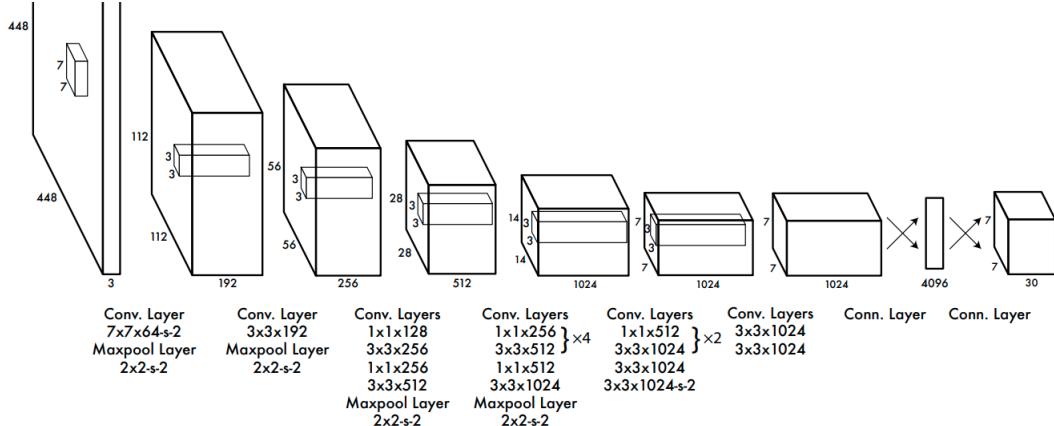
Fonte: Adaptado de Gholamalinezhad e Khosravi (2020).

localização e a detecção do objeto, dividindo a imagem em regiões menores e fazendo a previsão e a probabilidade da detecção para cada uma das regiões (REDMON; FARHADI, 2018).

A rede neural YOLO possui 24 camadas convolucionais seguidas por duas camadas totalmente conectadas (YAN, 2021). A arquitetura básica da primeira versão está na Figura 19.

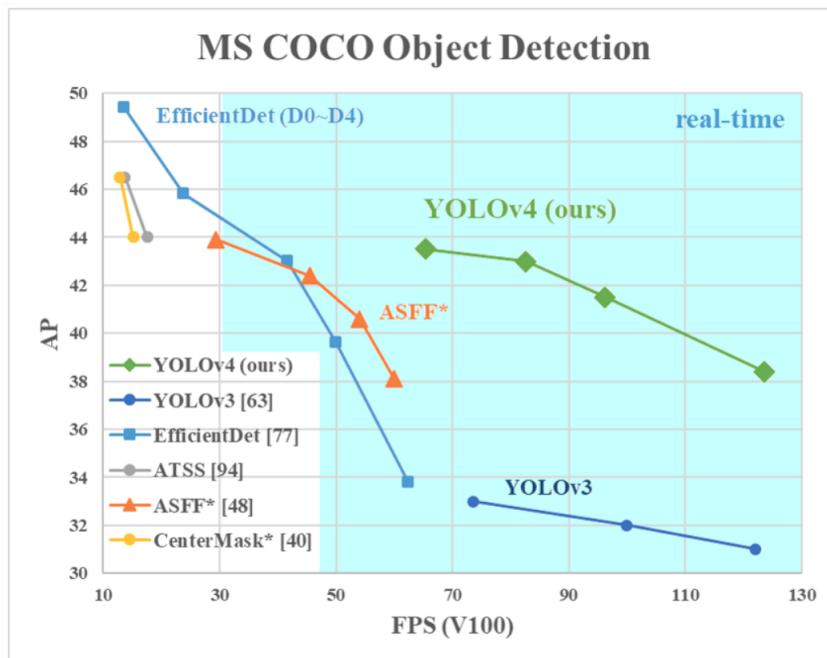
Atualmente, a rede neural YOLO está na versão 4, apresentando otimização em funções de ativação (utilizando a função Mish da Figura 8d para algumas camadas), processamento dos dados, treinamento, funções de perda, etc, em comparação com a versão anterior (WANG et al., 2021).

Uma comparação de desempenho com outras redes neurais utilizadas para detecção de objetos em tempo real está no gráfico da Figura 20, que avalia a precisão média em função da quantidade de *frames* por segundo que a rede consegue processar,

Figura 19 – Arquitetura da Rede Neural YOLO.

Fonte: Redmon et al. (2015).

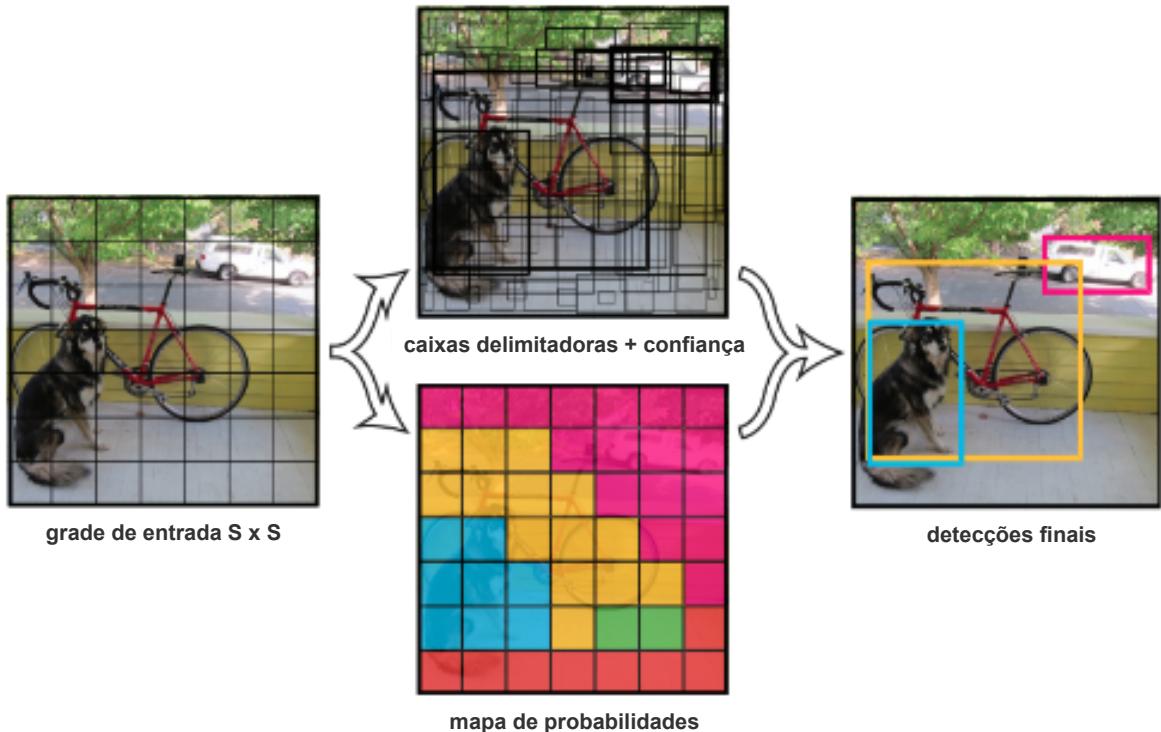
utilizando o *dataset Microsoft COCO: Common Objects in Context* (LIN et al., 2014).

Figura 20 – Comparação da precisão média em função do processamento em frames por segundo para diferentes redes neurais utilizadas em detecção de objetos.

Fonte: Bochkovskiy, Wang e Liao (2020).

A YOLO divide a imagem em blocos menores no formato de uma grade, e para cada uma dessas divisões, são previstas caixas delimitadoras, com suas confiâncias e probabilidades (REDMON et al., 2015). A maioria dessas divisões não contém um objeto detectado e a filtragem é feita com base na probabilidade da classe do objeto, onde apenas as caixas delimitadoras de maior probabilidade permanecerão (SIVARAJKUMAR, 2019). Esse processo é ilustrado na Figura 21.

Figura 21 – Processo de detecção e classificação na rede neural YOLO.



Fonte: Adaptado de Redmon et al. (2015).

A camada final é responsável por prever as probabilidades das classes treinadas e as coordenadas das caixas delimitadoras (REDMON et al., 2015). Tanto a largura e a altura das caixas delimitadoras quanto as coordenadas x e y são normalizadas pela altura e largura da imagem, de modo que os valores fiquem dentro do intervalo entre zero e um (REDMON et al., 2015).

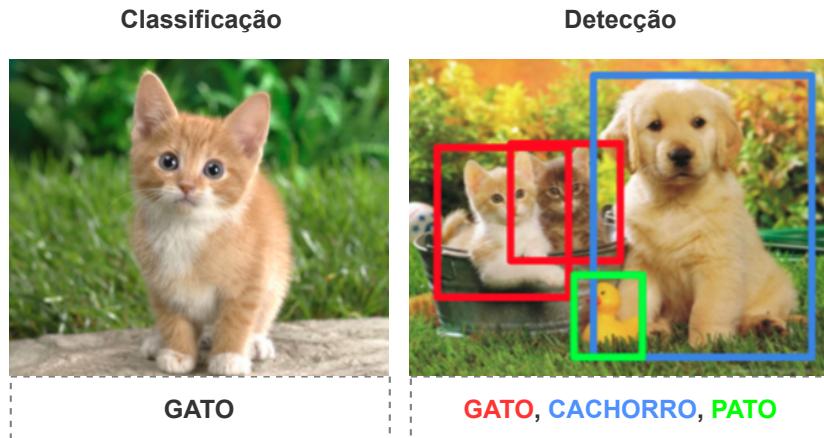
Para o treinamento, são necessárias algumas configurações de arquivos, além da adição das imagens do *dataset* utilizado, a partir do *download* do repositório do projeto disponibilizado por Bochkovskiy (2020). As instruções a serem seguidas, conforme Bochkovskiy (2020), estão no Apêndice B.

2.3 Classificação e Detecção de Defeitos em Placas de Circuito Impresso

A classificação e detecção de um objeto em uma imagem são duas tarefas distintas. Segundo Chen et al. (2015), “a tarefa de classificação de objetos visa prever a existência de objetos dentro das imagens, enquanto a tarefa de detecção de objetos visa localizar os objetos”, como mostra a Figura 22.

Considerando Placas de Circuito Impresso (PCIs), os defeitos da etapa de fabricação são localizados e detectados antes da etapa de soldagem dos componentes.

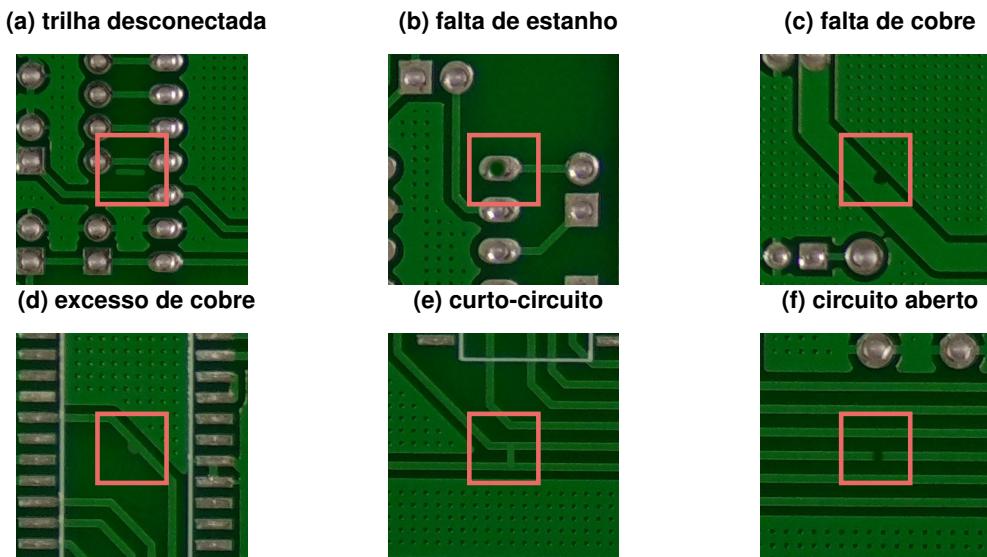
Figura 22 – Diferença entre as tarefas de classificação e detecção de objetos em uma imagem.



Fonte: Adaptado de Sivarajkumar (2019).

Esses defeitos podem ser classificados como circuito aberto, curto-circuito, falta de cobre, cobre excessivo, trilha desconectada e falta de estanho (DING et al., 2019), conforme a Figura 23.

Figura 23 – Tipos de defeito de fabricação em placas de circuito impresso.



Fonte: Adaptado de Huang et al. (2020).

2.3.1 Métodos Automatizados de Extração de Defeitos

Assim como os componentes eletrônicos utilizados em PCIs estão ficando cada vez menores, as placas de circuito impresso também estão diminuindo e se tornando mais delicadas e sofisticadas (HU; WANG, 2020). Dessa forma, utilizar técnicas automatizadas para detecção e classificação dos defeitos em PCIs é indispensável, já

que a inspeção humana além de imprecisa, está associada também à subjetividade, fadiga, lentidão e alto custo (LETA; FELICIANO; MARTINS, 2008).

Diversos métodos de detecção e classificação de defeitos em PCIs vem sendo utilizados nas últimas décadas, podendo ser classificados como métodos comparativos, não-referenciais e híbridos (MOGANTI et al., 1996 apud DING et al., 2019).

2.3.1.1 *Métodos Comparativos*

Os métodos comparativos de detecção, também conhecidos como métodos referenciais, utilizam uma imagem de referência para extrair os objetos da imagem a ser testada.

Uma das formas de fazer a extração dos defeitos é utilizando a operação lógica de ou-exclusivo (XOR) entre as duas imagens binarizadas, de forma que se o pixel da imagem a ser testada corresponde com o pixel da imagem de referência, o resultado será um, do contrário, o resultado será zero (HUANG et al., 2020).

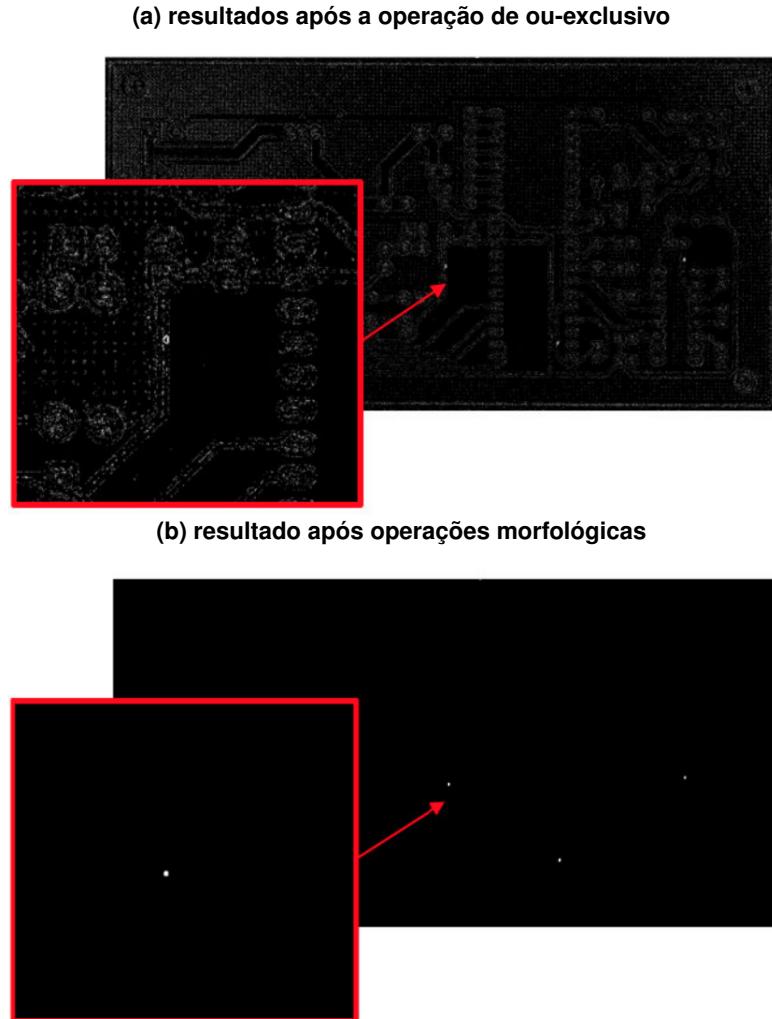
A grande dificuldade da operação XOR é determinar o alinhamento preciso entre as duas imagens (DING et al., 2019), além de que ruídos externos podem mascarar os defeitos. Huang et al. (2020) propõe o uso de algoritmos de registro de imagem para corrigir o desalinhamento entre as imagens. Esses algoritmos extraem as características de ambas imagens, calculando uma matriz de transformação que rotacionará a imagem teste deixando na mesma orientação da imagem de referência (HUANG et al., 2020). Já para remover os ruídos externos, é proposta por Huang et al. (2020) a utilização de operações morfológicas, tais como filtros medianos e operações morfológicas de fechamento e abertura. Essas operações são feitas após a operação de XOR, e uma comparação entre o resultado antes e depois das operações morfológicas está na Figura 24.

No entanto, de acordo com Ding et al. (2019), obter uma imagem de referência totalmente sem defeitos em um ambiente de produção é considerado fora da realidade, além de que problemas críticos de desalinhamento, variação de cor, variação de luz, e outras variações, tornam essa tarefa ainda mais custosa.

2.3.1.2 *Métodos Não-Referenciais*

De acordo com Ding et al. (2019), os métodos não-referenciais para detecção e classificação de defeitos são baseados na verificação de regras gerais de projeto e, nos últimos anos, o uso de redes neurais profundas tem sido empregadas nessa tarefa. Contudo, conforme Tang et al. (2019), usar redes neurais para essa aplicação implica em encontrar um equilíbrio entre eficiência e a alta precisão, já que detecções mais precisas requerem modelos de redes neurais mais profundas para obter características

Figura 24 – Extração de defeitos em placas de circuito impresso utilizando métodos comparativos.



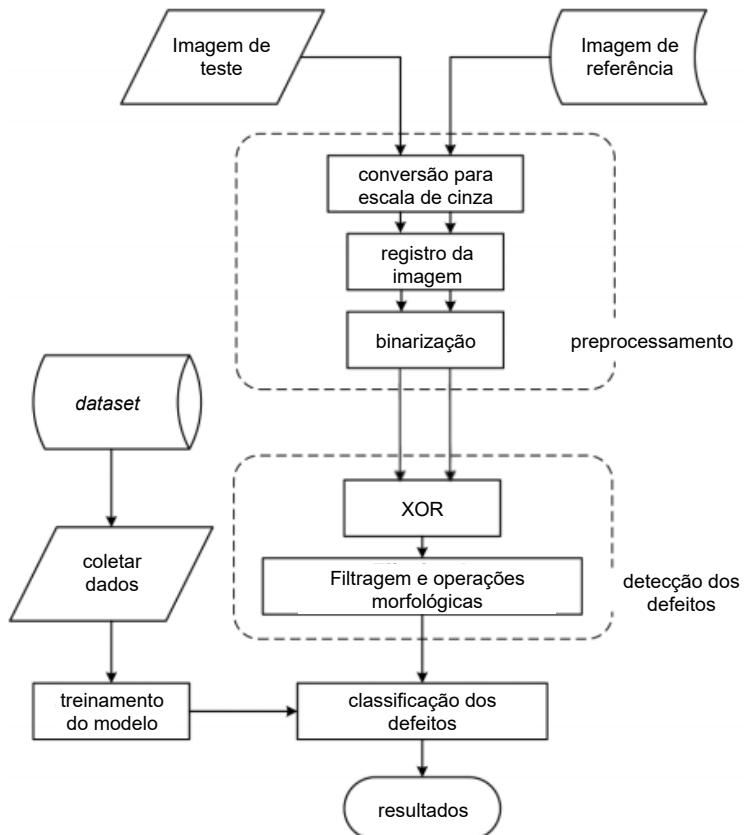
Fonte: Huang et al. (2020).

de mais alto nível e detecções mais eficientes precisam de modelos menos profundos para processamentos mais rápidos.

2.3.1.3 Métodos Híbridos

Os métodos híbridos combinam os métodos comparativo e não-referencial para a detecção e classificação de defeitos. Um exemplo para a aplicação de detecção e classificação de defeitos é a abordagem RBCNN proposta por Huang et al. (2020), onde os defeitos são inicialmente localizados utilizando métodos comparativos e, posteriormente, redes neurais convolucionais são utilizadas para a classificação, conforme o fluxograma da Figura 25.

Figura 25 – Fluxograma das etapas para classificação e detecção de defeitos utilizadas na abordagem híbrida RBCNN.



Fonte: Adaptado de Huang et al. (2020).

2.4 Frameworks e Bibliotecas para Detecção e Classificação de Objetos

Frameworks e Bibliotecas são códigos utilizados para resolver problemas comuns, evitando a criação de códigos repetidos. Uma biblioteca é responsável por oferecer um conjunto de funcionalidades prontas para o uso, enquanto frameworks são um conjunto de bibliotecas que não fornecem apenas funcionalidades, mas também a arquitetura para o desenvolvimento (TAMENAOUL, 2020).

2.4.1 Darknet

Darknet é um framework de código aberto para redes neurais escrito em C e CUDA, com fácil instalação e suporte à computação em CPU e GPU (REDMON, 2016).

2.4.2 Flask

Flask é um *framework* para *web* escrito em Python, comumente utilizado por empresas como ferramenta para soluções rápidas e simples (COPPERWAITE; LEIFER, 2015). Flask também é escalável para desenvolvimento de aplicativos de grande escala (AGGARWAL, 2014), fornecendo um conjunto de bibliotecas para lidar com tarefas mais comuns de desenvolvimento *web* (COPPERWAITE; LEIFER, 2015). Copperwaite e Leifer (2015) lista algumas de suas principais funcionalidades:

- Roteamento de URLs tornando o mapeamento mais fácil para o código;
- Gerenciamento de sessão e proteção de cookies;
- Depurador interativo baseado na web;
- Gerenciamento de configuração de aplicativo flexível e fácil de usar;
- Análise de solicitação HTTP.

3 METODOLOGIA

4 TREINAMENTO DA REDE NEURAL PARA DETECÇÃO E LOCALIZAÇÃO DE DEFEITOS EM PLACAS DE CIRCUITO IMPRESSO

4.1 Seleção da Rede Neural e do *Framework* para o Treinamento

Defeitos em placas de circuito impresso ocupam pequenas regiões da PCI, de forma que a proporção entre a área do objeto a ser detectado e a imagem inteira é muito pequena. Sendo assim, a escolha da rede neural deve considerar um bom desempenho para detecção de pequenos objetos. Segundo Redmon e Farhadi (2018), a partir da versão três, a YOLO apresentou uma melhor performance para esse tipo de detecção, sendo recomendada para essa aplicação por Valenti et al. (2020), quando o tempo de treinamento não é tão relevante. Já a versão quatro da YOLO apresenta melhor desempenho para o treinamento em GPUs quando comparada a sua versão anterior.

Sendo assim, a rede neural escolhida para o treinamento da detecção de defeitos em PCIs foi a YOLO em sua quarta versão, proposta por Bochkovskiy, Wang e Liao (2020) em conjunto com *framework* Darknet, que possui funções de cálculo das métricas de avaliação, desenho das caixas delimitadoras, entre outras.

4.2 Seleção do Conjunto de Dados

Para a detecção e classificação de objetos, o conjunto de dados escolhido deve incluir além das imagens, a localização e classificação dos objetos de interesse. O *Dataset* escolhido para a detecção e localização de defeitos em PCIs é o *HRIPCB: a challenging dataset for PCB defects detection and classification*, proposto por Huang et al. (2020).

As imagens das placas são capturadas por uma câmera do tipo industrial com dezesseis megapixel de resolução equipada com um sensor C-MOS (HUANG et al., 2020). O *dataset* baseia-se em dez diferentes tipos de PCIs sem defeitos, conforme as imagens do Apêndice C e, após a captura e ajustes, seis tipos de defeitos são adicionados manualmente em um *software* de edição de imagens, onde cada imagem contém de dois a seis defeitos da mesma categoria em diferentes lugares da placa (HUANG et al., 2020). A distribuição dos defeitos está na Tabela 1.

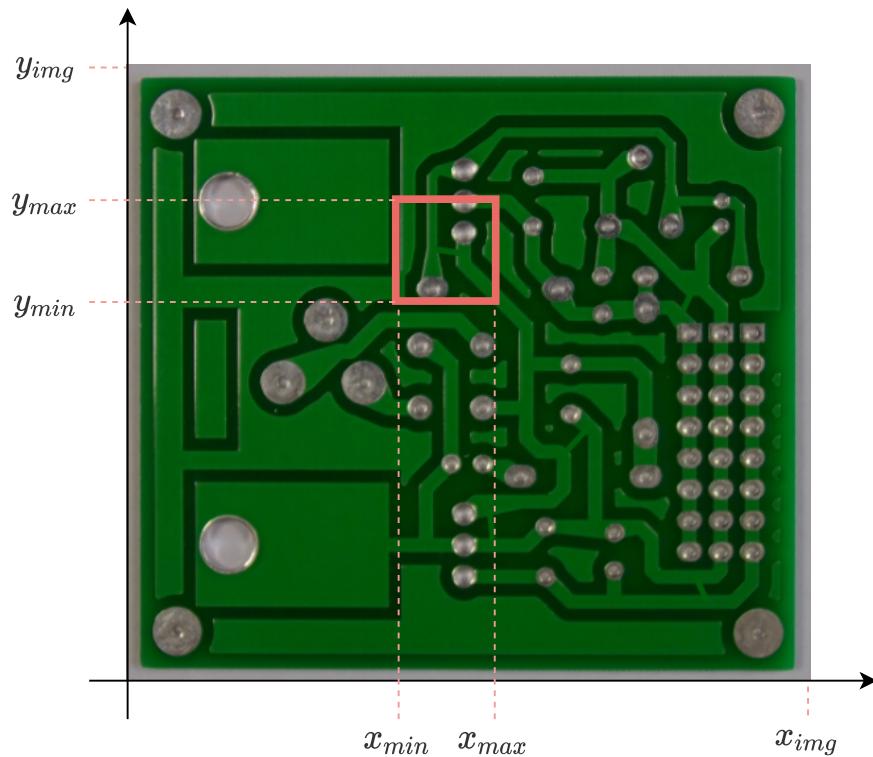
Para cada uma das imagens, existe um arquivo com extensão *.xml* que mapeia as informações das caixas delimitadoras de cada defeito. Um exemplo desse arquivo está no Apêndice D. As anotações do HRIPCB não estão normalizadas e são mapeadas em coordenadas cartesianas em pixels x_{min} , x_{max} , que variam entre zero e a largura da imagem e y_{min} , y_{max} , que variam entre zero e a altura da imagem, conforme a Figura 26.

Tabela 1 – Distribuição dos defeitos no conjunto de dados HRIPCB.

Tipo do Defeito	Número de Imagens	Quantidade Total de Defeitos
Falta de Estanho	115	497
Falta de Cobre	115	492
Circuito Aberto	116	482
Curto-Circuito	116	491
Excesso de Cobre	115	488
Trilha Desconectada	116	503
Total	693	2953

Fonte: Huang et al. (2020).

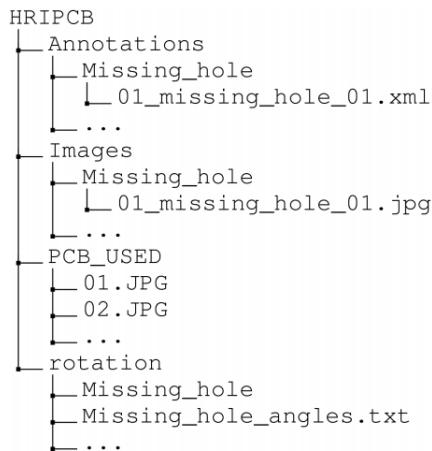
Figura 26 – Anotações das caixas delimitadoras do dataset HRIPCB.



Fonte: Adaptado de Huang et al. (2020).

A estrutura de arquivos do *dataset* está na Figura 27. Os arquivos dentro da pasta *rotation* não foram utilizados já que não possuem as delimitações dos defeitos em arquivos de anotação.

Figura 27 – Estrutura de arquivos do dataset HRIPCB.



Fonte: Huang et al. (2020).

4.3 Configuração dos Arquivos para o Treinamento

Para o treinamento, é necessário que os arquivos do *dataset* escolhido estejam de acordo com o padrão utilizado pela YOLO e as imagens estejam separadas nos subconjuntos de teste e treinamento.

Os arquivos de anotação do *dataset* HRIPC foram convertidos de *.xml* para *.txt* utilizando o *script* em Python que está no Apêndice E. Esse *script* percorre as pastas de anotação de cada defeito (Figura 27), abrindo os arquivos *.xml*, convertendo os dados conforme o padrão necessário para o treinamento e salvando esses dados em um arquivo de texto.

A conversão segue as Equações 4.1 e 4.2 para as coordenadas centrais x_{center} e y_{center} das caixas delimitadoras e as Equações 4.3 e 4.4 para a largura (*width*) e altura (*height*) das caixas delimitadoras, onde os resultados estão normalizados pela largura x_{img} e altura y_{img} da imagem conforme indicado nos passos do Apêndice B.

$$x_{center} = \frac{1}{x_{img}} \cdot (x_{min} + \frac{x_{max} - x_{min}}{2}) \quad (4.1)$$

$$y_{center} = \frac{1}{y_{img}} \cdot (y_{min} + \frac{y_{max} - y_{min}}{2}) \quad (4.2)$$

$$width = \frac{1}{x_{img}} \cdot (x_{max} - x_{min}) \quad (4.3)$$

$$height = \frac{1}{y_{img}} \cdot (y_{max} - y_{min}) \quad (4.4)$$

Além da conversão das coordenadas da caixa delimitadora, é necessário também especificar de maneira numérica qual é o defeito indicado por ela. Os defeitos estão numerados de acordo com a Tabela 2.

Tabela 2 – Representação dos defeitos do conjunto de dados HRIPCB para o treinamento da Rede Neural.

Defeito	Representação Numérica
Circuito Aberto	1
Curto-Circuito	2
Falta de Cobre	3
Excesso de Cobre	4
Trilha Desconectada	5
Falta de Estanho	6

Fonte: Elaboração própria (2021).

O resultado da conversão do arquivo de anotação em *.xml* do Apêndice D está no Código 4.1, onde cada linha representa um defeito suas coordenadas seguindo o padrão indicado por Bochkovskiy (2020) no Apêndice B.

Código 4.1 – Exemplo de arquivo de anotação para treinamento com YOLO.

```
1 5 0.8220171390903098 0.8203026481715007 0.023401450230718525 0.03467843631778058  
2 5 0.5425181278839816 0.23013871374527112 0.021753460777851022 0.03909205548549811  
3 5 0.5804218852999341 0.519546027742749 0.023401450230718525 0.03783102143757881
```

O *dataset* foi dividido aleatoriamente nos sub-conjuntos treinamento e teste utilizando o *script* do Apêndice F, com 20% dos arquivos para testes e o restante para o treinamento. Esse *script* separa tanto as imagens quanto os arquivos de anotação e, além disso, também é responsável por criar os arquivos de texto *train.txt* e *test.txt* que lista todos os arquivos de cada subconjunto de treinamento e testes, respectivamente, necessários para o treinamento com YOLO.

O arquivo de configuração *obj.data* é o apresentado no Código 4.2. Esse arquivo contém o número de classes a serem treinadas além dos caminhos para os principais arquivos requeridos para o treinamento. O arquivo *obj.names* está no Código 4.3 que lista todas as classes treinadas.

Código 4.2 – Arquivo de configuração *obj.data*.

```
1 classes = 6  
2 train = data/train.txt  
3 valid = data/test.txt  
4 names = data/obj.names  
5 backup = backup/
```

Código 4.3 – Arquivo de configuração *obj.names*.

```
1 circuito aberto
2 curto-circuito
3 falta de cobre
4 excesso de cobre
5 trilha desconectada
6 falta de estanho
```

O arquivo de configuração das camadas da rede neural foi ajustado de acordo com o Apêndice B. Sendo assim, o número máximo de *batches* necessários para o treinamento foi $n_{classes} \cdot 2000 = 12000$ e a quantidade de filtros considerada foi de $(n_{classes} + 5) \cdot 3 = 33$.

4.4 Treinamento

O treinamento foi feito no Google Colaboratory, um ambiente que permite a execução de códigos Python em *notebooks* Jupyter, adequado para Aprendizado de Máquina, análise de dados e educação, oferecendo recursos de computação como GPUs de forma gratuita (COLABORATORY..., 2021).

O código completo utilizado para o treinamento está disponível no Apêndice G, contendo todos os comandos utilizados.

O repositório do Darknet, disponibilizado *online* por Bochkovskiy (2020) no endereço <<https://github.com/AlexeyAB/darknet>>, foi adicionado ao ambiente, por meio do comando *git clone*. Além disso, foram manualmente adicionadas as pastas contendo os arquivos de configuração e do *dataset* para o treinamento.

Alterações no arquivo *makefile* foram feitas para que o treinamento ocorresse utilizando GPU (linhas quinze a dezessete do código do Apêndice G) e em seguida compilou-se o Darknet para o treinamento.

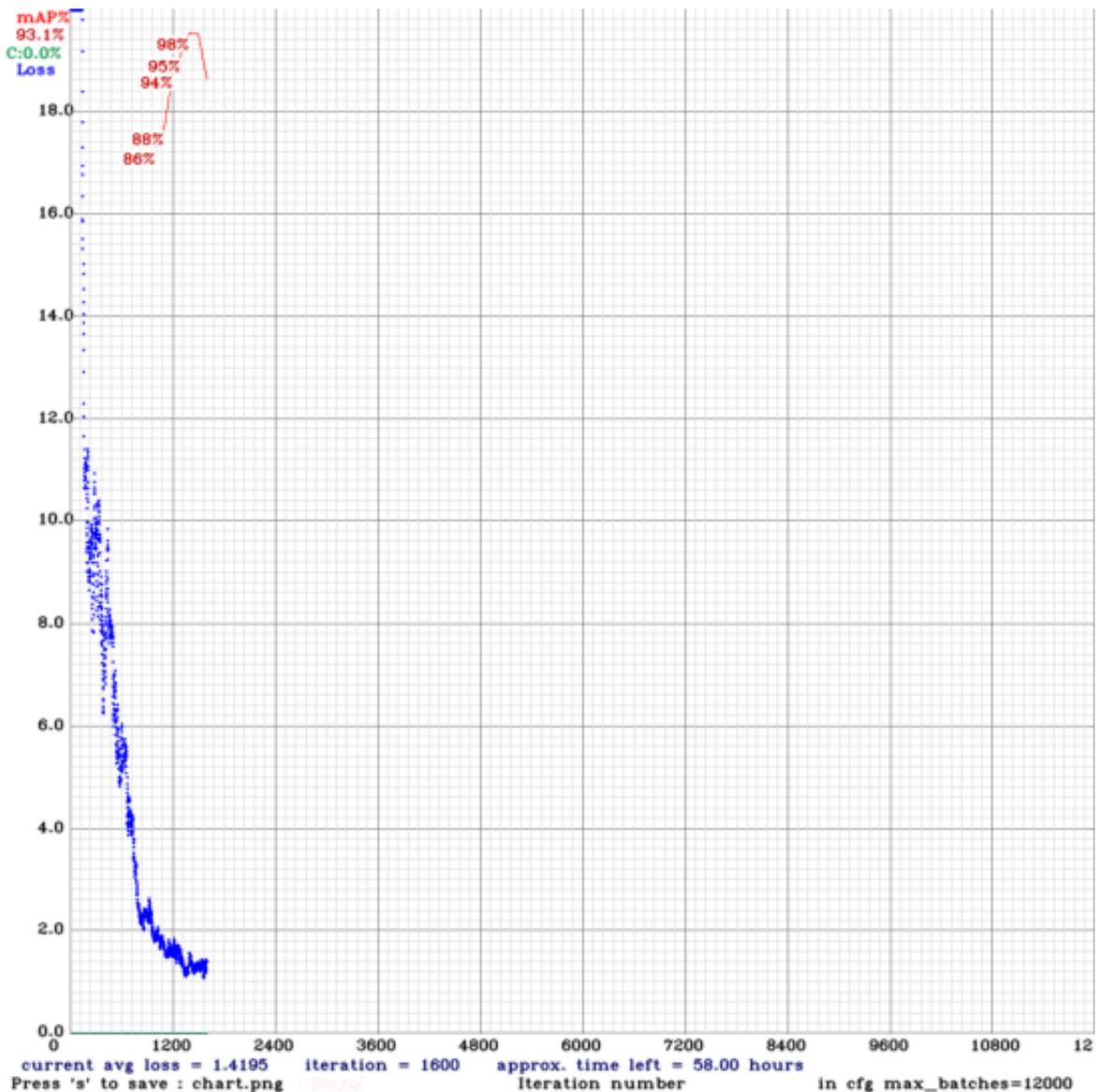
Com intuído de acelerar o treinamento, utilizou-se a transferência de aprendizado de forma que a rede foi inicializada com os pesos da *yolov4.conv.137*, pré-treinada com um *dataset* contendo mais de um milhão de imagens com objetos classificados em mais mil categorias (REDMON et al., 2015). A transferência de aprendizado consiste em acelerar um novo treinamento a partir de resultados obtidos em treinamentos anteriores (CAI; BILE SCHI; NIELSEN, 2020).

4.5 Resultados

O gráfico da Figura 28 mostra a evolução do desempenho durante o início do treinamento. Nota-se que para um pouco mais de mil e duzentos *batches* a métrica de mAP, em vermelho, já estava com pico em 98%. Da mesma forma, a função de perda, representada pelo traço azul, foi diminuindo consideravelmente conforme os

pesos da rede neural se atualizavam. Esse gráfico não está completo pois é gerado pelo Darknet e salvo em um arquivo temporário e o treinamento foi interrompido em mil e quinhentos *batches* e reiniciado a partir desse ponto.

Figura 28 – Desempenho do início do treinamento da rede neural.



Fonte: Elaboração própria (2021).

As métricas de treinamento para os pesos obtidos são calculadas por meio do comando do Darknet presente no Código 4.4, onde o valor da *flag thresh* indica o limiar de confiabilidade da predição utilizado para as métricas baseadas em matriz de confusão e a *flag points* representa o tipo de aproximação para o cálculo da métrica de AP e mAP, onde o valor 0 quer dizer que não há nenhuma aproximação para o cálculo da área da curva precisão em função da sensibilidade.

Código 4.4 – Comando Darknet para o cálculo das métricas de avaliação da rede neural.

```
1 !./darknet detector map data/obj.data cfg/yolov4_custom.cfg yolov4_custom_last.  
weights -thresh 0.75 -points 0
```

O treinamento foi finalizado após dois mil e quinhentos *batches*, já que os resultados apresentavam bom desempenho, conforme a Tabela 3, que apresenta a métrica AP calculadas para cada uma das classes treinadas e a Tabela 4, que apresenta os resultados médios das classes treinadas. O tempo de treinamento não foi cronometrado, durando aproximadamente entre quinze e vinte horas.

Tabela 3 – Resultados de Precisão Média para as classes treinadas de defeitos em placas de circuito impresso.

Defeito	Precisão Média (AP)
Circuito Aberto	97,07%
Curto-Círcito	98,82%
Falta de Cobre	99,20%
Excesso de Cobre	96,68%
Trilha Desconectada	98,87%
Falta de Estanho	100,00%

Fonte: Elaboração própria (2021).

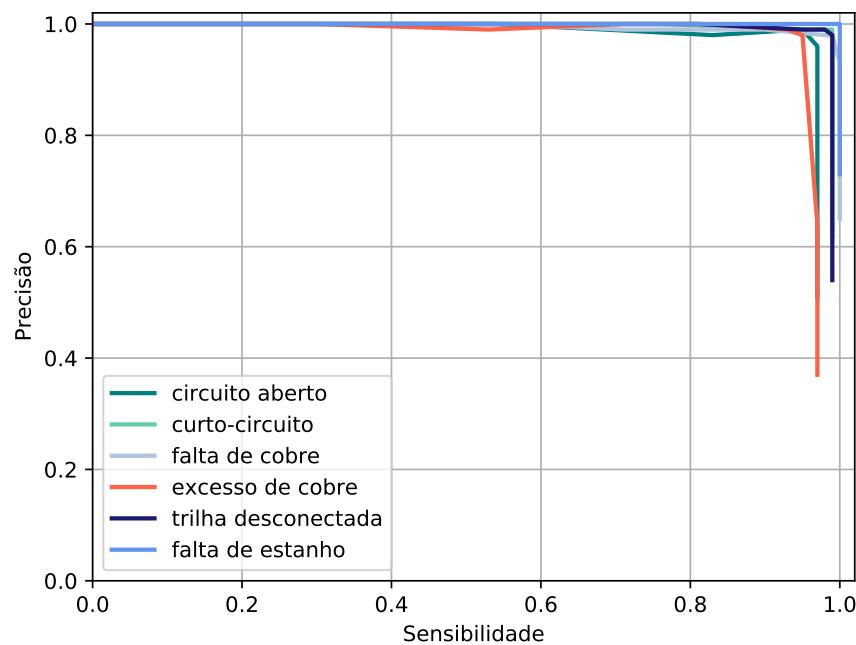
Tabela 4 – Métricas de Avaliação para o conjunto das classes treinadas de defeitos em placas de circuito impresso, considerando um *threshold* de confiabilidade da predição de 75%.

Métrica	Resultado
Precisão	99,00%
Sensibilidade	94,00%
<i>F1-score</i>	97,00%
mAP	98,44%

Fonte: Elaboração própria (2021).

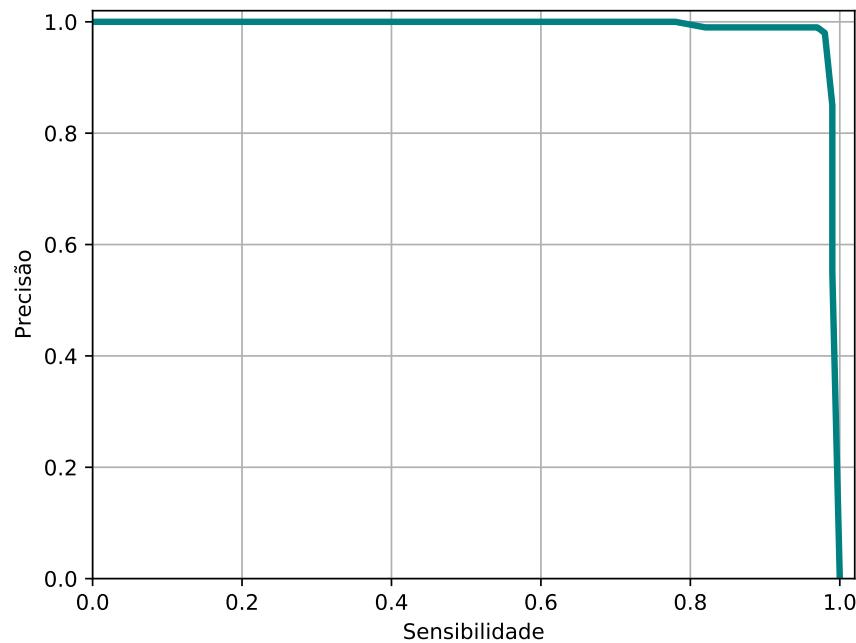
O gráfico de precisão em função da sensibilidade para cada uma das classes está na Figura 29. A Figura 30 apresenta o mesmo gráfico considerando uma média de todas as classes para a curva de precisão em função da sensibilidade. Esses gráficos foram construídos variando o valor de confiabilidade da predição (*flag thresh*) dentro do intervalo [0.001, 0.99] para as métricas de precisão e sensibilidade utilizando o comando do Código 4.4, além de alterar a função *qsort* do arquivo “detector.c” que está localizado dentro da pasta *Darknet/src/* e recompilar o Darknet para imprimir os valores de precisão e sensibilidade, já que sem isso apenas a métrica de AP é apresentada para cada classe.

Figura 29 – Curvas de precisão em função da sensibilidade para cada classe treinada.



Fonte: Elaboração própria (2021).

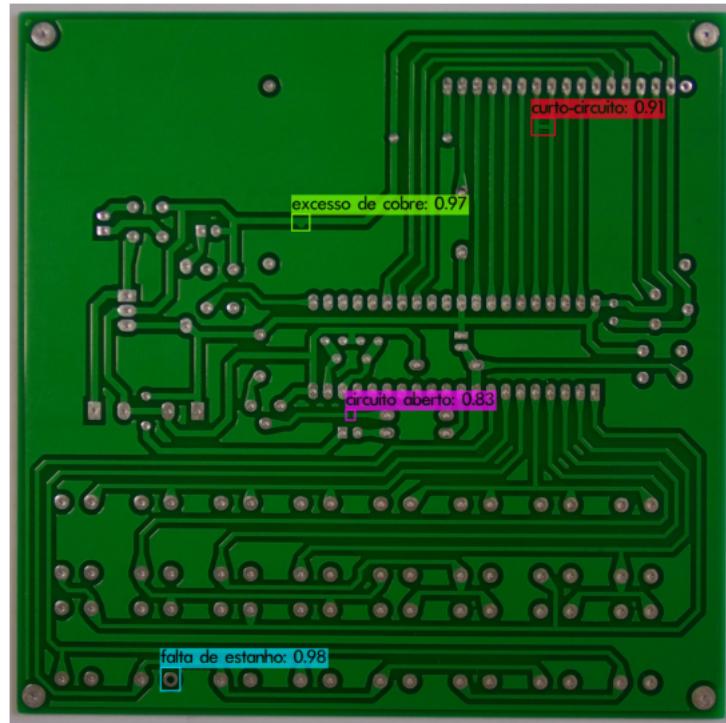
Figura 30 – Curva de precisão em função da sensibilidade para o conjunto de classes treinado.



Fonte: Elaboração própria (2021).

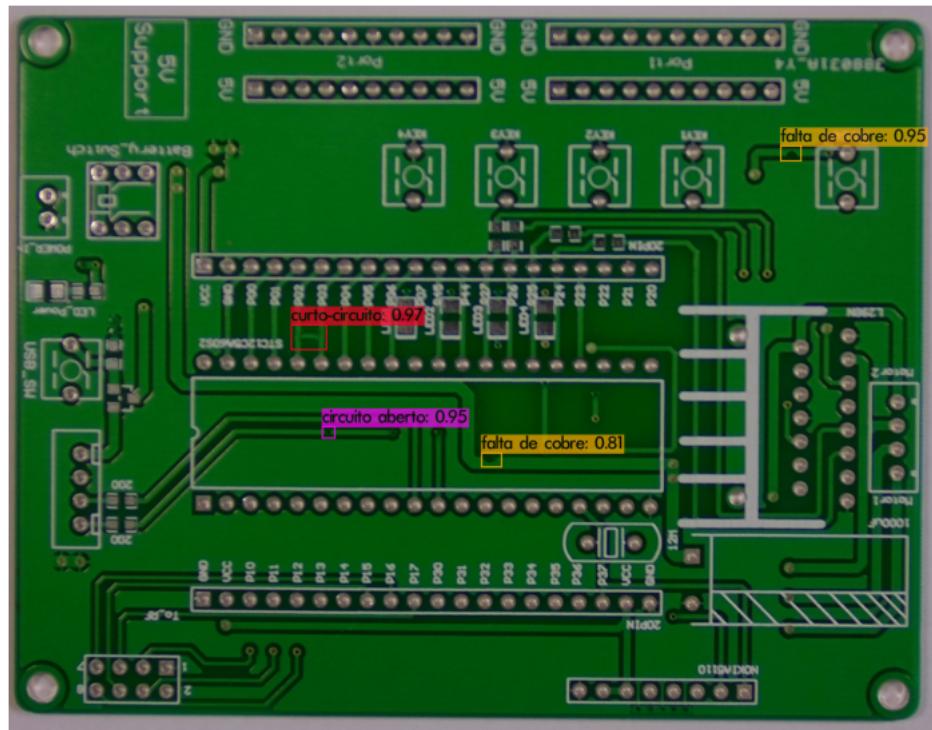
As Figuras 31, 32 e 33 apresentam resultados de detecção para imagens com defeitos variados adicionados manualmente em software de edição de imagem a partir de PCIs do *dataset* sem presença de defeitos. Essas imagens baseiam-se nas apresentadas no Apêndice C. O número indicado ao lado do nome do defeito significa a confiabilidade da predição, onde 1 indicaria uma predição 100% confiável.

Figura 31 – Resultado de predição para imagem com defeito, teste 1.



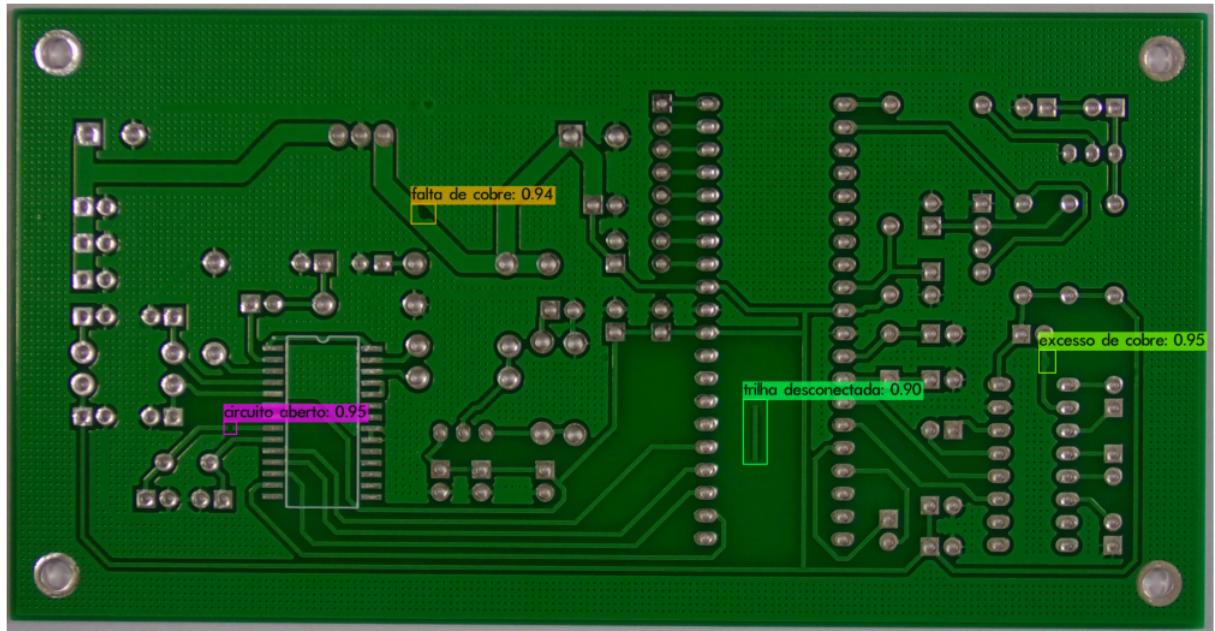
Fonte: Adaptado de Huang et al. (2020).

Figura 32 – Resultado de predição para imagem com defeito, teste 2.



Fonte: Adaptado de Huang et al. (2020).

Figura 33 – Resultado de predição para imagem com defeito, teste 3.

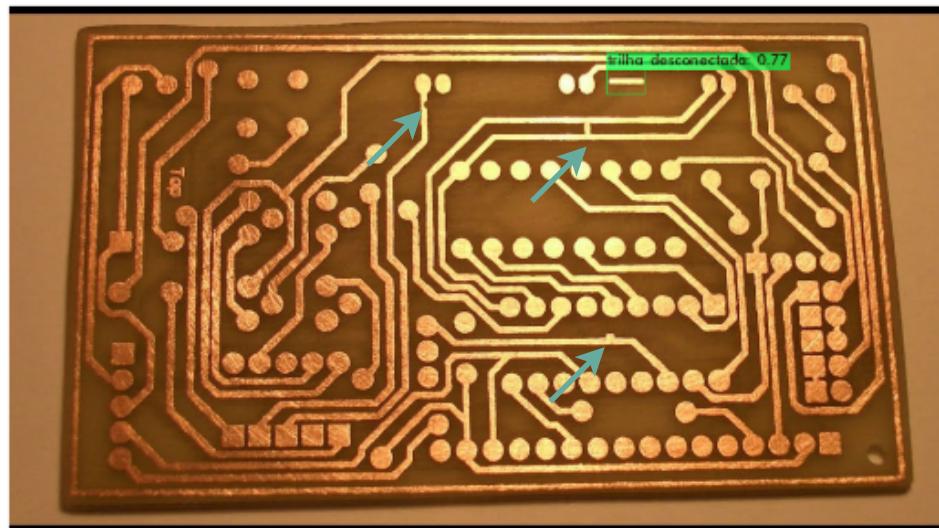


Fonte: Adaptado de Huang et al. (2020).

As Figuras 34 e 35 apresentam resultados de detecção para imagens de fora

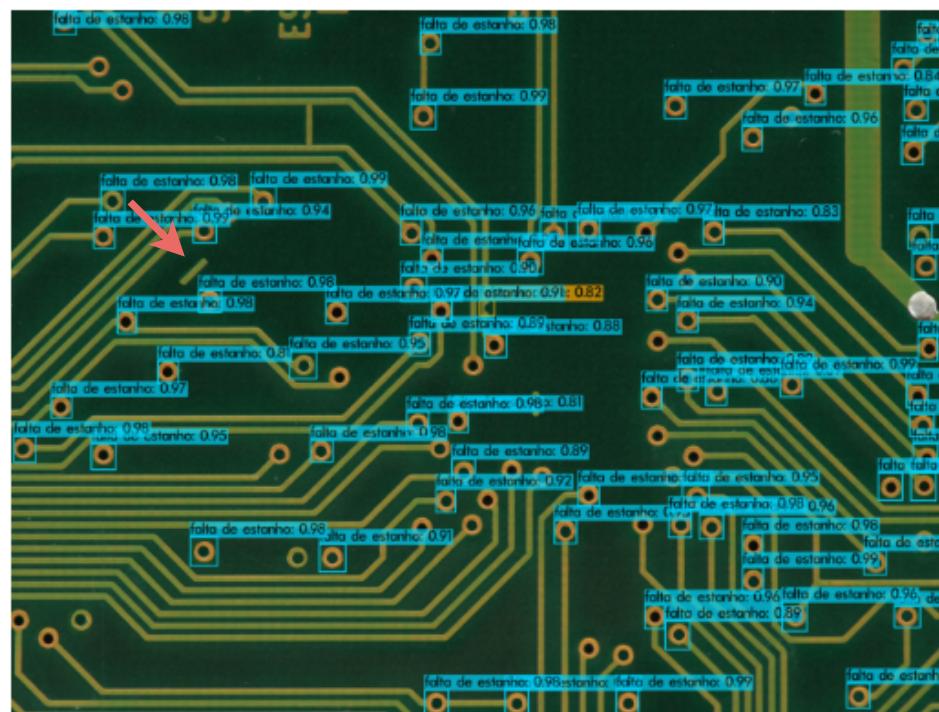
do *dataset*. Em ambas figuras, as setas indicam os defeitos que não foram detectados pela rede neural. A Figura 35 aponta os furos de vias como defeitos, já que o *dataset* utilizado considera vias ou furos sem a cobertura de estanho como um defeito.

Figura 34 – Resultado de predição para imagem com defeito, teste 4.



Fonte: Adaptado de Khambhala (2015).

Figura 35 – Resultado de predição para imagem com defeito, teste 5.



Fonte: Adaptado de Dratza (2015).

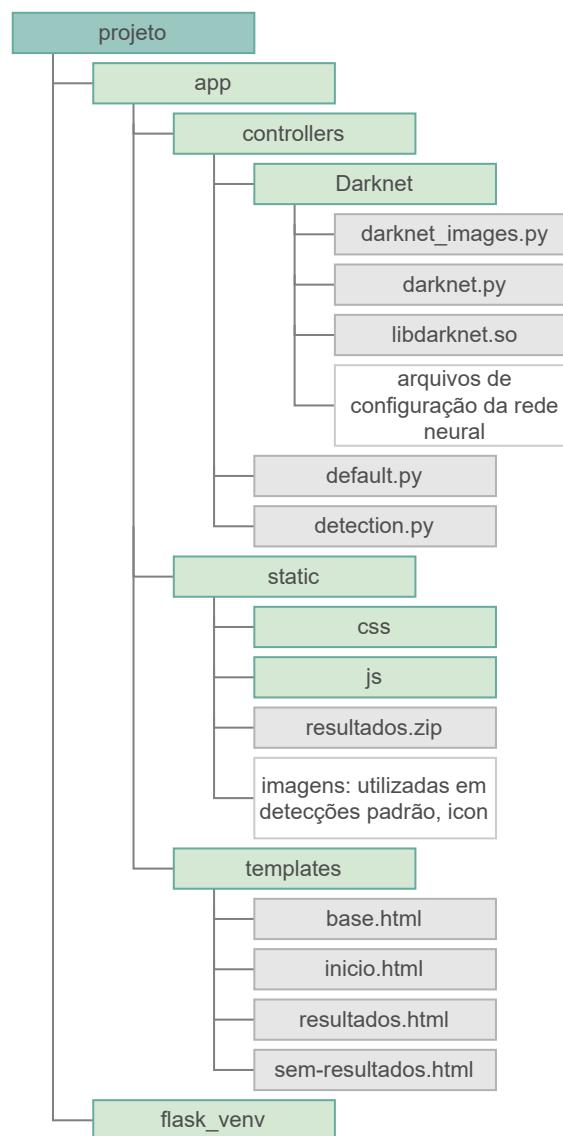
5 INTERFACE DE APLICAÇÃO

Para demonstração dos resultados obtidos com o treinamento da rede neural, uma aplicação *web* foi implementada utilizando Flask, um *framework* de fácil uso que possibilita a integração com outros módulos Python.

O código desenvolvido está disponível no repositório online <<https://github.com/anabdck/pcb-defect-detection-api>> e as etapas de instalação e configuração estão no Apêndice H.

A estrutura da aplicação foi construída de acordo com o apresentado na Figura 36.

Figura 36 – Estrutura da interface de aplicação proposta.



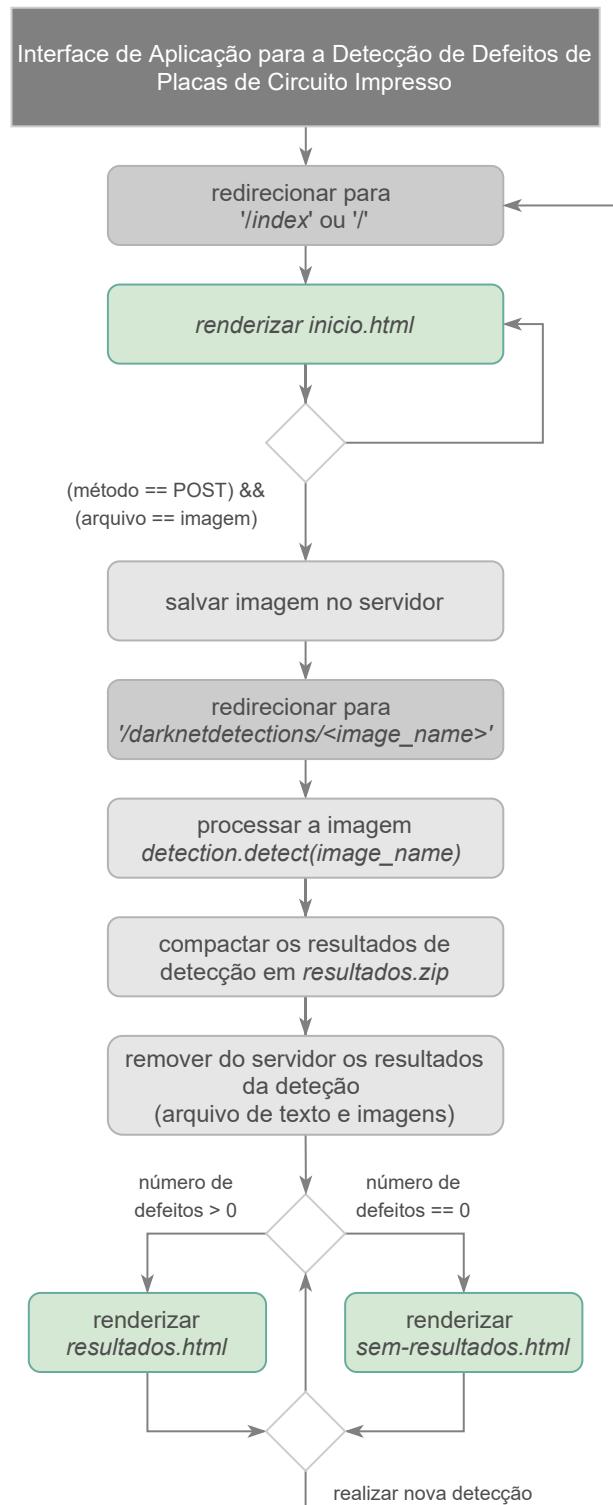
Fonte: Elaboração própria (2021).

As páginas *web* apresentadas ao usuário utilizam HTML5, CSS3, Java Script e Bootstrap, organizadas dentro das pastas *static* e *templates* e o roteamento das páginas ocorre em *default.py* dentro da pasta *controllers*.

O módulo de detecção *detection.py* é responsável por invocar o Darknet e processar a imagem para a detecção dos defeitos utilizando os arquivos de configuração da rede neural e os pesos treinados. Para isso, é necessária a biblioteca *libdarknet.so*, compilada para o servidor utilizado conforme o indicado nos passos do Apêndice H.

O Fluxograma da Figura 37 apresenta de maneira abreviada as etapas de detecção dos defeitos, indicando o momento em que as diferentes páginas são renderizadas e quais são os endereços que foram roteados para a aplicação utilizados no processo.

Figura 37 – Fluxograma da detecção de defeitos de placas de circuito impresso dentro da interface de aplicação proposta.



Fonte: Elaboração própria (2021).

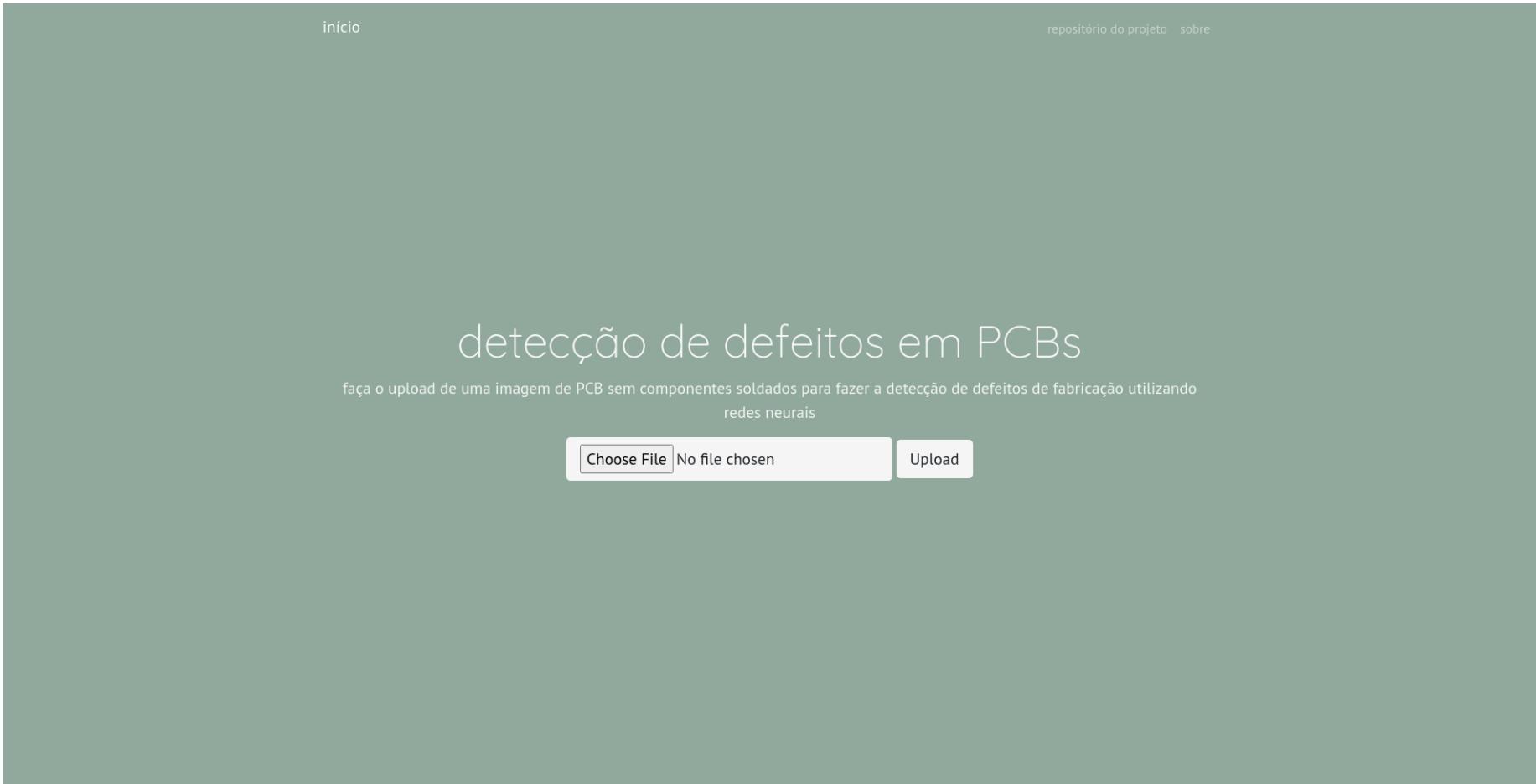
O endereço inicial apresenta a tela da Figura 38, onde o usuário pode fazer o *upload* de uma imagem para a detecção ou, ao rolar a página para baixo, pode

escolher uma das quatro imagens que já estão no servidor para a detecção, conforme a tela da Figura 39. É possível fazer apenas o *upload* de arquivos de imagens com extensões *png*, *jpg* ou *jpeg*. Caso seja feito o *upload* de outros tipos de arquivos, é feito o redirecionamento para a página inicial novamente, já que as demais extensões não são aceitas pela rede neural.

Se arquivo escolhido for de uma das extensões permitidas ou o usuário escolher uma das imagens que está no servidor, a detecção dos defeitos é feita. Se a rede neural encontrar defeitos na imagem, a tela da Figura 40 é apresentada ao usuário, onde a indicação dos defeitos encontrados é feita por meio das caixas delimitadoras resultantes impressas na imagem processada. Uma mensagem indicando o tempo de processamento e a quantidade de defeitos encontrados é exibido ao usuário. O botão de “*Download dos Resultados*” salva no computador do usuário um arquivo compactado contendo a imagem com as caixas delimitadoras e um arquivo de texto de anotação no padrão YOLO. Caso nenhum defeito é encontrado, a imagem original é mostrada ao usuário, além do aviso indicando o tempo de processamento, conforme a Figura 41. O botão de “*Download dos Resultados*” fica travado nesse caso.

A aplicação construída não considerou o uso de GPUs, mas caso esteja disponível no servidor utilizado, o Darknet pode ser compilado para isso, melhorando o tempo de processamento da detecção que para o servidor testado varia em torno de vinte segundos.

Figura 38 – Tela Inicial 1 da Interface de Aplicação.



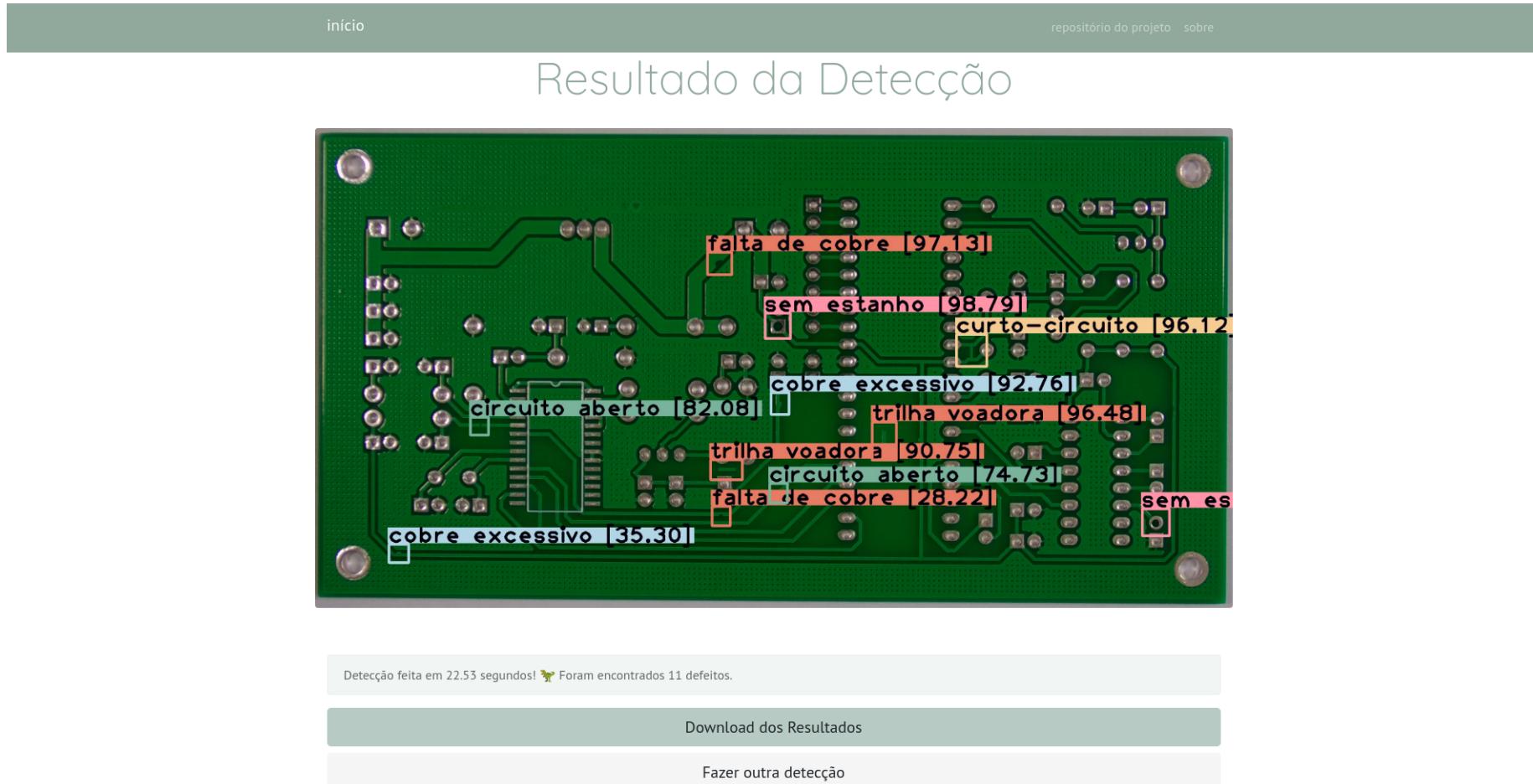
Fonte: Elaboração própria (2021).

Figura 39 – Tela Inicial 2 da Interface de Aplicação.



Fonte: Elaboração própria (2021).

Figura 40 – Tela da Interface de Aplicação para imagem com detecções.



Fonte: Elaboração própria (2021).

Figura 41 – Tela da Interface de Aplicação para imagem sem detecções.



Fonte: Elaboração própria (2021).

6 DISCUSSÃO

A rede neural treinada possui resultados satisfatórios para o *dataset* utilizado, conforme apresentado nas Tabelas 3 e 4. A Tabela 5 apresenta a métrica de mAP para diferentes estruturas de redes neurais treinadas com o mesmo conjunto de dados com a finalidade de comparação com os resultados obtidos utilizando a YOLO versão quatro.

Tabela 5 – Comparação da utilização do conjunto de dados HRIPCB para o reconhecimento de defeitos em placas de circuito impresso para diferentes modelos de Redes Neurais.

Modelo	Referência	mAP
Faster R-CNN / VGG-16	(REN et al., 2015 apud DING et al., 2019)	58,57%
Faster R-CNN / ResNet-101	(REN et al., 2015 apud DING et al., 2019)	94,27%
FPN	(LIN et al., 2017 apud DING et al., 2019)	92,23%
Faster R-CNN fine-tuned	(DING et al., 2019)	96,44%
TDD-Net	(DING et al., 2019)	98,90%
YOLO versão 4	proposto	98,44%

Fonte: Elaboração própria (2021).

Dentre as redes neurais apresentadas na Tabela 5, apenas a TDD-Net proposta por Ding et al. (2019) apresentou um melhor desempenho do que a YOLO em sua quarta versão, cujos resultados foram apontados no Capítulo 4. Esse resultado pode melhorar com o treinamento de mais *batches* da rede neural, ao invés de apenas dois mil e quinhentos.

Para imagens que não são do *dataset* porém, os resultados apresentaram um desempenho não satisfatório, conforme mostram as Figuras 34 e 35. Isso é consequência do conjunto de dados escolhido, que possui apenas imagens de placas de circuito impresso dentro de um padrão, conforme as imagens do Apêndice C, e na indústria existem diversos tipos de placas de circuito impresso, com cores e tamanhos distintos.

7 CONSIDERAÇÕES FINAIS

REFERÊNCIAS

- ABRAHAM, T. et al. Contributors. In: COHEN, S. (Ed.). *Artificial Intelligence and Deep Learning in Pathology*. Elsevier, 2021. p. xiii–xiv. ISBN 978-0-323-67538-3. Disponível em: <<https://www.sciencedirect.com/science/article/pii/B9780323675383010022>>. Citado na página 13.
- AGGARWAL, S. *Flask framework cookbook*. [S.I.]: Packt Publishing Ltd, 2014. Citado na página 37.
- ALBAWI, S.; MOHAMMED, T. A.; AL-ZAWI, S. Understanding of a convolutional neural network. In: *2017 International Conference on Engineering and Technology (ICET)*. IEEE, 2017. Disponível em: <<https://doi.org/10.1109%2Ficengtechnol.2017.8308186>>. Citado na página 29.
- BOCHKOVSKIY, A. *Darknet*. 2020. Disponível em: <<https://github.com/AlexeyAB/darknet>>. Citado 3 vezes nas páginas 32, 42 e 43.
- BOCHKOVSKIY, A.; WANG, C.-Y.; LIAO, H.-Y. M. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. Citado 2 vezes nas páginas 31 e 39.
- BROWNLEE, J. *Difference Between a Batch and an Epoch in a Neural Network*. 2018. Disponível em: <<https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/>>. Citado na página 68.
- CAI, S.; BILESCHE, S.; NIELSEN, E. *Deep Learning with JavaScript: Neural networks in TensorFlow.js*. Manning Publications, 2020. ISBN 9781617296178. Disponível em: <<https://books.google.com.br/books?id=N2dswgEACAAJ>>. Citado na página 43.
- CHAWLA, N. V. Data mining for imbalanced datasets: An overview. In: _____. *Data Mining and Knowledge Discovery Handbook*. Boston, MA: Springer US, 2005. p. 853–867. ISBN 978-0-387-25465-4. Disponível em: <https://doi.org/10.1007/0-387-25465-X_40>. Citado na página 23.
- CHEN, Q. et al. Contextualizing object detection and classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 37, n. 1, p. 13–27, 2015. Citado na página 32.
- CHIN, R. T.; HARLOW, C. A. Automated visual inspection: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4, n. 6, p. 557–573, 1982. Citado na página 11.
- COLABORATORY - Frequently Asked Questions. 2021. Acesso em: março/21. Disponível em: <<https://research.google.com/colaboratory/faq.html>>. Citado na página 43.
- COPPERWAITE, M.; LEIFER, C. *Learning Flask Framework*. [S.I.]: Packt Publishing Ltd, 2015. Citado na página 37.
- DICK, S. Artificial intelligence. *Harvard Data Science Review*, v. 1, n. 1, 7 2019. <Https://hdsr.mitpress.mit.edu/pub/0aytgrau>. Disponível em: <<Https://hdsr.mitpress.mit.edu/pub/0aytgrau>>. Citado na página 13.

- DING, R. et al. Tdd-net: A tiny defect detection network for printed circuit boards. *CAAI Transactions on Intelligence Technology*, v. 4, 04 2019. Citado 3 vezes nas páginas 33, 34 e 58.
- DRATZA. *Circuitos impresos*. 2015. Acesso em: abril/21. Disponível em: <<https://www.explicofacil.com/2016/11/circuitos-impresos-que-son.html>>. Citado na página 49.
- EDGAR, T. W.; MANZ, D. O. Chapter 6 - machine learning. In: EDGAR, T. W.; MANZ, D. O. (Ed.). *Research Methods for Cyber Security*. Syngress, 2017. p. 153–173. ISBN 978-0-12-805349-2. Disponível em: <<https://www.sciencedirect.com/science/article/pii/B9780128053492000066>>. Citado na página 14.
- ETTEN, A. V. Satellite imagery multiscale rapid detection with windowed networks. *CoRR*, abs/1809.09978, 2018. Disponível em: <<http://arxiv.org/abs/1809.09978>>. Citado na página 24.
- GHOLAMALINEZHAD, H.; KHOSRAVI, H. *Pooling Methods in Deep Neural Networks, a Review*. 2020. Citado 3 vezes nas páginas 27, 29 e 30.
- GOODFELLOW, I. J.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. <<http://www.deeplearningbook.org>>. Citado 12 vezes nas páginas 13, 14, 15, 16, 17, 19, 22, 23, 26, 27, 28 e 29.
- HU, B.; WANG, J. Detection of pcb surface defects with improved faster-rcnn and feature pyramid network. In: . [S.l.: s.n.], 2020. v. 8, p. 108335–108345. Citado 2 vezes nas páginas 11 e 33.
- HUANG, W. et al. Hripcb: a challenging dataset for pcb defects detection and classification. *The Journal of Engineering*, v. 2020, 01 2020. Citado 14 vezes nas páginas 33, 34, 35, 36, 39, 40, 41, 47, 48, 70, 71, 72, 73 e 74.
- KHAMHALA, N. *Make PCB using Laser Printer*. YouTube, 2015. Acesso em: abril/21. Disponível em: <<https://www.youtube.com/watch?v=GjXhPRYOMzg>>. Citado na página 49.
- KULKARNI, A.; CHONG, D.; BATARSEH, F. A. 5 - foundations of data imbalance and solutions for a data democracy. In: BATARSEH, F. A.; YANG, R. (Ed.). *Data Democracy*. Academic Press, 2020. p. 83–106. ISBN 978-0-12-818366-3. Disponível em: <<https://www.sciencedirect.com/science/article/pii/B9780128183663000058>>. Citado 3 vezes nas páginas 22, 23 e 25.
- LAWLESS, W. F.; MITTU, R.; SOFGE, D. A. Chapter 4 - context: Separating the forest and the trees—wavelet contextual conditioning for ai. In: *Human-Machine Shared Contexts*. Academic Press, 2020. p. 67–91. ISBN 978-0-12-820543-3. Disponível em: <<https://www.sciencedirect.com/science/article/pii/B9780128205433000043>>. Citado na página 13.
- LESORT, T. et al. State representation learning for control: An overview. *Neural Networks*, v. 108, p. 379–392, 2018. ISSN 0893-6080. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S0893608018302053>>. Citado na página 14.

- LETA, F. R.; FELICIANO, F. F.; MARTINS, F. P. R. Computer vision system for printed circuit board inspection. In: *ABCM Symposium Series in Mechatronics*. [S.I.: s.n.], 2008. v. 3, p. 623 – 632. Citado 2 vezes nas páginas 11 e 34.
- LIN, T. et al. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014. Disponível em: <<http://arxiv.org/abs/1405.0312>>. Citado na página 31.
- LIN, T.-Y. et al. *Feature Pyramid Networks for Object Detection*. 2017. Citado na página 58.
- MAO, S. et al. Opportunities and challenges of artificial intelligence for green manufacturing in the process industry. *Engineering*, v. 5, n. 6, p. 995–1002, 2019. ISSN 2095-8099. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S2095809919300074>>. Citado na página 14.
- MISHRA, A. misc, *Metrics to Evaluate your Machine Learning Algorithm*. towardsdatascience.com, 2018. Disponível em: <<https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234>>. Citado na página 25.
- MISRA, D. Mish: A self regularized non-monotonic neural activation function. *arXiv preprint arXiv:1908.08681*, 2019. Citado 3 vezes nas páginas 18, 19 e 20.
- MITCHELL, R.; MICHALSKI, J.; CARBONELL, T. *An artificial intelligence approach*. [S.I.]: Springer, 2013. Citado na página 13.
- MOGANTI, M. et al. Automatic pcb inspection algorithms: A survey. *Computer Vision and Image Understanding*, v. 63, n. 2, p. 287–313, 1996. ISSN 1077-3142. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S107731429690020X>>. Citado na página 34.
- NG, A. Notas de Aula, *Introduction To Deep Learning*. 2020. COURSERA: Neural Networks and Deep Learning. Disponível em: <<https://www.coursera.org/learn/neural-networks-deep-learning>>. Citado 6 vezes nas páginas 17, 18, 19, 20, 21 e 22.
- NICHOLSON, C. misc, *A Beginner's Guide to Neural Networks and Deep Learning*. Pathmind A.I. Wiki, 2020. Disponível em: <<https://wiki.pathmind.com/neural-network#define>>. Citado 3 vezes nas páginas 17, 18 e 26.
- NIELSEN, M. A. misc, *Neural Networks and Deep Learning*. Determination Press, 2018. Disponível em: <<http://neuralnetworksanddeeplearning.com/>>. Citado 2 vezes nas páginas 16 e 29.
- RASHID, T. *Make Your Own Neural Network*. 1. ed. [S.I.]: CreateSpace Independent Publishing Platform, 2016. ISBN 1530826608,9781530826605. Citado na página 13.
- REDMON, J. *Darknet: Open Source Neural Networks in C*. 2016. <<http://pjreddie.com/darknet/>>. Citado na página 36.
- REDMON, J. et al. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015. Disponível em: <<http://arxiv.org/abs/1506.02640>>. Citado 3 vezes nas páginas 31, 32 e 43.

- REDMON, J.; FARHADI, A. Yolov3: An incremental improvement. *arXiv*, 2018. Citado 2 vezes nas páginas 30 e 39.
- REN, S. et al. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015. Disponível em: <<http://arxiv.org/abs/1506.01497>>. Citado na página 58.
- REZATOFIGHI, H. et al. Generalized intersection over union. June 2019. Citado na página 25.
- ROBINS, M. The difference between artificial intelligence, machine learning and deep learning. *Intel Artificial Intelligence*, 5 2020. Disponível em: <<https://www.intel.com.br/content/www/br/pt/artificial-intelligence/posts/difference-between-ai-machine-learning-deep-learning.html>>. Citado 2 vezes nas páginas 14 e 16.
- SAUER, A. misc, *Quick Guide to Gradient Descent and Its Variants*. Morioh, 2020. Disponível em: <<https://morioh.com/p/15c995420be6>>. Citado na página 21.
- SHARMA, S. Activation functions in neural networks. *towards data science*, v. 6, 2017. Citado na página 18.
- SIVARAJKUMAR, S. misc, *Image Classification And Object Detection*. medium.com, 2019. Disponível em: <<https://medium.com/@sonish.sivarajkumar/image-classification-and-object-detection-c9803f854923>>. Citado 4 vezes nas páginas 25, 26, 31 e 33.
- STEEN, D. misc, *Precision-Recall Curves*. medium.com, 2020. Disponível em: <<https://medium.com/@douglaspsteen/precision-recall-curves-d32e5b290248>>. Citado na página 23.
- TAMENAOU, H. misc, *Framework vs library vs package vs module: The debate*. medium.com, 2020. Disponível em: <<https://medium.com/ieee-ensias-student-branch/framework-vs-library-vs-package-vs-module-the-debate-e1013a3e114d>>. Citado na página 36.
- TAN, R. J. misc, *Breaking Down Mean Average Precision (mAP)*. towardsdatascience.com, 2019. Disponível em: <<https://towardsdatascience.com/breaking-down-mean-average-precision-map-ae462f623a52>>. Citado 2 vezes nas páginas 24 e 25.
- TANG, S. et al. Online PCB defect detector on A new PCB defect dataset. *CoRR*, abs/1902.06197, 2019. Disponível em: <<http://arxiv.org/abs/1902.06197>>. Citado na página 34.
- VALENTI, C. F. et al. An evaluation of deep learning methods for small object detection. *Journal of Electrical and Computer Engineering*, 2020. Disponível em: <<https://www.hindawi.com/journals/jece/2020/3189691/>>. Citado na página 39.
- WALCZAK, S.; CERPA, N. Artificial neural networks. In: MEYERS, R. A. (Ed.). *Encyclopedia of Physical Science and Technology (Third Edition)*. Third edition. New York: Academic Press, 2003. p. 631–645. ISBN 978-0-12-227410-7. Disponível em:

- <<https://www.sciencedirect.com/science/article/pii/B0122274105008371>>. Citado na página 16.
- WANG, C. et al. Citrus recognition based on yolov4 neural network. *Journal of Physics: Conference Series*, IOP Publishing, v. 1820, n. 1, p. 012163, mar 2021. Disponível em: <<https://doi.org/10.1088/1742-6596/1820/1/012163>>. Citado na página 30.
- WOOLF, B. P. Chapter 7 - machine learning. In: WOOLF, B. P. (Ed.). *Building Intelligent Interactive Tutors*. San Francisco: Morgan Kaufmann, 2009. p. 221–297. ISBN 978-0-12-373594-2. Disponível em: <<https://www.sciencedirect.com/science/article/pii/B9780123735942000071>>. Citado na página 14.
- WU, W.-Y.; WANG, M.-J. J.; LIU, C.-M. Automated inspection of printed circuit boards through machine vision. *Computers in Industry*, v. 28, n. 2, p. 103–111, 1996. ISSN 0166-3615. Disponível em: <<https://www.sciencedirect.com/science/article/pii/0166361595000631>>. Citado na página 11.
- YAN, W. Q. *Computational Methods for Deep Learning: Theoretic, Practice and Applications*. [S.I.]: Springer, 2021. (Texts in Computer Science). ISBN 3030610802, 9783030610807. Citado 4 vezes nas páginas 20, 26, 29 e 30.
- ZHU, W. et al. Deep learning based soft sensor and its application on a pyrolysis reactor for compositions predictions of gas phase components. In: *13th International Symposium on Process Systems Engineering (PSE 2018)*. Elsevier, 2018, (Computer Aided Chemical Engineering, v. 44). p. 2245–2250. Disponível em: <<https://www.sciencedirect.com/science/article/pii/B9780444642417503694>>. Citado na página 26.
- ZUMBAHLEN, H. Chapter 12 - printed circuit-board design issues. In: ZUMBAHLEN, H. (Ed.). *Linear Circuit Design Handbook*. Burlington: Newnes, 2008. p. 821–895. ISBN 978-0-7506-8703-4. Disponível em: <<https://www.sciencedirect.com/science/article/pii/B9780750687034000122>>. Citado na página 11.

APÊNDICES

APÊNDICE A – NOTAÇÕES UTILIZADAS EM REDES NEURAIS

Standard notations for Deep Learning

This document has the purpose of discussing a new standard for deep learning mathematical notations.

1 Neural Networks Notations.

General comments:

- superscript (i) will denote the i^{th} training example while superscript [l] will denote the l^{th} layer

Sizes:

· m : number of examples in the dataset

· n_x : input size

· n_y : output size (or number of classes)

· $n_h^{[l]}$: number of hidden units of the l^{th} layer

In a for loop, it is possible to denote $n_x = n_h^{[0]}$ and $n_y = n_h^{[\text{number of layers} + 1]}$.

· L : number of layers in the network.

Objects:

· $X \in \mathbb{R}^{n_x \times m}$ is the input matrix

· $x^{(i)} \in \mathbb{R}^{n_x}$ is the i^{th} example represented as a column vector

· $Y \in \mathbb{R}^{n_y \times m}$ is the label matrix

· $y^{(i)} \in \mathbb{R}^{n_y}$ is the output label for the i^{th} example

· $W^{[l]} \in \mathbb{R}^{\text{number of units in next layer} \times \text{number of units in the previous layer}}$ is the weight matrix, superscript [l] indicates the layer

· $b^{[l]} \in \mathbb{R}^{\text{number of units in next layer}}$ is the bias vector in the l^{th} layer

· $\hat{y} \in \mathbb{R}^{n_y}$ is the predicted output vector. It can also be denoted $a^{[L]}$ where L is the number of layers in the network.

Common forward propagation equation examples:

$a = g^{[l]}(W_x x^{(i)} + b_1) = g^{[l]}(z_1)$ where $g^{[l]}$ denotes the l^{th} layer activation function

$\hat{y}^{(i)} = \text{softmax}(W_h h + b_2)$

· General Activation Formula: $a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}) = g^{[l]}(z_j^{[l]})$

· $J(x, W, b, y)$ or $J(\hat{y}, y)$ denote the cost function.

Examples of cost function:

· $J_{CE}(\hat{y}, y) = -\sum_{i=0}^m y^{(i)} \log \hat{y}^{(i)}$

· $J_1(\hat{y}, y) = \sum_{i=0}^m |y^{(i)} - \hat{y}^{(i)}|$

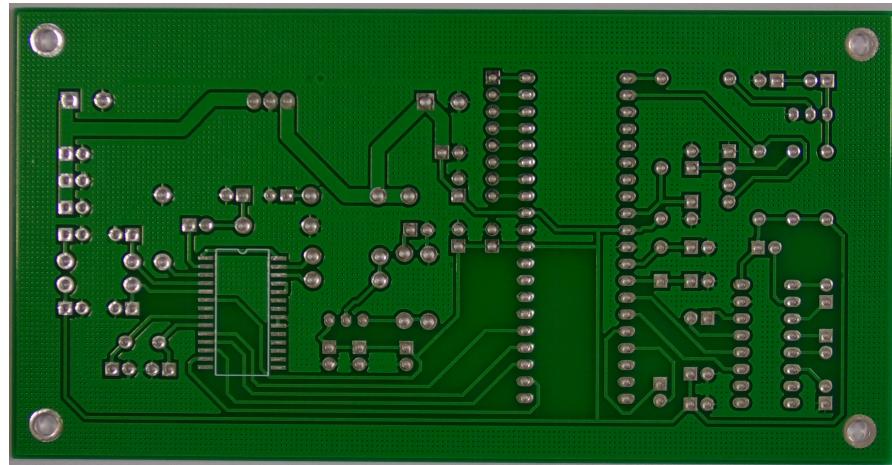
APÊNDICE B – ETAPAS DE TREINAMENTO UTILIZANDO A REDE NEURAL YOLO

- Criar o arquivo “yolo-obj.cfg” com o mesmo conteúdo de “yolov4-custom.cfg”;
- Alterar “yolo-obj.cfg” para:
 - Considerar *batches* com 64 imagens, onde um *batch* representa o número de amostras a serem trabalhadas antes de atualizar os parâmetros do modelo (BROWNLEE, 2018);
 - Subdividir cada *batch* em 16 imagens (para processamento paralelo em treinamentos utilizando GPUs - Unidade de Processamento Gráfico do inglês *Graphics Processing Unit*);
 - Utilizar um número máximo de *batches* igual ao produto $n_{classes} \cdot 2000$
 - Alterar o número de passos de acordo com o número máximo de *batches*;
 - Alterar a quantidade de filtros considerando a fórmula $(n_{classes} + 5) \cdot 3$.
- Criar o arquivo “obj.names” com os nomes das classes utilizadas, com cada nome em uma linha nova;
- Alterar no arquivo “obj.data” o número de classes utilizadas;
- Colocar as imagens de treinamento e teste no diretório “obj” que se encontra dentro de “data”;
- Criar um arquivo de texto com a extensão “.txt” para cada uma das imagens do *dataset* no mesmo diretório em que elas estão contendo os objetos e suas respectivas coordenadas, seguindo o padrão $<objectclass><x_{center}><y_{center}>< width >< height >$ de forma que cada linha represente um objeto, onde
 - $<objectclass>$ indica a classe do objeto de acordo com a ordem definida no arquivo “obj.names” com valores dentro do intervalo $[0, n_{classes} - 1]$;
 - $<x_{center}><y_{center}>$ são as coordenadas centrais da localização da caixa de limitadora do objeto, valores em ponto flutuante dentro do intervalo $[0.0, 1.0]$, normalizados de acordo com o tamanho da imagem utilizada.
 - $< width >< height >$ são a largura e a altura, respectivamente, da caixa de limitadora do objeto, valores em ponto flutuante dentro do intervalo $[0.0, 1.0]$, normalizados de acordo com o tamanho da imagem utilizada.
- Criar os arquivos de texto “train.txt” e “test.txt” que contém o nome das imagens utilizadas no treinamento e teste, respectivamente, onde para cada imagem utiliza-se uma nova linha;

- Fazer o *download* de pesos pré-treinados, com intuído de acelerar o treinamento;
- Iniciar o treinamento.

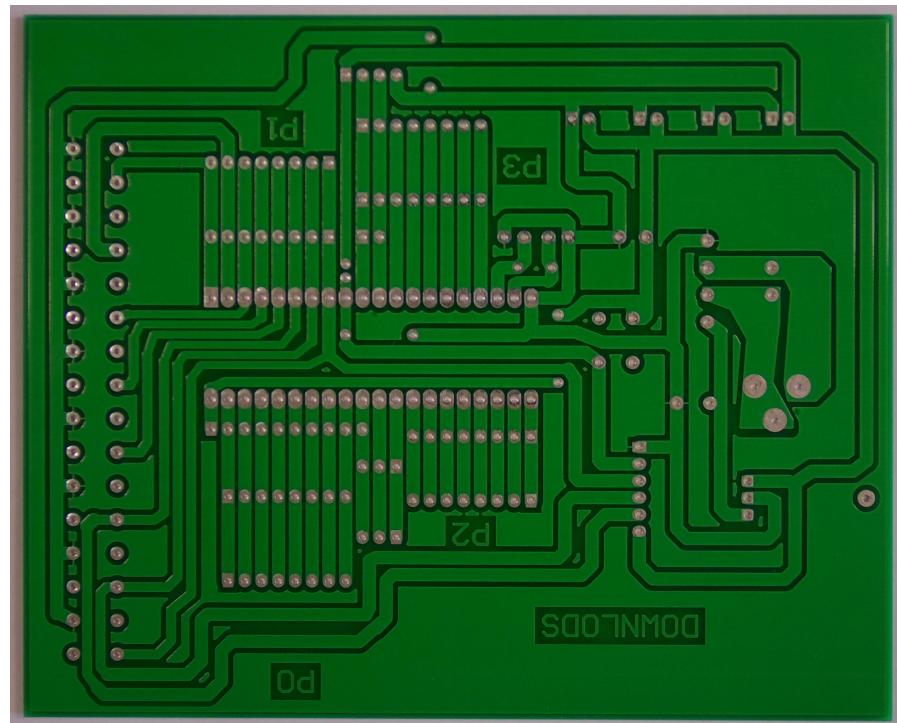
APÊNDICE C – PLACAS DE CIRCUITO IMPRESSO UTILIZADAS PARA O CONJUNTO DE DADOS HRIPCB

Figura 42 – PCI Número 01 do conjunto de dados HRIPCB.



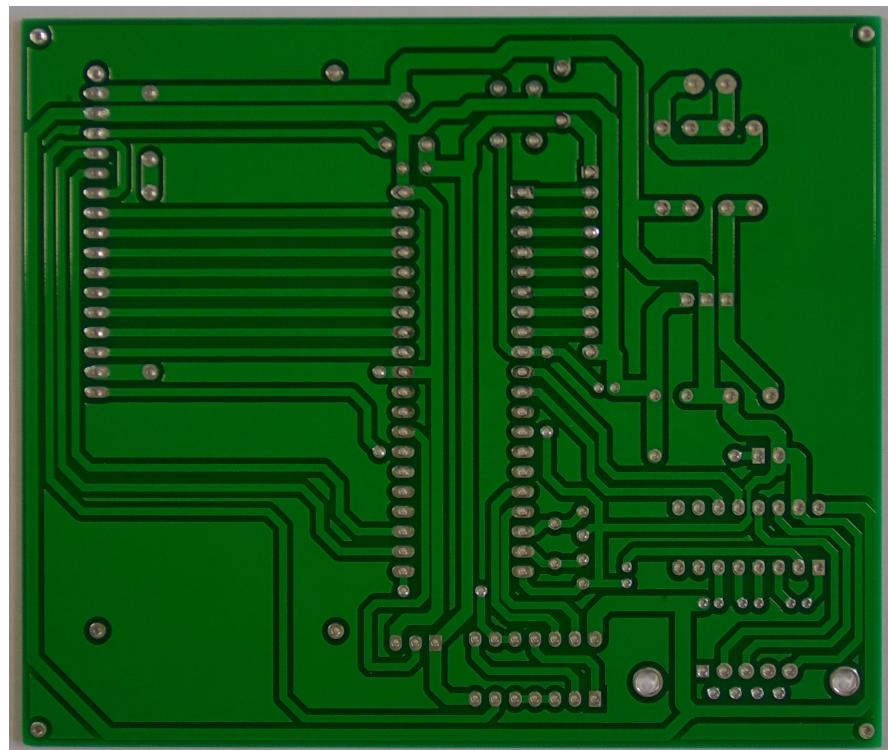
Fonte: Huang et al. (2020).

Figura 43 – PCI Número 04 do conjunto de dados HRIPCB.



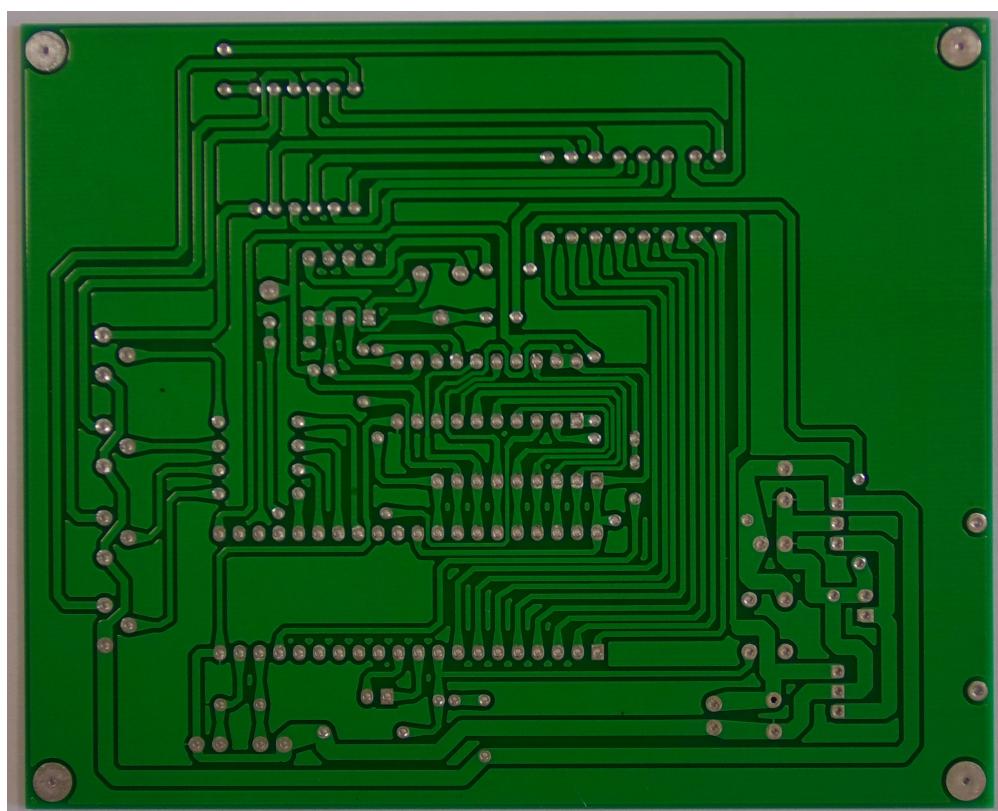
Fonte: Huang et al. (2020).

Figura 44 – PCI Número 05 do conjunto de dados HRIPCB.



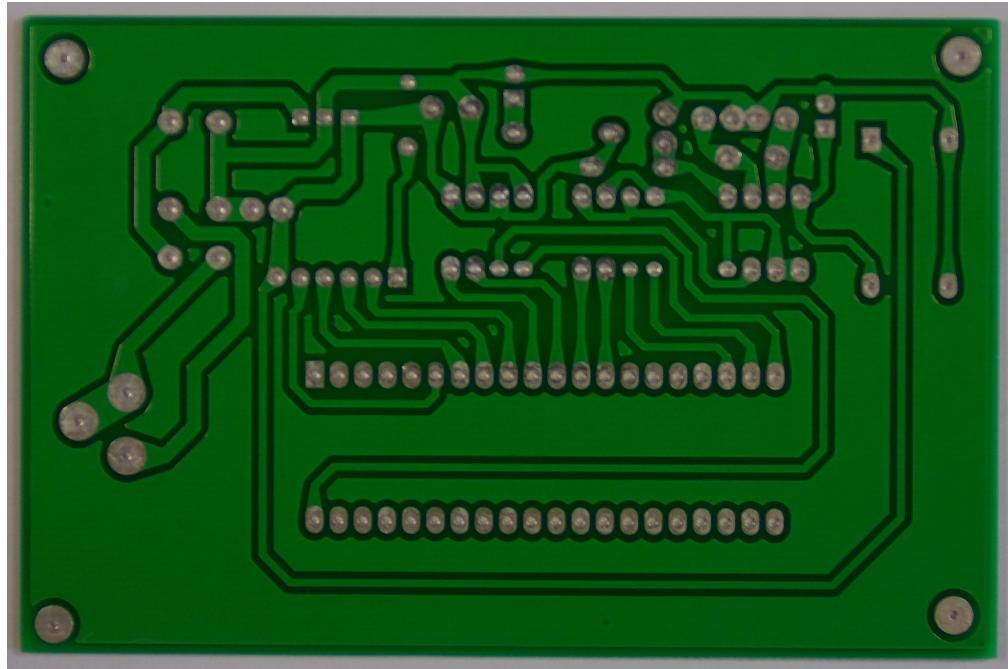
Fonte: Huang et al. (2020).

Figura 45 – PCI Número 06 do conjunto de dados HRIPCB.



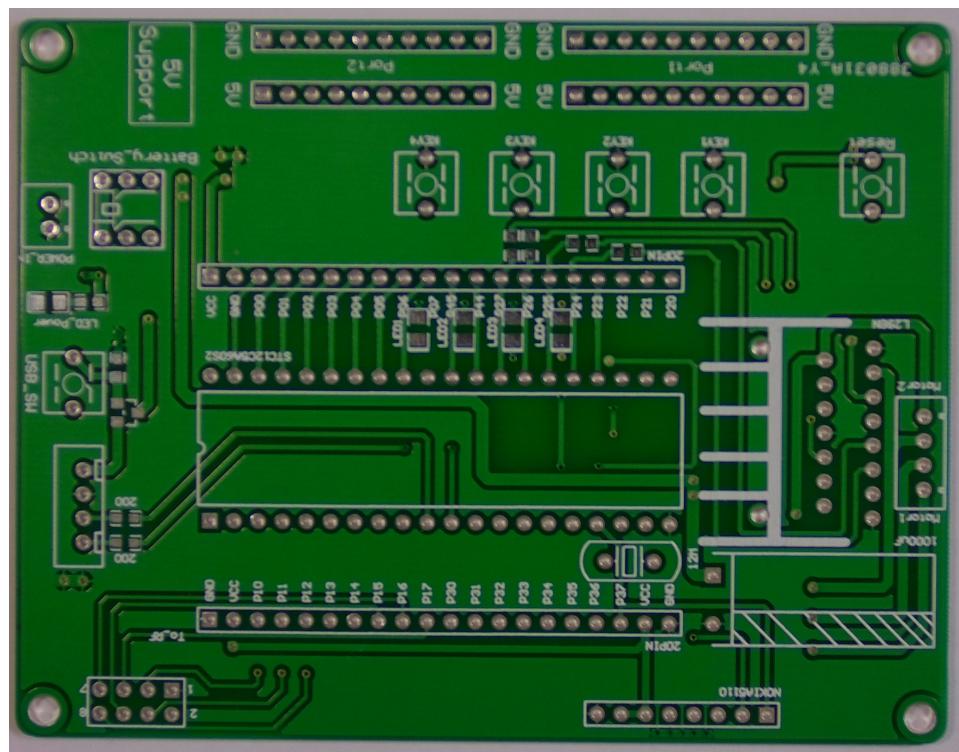
Fonte: Huang et al. (2020).

Figura 46 – PCI Número 07 do conjunto de dados HRIPCB.

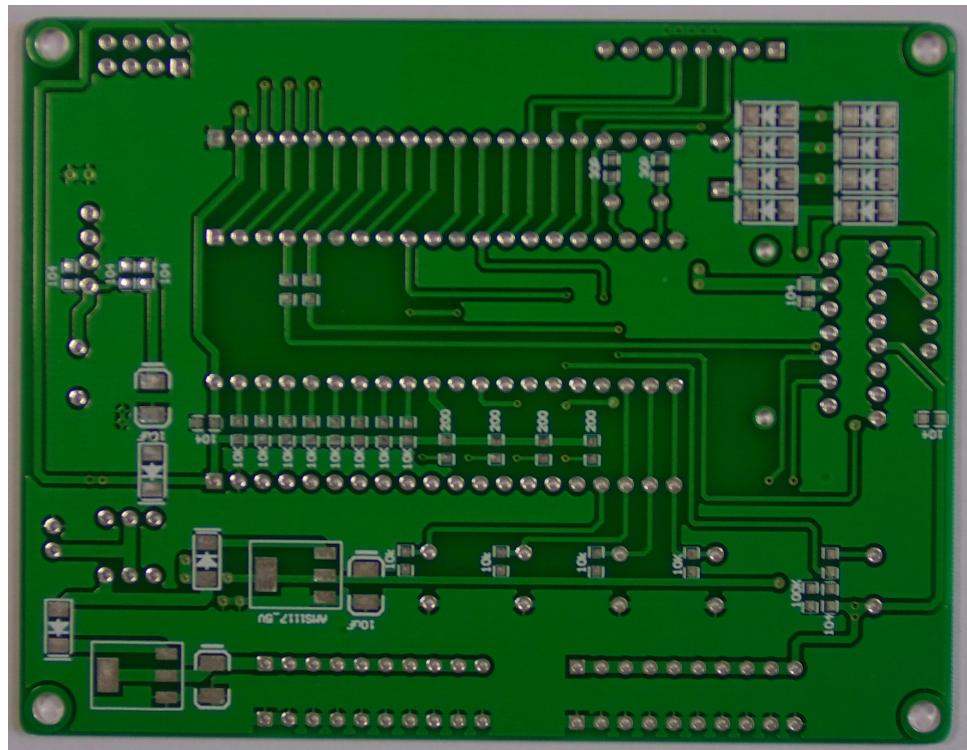


Fonte: Huang et al. (2020).

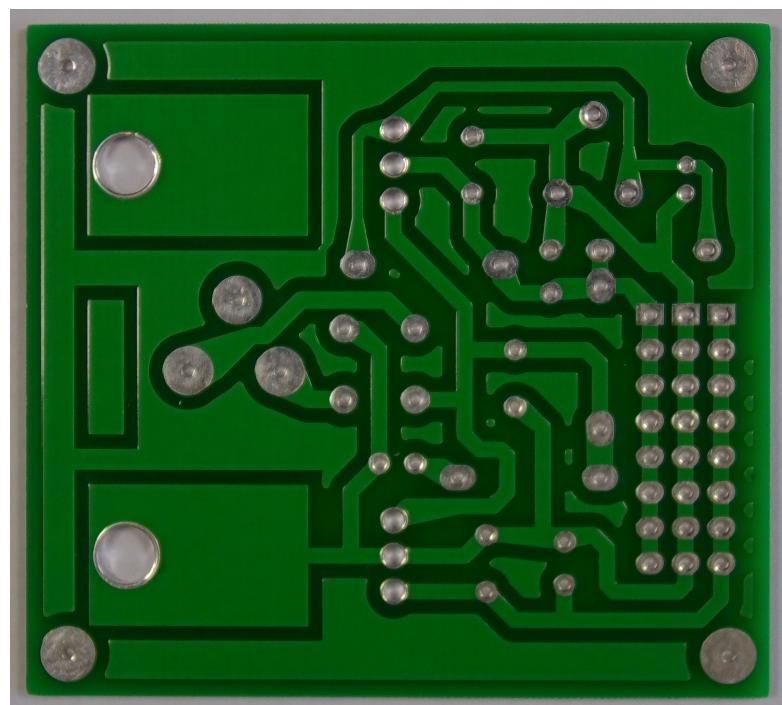
Figura 47 – PCI Número 08 do conjunto de dados HRIPCB.



Fonte: Huang et al. (2020).

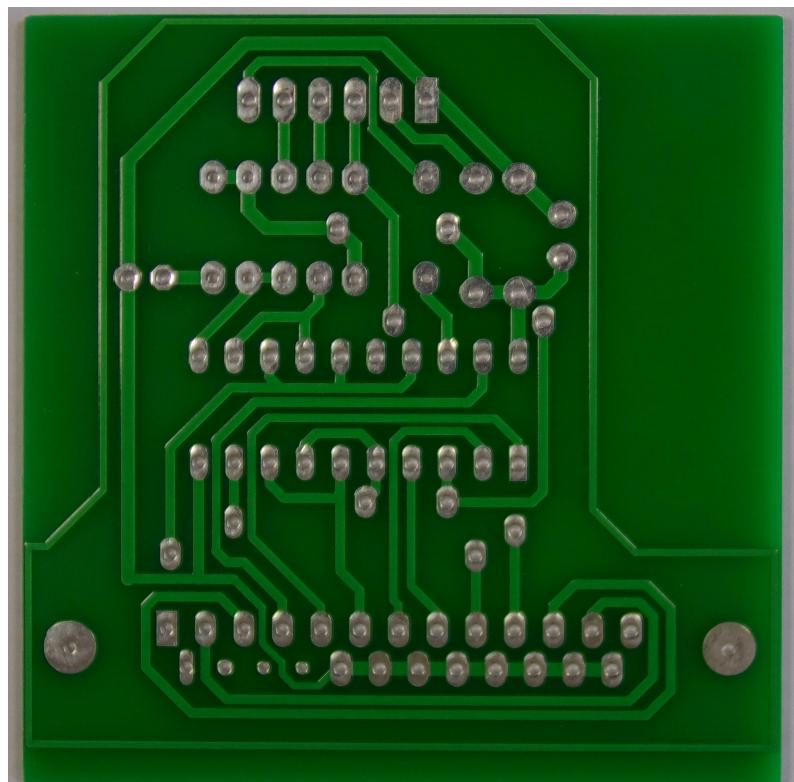
Figura 48 – PCI Número 09 do conjunto de dados HRIPCB.

Fonte: Huang et al. (2020).

Figura 49 – PCI Número 10 do conjunto de dados HRIPCB.

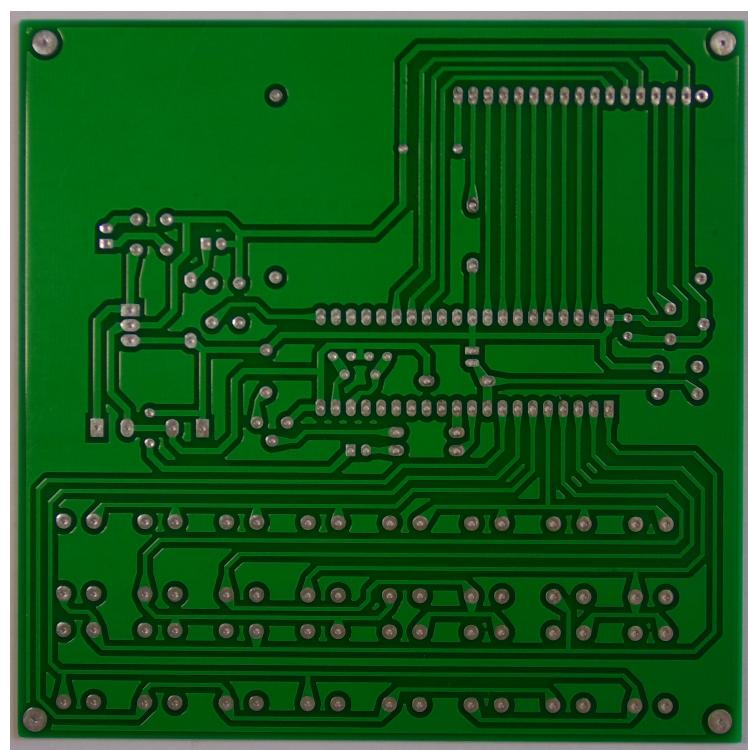
Fonte: Huang et al. (2020).

Figura 50 – PCI Número 11 do conjunto de dados HRIPCB.



Fonte: Huang et al. (2020).

Figura 51 – PCI Número 12 do conjunto de dados HRIPCB.



Fonte: Huang et al. (2020).

APÊNDICE D – EXEMPLO DE ARQUIVO DE ANOTAÇÃO DO CONJUNTO DE DADOS HRIPCB EM XML

Código D.1 – Arquivo de Anotação do *Dataset* HRIPCB.

```

1 <annotation>
2   <folder>Missing_hole</folder>
3   <filename>01_missing_hole_01.jpg</filename>
4   <path>/home/weapon/Desktop/PCB_DATASET/Missing_hole/01_missing_hole_01.jpg</path>
5   <source>
6     <database>Unknown</database>
7   </source>
8   <size>
9     <width>3034</width>
10    <height>1586</height>
11    <depth>3</depth>
12  </size>
13  <segmented>0</segmented>
14  <object>
15    <name>missing_hole</name>
16    <pose>Unspecified</pose>
17    <truncated>0</truncated>
18    <difficult>0</difficult>
19    <bndbox>
20      <xmin>2459</xmin>
21      <ymin>1274</ymin>
22      <xmax>2530</xmax>
23      <ymax>1329</ymax>
24    </bndbox>
25  </object>
26  <object>
27    <name>missing_hole</name>
28    <pose>Unspecified</pose>
29    <truncated>0</truncated>
30    <difficult>0</difficult>
31    <bndbox>
32      <xmin>1613</xmin>
33      <ymin>334</ymin>
34      <xmax>1679</xmax>
35      <ymax>396</ymax>
36    </bndbox>
37  </object>
38  <object>
39    <name>missing_hole</name>
40    <pose>Unspecified</pose>
41    <truncated>0</truncated>
42    <difficult>0</difficult>
43    <bndbox>
44      <xmin>1726</xmin>
45      <ymin>794</ymin>
46      <xmax>1797</xmax>
47      <ymax>854</ymax>
48    </bndbox>
49  </object>
50 </annotation>
```

APÊNDICE E – *SCRIPT UTILIZADO NA CONVERSÃO DOS ARQUIVOS DE ANOTAÇÃO*

Código E.1 – Script para conversão de arquivos de anotação de xml para txt.

```

1 """
2 Conversão das anotações de "HRIPCB dataset" em xml para as anotações em txt
3 utilizadas no YOLO
4 Referências:
5 https://stackoverflow.com/questions/19772288/python-parse-xml-and-save-as-txt
6 """
7 import os
8 import cv2
9 import numpy as np
10 from tqdm import tqdm
11 import argparse
12 import fileinput
13 import matplotlib.pyplot as plt
14 import xml.etree.ElementTree as ET
15
16 obj_names = ["open_circuit", "short", "mouse_bite", "spur", "spurious_copper", "missing_hole"]
17
18 def convert(splited_filename, coords, group):
19     img_dir = "/home/ana/github/PCB_DATASET/images/" + group + "/" +
20     splited_filename + ".jpg"
21     imagem = cv2.imread(img_dir)
22     coords[2] -= coords[0] # diferença largura (x_fim - x_inicio)
23     coords[3] -= coords[1] # diferença altura (y_fim - y_inicio)
24     x_dif = int(coords[2] / 2) # converte diferença_largura pra int
25     y_dif = int(coords[3] / 2) # converte diferença_altura pra int
26     coords[0] = coords[0] + x_dif # x_inicio + diferença largura
27     coords[1] = coords[1] + y_dif # y_inicio + diferença altura
28     largura = int(imagem.shape[1]) # = largura da imagem
29     altura = int(imagem.shape[0]) # = altura da imagem
30     coords[0] /= largura
31     coords[1] /= altura
32     coords[2] /= largura
33     coords[3] /= altura
34     return coords
35
36 if __name__ == "__main__":
37     annotations_folder = 'Annotations'
38     os.chdir('/home/ana/github/PCB_DATASET')
39     for folder in os.listdir(annotations_folder):
40         if (folder.endswith(".JPG") or folder.endswith(".jpg") or folder.endswith(".TXT") or folder.endswith(".txt")) :
41             pass
42         else:
43             print(' - - - ' + folder + ' - - - ')
44             for filename in tqdm(os.listdir(annotations_folder + '/' + folder)):
45                 if (filename.endswith(".XML") or filename.endswith(".xml")) :
46                     splited_filename = str.split(filename, ".")[0]
47                     annotations = []

```

```
47         tree = ET.parse(annotations_folder+'/'+folder+"/"+filename)
48         root = tree.getroot()
49         annotations = []
50         for object in root.findall('object'):
51             coords = []
52             name = object.find('name').text
53             coords.append(int(object.find('bndbox').find('xmin').text))
54             coords.append(int(object.find('bndbox').find('ymin').text))
55             coords.append(int(object.find('bndbox').find('xmax').text))
56             coords.append(int(object.find('bndbox').find('ymax').text))
57             coords = convert(splited_filename, coords, folder)
58             newline = str(obj_names.index(name)) + " " + str(coords[0])
59             + " " + str(coords[1]) + " " + str(coords[2]) + " " + str(coords[3])
60             annotations.append(newline)
61
62             os.chdir("/home/ana/github/pcb-fabrication-fault-detection/
63             pcb_data/data")
64             with open(splited_filename + ".txt", "w") as outfile:
65                 for line in annotations:
66                     outfile.write(line)
67                     outfile.write("\n")
68                     outfile.close()
69             os.chdir('/home/ana/github/PCB_DATASET')
```

APÊNDICE F – SCRIPT UTILIZADO NA DIVISÃO DO DATASET PARA TREINAMENTO DA REDE NEURAL

Código F.1 – Script para divisão do dataset.

```

1 """
2 Separação do dataset para treinamento e teste
3 """
4 import os
5 import cv2
6 import numpy as np
7 import fileinput
8 from random import seed
9 from random import random
10 import math
11
12 test_fraction = 0.2
13 train_fraction = 1 - test_fraction
14
15 seed()
16
17 ROOT_DIR = os.getcwd()
18 dir = ROOT_DIR + '/data'
19 os.chdir(dir)
20
21 dataset_size = int(len(os.listdir(dir))/2)
22
23 list = []
24 test_list = []
25 train_list = []
26
27 for filename in os.listdir(os.getcwd()):
28     if filename.endswith(".txt"):
29         file = str.split(filename, ".") [0]
30         list.append(file)
31
32
33 #Colocando aleatoriamente itens dentro da pasta de validacao
34 for x in range(int(test_fraction*dataset_size)):
35     filename = list.pop(int(random()*(dataset_size-x)))
36     os.rename(filename + ".txt", "../valid/" + filename + ".txt")
37     os.rename(filename + ".jpg", "../valid/" + filename + ".jpg")
38     test_list.append(filename)
39
40 #Colocando o restante dos itens dentro da pasta de treinamento
41 for filename in list:
42     os.rename(filename + ".txt", "../obj/" + filename + ".txt")
43     os.rename(filename + ".jpg", "../obj/" + filename + ".jpg")
44     train_list.append(filename)
45
46 # salvando arquivo txt com a lista de teste e treinamento
47 dir = ROOT_DIR + '/../yolov4_train'
48 os.chdir(dir)
49 with open("test.txt", "w") as outfile:
50     for line in test_list:

```

```
51     outfile.write('data/valid/' + line + '.jpg')
52     outfile.write("\n")
53     outfile.close()
54
55 with open("train.txt", "w") as outfile:
56     for line in train_list:
57         outfile.write('data/obj/' + line + '.jpg')
58         outfile.write("\n")
59     outfile.close()
```

APÊNDICE G – CÓDIGO UTILIZADO PARA O TREINAMENTO DA REDE NEURAL NO AMBIENTE GOOGLE COLAB

Código G.1 – Comandos executados no ambiente Google Colab para o treinamento da Rede Neural.

```

1 # -*- coding: utf-8 -*-
2 """treinamento_YOLOv4_hripcb.ipynb
3 Automatically generated by Colaboratory.
4 Original file is located at
5     https://colab.research.google.com/drive/10Lco0mSDQbwCXTLOVd0twqukQaZAzJzM
6 """
7 import cv2
8 import matplotlib.pyplot as plt
9 import tensorflow as tf
10
11 !git clone https://github.com/AlexeyAB/darknet
12
13 %cd darknet
14
15 !sed -i 's/OPENCV=0/OPENCV=1/' Makefile
16 !sed -i 's/GPU=0/GPU=1/' Makefile #habilitar treinamento em GPU
17 !sed -i 's/CUDNN=0/CUDNN=1/' Makefile
18 !make
19
20 #incluir arquivos de treinamento para a pasta darknet
21 !cp -R /content/pcb_data/obj /content/darknet/data
22 !cp -R /content/pcb_data/valid /content/darknet/data
23 #incluir arquivos de configurações para a pasta darknet
24 !cp /content/yolov4/cfg/obj.data /content/darknet/data
25 !cp /content/yolov4/cfg/obj.names /content/darknet/data
26 !cp /content/yolov4/cfg/yolov4_custom.cfg /content/darknet/cfg
27 !cp /content/yolov4/train.txt /content/darknet/data
28 !cp /content/yolov4/test.txt /content/darknet/data
29
30 #transferencia de aprendizagem
31 !wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal
32 #treinamento
33 !./darknet detector train data/obj.data cfg/yolov4_custom.cfg yolov4.conv.137 -
    dont_show -map

```

APÊNDICE H – CONFIGURAÇÕES PARA A EXECUÇÃO DA INTERFACE DE APLICAÇÃO DE DETECCÃO DE DEFEITOS EM PLACAS DE CIRCUITO IMPRESSO

Código H.1 – Compilando o Darknet

```
1 $ git clone https://github.com/AlexeyAB/darknet.git
2 $ cd Darknet
3 $ sed -i 's/OPENCV=0/OPENCV=1/' Makefile
4 $ sed -i 's/LIBSO=0/LIBSO=1/' Makefile
5 $ sed -i 's/DEBUG=0/DEBUG=1/' Makefile
6 $ make
```

Código H.2 – Executando a Interface de Aplicação

```
1 $ cd ../
2 $ git clone https://github.com/anabdck/pcb-defect-detection-api.git
3 $ cd pcb-defect-detection-api
4 $ pip3 install virtualenv
5 $ python3 -m venv flask_venv
6 $ source flask_venv/bin/activate
7 $ pip3 install -r requirements.txt
8 $ cp ../../Darknet/libdarknet.so libdarknet.so
9 $ python run.py runserver --host 0.0.0.0
```