



# Universidade Federal de Itajubá

## Instituto de Engenharia de Sistemas e Tecnologia da Informação

**Modelagem matemática para elucidar o  
potencial de redução de dose de Inibidores de  
Tirosina-Quinase no tratamento de Leucemia  
Mieloide Crônica**

RELATÓRIO FINAL  
PROGRAMA PIBIC

Universidade Federal de Itajubá/União  
CICLO 2019/2020

**Aluno: Fabio Augusto Ramalho  
Matrícula: 2018006250  
Curso: Engenharia de Computação  
Orientador: Artur César Fassoni**

**Fase/Período: 6º Período**

**Vigência: setembro/2019 a agosto/2020**

## **AGRADECIMENTOSX**

Meus agradecimentos ao professor Artur C. Fassoni pelo convite para a pesquisa e confiança no meu trabalho e aos meus professores do IESTI, principalmente o professor Edmilson Marmo Moreira, que me fez enxergar os dados e algoritmos como grandes ferramentas da humanidade, capazes de realizar mudanças no rumo da história.

## **RESUMO**

O uso da inteligência artificial e suas aplicações em diversos temas tem crescido cada vez mais. O objetivo deste trabalho era estudar a construção de redes neurais e algumas de suas aplicações à medicina. Inicialmente, estudamos na literatura as fases do desenvolvimento de uma Rede Neural Profunda e as escolhas de algoritmos e parâmetros a serem tomadas nas configurações destas fases. Também foram estudadas aplicações de Redes Neurais Profundas em problemas relacionados à área médica a fim de entender suas limitações. Os problemas foram a falta de confiabilidade nos resultados e riscos atrelados a eles. Finalmente, a principal contribuição do trabalho foi desenvolver uma interface gráfica para facilitar a implementação da codificação de uma Rede Neural Profunda, baseado na biblioteca Keras, desenvolvida por François Chollet. Este tipo de rede neural tem sido cada vez mais utilizado, devido ao crescente aumento no poder de processamento dos computadores. Esta interface gráfica facilita a criação e uso de redes neurais por usuários não programadores, aproximando esta nova tecnologia dos campos de estudo onde ela pode ser útil, agilizando processos humanos.

## LISTA DE ILUSTRAÇÕES

Figura 1 – Representação da função relu. ....	12
Figura 2 – Representação da função sigmoid. ....	12
Figura 3 – Ilustração das possíveis janelas em uma operação de convolução. ....	13
Figura 4 – entropia cruzada para $y=1$ . ....	15
Figura 5 – entropia cruzada para $y=0$ . ....	15
Figura 6 – entropia cruzada para $y=0.3$ . ....	15
Figura 7 – entropia cruzada para $y=0.5$ . ....	15
Figura 8 – entropia cruzada para $y=0.7$ . ....	15
Figura 9 – Tela Inicial da Interface Gráfica. ....	18
Figura 10 – Menu para escolha do conjunto de dados. ....	19
Figura 11 – Arquivos do programa. ....	19
Figura 12 – Pasta onde se encontram os conjuntos de dados. ....	20
Figura 13 – Conjunto chest_xray separado entre imagens para treino, teste e validação.	21
Figura 14 – Conjunto de testes separado entre saudáveis (NORMAL) e com pneumonia. ....	21
Figura 15 – Imagens de raio-X classificados como saudáveis (NORMAL). ....	22
Figura 16 – Conjunto histopathologic-cancer-detection separado entre imagens para treino e teste, juntamente com os arquivos CSV com as classificações. ....	22
Figura 17 – Imagens do conjunto de teste. ....	23
Figura 18 – Conteúdo do arquivo CSV relativo ao conjunto de treino. ....	23
Figura 19 – Menu para escolha dos dados com um conjunto selecionado. ....	24
Figura 20 – Menu inicial após a escolha dos dados. ....	24
Figura 21 – Primeira página do menu de tratamento de dados. ....	25
Figura 22 – Segunda página do menu de tratamento de dados. ....	26
Figura 23 – Menu de tratamento de dados e janela aberta após clicar em ‘tamanho’ com o botão ‘Info’ habilitado. ....	26
Figura 24 – Menu de tratamento de dados com algumas opções selecionadas. ....	27
Figura 25 – Menu inicial após configurar o tratamento dos dados. ....	28
Figura 26 – Menu para criação da arquitetura da rede. ....	28
Figura 27 – Menu para a escolha das camadas. ....	29
Figura 28 – Menu das Core Layers. ....	29
Figura 29 – Menu das Pooling Layers. ....	30
Figura 30 – Menu das Convolutional Layers. ....	30
Figura 31 – Menu das Core Layers e janela de informações sobre a camada Dense. ....	31
Figura 32 – Menu de configuração da camada Dense. ....	31
Figura 33 – Menu de criação da arquitetura após a inserção das camadas Dense, Activation, Flatten e Dense, respectivamente. ....	32
Figura 34 – Menu de criação da arquitetura e opções para começar uma nova, carregar uma salva anteriormente e salvar a atual. ....	33
Figura 35 – Menu para salvar um arquivo de arquitetura de rede. ....	33
Figura 36 – Menu para carregar um arquivo de arquitetura de rede. ....	34
Figura 37 – Último menu de configurações. ....	34
Figura 38 – Código Python gerado a partir do exemplo. ....	35
Figura 39 – Código usado para gerar mil imagens azuis e mil imagens vermelhas. ....	36
Figura 40 – Algumas das imagens de tons azulados geradas. ....	36
Figura 41 – Algumas das imagens de tons avermelhados geradas. ....	36
Figura 42 – Escolha do conjunto de dados das tonalidades no software. ....	37
Figura 43 – Configuração da etapa de tratamento de dados, página um. ....	38

Figura 44 – Configuração da etapa de tratamento de dados, página dois. ....	38
Figura 45 – Etapa da configuração da arquitetura da rede. ....	39
Figura 46 – Configuração da primeira camada densa da rede. ....	39
Figura 47 – Configuração da segunda camada densa da rede. ....	39
Figura 48 – Configuração final da etapa de arquitetura da rede. ....	40
Figura 49 – Código final gerado (rede.py). ....	40
Figura 50 – Comandos para o treinamento da rede no computador. ....	41
Figura 51 – Sumário do modelo da rede. ....	41
Figura 52 – Resultado do processo de treinamento. ....	41
Figura 53 – Imagens avaliadas na rede e respectivos nomes. ....	42
Figura 54 – Comandos para tratamento de uma imagem e avaliação na rede. ....	42
Figura 55 – Avaliação das cinco imagens na rede. ....	42
Figura 56 – Código usado para gerar as imagens de tom verde. ....	43
Figura 57 – Algumas das imagens de tons esverdeados geradas. ....	43
Figura 58 – Configurações da última camada da rede. ....	43
Figura 59 – Última etapa de configurações da rede. ....	44
Figura 60 – Trecho de código referente ao modelo da rede. ....	44
Figura 61 – Resultado do processo de treinamento com a função de perda MSE. ....	44
Figura 62 – Sumário da rede com MSE. ....	44
Figura 63 – Trecho de código referente ao modelo da rede com MAE. ....	45
Figura 64 – Resultado do processo de treinamento com a função de perda MAE. ....	45
Figura 65 – Sumário da rede com 9 saídas na primeira camada. ....	45
Figura 66 – Resultado do processo de treinamento com apenas 600 imagens. ....	46
Figura 67 – Avaliação da rede sob o conjunto de testes. ....	46
Figura 68 – Resultado do processo de treinamento com apenas 60 imagens. ....	46
Figura 69 – Processo de avaliação da rede sob as 2400 imagens de teste. ....	46
Figura 70 – Configurações da última etapa da rede. ....	47
Figura 71 – Seção de código do modelo com entropia cruzada categórica. ....	47
Figura 72 – Processo de treinamento da rede com entropia cruzada categórica e 60 imagens. ....	47
Figura 73 – Etapas finais do treinamento com 30 épocas. ....	48
Figura 74 – Processo de treinamento da rede com 600 imagens. ....	48
Figura 75 – Avaliação da rede sob o conjunto de testes (2400 imagens). ....	48
Figura 76 – Imagens avaliadas. ....	49
Figura 77 – Saída da primeira camada densa sob a imagem azul, rede com MAE. ....	49
Figura 78 – Saída da primeira camada densa sob a imagem verde, rede com MAE. ....	49
Figura 79 – Saída da primeira camada densa sob a imagem vermelha, rede com MAE. ....	50
Figura 80 – Saída da primeira camada densa sob a imagem azul, rede com ECC. ....	50
Figura 81 – Saída da primeira camada densa sob a imagem verde, rede com ECC. ....	50
Figura 82 – Saída da primeira camada densa sob a imagem vermelha, rede com ECC. ....	51
Figura 83 – Parte do código que gerou as imagens com quadrados. ....	51
Figura 84 – Parte do código que gerou as imagens com triângulos. ....	52
Figura 85 – Imagens com quadrados geradas. ....	52
Figura 86 – Imagens com triângulos geradas. ....	53
Figura 87 – Menu de seleção dos dados, conjunto de figuras geométricas. ....	53
Figura 88 – Menu de tratamento de dados, página um. ....	54
Figura 89 – Menu de tratamento de dados, página dois. ....	54
Figura 90 – Menu de criação de modelos para a rede densa. ....	55
Figura 91 – Primeira camada densa. ....	55
Figura 92 – Segunda camada densa. ....	55

Figura 93 – Terceira camada densa. ....	55
Figura 94 – Menu de compilação da rede dense. ....	56
Figura 95 – Precisão da rede dense através das 30 épocas. ....	56
Figura 96 – Perda computada da rede dense através das 30 épocas. ....	56
Figura 97 - Menu de criação de modelos para a rede convolutiva. ....	57
Figura 98 – Primeira camada convolutiva. ....	57
Figura 99 – Segunda camada convolutiva. ....	57
Figura 100 – Camada maxpooling. ....	58
Figura 101 – Primeira camada densa. ....	58
Figura 102 – Segunda e última camada densa. ....	58
Figura 103 – Menu de compilação da rede conv. ....	58
Figura 104 – Etapa de treinamento da rede conv. ....	59
Figura 105 – Comandos usados para plotar a precisão e a perda do treino. ....	59
Figura 106 - Precisão da rede conv através das épocas. ....	59
Figura 107 – Perda computada da rede conv através das épocas. ....	60
Figura 108 – Comandos usados para obter o caminho do conjunto de testes. ....	60
Figura 109 – Processo de avaliação do conjunto de testes. ....	60
Figura 110 – Precisão durante o processo de treinamento da nova rede. ....	61
Figura 111 – Resultado da avaliação da rede sob o conjunto de testes. ....	61
Figura 112 – Precisão da rede com janelas de tamanho 3 através das épocas. ....	61
Figura 113 – Perda computada pela rede com janelas de tamanho 3 através das épocas. ....	62
Figura 114 – Resultado da avaliação da rede sob o conjunto de treino. ....	62
Figura 115 – Código da rede antes das mudanças. ....	62
Figura 116 – Código da rede depois das mudanças. ....	63
Figura 117 – Processo de treinamento da segunda rede conv. ....	63
Figura 118 – Avaliação do conjunto de testes da segunda rede conv. ....	63
Figura 119 – Processo de treinamento da rede com 3000 imagens. ....	64
Figura 120 – Avaliação da rede com 3000 imagens sob o conjunto de testes. ....	64
Figura 121 – Sumário da rede. ....	65
Figura 122 – Configurações da primeira página do menu de tratamento de dados. ....	65
Figura 123 – Etapa de treinamento da rede final. ....	66
Figura 124 – Avaliação sob o conjunto de testes da rede final. ....	66
Figura 125 – Exemplos com pneumonia. ....	67
Figura 126 – Exemplos sem pneumonia. ....	67
Figura 127 – Primeira página do menu de tratamento de dados. ....	68
Figura 128 – Segunda página do menu de tratamento de dados. ....	68
Figura 129 – Modelo da rede e suas camadas. ....	69
Figura 130 – Primeira parte do código gerado para a rede. ....	69
Figura 131 – Segunda parte do código gerado para a rede. ....	70
Figura 132 – Primeira parte do sumário da rede. ....	70
Figura 133 – Segunda parte do sumário da rede. ....	71
Figura 134 – Etapa de treinamento da primeira rede. ....	71
Figura 135 – Avaliação da primeira rede sob o conjunto de testes. ....	72
Figura 136 – Etapa de treinamento da segunda rede. ....	72
Figura 137 – Avaliação da segunda rede sob o conjunto de testes. ....	72
Figura 138 – exemplos de imagens rotuladas com ‘1’. ....	73
Figura 139 – exemplos de imagens rotuladas com ‘0’. ....	74
Figura 140 – Etapa de treinamento da rede para detecção de metástase. ....	74
Figura 141 – Avaliação sob o conjunto de testes da rede para detecção de metástase. .	74

## **LISTA DE ABREVIATURAS E SIGLAS**

UMCU Radboud University Medical Center

RUMC University Medical Center Utrecht

MAE Mean Absolute Error

MSE Mean Squared Error

ECC Entropia Cruzada Categórica

ECB Entropia Cruzada Binária

RNP Rede Neural Profunda

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO .....</b>	<b>9</b>
<b>2</b>	<b>OBJETIVOS PROPOSTOS .....</b>	<b>10</b>
<b>3</b>	<b>REFERENCIAL TEÓRICO (revisão bibliográfica) .....</b>	<b>11</b>
<b>4</b>	<b>DESCRIÇÃO DAS ATIVIDADES DESENVOLVIDAS .....</b>	<b>17</b>
<b>5</b>	<b>RESULTADOS OBTIDOS E ANÁLISE (e/ou discussão) .....</b>	<b>18</b>
<b>6</b>	<b>CONCLUSÕES .....</b>	<b>75</b>
	<b>REFERÊNCIAS BIBLIOGRÁFICAS .....</b>	<b>76</b>
	<b>APÊNDICE A – ASSINATURA DO ALUNO E DO ORIENTADOR .....</b>	<b>77</b>

## 1 INTRODUÇÃO

As Redes Neurais Profundas são algoritmos de Aprendizagem de Máquina que buscam criar uma inteligência artificial capaz de identificar padrões em dados dos mais variados tipos (imagens, áudios, vídeos, eletrocardiogramas, etc). Aprendizagem de Máquina (*Machine Learning*, em inglês) é o nome dado à algoritmos que, dado um conjunto de dados e respostas, buscam encontrar regras que os relacionam.

A primeira aplicação com sucesso de uma rede neural data de 1989, quando Yann LeCun uniu arquiteturas de redes neurais convolutivas e algoritmos de *backpropagation* e uniu-os para realizar a tarefa de ler dígitos manuscritos. Apesar disso, novos estudos relevantes relacionados à redes neurais só voltaram a aparecer a partir de 2010, quando Redes Neurais Profundas começaram a aparecer em competições de classificação de dados e obter resultados promissores, com precisões de mais de 90%, quebrando assim recordes antes mantidos por outros algoritmos de classificação e aprendizagem de máquina.

O retorno dos estudos nesta área durante a última década se dá pelo grande aumento da capacidade de processamento e armazenamento dos hardwares, possibilitando trabalhar com redes e dados cada vez mais complexos. Os modelos de Redes Neurais Profundas para tarefas como classificação de imagens, reconhecimento de fala e transcrição de textos escritos a mão são o que se tem de mais avançado nessas áreas, com performances quase humanas.

Deste modo, este trabalho busca implementar uma interface gráfica para a criação de Redes Neurais Profundas, e utilizá-la para avaliar alguns modelos de classificação de imagens aplicados em diagnósticos de pneumonia e outras doenças pulmonares em imagens de raio-X, além da detecção de células cancerosas em imagens histopatológicas obtidas através de microscópios digitais.

## 2 OBJETIVOS PROPOSTOS

Os objetivos específicos desta pesquisa foram:

- Estudar os conceitos fundamentais sobre a teoria e a utilização de Redes Neurais Profundas.
- Estudar exemplos de arquiteturas de Redes Neurais Profundas na biblioteca Keras.
- Desenvolver uma interface gráfica para construção de Redes Neurais Profundas.
- Analisar a eficiência de Redes Neurais Profundas quando aplicadas à realização de diagnósticos médicos através de duas aplicações: uma no diagnóstico de metástase em imagens de amostras de tecidos patológicos e outra na detecção de pneumonia em imagens de Raio-X de pulmões.

### 3 REFERENCIAL TEÓRICO

#### 3.1 Inteligência Artificial

O conceito de Inteligência Artificial surge da tentativa de automatizar tarefas intelectuais geralmente realizada por humanos. Os primeiros modelos que surgiram, em 1950, são chamados de Inteligências Artificiais Simbólicas, que consistem em um conjunto de regras definidas à mão por programadores. Acreditava-se que era possível alcançar níveis humanos de eficiência se houvesse um conjunto suficientemente grande de regras, o que é verdadeiro para tarefas bem-definidas como jogar xadrez, porém essa eficiência não se estende para tarefas mais complexas e instáveis como a visão computacional, o reconhecimento de fala e traduções entre linguagens naturais. Se faz necessário uma nova abordagem, o Aprendizado de Máquina.

##### 3.1.1 Aprendizado de Máquina

Segundo François Chollet [X], sob a perspectiva do paradigma de programação clássico um programa é um conjunto de regras e dados que tem como objetivo gerar uma resposta. Já sob a perspectiva do Aprendizado de Máquina, um programa é um conjunto de dados e respostas cujo objetivo é encontrar regras que definem esta dedução e usá-las para classificar dados do mesmo tipo, a fim de gerar respostas originais acerca de uma amostra que a rede não conheceu antes, mas cujos padrões ela aprendeu através de exemplos similares.

Um sistema de Aprendizado de Máquina treina sob um conjunto de dados ao invés de ser explicitamente programado. O processo de treinamento é a tentativa de encontrar a função que define as relações apresentadas ao sistema, através do uso de uma função de perda. As funções de perda dos sistemas atuais são implementadas com base em diferentes campos da ciência como: estatística, álgebra linear, cálculo diferencial, ciência da computação, entre outros, e buscam de alguma maneira minimizar o erro nas respostas que o sistema dá.

##### 3.1.2 Rede Neural Profunda

Uma Rede Neural Profunda é um sistema de Aprendizado de Máquina que utiliza o encadeamento de camadas, que são blocos matemáticos que envolvem operações elementares, convoluções, produtos vetoriais e escalares, entre outros. A primeira camada agirá diretamente sob o dado de entrada, já a segunda camada agirá sob o resultado da anterior, e assim consecutivamente por toda a rede. O processo de aprendizado é feito através de um algoritmo chamado *backpropagation*, que mapeará o quanto uma mudança em cada parâmetro influencia no resultado, encontrando os pontos ótimos para a tarefa que está realizando através do conceito de vetor gradiente do cálculo diferencial.

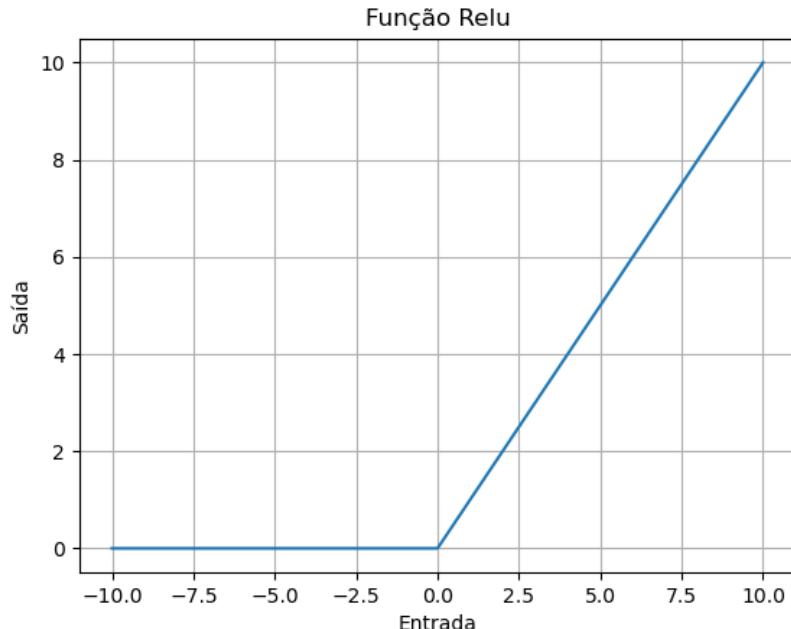
##### 3.1.2.1 Camadas

As camadas usadas nesta pesquisa serão:

###### 3.1.2.1.1 Ativações

A definição matemática de linearidade implica que a concatenação de blocos lineares poderia ser resumida a apenas um bloco, também linear. Desta maneira, as camadas de ativação (*activation layers*, em inglês) buscam dar mais dinâmica ao processo de treinamento através da aplicação de funções não lineares nos dados. Uma das funções de ativação é a *relu* (unidade de retificação linear), que checa termo-a-termo os valores, atribuindo zero àqueles que são negativos e mantendo aqueles que são positivos. Eis um gráfico que representa a função *relu*:

Figura 1 – Representação da função relu.

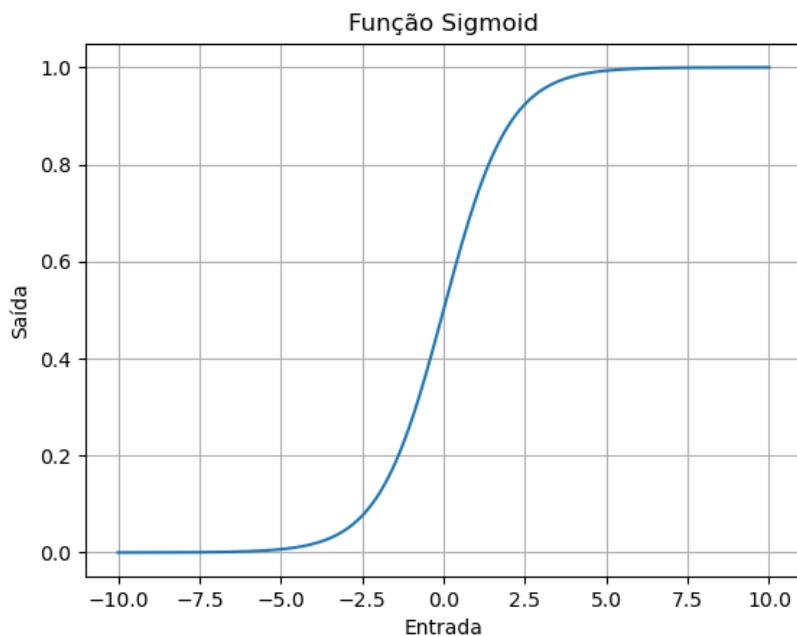


Fonte: o Autor

Desta maneira, a rede é capaz de identificar não apenas pontos relevantes na entrada como também ignorar irrelevâncias, fazendo com que pontos com valores menores que zero não influenciem no resultado. A relu é amplamente aplicada entre camadas intermediárias da rede.

Outra função de ativação amplamente utilizada é a sigmoid. Eis um gráfico que a representa:

Figura 2 – Representação da função sigmoid.



Fonte: o Autor

Esta função define bem um classificador, pois tende a levar os dados de entrada às extremidades da curva. É usada nas etapas finais de redes classificadoras.

### 3.1.2.1.2 Densas

Uma camada densa (ou *dense layers*, em inglês) realiza um produto matricial seguido de uma adição termo a termo de matrizes:

$$X = (M \cdot A) + B$$

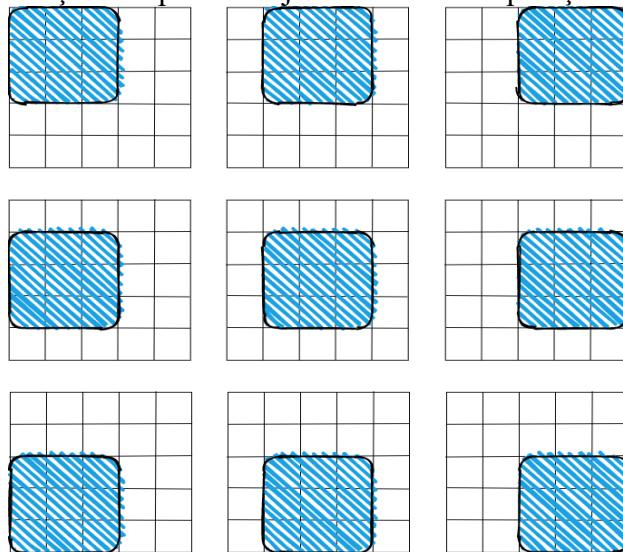
Onde M é a matriz de entrada, A e B são matrizes de pesos reajustáveis da camada e X é o resultado desse produto seguido de uma adição. A e B têm as mesmas dimensões de M para evitar distorções no tamanho da imagem e para tornar válida a operação. Uma camada densa pode gerar mais de um conjunto de matrizes A e B, resultando em várias saídas X diferentes, pois as matrizes de pesos são programadas para não repetirem os padrões umas das outras.

Se aplicada à uma imagem RGB (três canais), a saída será um conjunto de N variações desta imagem, modificando o tamanho da dimensão dos canais. Quando aplicada a um vetor, a interpretação é que cada elemento representa um canal, logo a saída será um outro vetor com o tamanho do número de canais configurado pela rede. O número de parâmetros treináveis desta camada é igual ao tamanho do dado de entrada multiplicado pelo número de saídas X. Por exemplo: uma camada densa aplicada sob uma imagem 20x20 terá  $400 + 400 = 800$  pesos (matrizes A e B) para cada canal de saída, então se configurada para gerar 10 canais, serão 8000 pesos. Para entradas em formato de vetores unidimensionais, uma camada densa terá  $(T + 1)$  pesos por canal, onde T é o tamanho do vetor e o + 1 é o elemento de adição.

### 3.1.2.1.3 Convolutiva

Esta camada aplica uma operação de convolução sobre a entrada. A operação consiste em usar uma matriz de pesos de tamanho arbitrário, porém menor que a dos dados de entrada, e realizar produtos escalares entre a matriz de pesos e uma janela de mesmo tamanho no dado de entrada. Isso acontece para todas as janelas possíveis, e elas são então rearranjadas em uma nova saída com todos os pontos criados. Geralmente as matrizes de peso são de dimensões 3x3 ou 5x5. Eis uma ilustração de uma operação de convolução com matriz de pesos de tamanho 3x3 em uma entrada de tamanho 5x5:

Figura 3 – Ilustração das possíveis janelas em uma operação de convolução.



Fonte: o Autor.

Como pode-se observar na figura, existem 9 possibilidades de produtos escalares diferentes entre uma matriz de pesos 3x3 e uma entrada 5x5. A saída terá sua dimensão afetada pelos efeitos de borda, que são os valores que não possuem uma zona 3x3 ao seu redor. A saída desta camada terá um valor para cada janela possível. Neste caso a dimensão da saída será 3x3,

onde cada valor será o resultado do produto escalar de sua respectiva janela. Assim como na camada densa, existe um fator de adição, que neste caso é um valor único, somado ao resultado do produto escalar. Também há a possibilidade de gerar múltiplas saídas, cada uma sendo o resultado da operação com uma matriz de pesos diferentes. Uma camada convolutiva (*convolutional layer*; em inglês) de janela 3x3 com 32 saídas terá 320 pesos reajustáveis, pois cada uma delas possui uma janela 3x3 (9 valores) e 1 fator de adição, sendo 10 pesos por saída.

#### **3.1.2.1.4 Pooling**

A camada pooling utiliza do mesmo conceito de janelas da camada convolutiva, porém realiza outras operações que visam diminuir o tamanho dos dados conforme percorrem as camadas através da rede. As implementações presentes na biblioteca Keras são: maxpooling, que substitui cada janela pelo maior valor dentre seus parâmetros, e averagepooling, que substitui a janela pelo valor médio de seus parâmetros. As janelas são de tamanho arbitrário e geralmente são configuradas como um espaço 2x2, diminuindo o tamanho das dimensões do dado de entrada pela metade. Esta camada não possui parâmetros treináveis.

#### **3.1.2.1.5 Retificadora**

A camada retificadora (ou *flatten layer*; em inglês) muda o formato dos dados de entrada, transformando-os em um vetor de dimensão única, sem mudar o valor de seus parâmetros. Esta operação é realizada com o intuito de mudar o resultado de uma aplicação de camada densa, que se aplicada em uma entrada de múltiplas dimensões fará um produto matricial, porém se houver apenas uma dimensão tal operação se torna uma multiplicação termo a termo. Esta camada também não possui parâmetros treináveis.

#### **3.1.2.2 Modelos**

Os modelos definem a sequência que as camadas seguirão. É possível criar redes onde os dados de entrada seguirão diferentes caminhos dentro da arquitetura, com a saída de uma camada sendo conectada à entrada de múltiplas outras, bifurcando a rede. Neste estudo serão usados apenas modelos sequenciais, onde as camadas estão conectadas umas às saídas das outras sempre com uma relação de um para um.

#### **3.1.2.3 Otimizadores e funções de perda**

O algoritmo de *backpropagation* é o responsável pelo ajuste dos pesos da rede, e funciona com base em duas funções: a função de perda e o otimizador.

##### **3.1.2.3.1 Funções de perda**

A função de perda, ou também chamada de função de custo, é o guia que a rede usará para saber que ajustes devem ser feitos, pois o otimizador tentará minimizar essa função. São implementações que relacionam o resultado esperado de uma camada com o resultado obtido, e sua escolha é de grande influência no treinamento da rede. Para tarefas de classificação geralmente são usadas funções de entropia cruzada, e para outras tarefas são muito comuns funções baseadas nos erros, como a MSE (mean-squared error) e a MAE (mean-absolute error).

Na MSE, a perda da rede é computada da seguinte maneira:

$$C = (y - a)^2$$

Onde  $C$  é o valor computado para cada peso na rede,  $y$  é o resultado esperado e  $a$  é o resultado obtido. No processo de treinamento, o parâmetro que será passado ao otimizador para que os pesos sejam reajustados é a média dos  $C$ 's calculados por toda a rede.

A MAE segue a mesma ideia, porém com uma equação não quadrática:

$$C = |y - a|$$

As funções de entropia cruzada têm duas versões diferentes. Sua versão binária, usada em problemas de classificação com apenas duas classes distintas, pode ser descrita pela seguinte equação:

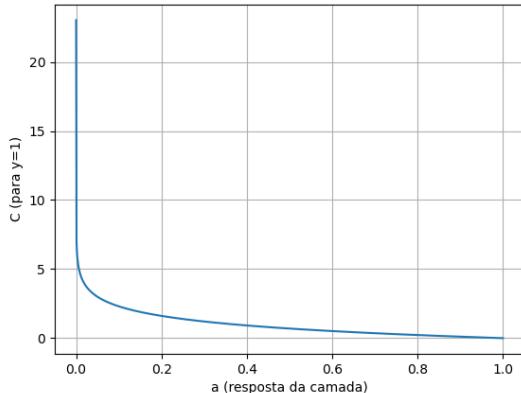
$$C = -[y \ln(a) + (1 - y) \ln(1 - a)]$$

Onde  $y$  é a resposta esperada,  $a$  é a resposta obtida e ambos estão no intervalo [0,1]. Fazendo uma análise da equação, nota-se: na última saída,  $y$  deve assumir o valor 0 ou o valor

1, pois se trata da classificação do dado em uma de duas classes. Isso nos diz que um dos parâmetros de dentro dos colchetes será nulo, o que leva à seguinte dedução: se  $y = 1$ , então  $C = -\ln(a)$ , e se  $y = 0$ , então  $C = -\ln(1 - a)$ . Para quaisquer casos em que  $a$  assume valores na extremidade do intervalo,  $C$  tende a valores muito grandes, como expressam seus respectivos limites e representações gráficas:

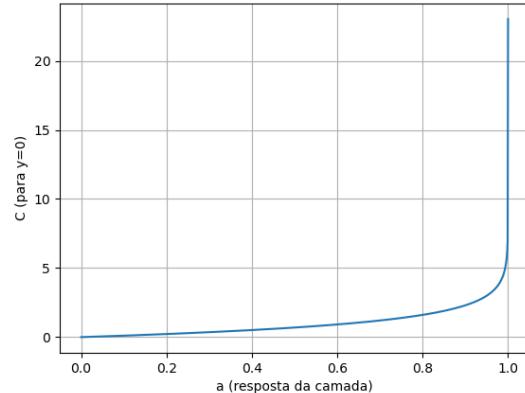
$$\lim_{a \rightarrow 0^+} -\ln(a) = \infty \quad e \quad \lim_{a \rightarrow 1^-} -\ln(1 - a) = \infty$$

Figura 4 – entropia cruzada para  $y=1$ .



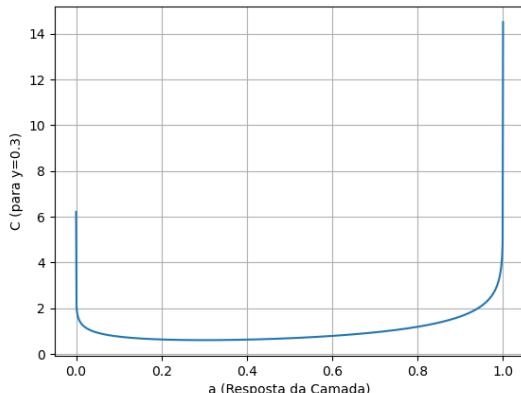
Fonte: o Autor.

Figura 5 – entropia cruzada para  $y=0$ .



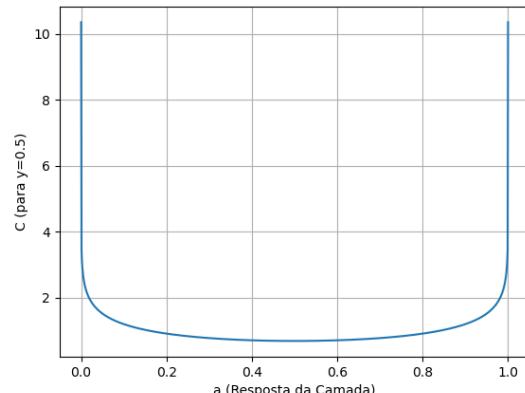
Fonte: o Autor.

Figura 6 – entropia cruzada para  $y=0.3$ .



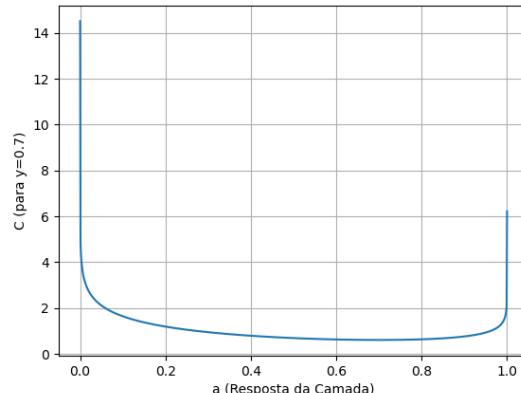
Fonte: o Autor.

Figura 7 – entropia cruzada para  $y=0.5$ .



Fonte: o Autor.

Figura 8 – entropia cruzada para  $y=0.7$ .



Fonte: o Autor.

Em pesos intermediários o resultado será baseado na aplicação da regra da cadeia, e a interpretação da equação é a combinação da tendência do peso em ser elevado ou diminuído. Assim como nas funções baseadas em erro, o valor passado ao otimizador é a média dos  $C$ 's calculados por toda a rede.

A arquitetura de redes classificadoras não binárias prevê um parâmetro final para cada classe, que dirá a possibilidade de um dado pertencer àquela definição ou não. Por exemplo: na camada final de um classificador de dígitos escritos à mão deve haver 10 valores na saída, onde cada um representará a probabilidade que a entrada tem de representar um dos dez dígitos possíveis. Para estes problemas é usada a entropia cruzada categórica, que é uma generalização da implementação binária para N parâmetros  $y$  e computa a soma destes valores.

### 3.1.2.3.2 Otimizadores

O otimizador compõe a estrutura final da rede, sendo o responsável por minimizar o valor computado pela função de perda. Como as camadas são operações matemáticas cujas derivadas são conhecidas, é possível aplicar a regra da cadeia sob seus resultados ( $\frac{dy}{dx} = \frac{dy}{du} * \frac{du}{dx}$ ), e assim reajustar os valores nas direções que correspondem aos mínimos locais da função (onde o valor da derivada é 0), que serão os valores dos pesos que minimizam a função de perda. Um otimizador simples é o SGD (inclinação com gradiente estocástico), que computa o vetor gradiente da operação, aplicando um reajuste baseado neste resultado. Como o gradiente aponta para a direção de maior crescimento da função, os reajustes têm de ser feitos na direção contrária. Nesta implementação mais simples, um peso receberá um ajuste de  $-k\nabla$ , onde  $k$  é um parâmetro configurável e  $\nabla$  é o gradiente. A magnitude de  $k$  irá ditar a magnitude dos ajustes. Um  $k$  pequeno exigirá muitas interações e tenderá a mínimos locais, enquanto um  $k$  muito grande pode levar a rede a nunca encontrar pontos mínimos. Diferentes implementações do SGD variam também na frequência em que os ajustes são feitos. O mais comum é o baseado em lotes, que executa os ajustes a cada vez que  $N$  dados são passados pela função de perda, com  $N$  sendo um valor arbitrário e deve ser pequeno o suficiente para que as alterações aconteçam em um ritmo que não prejudique a rede.

Diferentes implementações de otimizadores levam em conta o conceito de momentum, em que as magnitudes dos ajustes anteriores influenciarão no próximo. Suas diferenças se encontram na representação desta grandeza.

## 4 DESCRIÇÃO DAS ATIVIDADES DESENVOLVIDAS

### 4.1 Etapa dos estudos

Durante os primeiros meses (Setembro/2019 – Dezembro/2019) de pesquisa foi estudado todo o livro do François Chollet [1] a fim de entender o funcionamento das RNP (*Deep Neural Network*, em inglês). Foram também estudadas competições de tarefas relacionadas à classificação de imagens através do portal Kaggle. Também foram avaliados problemas médicos que poderiam ser incorporados à pesquisa em diferentes revistas.

### 4.2 Etapa do desenvolvimento de software

Após a etapa dos estudos foi proposto o desenvolvimento de uma interface para geração de códigos de RNP, e logo foram iniciadas as atividades. Foi escolhida a linguagem Python com a biblioteca gráfica Tkinter, devido à experiência anterior do aluno com a tecnologia. O desenvolvimento durou de Janeiro/2020 à Julho/2020.

### 4.3 Desenvolvimento do relatório

Após o desenvolvimento do software, foi iniciado o desenvolvimento do relatório, com a geração dos conjuntos de dados usados como exemplo, download dos dados médicos e treinamento das redes, com fim em Novembro/2020.

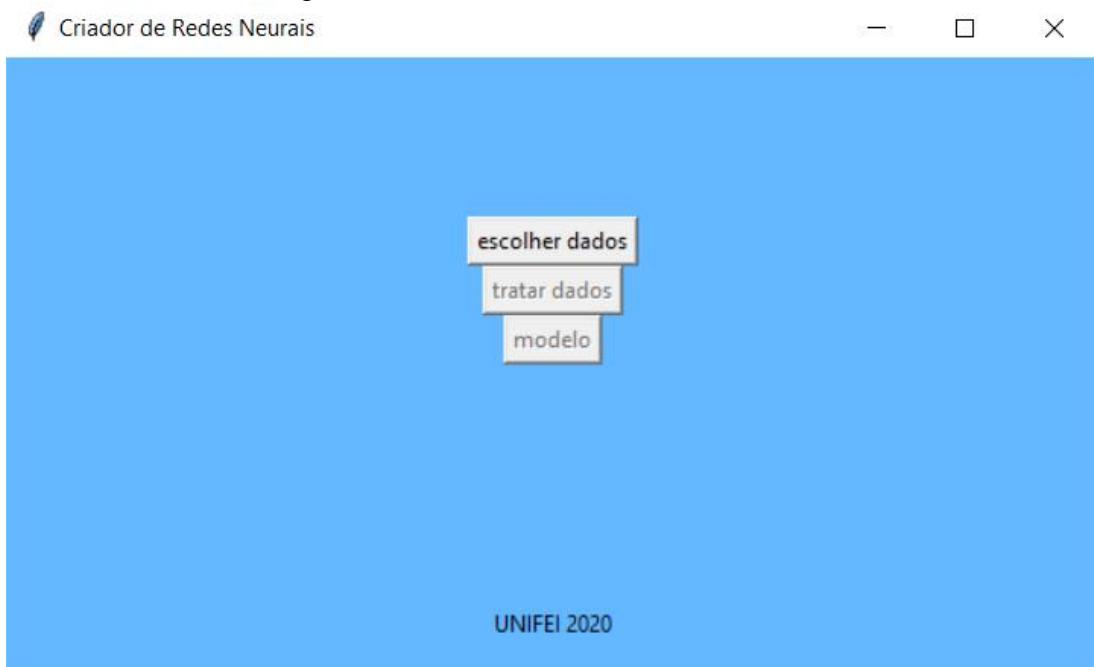
## 5 RESULTADOS OBTIDOS E ANÁLISE

### 5.1 O software desenvolvido

O processo de criação de uma Rede Neural Profunda é baseado em três estágios: a escolha dos dados, o tratamento destes dados e a construção do modelo. As atuais bibliotecas de código da área dão suporte apenas à uma construção através de linguagens de programação, como *Python* e *R*. Diante desta realidade, foi realizado o desenvolvimento de uma Interface Gráfica onde o usuário é capaz de construir seus modelos sem a necessidade de escrever códigos e também visualizar as camadas de maneira mais intuitiva, em blocos. A interface foi desenvolvida toda em *Python*, com o uso da biblioteca gráfica Tkinter.

A tela inicial do possui três botões, um para cada estágio do processo de criação, ilustrados na Figura 9.

Figura 9 – Tela Inicial da Interface Gráfica.

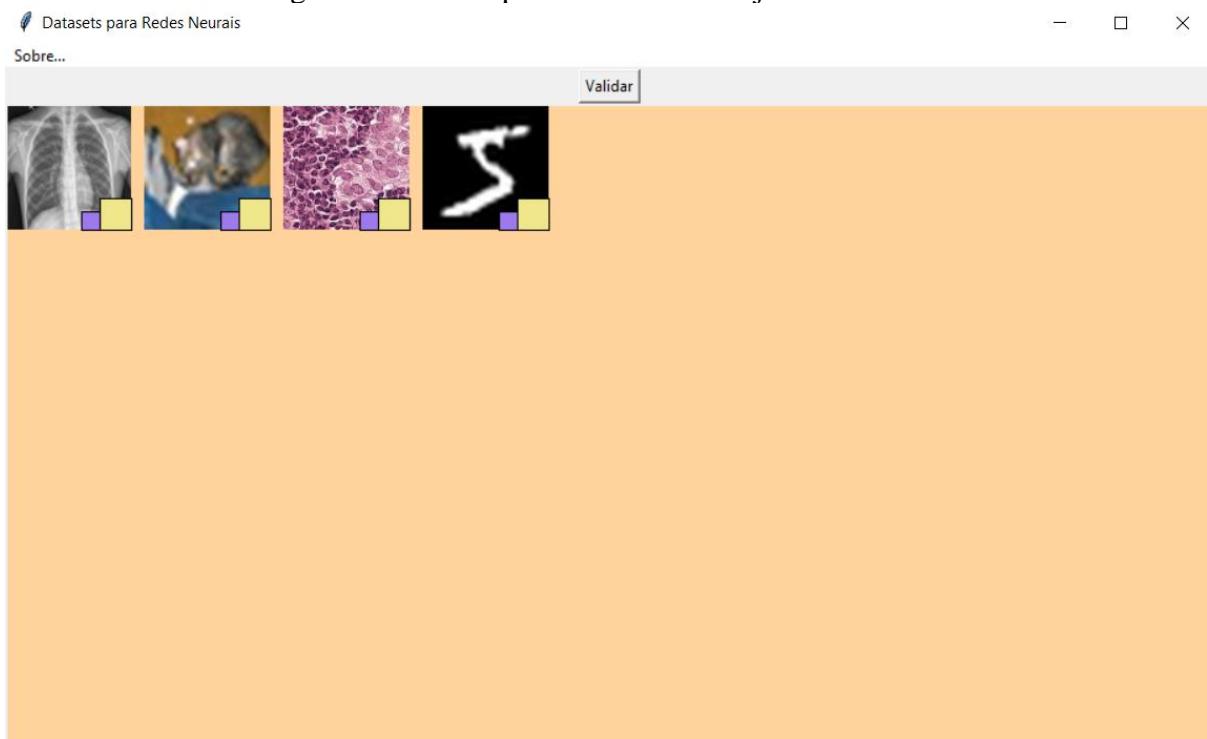


Fonte: o Autor.

#### 5.1.1 Escolha dos dados

A opção ‘escolher dados’ leva o usuário a um menu que mostra todos os conjuntos de dados disponíveis, como ilustra a Figura 10.

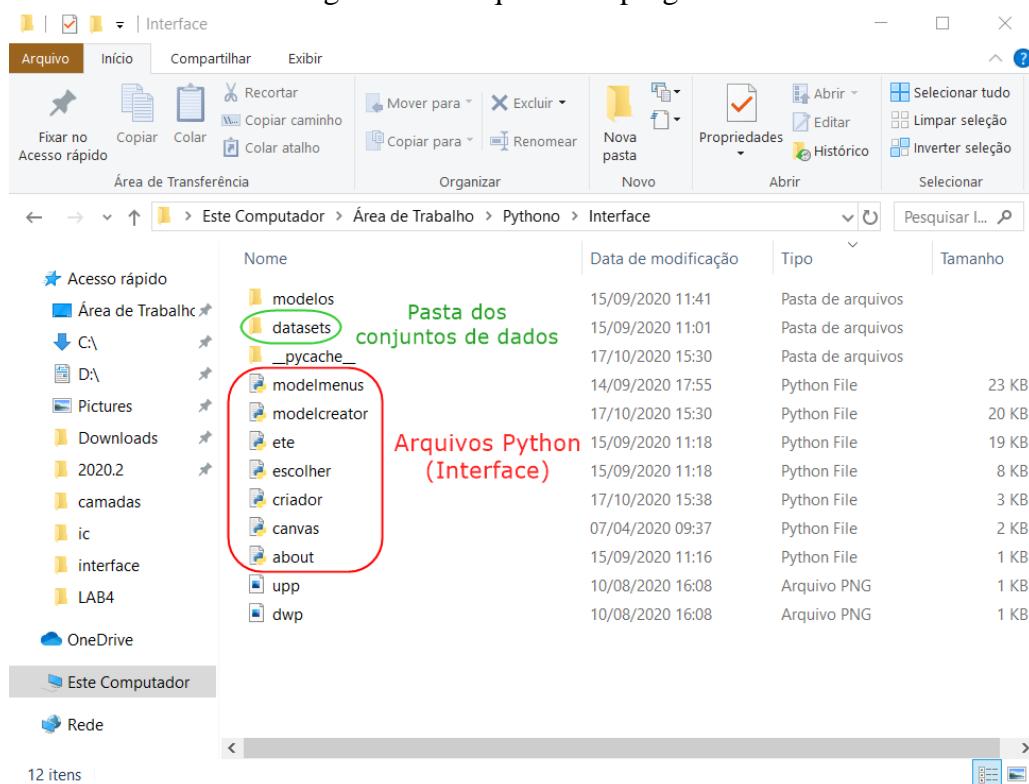
Figura 10 – Menu para escolha do conjunto de dados.



Fonte: o Autor.

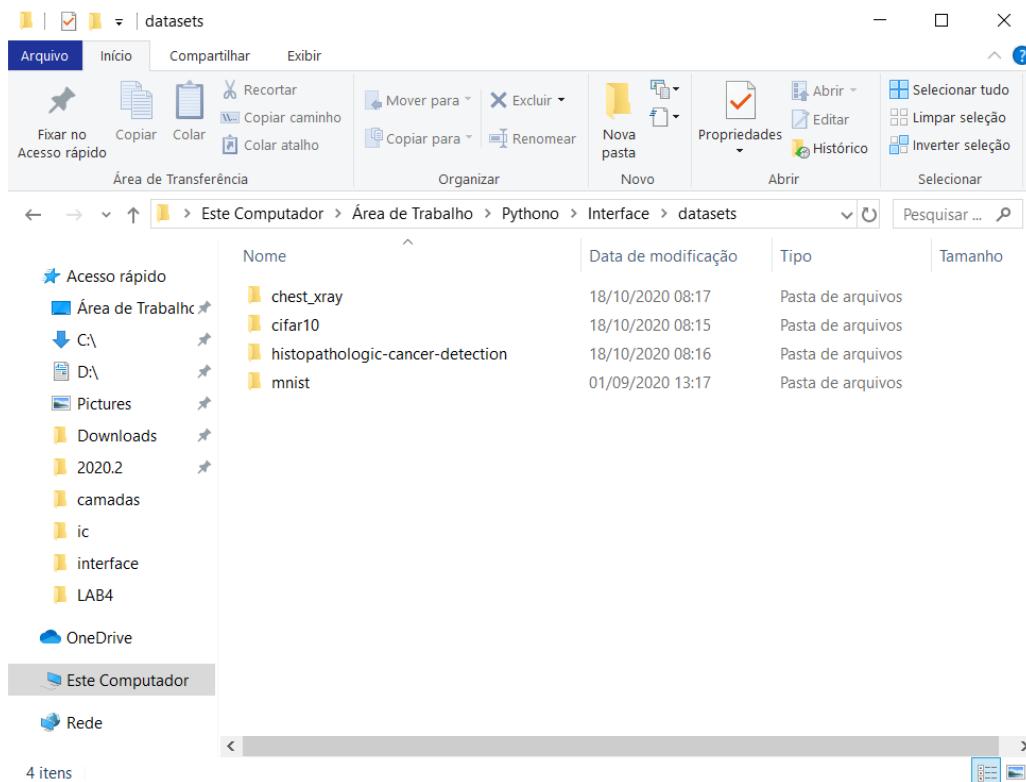
Para que um conjunto de dados esteja disponível neste menu, basta que as imagens se encontrem em uma pasta chamada ‘datasets’, no mesmo diretório do programa. As Figuras 11 e 12 ilustram a estrutura de arquivos do programa e conjunto de dados presentes na Figura 10, respectivamente.

Figura 11 – Arquivos do programa.



Fonte: o Autor.

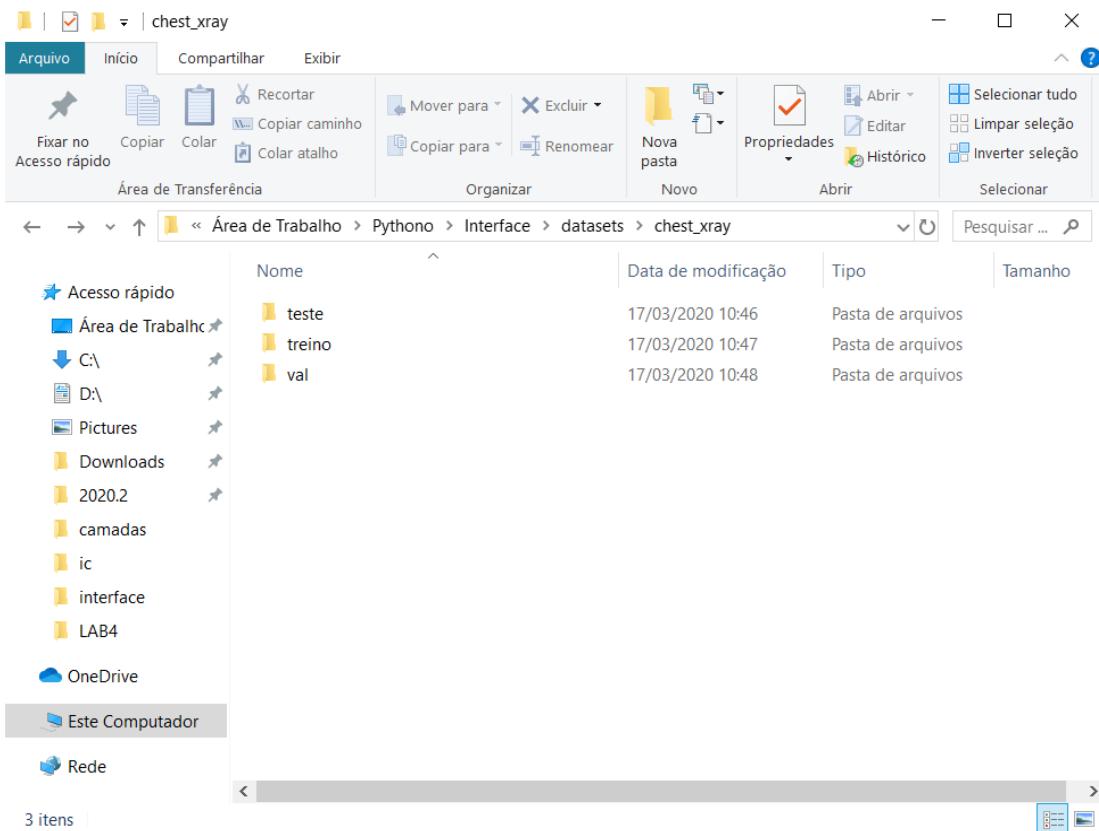
Figura 12 – Pasta onde se encontram os conjuntos de dados.



Fonte: o Autor.

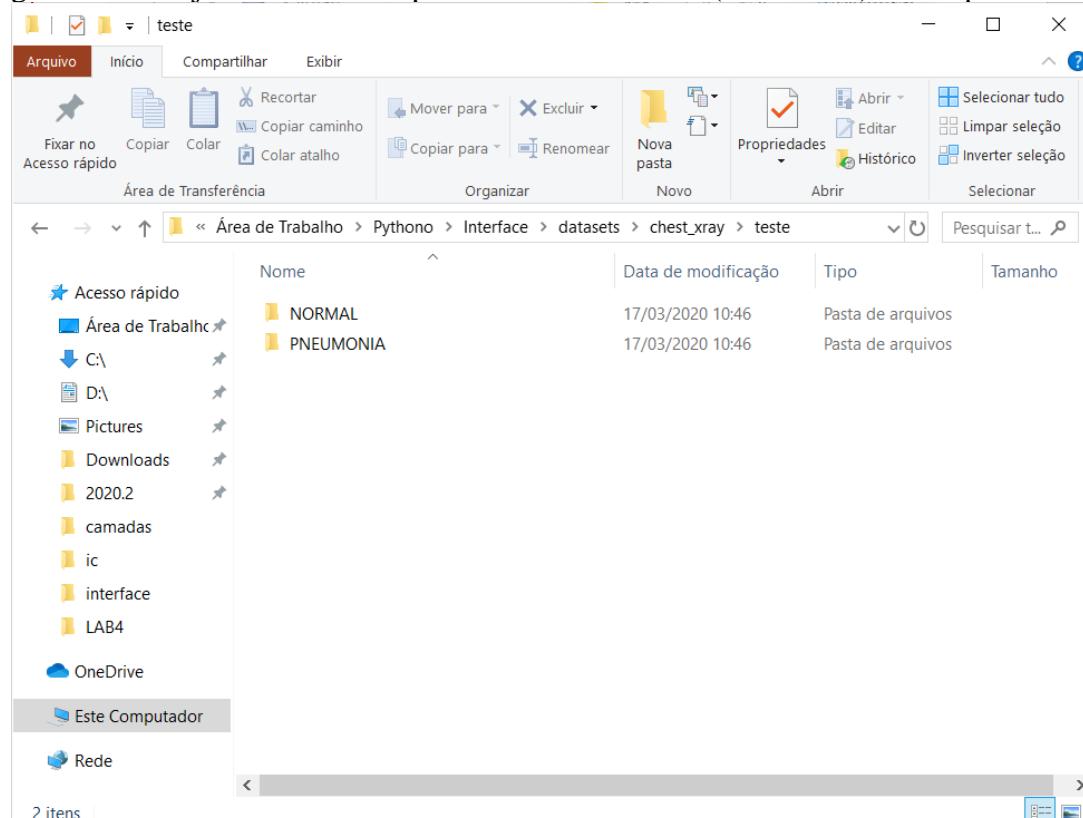
Para que o programa seja capaz de reconhecer as classificações atreladas a cada imagem, é necessário que haja ou a separação das diferentes classificações em diferentes pastas, ou um arquivo do tipo CSV contendo o nome dos arquivos de imagens e suas respectivas classificações. Por exemplo: o conjunto de imagens de raio-X (chest\_xray) está organizado em diferentes pastas para diferentes classificações (Figura 14), enquanto o de amostras microscópicas de tecidos possui um arquivo CSV com as classificações (Figura 16). O programa também reconhece caso o conjunto possua uma separação para o conjunto de treino, o conjunto de testes e o conjunto para validação dos treinos (Figuras 13 e 16). Caso não possua essa distinção, o programa fará automaticamente.

Figura 13 – Conjunto *chest\_xray* separado entre imagens para treino, teste e validação.



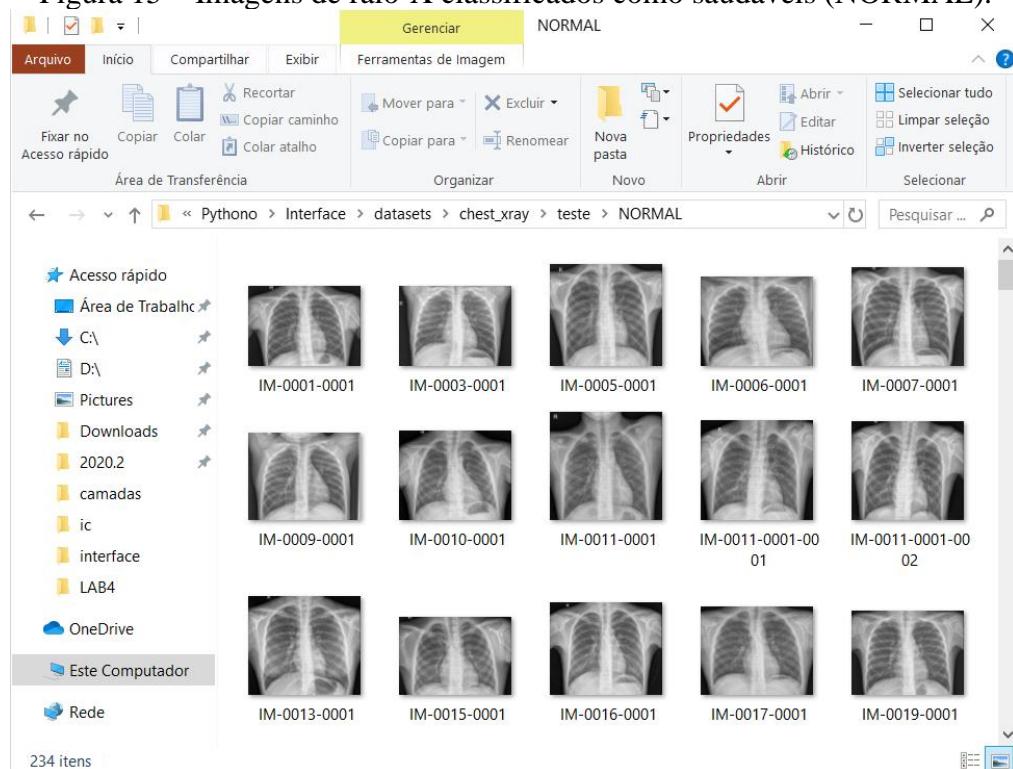
Fonte: o Autor.

Figura 14 – Conjunto de testes separado entre saudáveis (NORMAL) e com pneumonia.



Fonte: o Autor.

Figura 15 – Imagens de raio-X classificados como saudáveis (NORMAL).



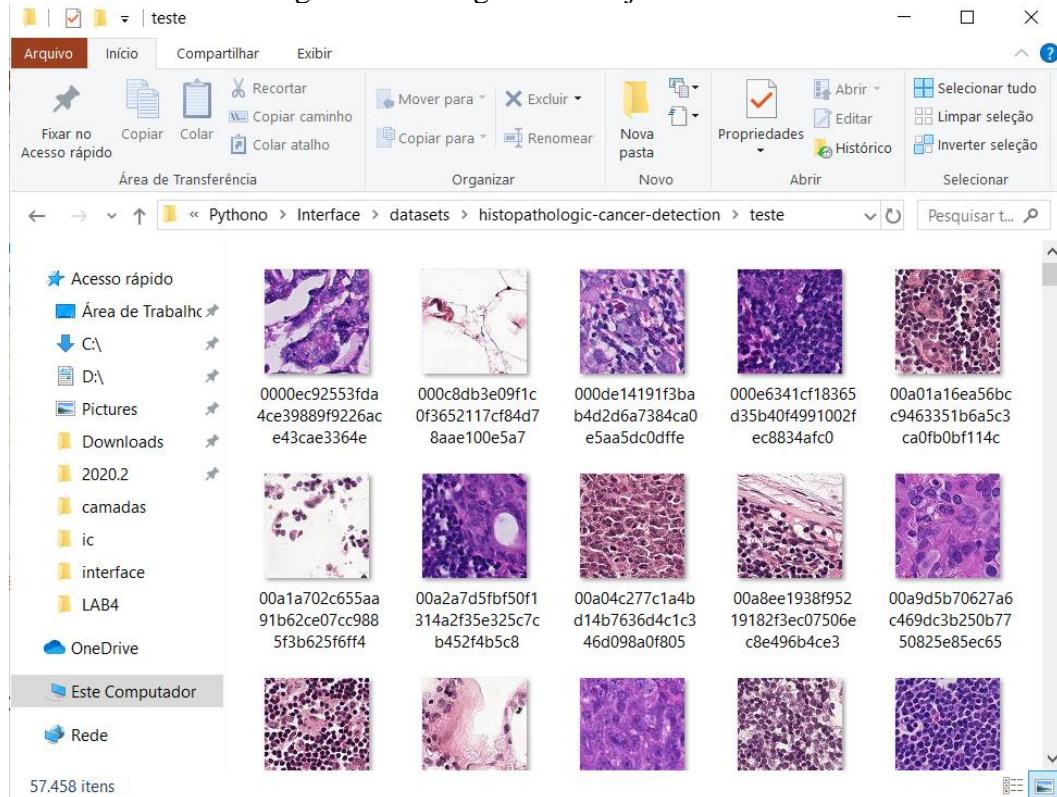
Fonte: o Autor.

Figura 16 – Conjunto *histopathologic-cancer-detection* separado entre imagens para treino e teste, juntamente com os arquivos CSV com as classificações.

Nome	Data de modificação	Tipo	Tamanho
teste	18/10/2020 08:16	Pasta de arquivos	
treino	18/10/2020 08:16	Pasta de arquivos	
teste_rotulos	30/03/2020 13:40	Arquivo de Valores	2.413 KB
treino_rotulos	30/03/2020 23:42	Arquivo de Valores	9.240 KB

Fonte: o Autor.

Figura 17 – Imagens do conjunto de teste.



Fonte: o Autor.

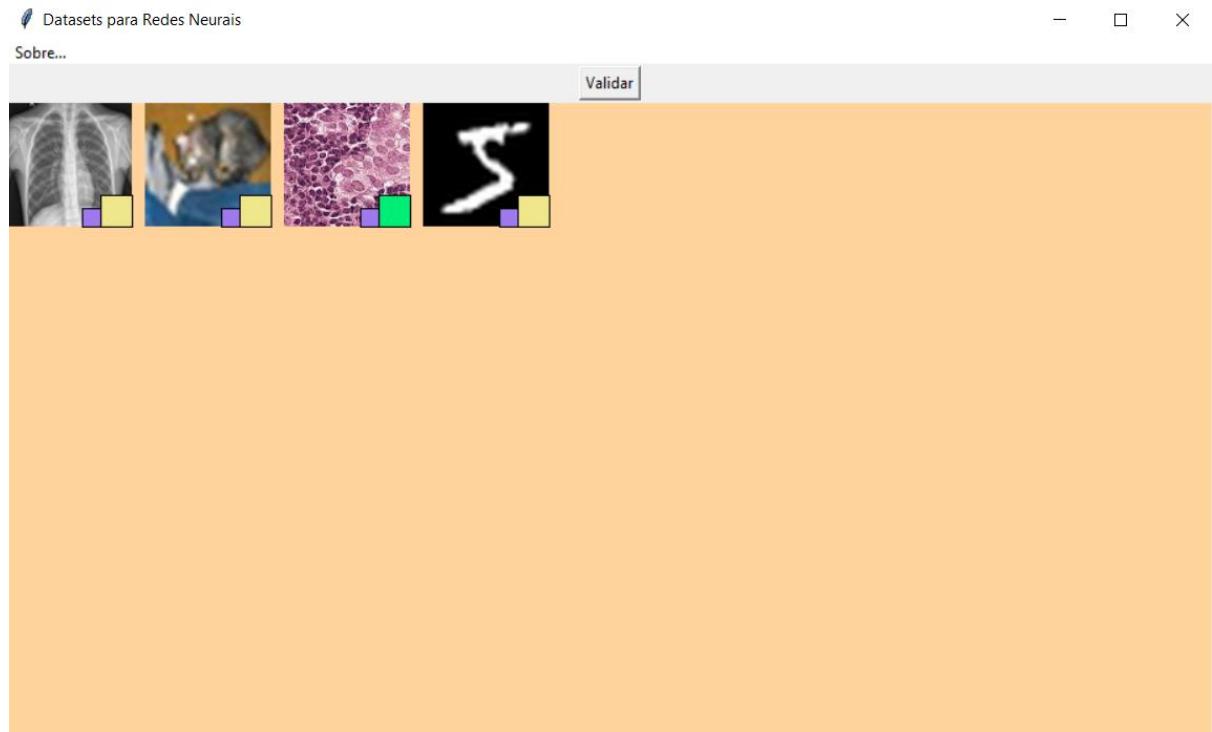
Figura 18 – Conteúdo do arquivo CSV relativo ao conjunto de treino.

A1	B	C	D	E	F
1 id,label					
2 f38a6374c348f90b587e046aac6079959adf3835,0					
3 c18f2d887b7ae4f6742ee445113fa1ef383ed77,1					
4 755db6279dae599ebb4d39a9123cce439965282d,0					
5 bc3f0c64fb968ff4a8bd33af6971ecae77c75e08,0					
6 068aba587a4950175d04c680d38943fd488d6a9d,0					
7 acfe80838488fae3c89bd21ade75be534e66be7,0					
8 a24ce148f6ffa7ef8eefb4efb12ebffe8dd700da,1					
9 7f6ccae485af121e0b6ee733022e226ee6b0c65f,1					
10 559e55a64c9ba828f700e948f6886f4cea919261,0					
11 8aaaa7400aa79d36c2440a4aa101c14256cda4,0					
12 a106469bbfd4cdc5a9da7ac0152927bf1b4a92d,0					
13 c3d660212bf2a11c994e0eadff13770a9927b731,1					
14 a1991e73a9b676faddd2bd47c39754b14d1eb923,0					
15 08566ce82d4406f464c9c2a3cd014704735db7a9,0					
16 94fa32b29cc1c00403176c0795ffa3cfaa0f20e,1					
17 f416de7491a31951f79b3cee75b002f4d1bf0162,0					
18 a1c001fb242c72d3066f15ac6eb059ea72d30ba,0					
19 0b820b71670c039dd0a51333d1c919f471a9e940,1					
20 730431efa2f79927156dcc4382819e9a6cc2c5bb,0					
21 d34af1e7500f2f3de41b0e6fdb2ed245d814590,1					
22 4b7a73f1fe1daf2fb7d2c0b83107f060b8d693,0					
23 5fc468030b7fdb5a0b656a45fa0dde5553dd9064,0					
24 4e1e69b64cdeb757178fc5b657b4e5ea07e53935,0					
25 464327050ef07bb927fbfb5c4e4dd5ebd4d3c09,1					
26 6961bdcc16f6c1d7db88fc6a7823178288c2a29e,1					
27 8e294a0fe82d36c9ae921ef2cc76ed0d0691b90f,0					

Fonte: o Autor.

Para escolher um conjunto de dados, basta o usuário selecioná-lo clicando no quadrado amarelo no canto inferior direito de cada imagem, que ficará verde, e em seguida clicar no botão ‘Validar’.

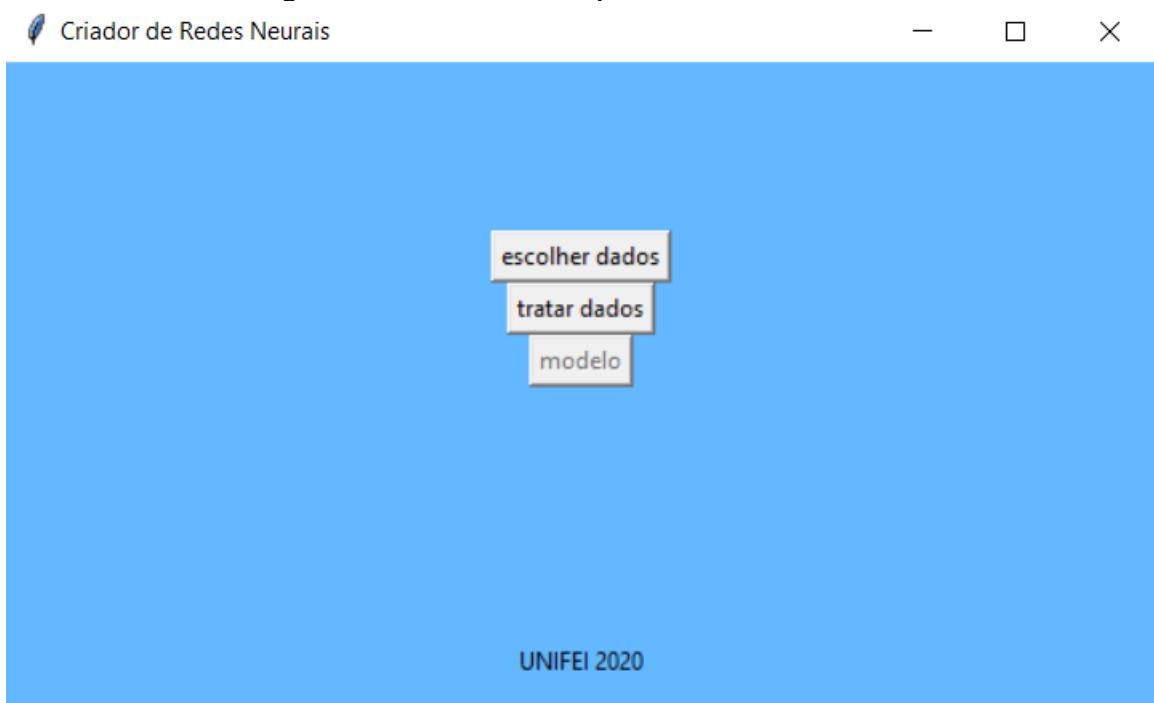
Figura 19 – Menu para escolha dos dados com um conjunto selecionado.



Fonte: o Autor.

Após clicar em ‘Validar’, o botão ‘tratar dados’ do menu inicial ficará disponível.

Figura 20 – Menu inicial após a escolha dos dados.



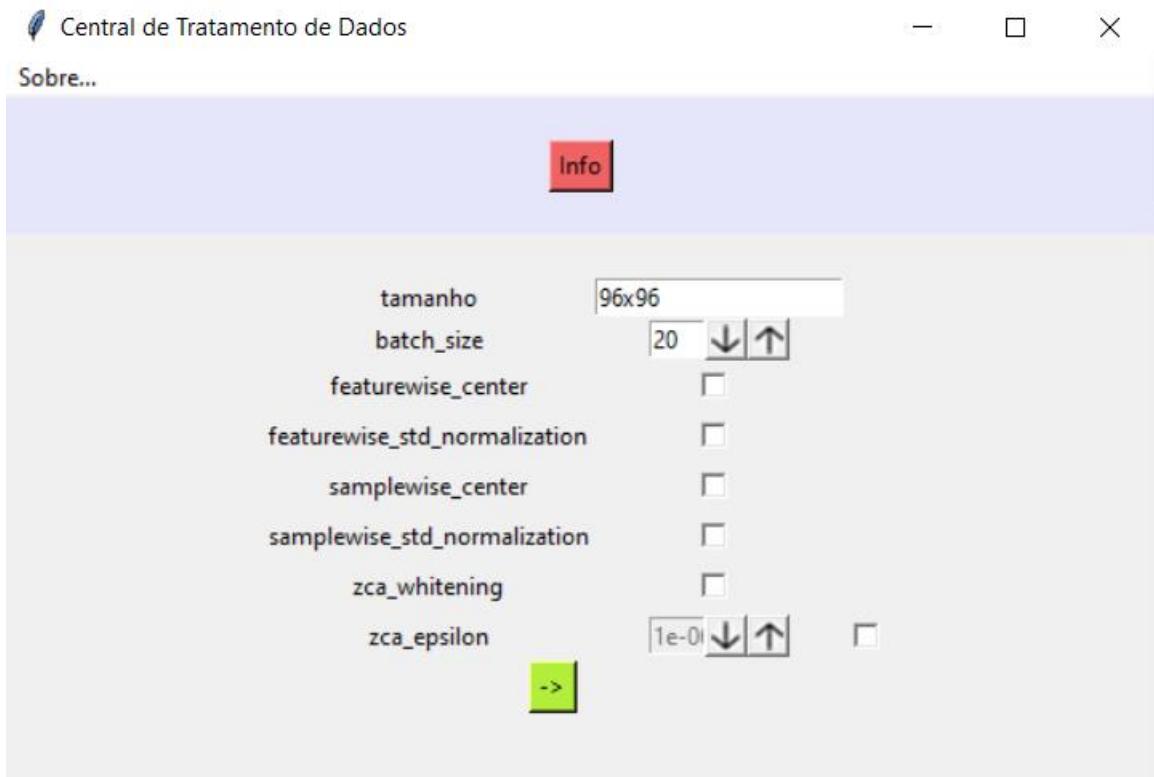
Fonte: o Autor.

### 5.1.2 Tratamento dos dados

O processo de tratamento dos dados consiste em realizar operações nas imagens antes de treiná-las na rede, como: normalizar os dados, que consiste em redefinir os valores dos pixels dentro do intervalo de 0 a 1 (em uma imagem RGB, por exemplo, esse intervalo é de 0 a 255), ou inserir outras versões das imagens no conjunto, como versões espelhadas, cortadas e esticadas. É muito útil para casos em que o conjunto de dados é pequeno e essas pequenas modificações não afetam o significado da imagem.

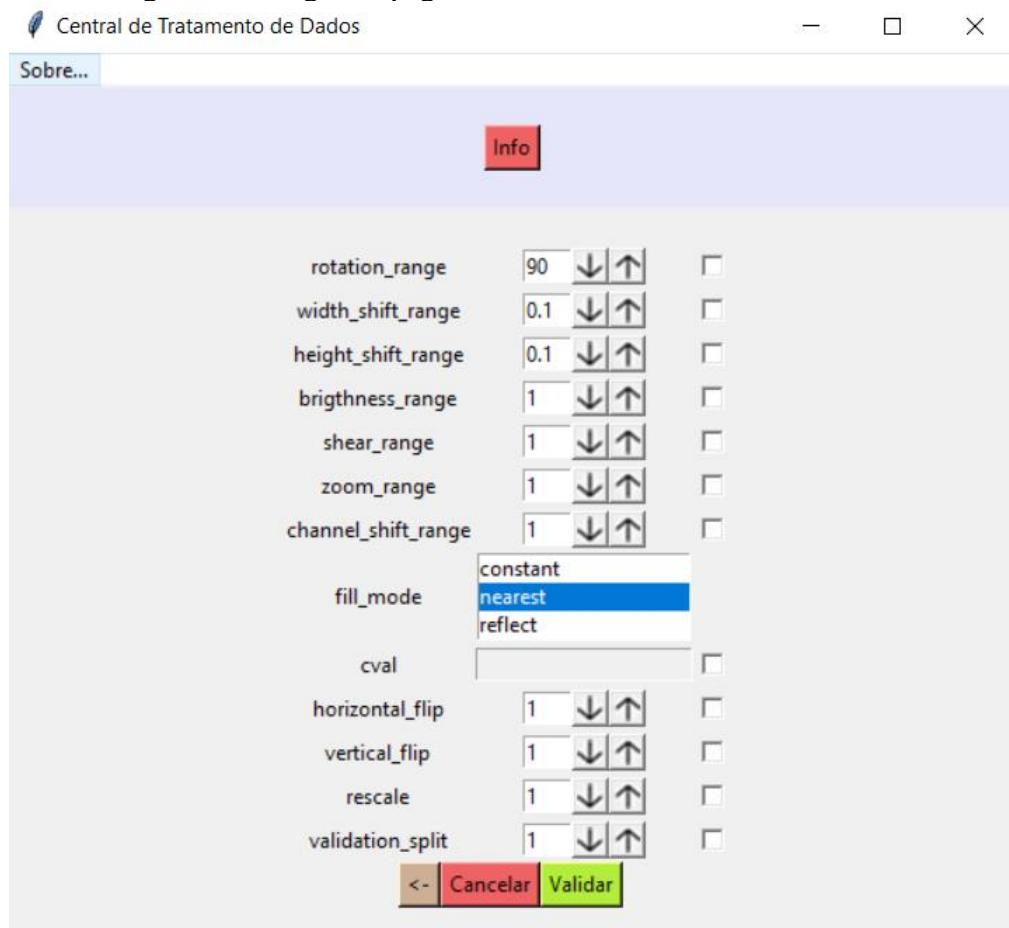
Ao clicar no botão ‘tratar dados’, um menu de duas páginas é aberto onde o usuário pode selecionar quais operações dentre as disponíveis ele quer realizar, além de poder definir o tamanho das imagens e dos lotes de treino.

Figura 21 – Primeira página do menu de tratamento de dados.



Fonte: o Autor.

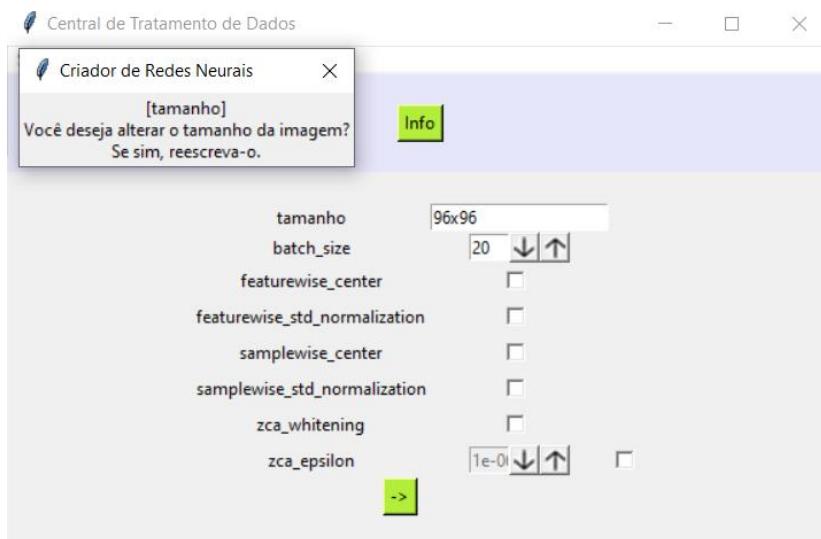
Figura 22 – Segunda página do menu de tratamento de dados.



Fonte: o Autor.

O botão ‘Info’ no canto superior habilita/desabilita a opção de informação a respeito das operações. Quando habilitado, uma janela com uma pequena descrição se abre sempre que o usuário clica em algum dos nomes das operações.

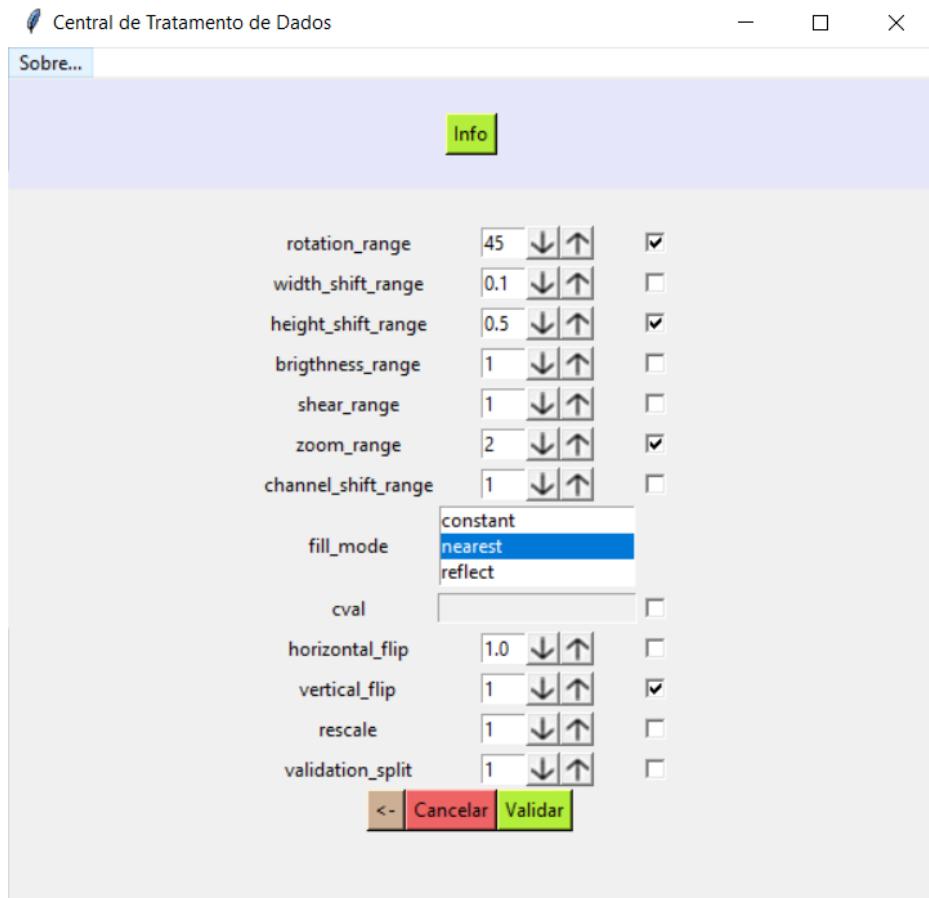
Figura 23 – Menu de tratamento de dados e janela aberta após clicar em ‘tamanho’ com o botão ‘Info’ habilitado.



Fonte: o Autor.

Neste menu o usuário escolhe as operações que quer incluir no tratamento das imagens selecionando a caixa branca à direita do nome da operação, que ficará marcada. Há três opções que não possuem a caixa, e são configurações obrigatórias do processo. As três opções são: *tamanho*, que configura o tamanho da imagem; *batch\_size*, que configura o tamanho do lote de treino, e *fill\_mode*, que diz como serão preenchidos os espaços extras da imagem que podem surgir ao realizar outras operações, como por exemplo uma compressão da altura da imagem (*height\_shift\_range*), que deixará espaços vazios nos bordas superiores e inferiores.

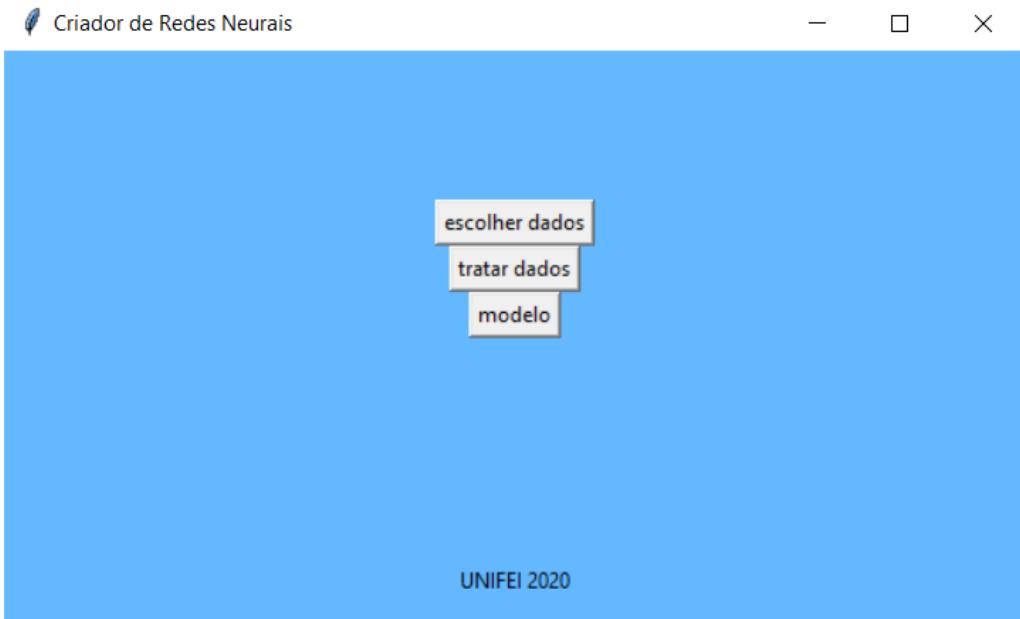
Figura 24 – Menu de tratamento de dados com algumas opções selecionadas.



Fonte: o Autor.

Para finalizar a configuração da fase de tratamento de dados, basta o usuário clicar no botão ‘Validar’, que disponibilizará o botão ‘modelo’ do menu inicial.

Figura 25 – Menu inicial após configurar o tratamento dos dados.



Fonte: o Autor.

### 5.1.3 Criação do Modelo

Ao clicar no botão ‘modelo’, é aberto um menu para a criação da arquitetura da rede, onde o usuário escolherá as camadas que deseja adicionar ao seu modelo. As camadas são adicionadas na mesma ordem que aparecerão no modelo, então deve-se adicionar as primeiras camadas primeiro.

Figura 26 – Menu para criação da arquitetura da rede.

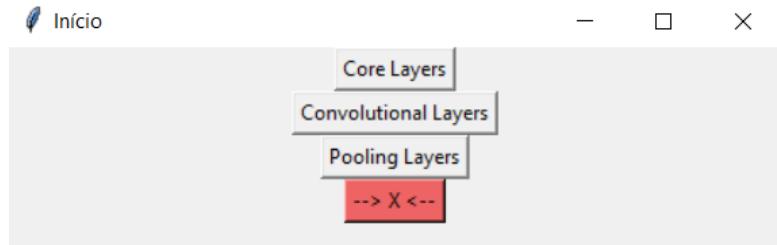


Fonte: o Autor.

O botão ‘+ Camada’ abre um menu onde o usuário pode escolher dentre as camadas disponíveis, que estão separadas em três grupos diferentes: Core Layers (*camadas núcleo*, em inglês), que agrupa diversos tipos de operações básicas, Convolutional Layers (*camadas*

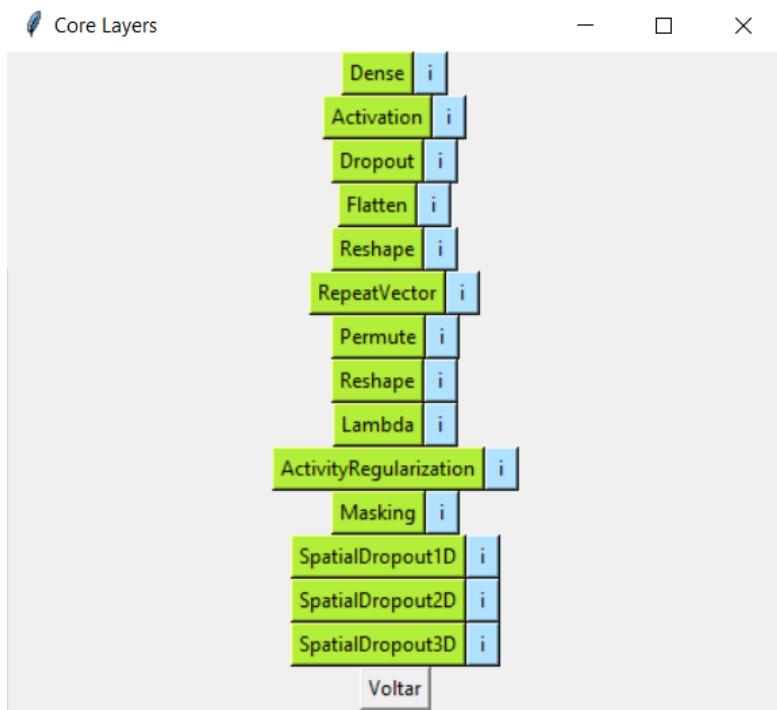
convolucionais, em inglês), que agrupa diversos tipos de operações convolutivas, e Pooling Layers (*camadas minadoras*, em inglês), que agrupa operações de pooling (Figura X21).

Figura 27 – Menu para a escolha das camadas.



Fonte: o Autor.

Figura 28 – Menu das *Core Layers*.



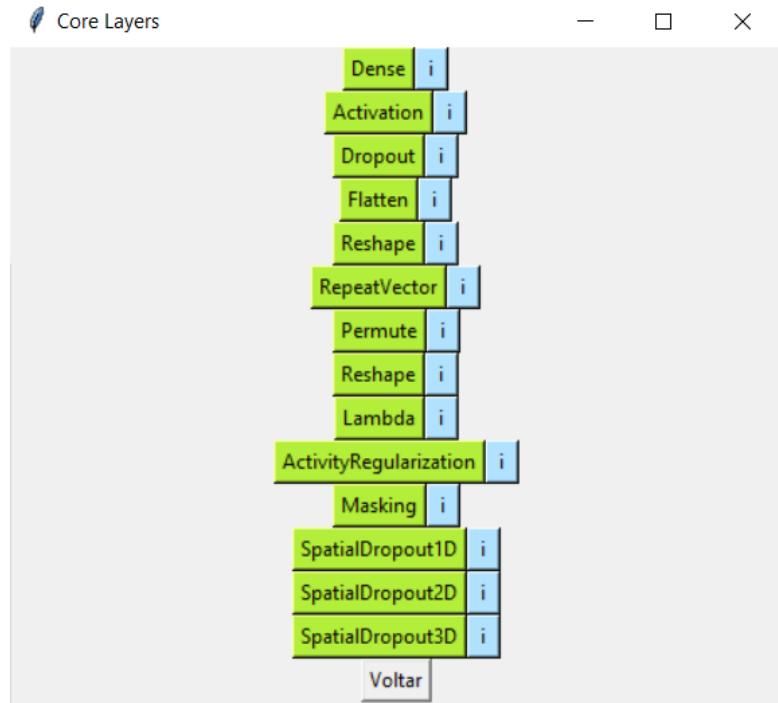
Fonte: o Autor.

Figura 29 – Menu das *Pooling Layers*.



Fonte: o Autor.

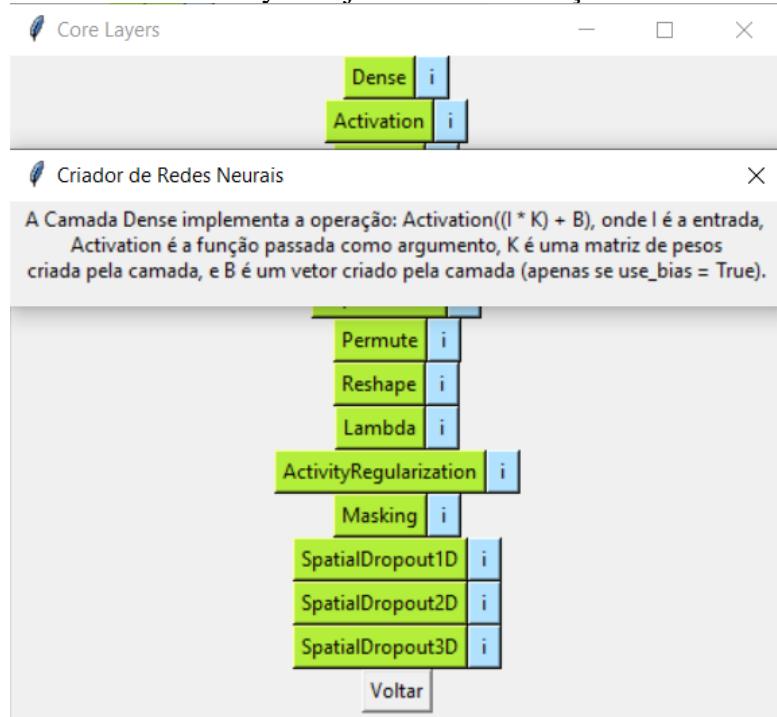
Figura 30 – Menu das *Convolutional Layers*.



Fonte: o Autor.

Cada camada dentro do menu é acompanhada de um botão ‘i’, que abre uma janela que contém uma breve explicação sobre a camada.

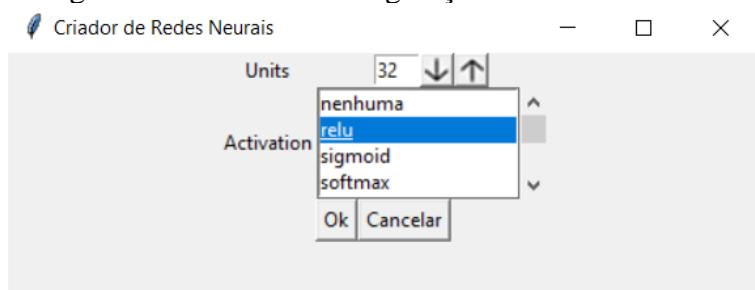
Figura 31 – Menu das Core Layers e janela de informações sobre a camada Dense.



Fonte: o Autor.

Ao clicar em uma camada, será aberto um menu onde o usuário irá configurar os parâmetros daquela camada. Cada camada possui seus parâmetros específicos.

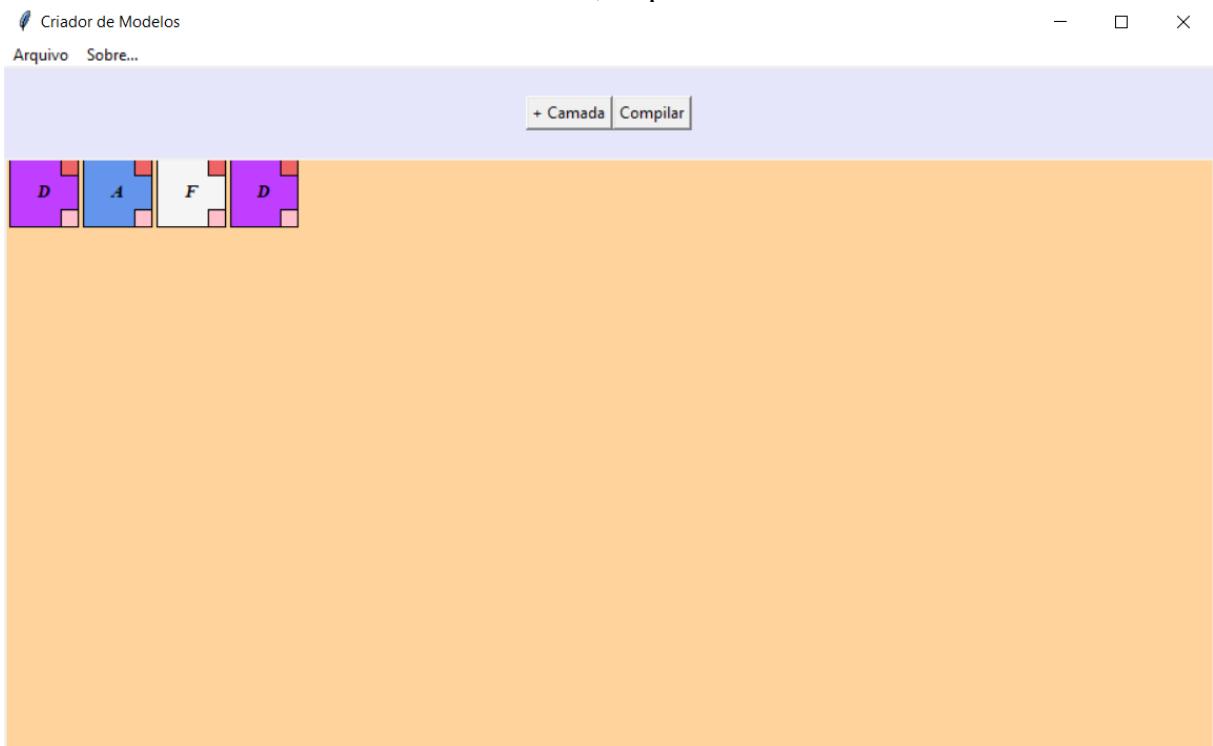
Figura 32 – Menu de configuração da camada *Dense*.



Fonte: o Autor.

Após clicar no botão ‘Ok’, a camada será adicionada ao menu de criação da arquitetura e aparecerá como um quadrado com a inicial da camada. Dentro do quadrado que representa a camada, também há outros dois itens iterativos: um quadrado vermelho no canto superior direito, que exclui a camada da arquitetura, e um quadrado salmão no canto inferior direito, que reabre o menu de configurações daquela camada.

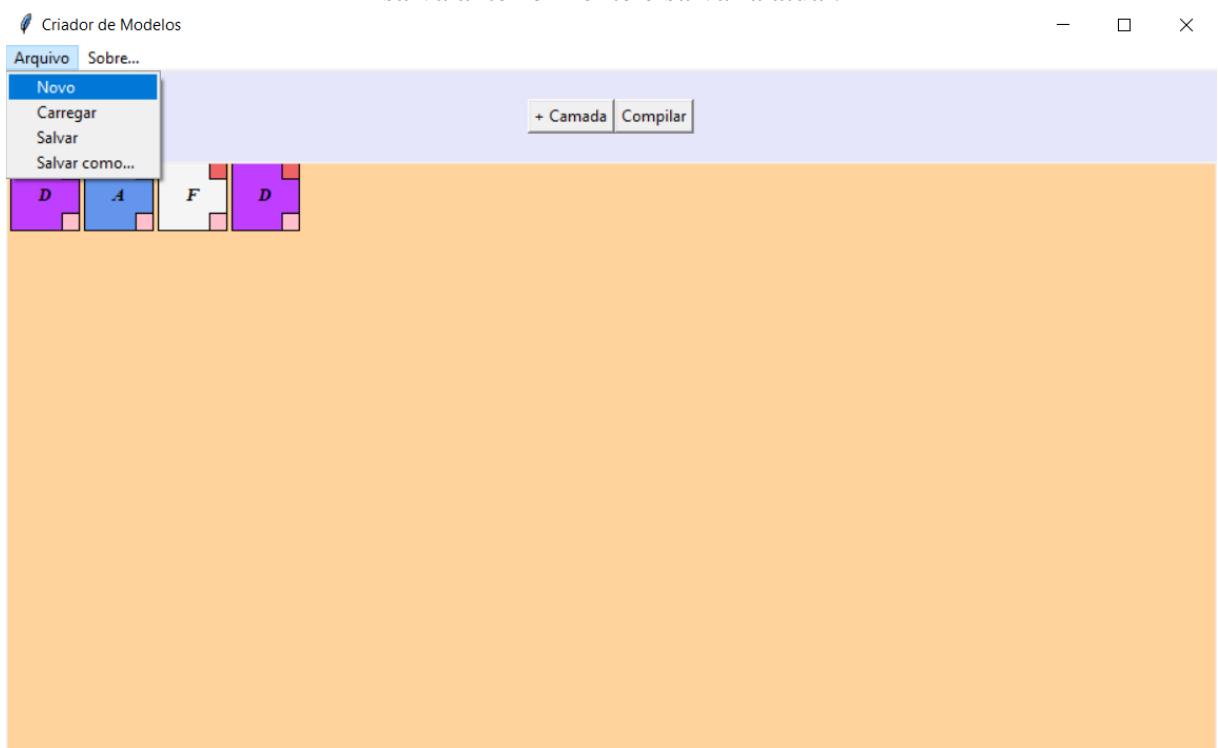
Figura 33 – Menu de criação da arquitetura após a inserção das camadas *Dense*, *Activation*, *Flatten* e *Dense*, respectivamente.



Fonte: o Autor.

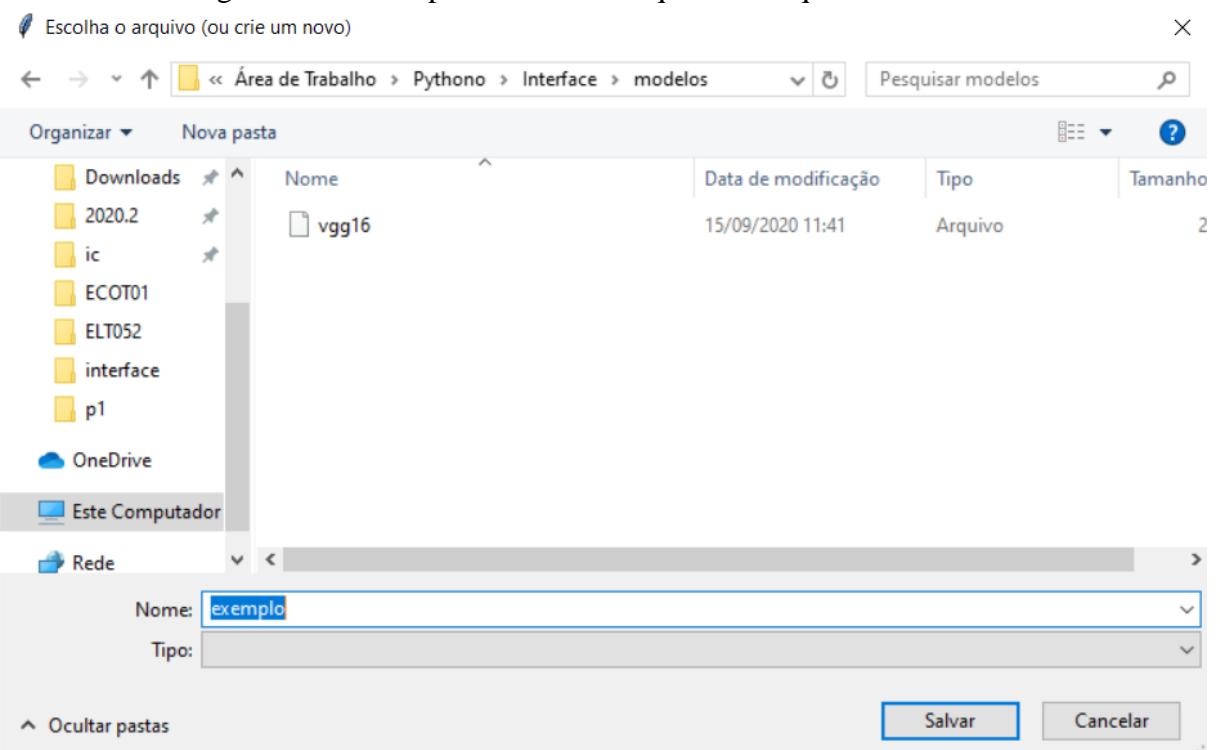
O programa também dá a opção de salvar o modelo, carregar um outro salvo previamente e iniciar a construção de uma nova arquitetura, reiniciando o menu. A opção ‘Salvar como’ abre uma janela para definição do nome do arquivo, enquanto a ‘Salvar’ atualizará o arquivo já salvo com as novas informações. Caso não exista uma versão anterior do arquivo, o programa abrirá a mesma janela da opção ‘Salvar como’. O menu para salvamento do arquivo inicia na pasta ‘modelos’ que está entre os arquivos do programa, porém é possível salvá-lo em qualquer local do computador.

Figura 34 – Menu de criação da arquitetura e opções para começar uma nova, carregar uma salva anteriormente e salvar a atual.



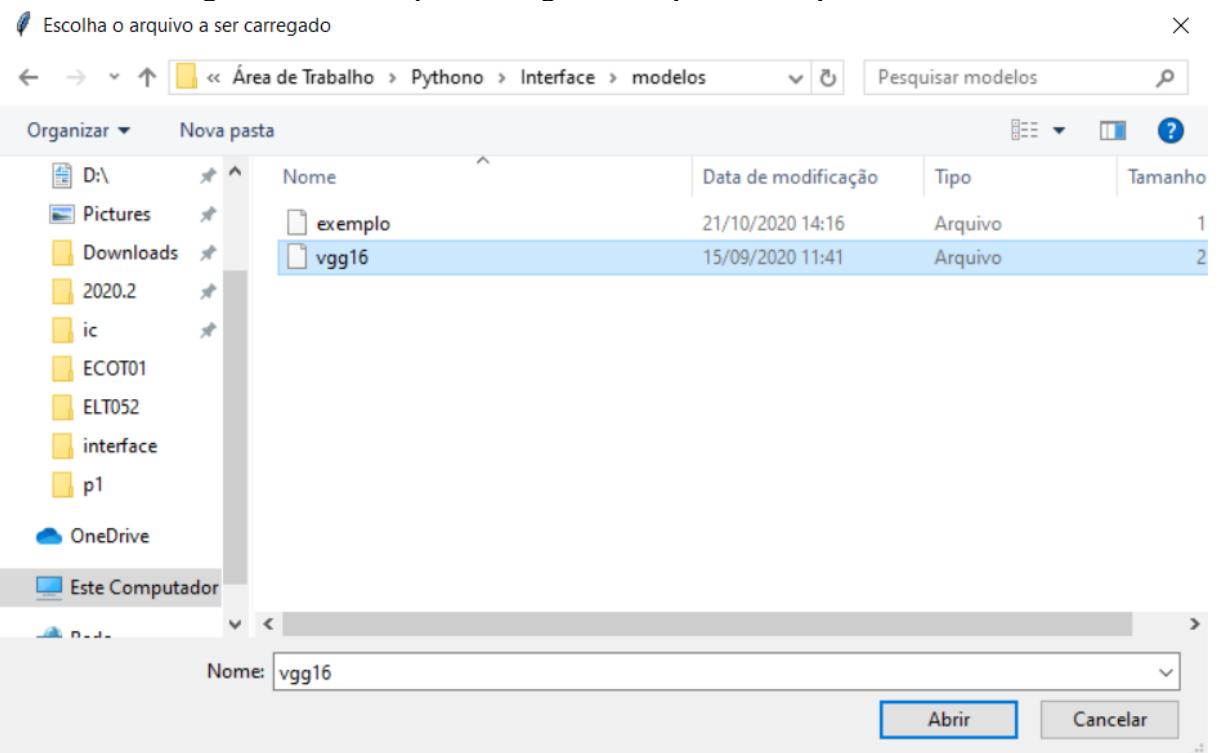
Fonte: o Autor.

Figura 35 – Menu para salvar um arquivo de arquitetura de rede.



Fonte: o Autor.

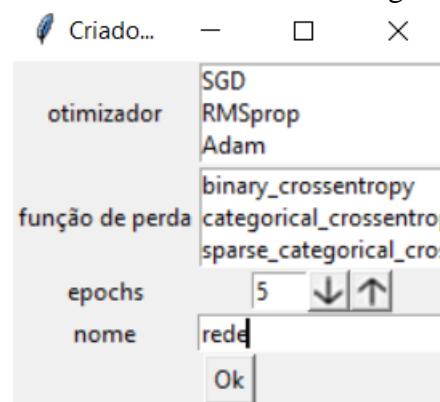
Figura 36 – Menu para carregar um arquivo de arquitetura de rede.



Fonte: o Autor.

Após finalizar a adição das camadas, o usuário deve clicar no botão ‘Compilar’ para ser direcionado ao último menu de configurações, onde ele escolherá o otimizador, a função de perda, o número de épocas e o nome do arquivo python gerado. O código gerado estará junto dos arquivos do sistema, e basta interpretá-lo para treinar a Rede Neural Profunda.

Figura 37 – Último menu de configurações.



Fonte: o Autor.

Figura 38 - Código Python gerado a partir do exemplo.

```

1 import tensorflow as tf
2 import numpy as np
3 from keras import layers
4 from keras import models
5 from keras.preprocessing import image
6 from pathlib import Path
7 import csv
8 from PIL import Image
9 from keras.utils import to_categorical
10 path='C:/Users/binho/Desktop/Pythono/Interface/datasets/histopathologic-cancer-detection'
11 path = Path(path)
12 labels = 'inferred'
13 itemes = list(path.rglob('*.*'))
14 features = len(itemes)
15 input_shape = (96, 96, 3)
16 generator = image.ImageDataGenerator(featurewise_center=True, featurewise_std_normalization=True,
17 rotation_range=45, height_shift_range=0.5, zoom_range=2, fill_mode='nearest', vertical_flip=1)
18 batch_size = 20
19 input_size = (96,96)
20 timesteps = int(features/batch_size)
21 for x in range(0,len(itemes)):
22     itemes[x] = Image.open(itemes[x])
23     itemes[x] = itemes[x].resize(size=input_size)
24     itemes[x] = np.asarray(itemes[x])
25 itemes = np.asarray(itemes)
26 y = itemes.shape
27 if(len(y)==3):
28     itemes = np.reshape(itemes, (y[0],y[1],y[2],1))
29 generator.fit(itemes)
30 traingen = generator.flow_from_directory(path, target_size=input_size,batch_size=batch_size)
31 model = models.Sequential()
32 model.add(layers.Dense(input_shape=input_shape,units=32,activation='nenhuma'))
33 model.add(layers.Activation(activation='nenhuma'))
34 model.add(layers.Flatten())
35 model.add(layers.Dense(units=32,activation='nenhuma'))
36 model.compile(optimizer='SGD',loss='binary_crossentropy',metrics=['accuracy'])
37 history = model.fit(traingen, steps_per_epoch = timesteps,epochs = 5)

```

Fonte: o Autor.

## 5.2 Exemplos

Nesta seção serão desenvolvidos alguns exemplos para elucidar o funcionamento das camadas, funções de perda e otimizadores.

### 5.2.1 Duas tonalidades

Como visto na seção anterior, existem uma função de perda (*entropia cruzada binária*) voltada para problemas de classificação binária. Neste exemplo uma rede será submetida a imagens classificadas entre tons azulados e tons avermelhados utilizando esta função de perda.

### 5.2.1.1 As imagens

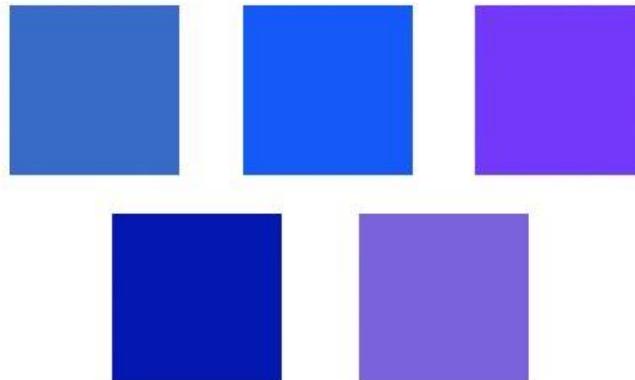
Figura 39 – Código usado para gerar mil imagens azuis e mil imagens vermelhas.

```

1 import numpy as np
2 from PIL import Image
3 from pathlib import Path
4 ptr = Path('./tonalidades/treino/vermelho') #caminho: Interface/datasets/tonalidades/treino/vermelho
5 ptb = Path('./tonalidades/treino/azul') #caminho: Interface/datasets/tonalidades/treino/azul
6 z = np.zeros((128,128), 'float32') #matriz de zeros, de tamanho 128x128
7 for n in range(1000): #1000 iterações
8     rgbR = np.zeros((128,128,3), 'uint8') #matriz de zeros, de tamanho 128x128x3
9     rgbR[:, :, 0] = (z+np.random.rand()*0.4+0.6)*256 #canal R = 0.6 + x*0.4, 0 <= x <= 1
10    rgbR[:, :, 1] = (z+np.random.rand()*0.5)*256 #canal G = x*0.5, 0 <= x <= 1
11    rgbR[:, :, 2] = (z+np.random.rand()*0.5)*256 #canal B = x*0.5, 0 <= x <= 1
12    img_vermelha = Image.fromarray(rgbR) #gerar imagem
13    img_vermelha.save(ptr/('imgred'+str(n)+'.png')) #salvar na pasta 'vermelho'
14
15    rgbB = np.zeros((128,128,3), 'uint8')
16    rgbB[:, :, 0] = (z+np.random.rand()*0.5)*256 #canal R = x*0.5, 0 <= x <= 1
17    rgbB[:, :, 1] = (z+np.random.rand()*0.5)*256 #canal G = x*0.5, 0 <= x <= 1
18    rgbB[:, :, 2] = (z+np.random.rand()*0.4+0.6)*256 #canal B = 0.6 + x*0.4, 0 <= x <= 1
19    img_azul = Image.fromarray(rgbB) #gerar imagem
20    img_azul.save(ptb/('imgblue'+str(n)+'.png')) #salvar na pasta 'azul'
```

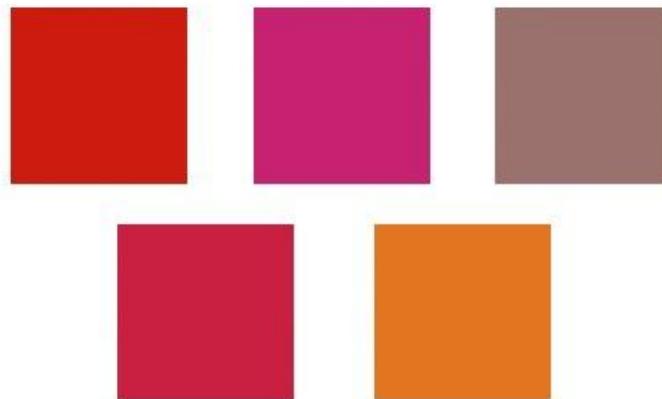
Fonte: o Autor.

Figura 40 – Algumas das imagens de tons azulados geradas.



Fonte: o Autor.

Figura 41 – Algumas das imagens de tons avermelhados geradas.



Fonte: o Autor.

### 5.2.1.2 A rede

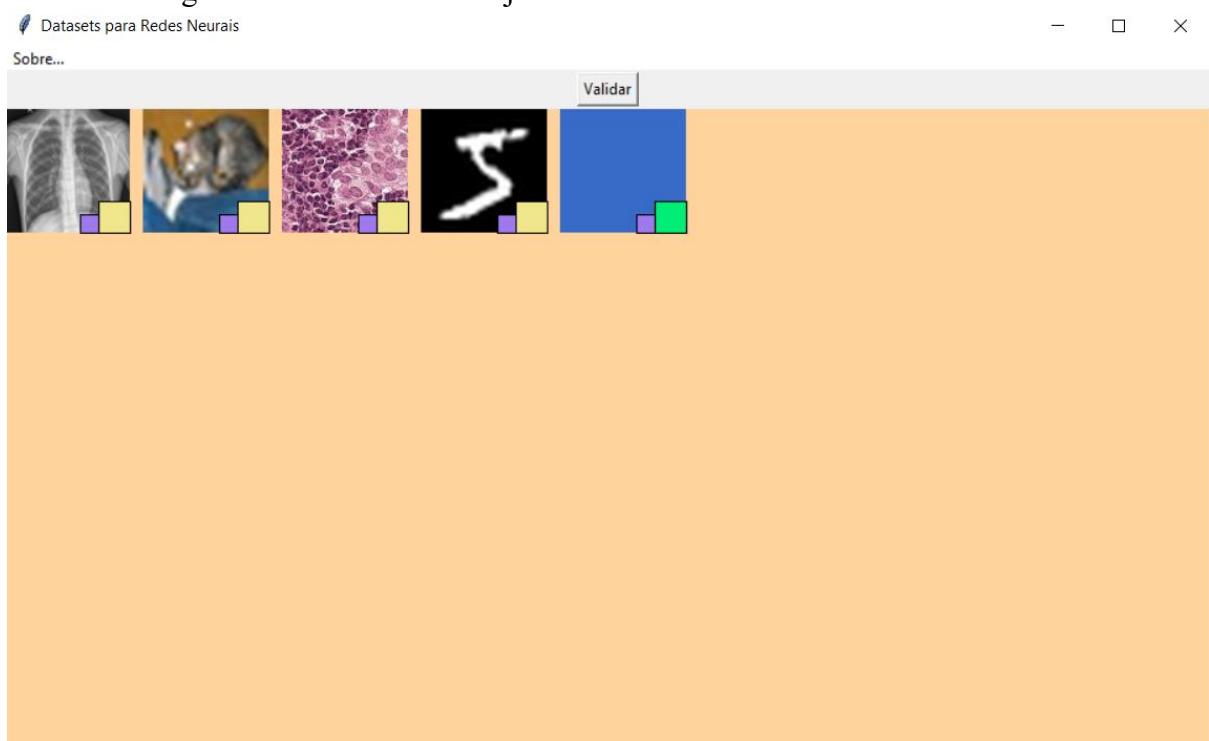
Neste problema estamos lidando com uma classificação binária, o que já nos indica que o uso da função de perda de entropia cruzada binária pode ser uma boa opção. Um otimizador simples pode ser usado, como o SGD.

A informação que buscamos está presente em toda a imagem, então camadas densas são uma boa opção para a arquitetura. Uma camada retificadora pode ser inserida a fim de transformar a operação da camada densa em uma multiplicação termo a termo. Apesar de as camadas de ativação existirem sozinhas, existe nas camadas densas da biblioteca Keras a opção de aplicar uma função de ativação ao final da operação. A função de ativação *relu*, que levará os valores negativos para zero, é uma boa opção para camadas intermediárias da arquitetura, enquanto nas camadas finais é mais indicada a *sigmoid*, que buscará classificar os valores entre dois extremos do intervalo ([Seção 3.1.2.1.1](#)).

As figuras a seguir ilustram o processo de construção da rede utilizando o software desenvolvido. Para a etapa de tratamento de dados, apenas a opção *featurewise\_std\_normalization* será selecionada, que divide os valores da imagem pela média dos valores de todo o conjunto. A arquitetura usada será:

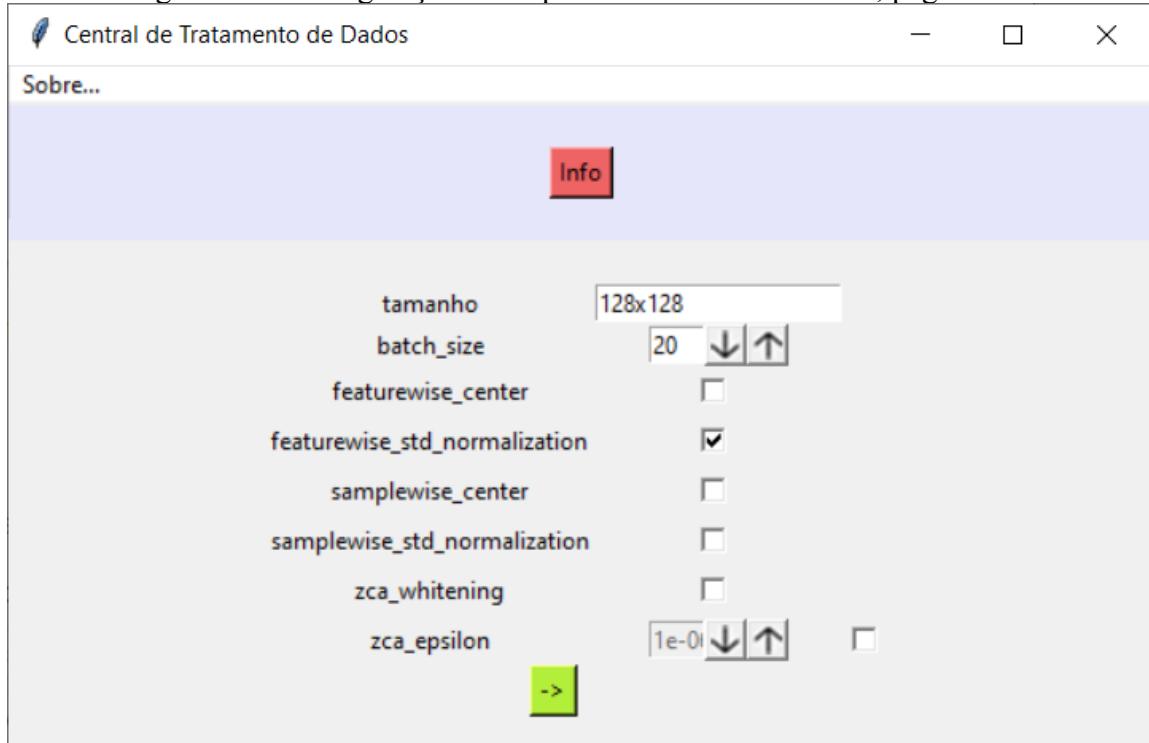
Retificadora → Densa(512) → Ativação(*relu*) → Densa(2) → Ativação(*sigmoid*)

Figura 42 – Escolha do conjunto de dados das tonalidades no software.



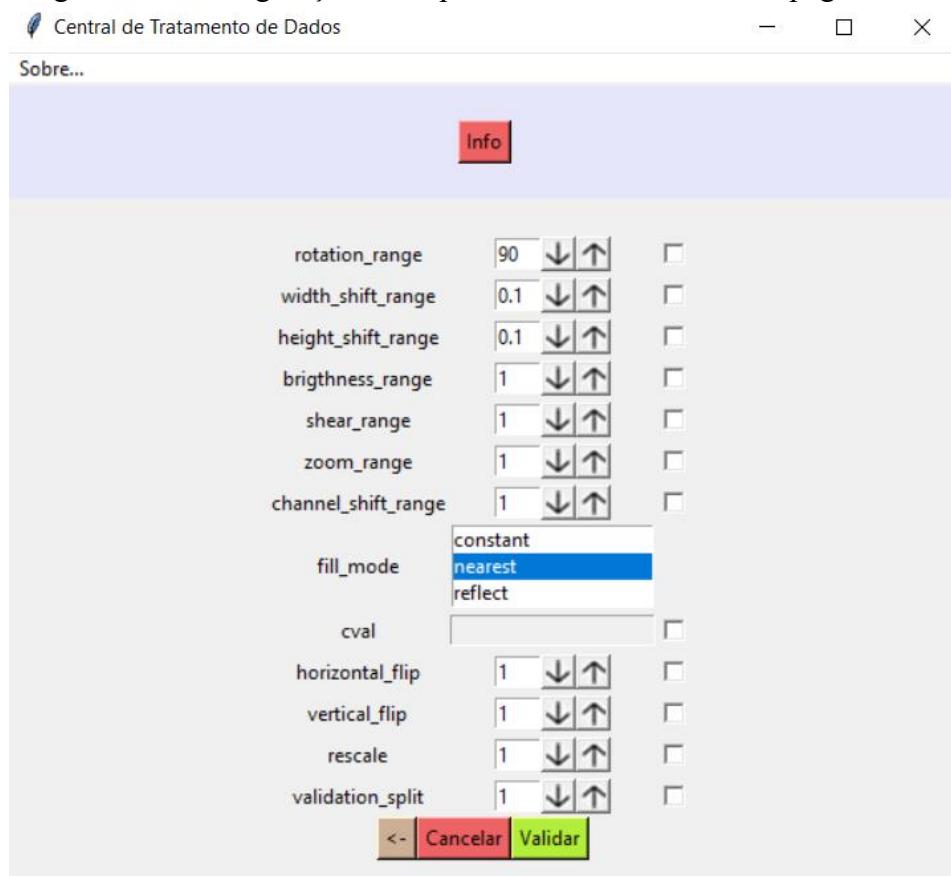
Fonte: o Autor.

Figura 43 – Configuração da etapa de tratamento de dados, página um.



Fonte: o Autor.

Figura 44 – Configuração da etapa de tratamento de dados, página dois.



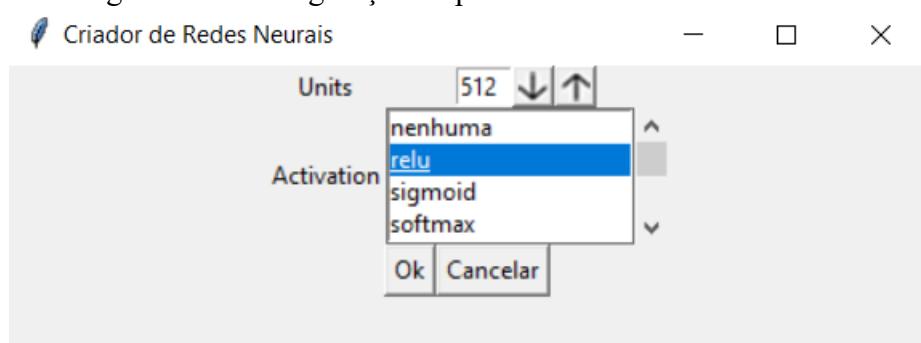
Fonte: o Autor.

Figura 45 – Etapa da configuração da arquitetura da rede.



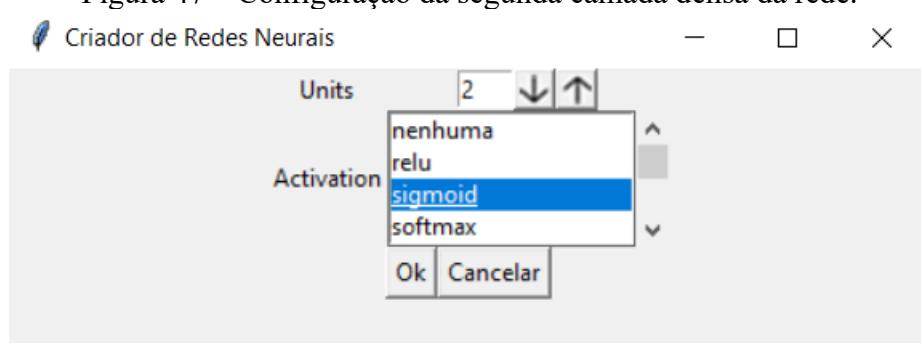
Fonte: o Autor.

Figura 46 – Configuração da primeira camada densa da rede.



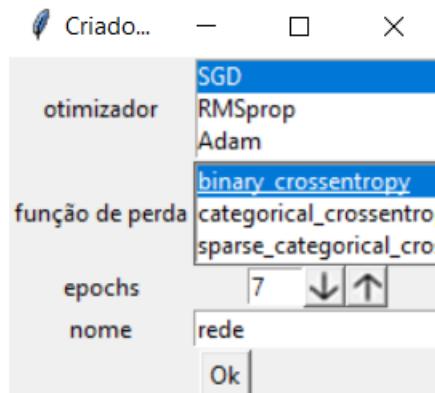
Fonte: o Autor.

Figura 47 – Configuração da segunda camada densa da rede.



Fonte: o Autor.

Figura 48 – Configuração final da etapa de arquitetura da rede.



Fonte: o Autor.

Figura 49 – Código final gerado (rede.py).

```

1 import tensorflow as tf
2 import numpy as np
3 from keras import layers
4 from keras import models
5 from keras.preprocessing import image
6 from pathlib import Path
7 import csv
8 from PIL import Image
9 from keras.utils import to_categorical
10 #seção da escolha dos dados
11 path='C:/Users/binho/Desktop/Pythono/Interface/datasets/tonalidades/treino'
12 path = Path(path)
13 labels = 'inferred'
14 itemes = list(path.rglob('*.*'))
15 features = len(itemes)
16 input_shape = (128, 128, 3)
17 #seção do tratamento de dados
18 generator = image.ImageDataGenerator(featurewise_std_normalization=True,fill_mode='constant')
19 batch_size = 20
20 input_size = (128,128)
21 timesteps = int(features/batch_size)
22 for x in range(0,len(itemes)):
23     itemes[x] = Image.open(itemes[x])
24     itemes[x] = itemes[x].resize(size=input_size)
25     itemes[x] = np.asarray(itemes[x])
26 itemes = np.asarray(itemes)
27 y = itemes.shape
28 if(len(y)==3):
29     itemes = np.reshape(itemes, (y[0],y[1],y[2],1))
30 generator.fit(itemes)
31 traingen = generator.flow_from_directory(path, target_size=input_size,batch_size=batch_size)
32 #seção do modelo de arquitetura
33 model = models.Sequential()
34 model.add(layers.Flatten(input_shape=input_shape))
35 model.add(layers.Dense(units=512,activation='relu'))
36 model.add(layers.Dense(units=2,activation='sigmoid'))
37 model.compile(optimizer='SGD',loss='binary_crossentropy',metrics=['accuracy'])
38 history = model.fit(traingen, steps_per_epoch = timesteps,epochs = 7)

```

Fonte: o Autor.

A etapa de treinamento da rede não foi incorporada ao software, e pode ser feita através de um interpretador Python na interface por linhas de comando do computador.

Figura 50 – Comandos para o treinamento da rede no computador.

```
C:\Users\binho\Desktop\Pythono\Interface>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import rede
```

Fonte: o Autor.

Ao realizar o comando *import rede*, a rede é treinada e as variáveis *rede.model* e *rede.history* podem ser carregadas, contendo informações sobre o modelo e o processo de treinamento. É possível obter um sumário da rede, que mostra o total de parâmetros e formato das saídas de cada camada.

Figura 51 – Sumário do modelo da rede.

```
>>> modelo = rede.model
>>> modelo.summary()
Model: "sequential"
=====
Layer (type)          Output Shape       Param #
=====
flatten (Flatten)     (None, 49152)      0
=====
dense (Dense)         (None, 512)        25166336
=====
dense_1 (Dense)       (None, 2)          1026
=====
Total params: 25,167,362
Trainable params: 25,167,362
Non-trainable params: 0
=====
>>>
```

Fonte: o Autor.

A camada retificadora (*flatten*, em inglês) não possui parâmetros, pois só muda o formato da entrada. O formato de sua saída é dado pela multiplicação  $128 \cdot 128 \cdot 3 = 49.152$ , pois as imagens têm dimensões 128x128 e 3 canais de cores (RGB). A segunda camada é uma camada densa, que possui para cada um de seus 512 valores  $49.152 + 1$  parâmetros, referentes à multiplicação termo a termo sob o dado de entrada e a adição final, ou seja, ela possui  $(49152 + 1) \cdot 512 = 25.166.336$  parâmetros. A última camada, que também é uma densa, possui apenas dois valores de saída, com  $512 + 1$  parâmetros para cada um, totalizando 1026. O processo de treinamento acontece logo após o comando *import rede*.

Figura 52 – Resultado do processo de treinamento.

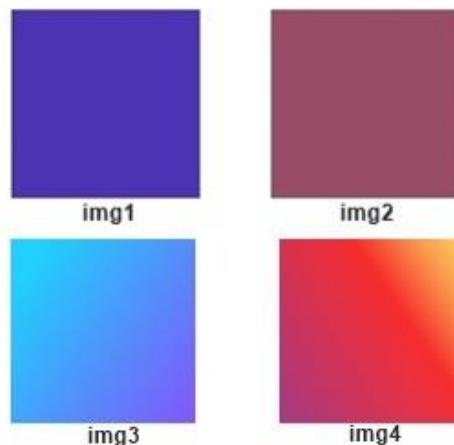
```
Epoch 1/7
2020-10-26 10:28:02.822718: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library cublas64_10.dll
100/100 [=====] - 3s 26ms/step - loss: 0.0122 - accuracy: 0.9920
Epoch 2/7
100/100 [=====] - 3s 25ms/step - loss: 3.9093e-05 - accuracy: 1.0000
Epoch 3/7
100/100 [=====] - 3s 25ms/step - loss: 2.9694e-05 - accuracy: 1.0000
Epoch 4/7
100/100 [=====] - 3s 25ms/step - loss: 2.4899e-05 - accuracy: 1.0000
Epoch 5/7
100/100 [=====] - 3s 25ms/step - loss: 2.1931e-05 - accuracy: 1.0000
Epoch 6/7
100/100 [=====] - 3s 25ms/step - loss: 1.9830e-05 - accuracy: 1.0000
Epoch 7/7
100/100 [=====] - 3s 25ms/step - loss: 1.8256e-05 - accuracy: 1.0000
>>>
```

Fonte: o Autor.

Ao final da primeira época a rede já havia obtido 99% de precisão sob os dados de treinamento, e com o passar das épocas a perda computada foi ficando cada vez menor. Neste caso a rede obtém uma boa performance pois os limites que ela precisa encontrar são claros, uma vez o canal azul ou vermelho das imagens sempre terá um valor entre 60% ~ 100%, enquanto os outros canais estarão sempre abaixo de 50%. Em tarefas do mundo real esses limites não são nada claros, e a rede pode perder precisão ao passar das épocas, pois ficará tendenciosa aos exemplos que mais aparecem no conjunto. Este fenômeno é chamado de overfitting e será ilustrado mais a frente nessa pesquisa.

Para avaliar uma imagem nesta rede é necessário passa-la pela etapa de tratamento de dados e usar a função *predict()* da variável do modelo, a mesma variável usada para obter o sumário. Esta função retornará o resultado da saída da última camada da rede, que neste caso são dois valores. As figuras a seguir mostram alguns testes feitos com imagens que não estavam presentes no conjunto de treino.

Figura 53 – Imagens avaliadas na rede e respectivos nomes.



Fonte: o Autor.

Figura 54 – Comandos para tratamento de uma imagem e avaliação na rede.

```
>>> from pathlib import Path
>>> import numpy as np
>>> from PIL import Image
>>> path = Path('.')
>>> img1 = Image.open(path/'datasets/img1.png')
>>> img1 = next(reden.generator.flow(np.asarray([np.asarray(img1)])))
>>> modelo.predict(img1)
array([[1.000000e+00, 1.8530133e-16]], dtype=float32)
>>>
```

Fonte: o Autor.

O primeiro valor se refere à probabilidade da imagem em pertencer à primeira categoria em ordem alfabética (azul), enquanto o outro se refere à segunda (vermelho).

Figura 55 – Avaliação das cinco imagens na rede.

```
>>> modelo.predict(img1)
array([[1.000000e+00, 3.659406e-14]], dtype=float32)
>>> modelo.predict(img2)
array([[1.000973e-06, 9.994776e-01]], dtype=float32)
>>> modelo.predict(img3)
array([[1.000000e+00, 3.250342e-23]], dtype=float32)
>>> modelo.predict(img4)
array([[9.2389134e-20, 1.000000e+00]], dtype=float32)
```

Fonte: o Autor.

As avaliações foram precisas, confirmando que a rede está funcionando corretamente.

### 5.2.2 Três tonalidades

Com o intuito de mostrar o funcionamento de outras funções de perda, foram adicionadas imagens de tons de verde ao conjunto de dados do exemplo anterior.

#### 5.2.2.1 As imagens

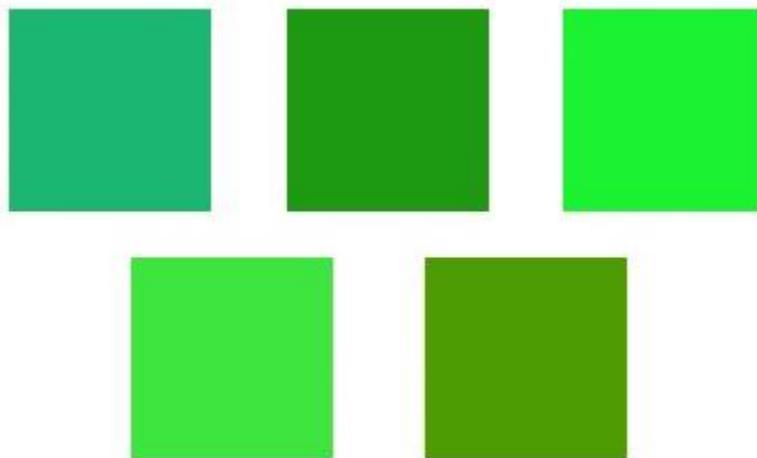
Figura 56 – Código usado para gerar as imagens de tom verde.

```

1 import numpy as np
2 from PIL import Image
3 from pathlib import Path
4 ptg = Path('./tonalidades/treino/verde') #caminho: Interface/datasets/tonalidades/treino/verde
5 z = np.zeros((128,128), 'float32') #matriz de zeros, de tamanho 128x128
6 for n in range(1000): #1000 iterações
7     rgbG = np.zeros((128,128,3), 'uint8')
8     rgbG[:, 0] = (z+np.random.rand()*0.5)*256 #canal R = x*0.5, 0 <= x <= 1
9     rgbG[:, 1] = (z+np.random.rand()*0.4+0.6)*256 #canal G = 0.6 + x*0.4, 0 <= x <= 1
10    rgbG[:, 2] = (z+np.random.rand()*0.5)*256 #canal B = x*0.5, 0 <= x <= 1
11    img_verde = Image.fromarray(rgbG) #gerar imagem
12    img_verde.save(ptg/('imggreen'+str(n)+'.png')) #salvar na pasta 'verde'
```

Fonte: o Autor.

Figura 57 – Algumas das imagens de tons esverdeados geradas.

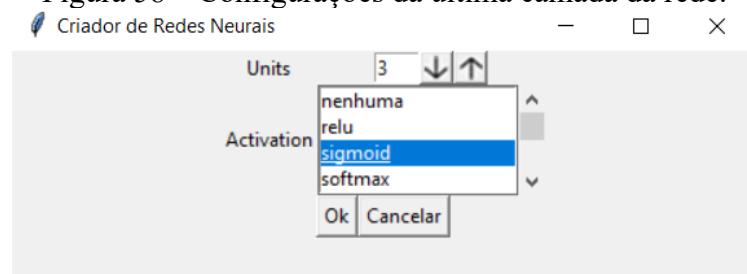


Fonte: o Autor.

#### 5.2.2.2 As redes

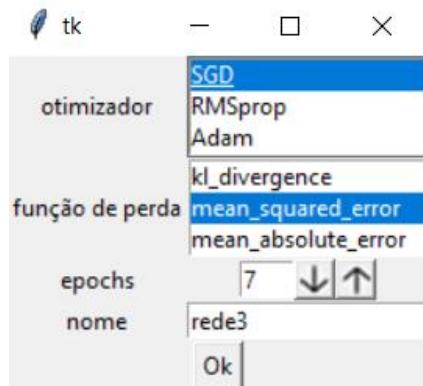
As alterações a feitas na rede foram: mudar o número de saídas da última camada de 2 para 3, e alterar a função de perda no último menu de configuração.

Figura 58 – Configurações da última camada da rede.



Fonte: o Autor.

Figura 59 – Última etapa de configurações da rede.



Fonte: o Autor.

Figura 60 – Trecho de código referente ao modelo da rede.

```

35 #seção do modelo da arquitetura
36 model = models.Sequential()
37 model.add(layers.Flatten(input_shape=input_shape))
38 model.add(layers.Dense(units=512,activation='relu'))
39 model.add(layers.Dense(units=3,activation='sigmoid'))
40 model.compile(optimizer='SGD',loss='mean_squared_error',metrics=['accuracy'])
41 history = model.fit(traiingen, steps_per_epoch = timesteps,epochs = 7)

```

Fonte: o Autor.

Figura 61 – Resultado do processo de treinamento com a função de perda MSE.

```

Epoch 1/7
2020-10-28 09:27:52.485418: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic
library cublas64_10.dll
150/150 [=====] - 4s 25ms/step - loss: 0.0030 - accuracy: 0.9950
Epoch 2/7
150/150 [=====] - 4s 25ms/step - loss: 3.4680e-05 - accuracy: 1.0000
Epoch 3/7
150/150 [=====] - 4s 25ms/step - loss: 2.4621e-05 - accuracy: 1.0000
Epoch 4/7
150/150 [=====] - 4s 26ms/step - loss: 1.9734e-05 - accuracy: 1.0000
Epoch 5/7
150/150 [=====] - 8s 53ms/step - loss: 1.6744e-05 - accuracy: 1.0000
Epoch 6/7
150/150 [=====] - 11s 74ms/step - loss: 1.4736e-05 - accuracy: 1.0000
Epoch 7/7
150/150 [=====] - 17s 113ms/step - loss: 1.3263e-05 - accuracy: 1.0000

```

Fonte: o Autor.

Figura 62 – Sumário da rede com MSE.

```

>>> modelo = rede3.model
>>> modelo.summary()
Model: "sequential"

```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 49152)	0
dense (Dense)	(None, 512)	25166336
dense_1 (Dense)	(None, 3)	1539

```

Total params: 25,167,875
Trainable params: 25,167,875
Non-trainable params: 0

```

```

>>>

```

Fonte: o Autor.

Para uma tarefa tão simples, 512 pode ser um número muito grande de saídas para a primeira camada, e dificulta a visualização dos pesos. As próximas figuras ilustram o resultado da rede se a primeira camada contiver apenas 9 saídas, e utilização da função de perda MAE.

Figura 63 – Trecho de código referente ao modelo da rede com MAE.

```

35 #seção do modelo da arquitetura
36 model = models.Sequential()
37 model.add(layers.Flatten(input_shape=input_shape))
38 model.add(layers.Dense(units=9,activation='relu'))
39 model.add(layers.Dense(units=3,activation='sigmoid'))
40 model.compile(optimizer='SGD',loss='mean_absolute_error',metrics=['accuracy'])
41 history = model.fit(training, steps_per_epoch = timesteps, epochs = 7)

```

Fonte: o Autor.

Figura 64 – Resultado do processo de treinamento com a função de perda MAE.

```

Epoch 1/7
2020-10-28 10:15:18.343680: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic
library cublas64_10.dll
150/150 [=====] - 3s 17ms/step - loss: 0.0140 - accuracy: 0.9937
Epoch 2/7
150/150 [=====] - 3s 18ms/step - loss: 5.3675e-04 - accuracy: 1.0000
Epoch 3/7
150/150 [=====] - 3s 17ms/step - loss: 3.4147e-04 - accuracy: 1.0000
Epoch 4/7
150/150 [=====] - 3s 17ms/step - loss: 2.5695e-04 - accuracy: 1.0000
Epoch 5/7
150/150 [=====] - 3s 17ms/step - loss: 2.0831e-04 - accuracy: 1.0000
Epoch 6/7
150/150 [=====] - 3s 17ms/step - loss: 1.7493e-04 - accuracy: 1.0000
Epoch 7/7
150/150 [=====] - 3s 17ms/step - loss: 1.5107e-04 - accuracy: 1.0000
>>>

```

Fonte: o Autor.

Figura 65 – Sumário da rede com 9 saídas na primeira camada.

```

>>> modelo = rede3.model
>>> modelo.summary()
Model: "sequential"

```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 49152)	0
dense (Dense)	(None, 9)	442377
dense_1 (Dense)	(None, 3)	30

```

Total params: 442,407
Trainable params: 442,407
Non-trainable params: 0

```

Fonte: o Autor.

A mudança de 512 para 9 não alterou significativamente a eficiência da etapa de treinamento da rede. A quantidade de dados também é um fator importante, e a figura a seguir ilustra o processo de treinamento, porém agora com 200 imagens de cada tonalidade, ao invés de 1000.

Figura 66 – Resultado do processo de treinamento com apenas 600 imagens.

```
Epoch 1/7
2020-10-28 10:48:18.664618: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic
library cublas64_10.dll
30/30 [=====] - 1s 17ms/step - loss: 0.0933 - accuracy: 0.9683
Epoch 2/7
30/30 [=====] - 0s 17ms/step - loss: 0.0090 - accuracy: 1.0000
Epoch 3/7
30/30 [=====] - 1s 17ms/step - loss: 0.0040 - accuracy: 1.0000
Epoch 4/7
30/30 [=====] - 0s 16ms/step - loss: 0.0026 - accuracy: 1.0000
Epoch 5/7
30/30 [=====] - 1s 17ms/step - loss: 0.0020 - accuracy: 1.0000
Epoch 6/7
30/30 [=====] - 0s 16ms/step - loss: 0.0015 - accuracy: 1.0000
Epoch 7/7
30/30 [=====] - 1s 17ms/step - loss: 0.0013 - accuracy: 1.0000
```

Fonte: o Autor.

Também é possível avaliar a rede sob um conjunto contendo dados e classificações. As 2400 imagens que foram retiradas para esse treinamento foram movidas para uma pasta de teste, e a função *evaluate()* da variável do modelo sob essa pasta gera o seguinte resultado:

Figura 67 – Avaliação da rede sob o conjunto de testes.

```
>>> testegen = rede3.generator.flow_from_directory(testepath, target_size = rede3.input_size, batch_size=2400)
Found 2400 images belonging to 3 classes.
>>> modelo = rede3.model
>>> modelo.evaluate(testegen, batch_size=2400, steps=1)
1/1 [=====] - 0s 0s/step - loss: 0.0018 - accuracy: 1.0000
[0.0017644099425524473, 1.0]
```

Fonte: o Autor.

A rede foi capaz de classificar com precisão as 2400 imagens, treinando apenas sob as 600. Nota-se que neste caso também é preciso passar as imagens pela etapa de tratamento dos dados, com comandos descritos nas primeiras linhas da Figura X28 e mais detalhadamente na figura X36. Os valores retornados ao final do processo (0.0017644..., 1.0) são os referentes à perda computada e à precisão.

A figura X29 ilustra o processo de treinamento para um conjunto reduzido de 20 imagens de cada tonalidade, e a figura X30 ilustra a avaliação sob as mesmas 2400 imagens da figura X28.

Figura 68 – Resultado do processo de treinamento com apenas 60 imagens.

```
Epoch 1/7
2020-10-28 11:30:32.223856: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic
library cublas64_10.dll
3/3 [=====] - 0s 12ms/step - loss: 0.1459 - accuracy: 0.8500
Epoch 2/7
3/3 [=====] - 0s 13ms/step - loss: 0.0206 - accuracy: 1.0000
Epoch 3/7
3/3 [=====] - 0s 13ms/step - loss: 0.0140 - accuracy: 1.0000
Epoch 4/7
3/3 [=====] - 0s 12ms/step - loss: 0.0105 - accuracy: 1.0000
Epoch 5/7
3/3 [=====] - 0s 12ms/step - loss: 0.0088 - accuracy: 1.0000
Epoch 6/7
3/3 [=====] - 0s 12ms/step - loss: 0.0075 - accuracy: 1.0000
Epoch 7/7
3/3 [=====] - 0s 12ms/step - loss: 0.0066 - accuracy: 1.0000
```

Fonte: o Autor.

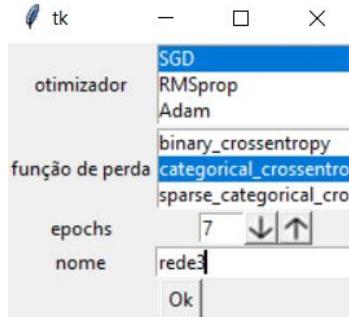
Figura 69 – Processo de avaliação da rede sob as 2400 imagens de teste.

```
>>> modelo = rede3.model
>>> modelo.evaluate(testegen, batch_size=2400, steps=1)
1/1 [=====] - 0s 998us/step - loss: 0.0099 - accuracy: 0.9958
[0.009926248341798782, 0.995833373069763]
```

Fonte: o Autor.

Neste caso a rede errou em 0.0042% das classificações, que corresponde a 10 casos. A seguir foi treinada a mesma rede, porém usando a função de perda de entropia cruzada categórica, a mais indicada para problemas de classificação em múltiplas classes.

Figura 70 – Configurações da última etapa da rede.



Fonte: o Autor.

Figura 71 – Seção de código do modelo com entropia cruzada categórica.

```
35 #seção do modelo da arquitetura
36 model = models.Sequential()
37 model.add(layers.Flatten(input_shape=input_shape))
38 model.add(layers.Dense(units=9,activation='relu'))
39 model.add(layers.Dense(units=3,activation='sigmoid'))
40 model.compile(optimizer='SGD',loss='categorical_crossentropy',metrics=['accuracy'])
41 history = model.fit(traingen, steps_per_epoch = timesteps,epochs = 7)
```

Fonte: o Autor.

Figura 72 –Processo de treinamento da rede com entropia cruzada categórica e 60 imagens.

```
Epoch 1/7
2020-10-28 11:56:14.086585: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library cublas64_10.dll
3/3 [=====] - 0s 10ms/step - loss: 0.9761 - accuracy: 0.5667
Epoch 2/7
3/3 [=====] - 0s 12ms/step - loss: 0.6601 - accuracy: 0.6667
Epoch 3/7
3/3 [=====] - 0s 12ms/step - loss: 0.6471 - accuracy: 0.6667
Epoch 4/7
3/3 [=====] - 0s 13ms/step - loss: 0.5920 - accuracy: 0.6833
Epoch 5/7
3/3 [=====] - 0s 13ms/step - loss: 0.5777 - accuracy: 0.7167
Epoch 6/7
3/3 [=====] - 0s 12ms/step - loss: 0.5774 - accuracy: 0.7167
Epoch 7/7
3/3 [=====] - 0s 12ms/step - loss: 0.5772 - accuracy: 0.7167
```

Fonte: o Autor.

Neste caso fica claro que os dados são insuficientes para o treinamento dessa rede. Para tratar este problema ou aumenta-se o número de dados ou o número de épocas. A figura 74 ilustra as etapas finais de um treinamento com 30 épocas e a figura 75 ilustra o treino com 600 imagens, 200 de cada tonalidade.

Figura 73 – Etapas finais do treinamento com 30 épocas.

```
Epoch 25/30
3/3 [=====] - 0s 12ms/step - loss: 0.2331 - accuracy: 0.7500
Epoch 26/30
3/3 [=====] - 0s 12ms/step - loss: 0.2323 - accuracy: 0.7500
Epoch 27/30
3/3 [=====] - 0s 12ms/step - loss: 0.2320 - accuracy: 0.7667
Epoch 28/30
3/3 [=====] - 0s 12ms/step - loss: 0.2318 - accuracy: 0.7667
Epoch 29/30
3/3 [=====] - 0s 12ms/step - loss: 0.2316 - accuracy: 0.7667
Epoch 30/30
3/3 [=====] - 0s 14ms/step - loss: 0.2315 - accuracy: 0.7667
```

Fonte: o Autor.

Isso confirma que são necessários mais dados para que a função ECC seja eficiente.

Figura 74 – Processo de treinamento da rede com 600 imagens.

```
Epoch 1/7
2020-10-28 12:04:28.186784: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic
library cublas64_10.dll
30/30 [=====] - 1s 17ms/step - loss: 0.1137 - accuracy: 0.9400
Epoch 2/7
30/30 [=====] - 1s 18ms/step - loss: 0.0059 - accuracy: 1.0000
Epoch 3/7
30/30 [=====] - 1s 17ms/step - loss: 0.0026 - accuracy: 1.0000
Epoch 4/7
30/30 [=====] - 1s 17ms/step - loss: 0.0017 - accuracy: 1.0000
Epoch 5/7
30/30 [=====] - 1s 17ms/step - loss: 0.0010 - accuracy: 1.0000
Epoch 6/7
30/30 [=====] - 1s 17ms/step - loss: 8.2328e-04 - accuracy: 1.0000
Epoch 7/7
30/30 [=====] - 1s 17ms/step - loss: 7.2166e-04 - accuracy: 1.0000
```

Fonte: o Autor.

Com 600 imagens a rede com entropia cruzada categórica já obteve um resultado próximo da com MAE, inclusive com uma perda menor. Isso elucida que não existe uma função específica para tratar cada tipo de problema, apenas indicações, e as possibilidades devem ser testadas.

Figura 75 – Avaliação da rede sob o conjunto de testes (2400 imagens).

```
>>> modelo = rede3.modelo
>>> testepath = rede3.path '/../teste'
>>> testepath = testepath.resolve()
>>> testepath
WindowsPath('C:/Users/binho/Desktop/Pythono/Interface/datasets/tonalidades/teste')
>>> testegen = rede3.generator.flow_from_directory(testepath, target_size=rede3.input_size, batch_size=2400)
Found 2400 images belonging to 3 classes.
>>> modelo.evaluate(testegen, steps=1)
1/1 [=====] - 0s 998us/step - loss: 0.0012 - accuracy: 1.0000
[0.0012252230662852526, 1.0]
```

Fonte: o Autor.

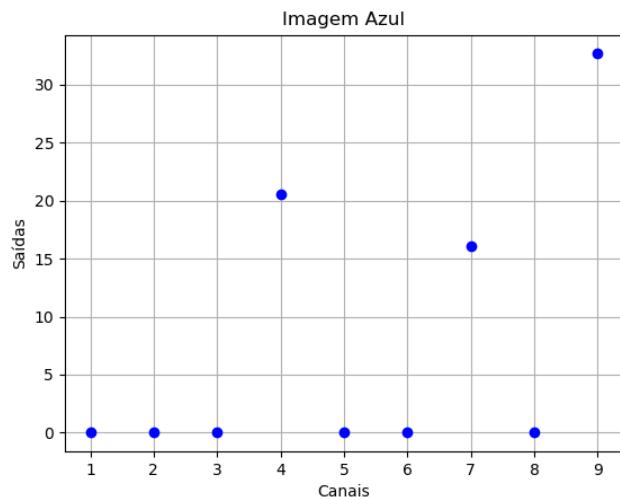
Através da variável do modelo também é possível obter as saídas separadas das camadas, e as figuras a seguir ilustram a saída da primeira camada densa para três imagens diferentes, uma de cada tonalidade, das redes com a função MAE e a função de entropia cruzada categórica, ambas treinadas sob o conjunto de 600 imagens.

Figura 76 – Imagens avaliadas.



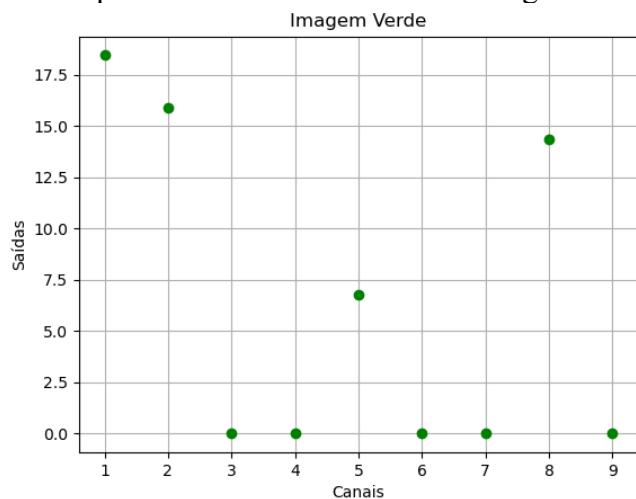
Fonte: o Autor.

Figura 77 – Saída da primeira camada densa sob a imagem azul, rede com MAE.



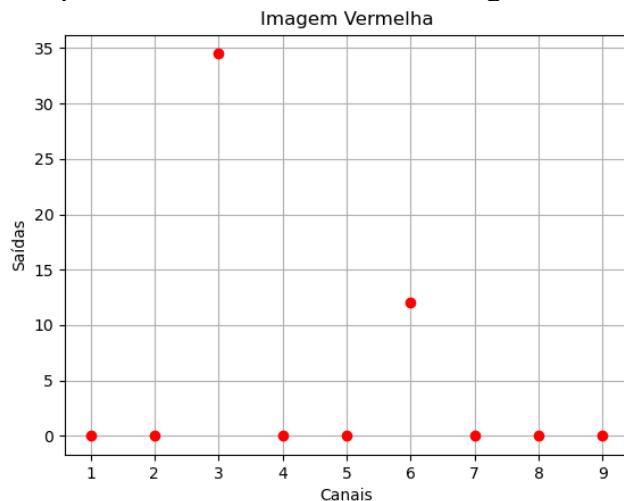
Fonte: o Autor.

Figura 78 – Saída da primeira camada densa sob a imagem verde, rede com MAE.



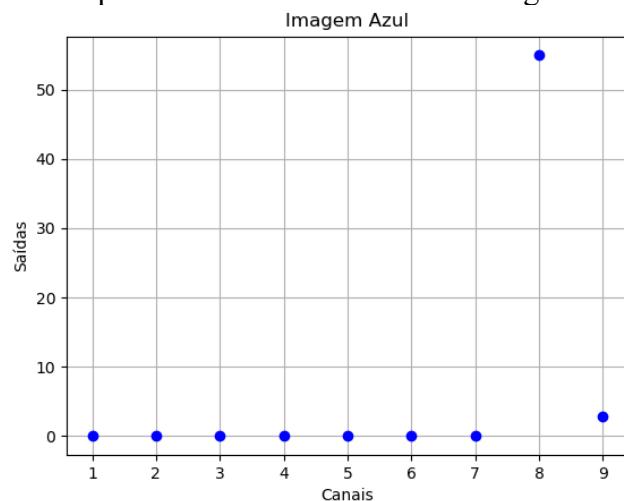
Fonte: o Autor.

Figura 79 – Saída da primeira camada densa sob a imagem vermelha, rede com MAE.



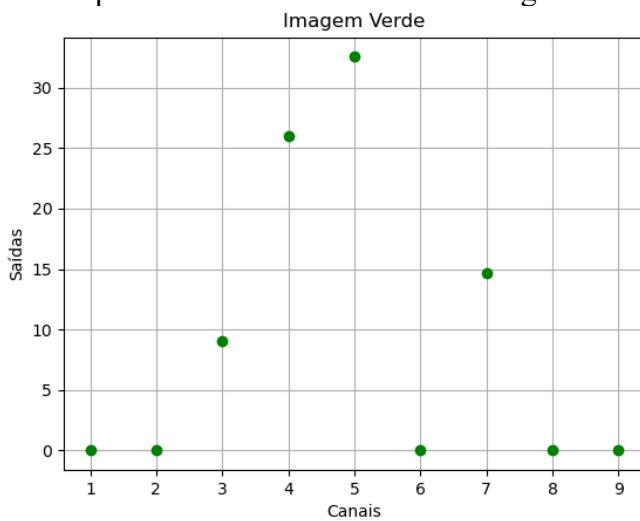
Fonte: o Autor.

Figura 80 – Saída da primeira camada densa sob a imagem azul, rede com ECC.



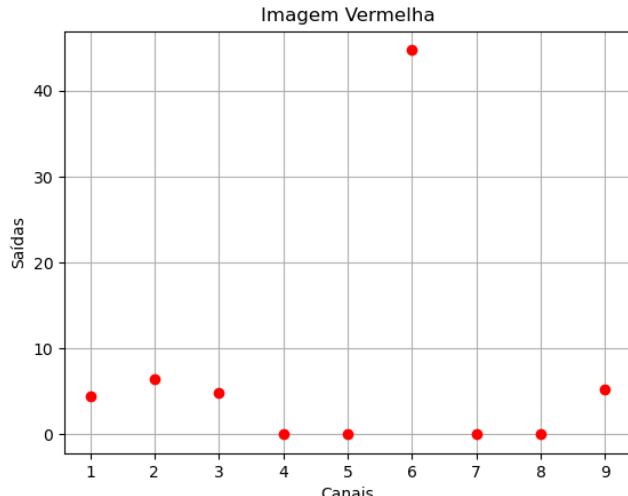
Fonte: o Autor.

Figura 81 – Saída da primeira camada densa sob a imagem verde, rede com ECC.



Fonte: o Autor.

Figura 82 – Saída da primeira camada densa sob a imagem vermelha, rede com ECC.



Fonte: o Autor.

### 5.2.3 Formas geométricas

Este exemplo usa uma RNP para identificar se uma imagem contém o desenho de um quadrado ou de um triângulo.

#### 5.2.3.1 As imagens

Para gerar as imagens deste exemplo foi usado o conjunto do exemplo de tonalidades, adicionando as formas geométricas às imagens. Ao todo foram usados 600 tons, 200 de cada, gerando 1200 imagens, 600 com quadrados e 600 com triângulos. O código usado para tal foi:

Figura 83 – Parte do código que gerou as imagens com quadrados.

```

1  import numpy as np
2  from PIL import Image
3  from pathlib import Path
4  ptvd = Path('./tonalidades/treino/verde')
5  ptvr = Path('./tonalidades/treino/vermelho')
6  ptaz = Path('./tonalidades/treino/azul')
7  pathqua = Path('./geometricas/treino/quadrado')
8  pathtri = Path('./geometricas/treino/triangulo')
9
10 square = np.zeros((30,30,3), 'uint8') #quadrado inteiro preto
11 square[2:28, 2:28, ...] = np.ones((26,26,3), 'uint8') #quadrado interno pintado
12
13 for n in range(200):
14     pontox = int(np.random.rand()*94)
15     pontoy = int(32 + np.random.rand()*95)
16     sqazul = np.asarray(Image.open(ptaz/('imgblue'+str(n)+'.png'))).copy() #imagem azul
17     sqazul[pontox:pontox+30, pontoy-30:pontoy, ...] *= square
18     img_qua = Image.fromarray(sqazul)
19     img_qua.save(pathqua/('imgsquare'+str(n)+'.png'))
20
21     pontox = int(np.random.rand()*95)
22     pontoy = int(32 + np.random.rand()*95)
23     sqverm = np.asarray(Image.open(ptvr/('imgred'+str(n)+'.png'))).copy() #imagem vermelha
24     sqverm[pontox:pontox+30, pontoy-30:pontoy, ...] *= square
25     img_qua = Image.fromarray(sqverm)
26     img_qua.save(pathqua/('imgsquare'+str(200+n)+'.png'))
27
28     pontox = int(np.random.rand()*95)
29     pontoy = int(32 + np.random.rand()*95)
30     sqverd = np.asarray(Image.open(ptvd/('imggreen'+str(n)+'.png'))).copy() #imagem verde
31     sqverd[pontox:pontox+30, pontoy-30:pontoy, ...] *= square
32     img_qua = Image.fromarray(sqverd)
33     img_qua.save(pathqua/('imgsquare'+str(400+n)+'.png'))
34

```

Fonte: o Autor.

Figura 84 – Parte do código que gerou as imagens com triângulos.

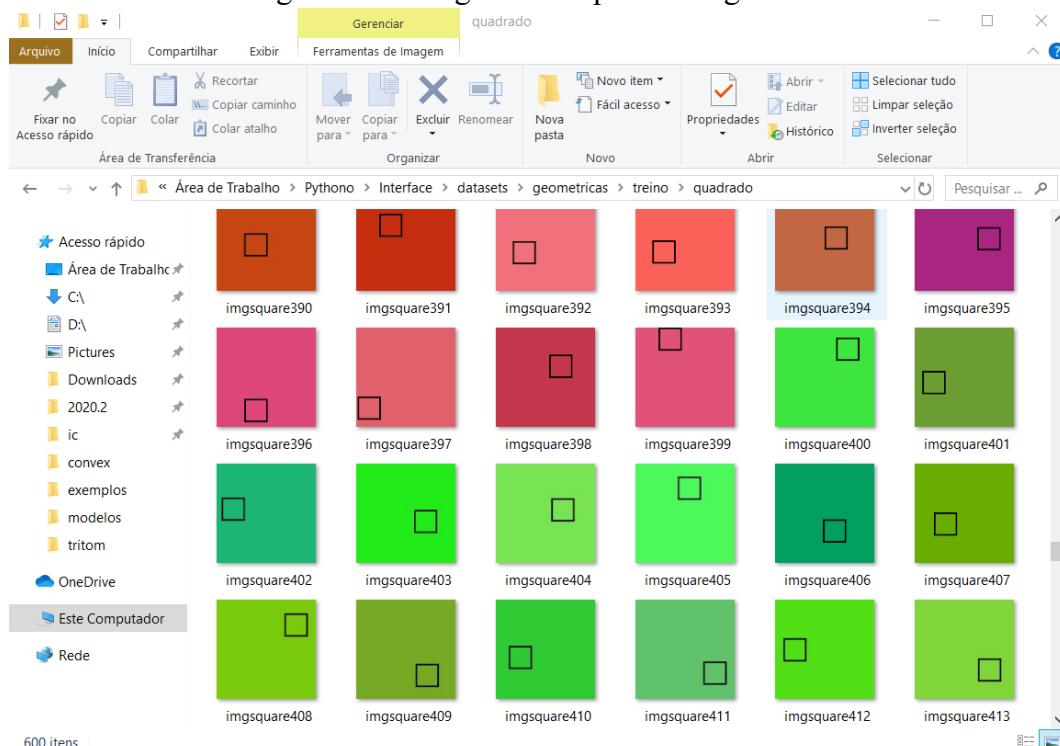
```

35     for n in range(200):
36         x = int(32 + np.random.rand()*94)
37         y = int(np.random.rand()*95)
38         triang = np.ones((128,128), 'uint8')
39         triang[x:x+2, y:y+30] = np.zeros((2,30), 'uint8') #linha inferior
40         for i in range(15): #linha lateral esquerda
41             triang[x-i, y+i] = 0
42             triang[x-i-1, y+i] = 0
43         for i in range(15,30): #linha lateral direita
44             triang[x+15-i, y+44-i] = 0
45             triang[x+14-i, y+44-i] = 0
46
47         #carregar imagem de tom verde
48         triverd = np.asarray(Image.open(ptvd/('imggreen'+str(n)+'.png'))).copy()
49         triverd[:, 0]=triang
50         triverd[:, 1]=triang
51         triverd[:, 2]=triang
52         img = Image.fromarray(triverd)
53         img.save(pathtri/('imgtriangle'+str(n)+'.png'))
54
55         #carregar imagem de tom vermelho
56         triverm = np.asarray(Image.open(ptvr/('imgred'+str(n)+'.png'))).copy()
57         triverm[:, 0]=triang
58         triverm[:, 1]=triang
59         triverm[:, 2]=triang
60         img = Image.fromarray(triverm)
61         img.save(pathtri/('imgtriangle'+str(200+n)+'.png'))
62
63         #carregar imagem de tom azul
64         triazul = np.asarray(Image.open(ptaz/('imgblue'+str(n)+'.png'))).copy()
65         triazul[:, 0]=triang
66         triazul[:, 1]=triang
67         triazul[:, 2]=triang
68         img = Image.fromarray(triazul)
69         img.save(pathtri/('imgtriangle'+str(400+n)+'.png'))

```

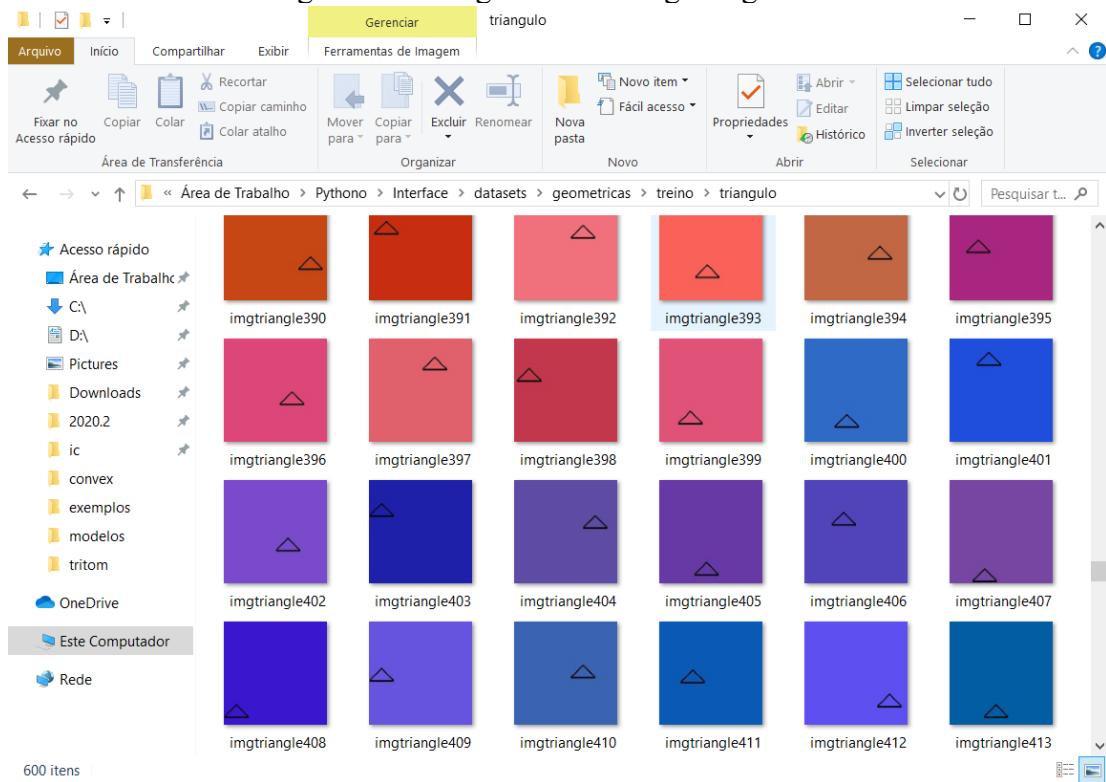
Fonte: o Autor.

Figura 85 – Imagens com quadrados geradas.



Fonte: o Autor.

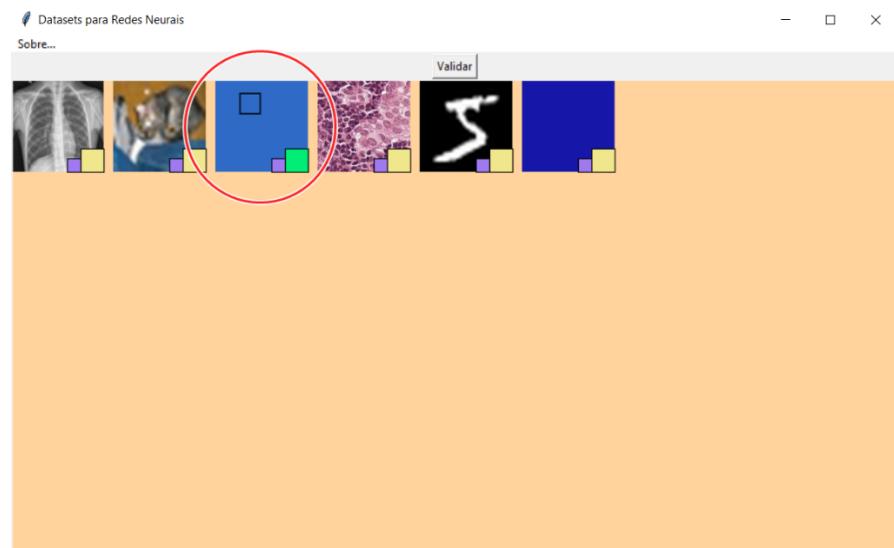
Figura 86 – Imagens com triângulos geradas.



Fonte: o Autor.

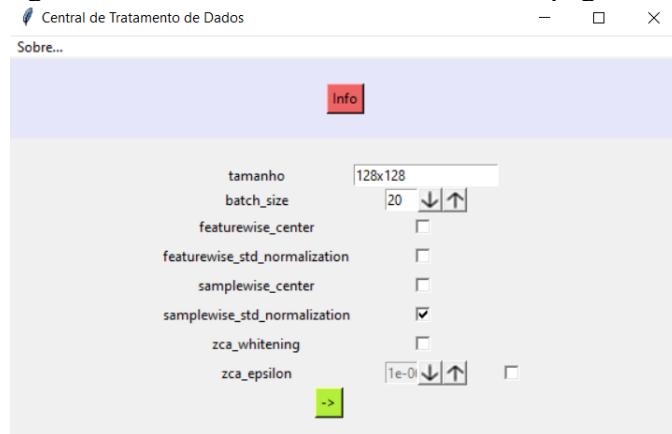
As figuras a seguir ilustram as etapas de seleção e tratamento dos dados através do software desenvolvido.

Figura 87 – Menu de seleção dos dados, conjunto de figuras geométricas.



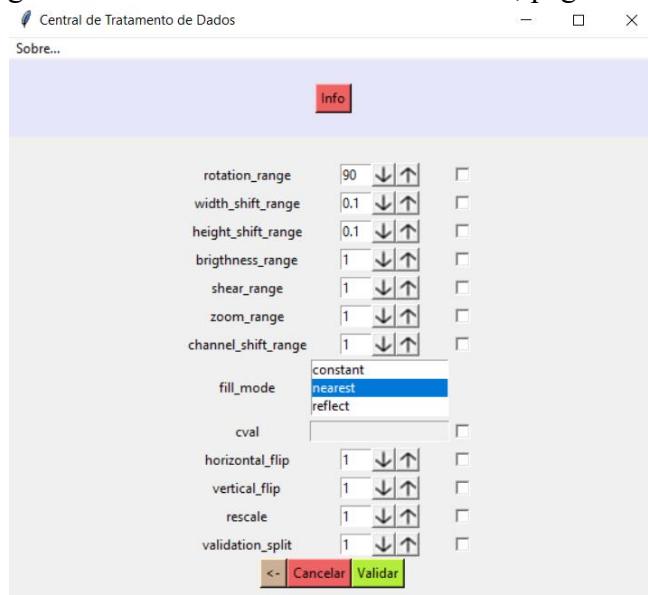
Fonte: o Autor.

Figura 88 – Menu de tratamento de dados, página um.



Fonte: o Autor.

Figura 89 – Menu de tratamento de dados, página dois.



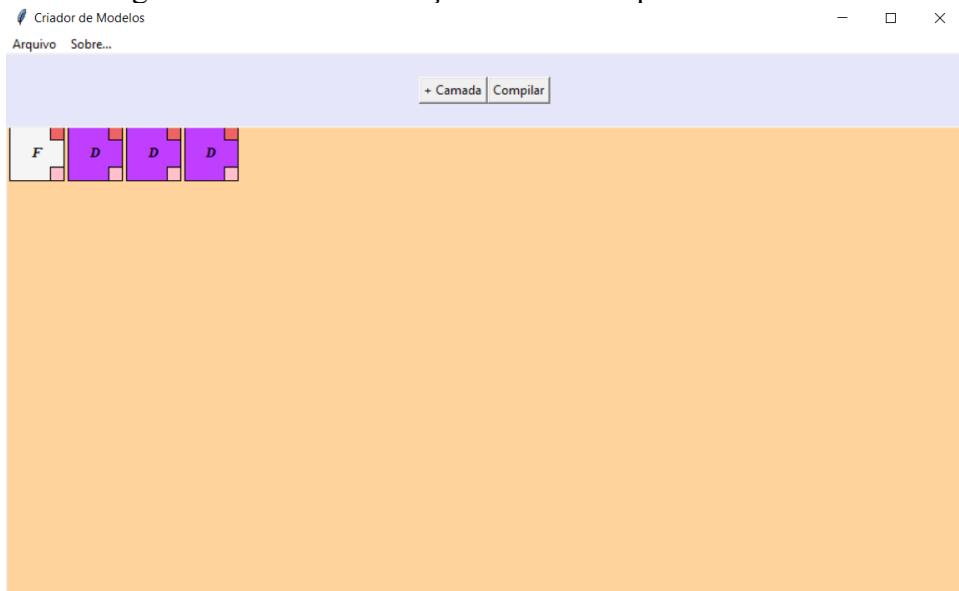
Fonte: o Autor.

### 5.2.3.2 As redes

A princípio foram testadas duas redes para o problema: uma com uma camada retificadora e três densas, com 512, 9 e 2 saídas cada, respectivamente; e outra com duas camadas convolutivas, uma MaxPooling, uma retificadora e duas densas, com 256 e 2 saídas cada, respectivamente. As funções de ativação das camadas intermediárias (convolutivas e densas) foram a *relu*, e das finais, que possuem com 2 saídas, foi a função sigmoid.

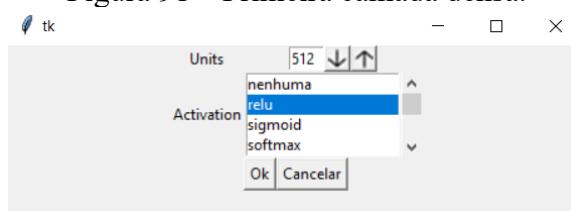
As figuras a seguir ilustram o menu de criação de modelos do software, a configuração das camadas e os resultados do treinamento da rede.

Figura 90 – Menu de criação de modelos para a rede densa.



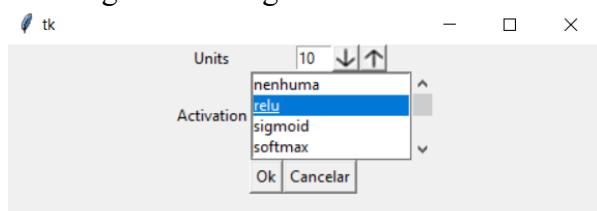
Fonte: o Autor.

Figura 91 – Primeira camada densa.



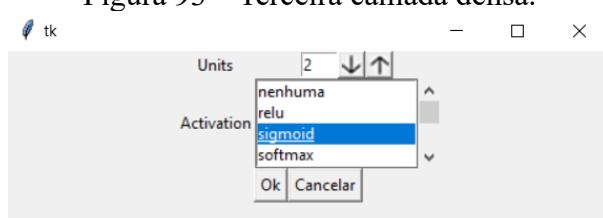
Fonte: o Autor.

Figura 92 – Segunda camada densa.



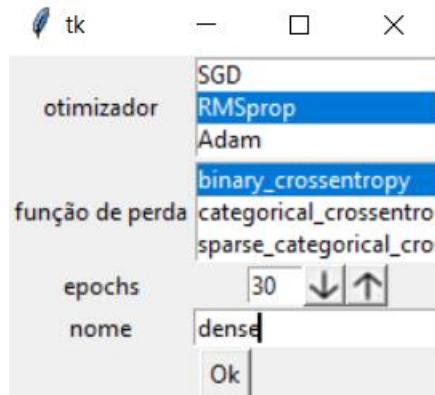
Fonte: o Autor.

Figura 93 – Terceira camada densa.



Fonte: o Autor.

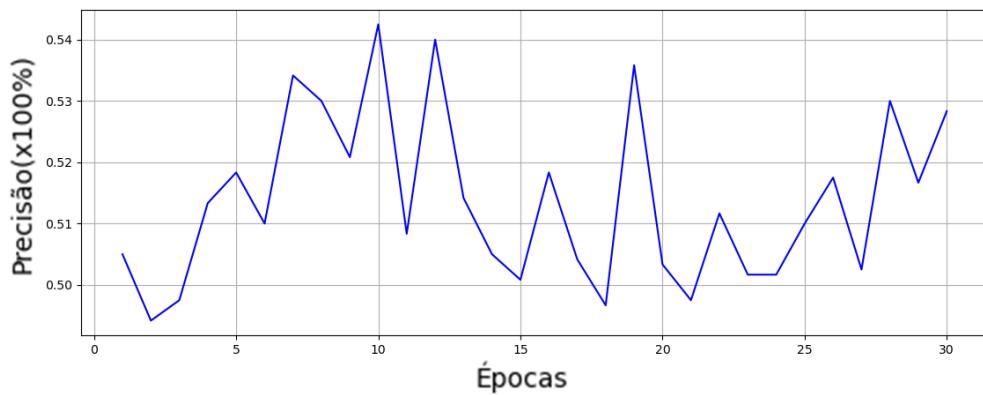
Figura 94 – Menu de compilação da rede dense.



Fonte: o Autor.

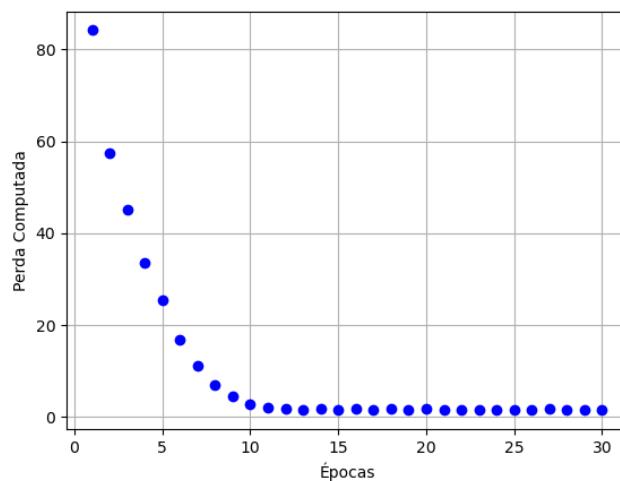
Cada época tem como saída dois valores: *loss*, que é a média das perdas computadas naquela época; e *accuracy*, que é a precisão das previsões que a rede fez sob o conjunto de treino, ou seja, a porcentagem de acertos. As figuras a seguir representam a *loss* e a *accuracy* através das 30 épocas.

Figura 95 – Precisão da rede dense através das 30 épocas.



Fonte: o Autor.

Figura 96 – Perda computada da rede dense através das 30 épocas.

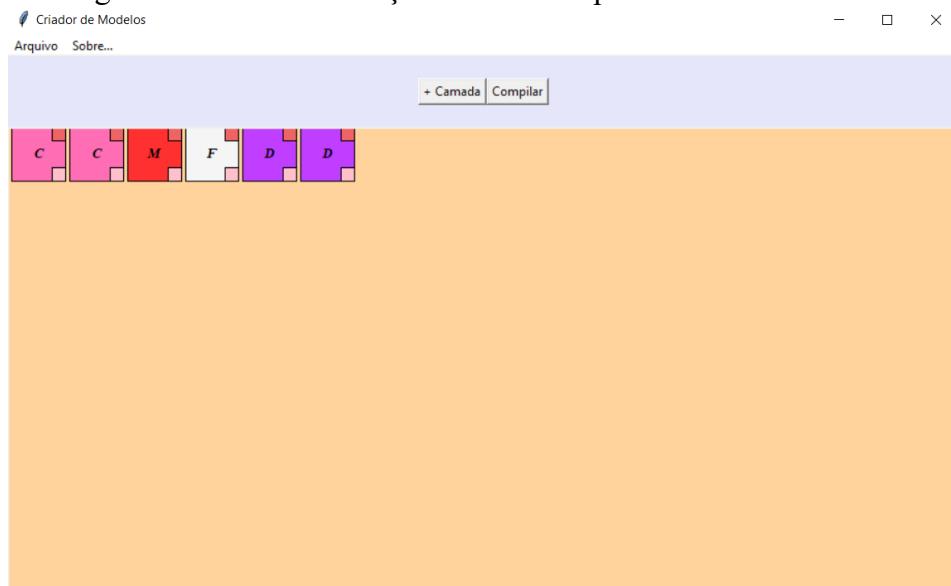


Fonte: o Autor.

O baixo valor de perda computada a partir da décima época acompanhada do baixo valor de precisão, com maior valor próximo de 54.2% e menor valor próximo de 49.3%, significa que a partir daquele ponto a rede já não é mais capaz de adquirir nenhuma informação nova a respeito do conjunto de dados, e a baixa precisão era esperada pois redes com apenas camadas densas não são capazes de identificar informações que podem estar contidas em qualquer lugar da imagem, como é o caso deste conjunto.

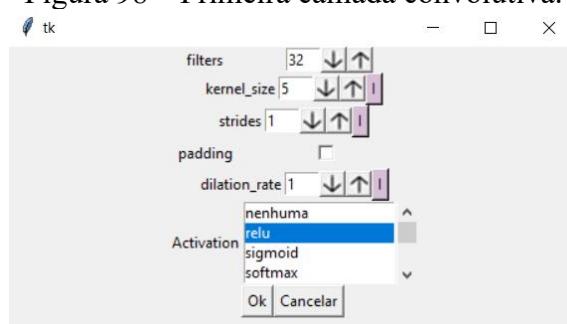
As figuras a seguir ilustram as configurações das camadas, os resultados e os comandos usados para plotar os valores com a biblioteca matplotlib, em python.

Figura 97 - Menu de criação de modelos para a rede convolutiva.



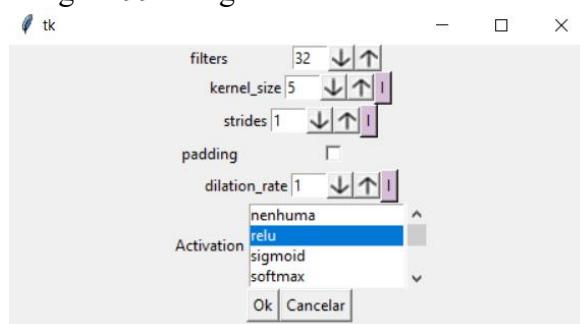
Fonte: o Autor.

Figura 98 – Primeira camada convolutiva.



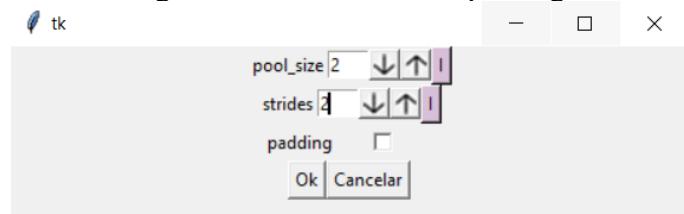
Fonte: o Autor.

Figura 99 – Segunda camada convolutiva.



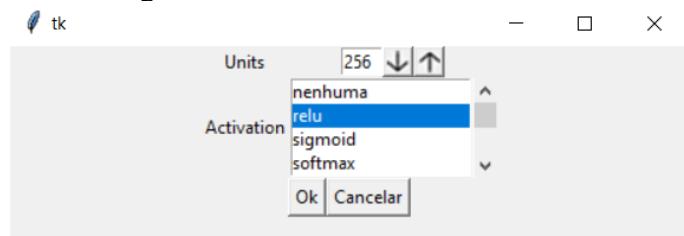
Fonte: o Autor.

Figura 100 – Camada maxpooling.



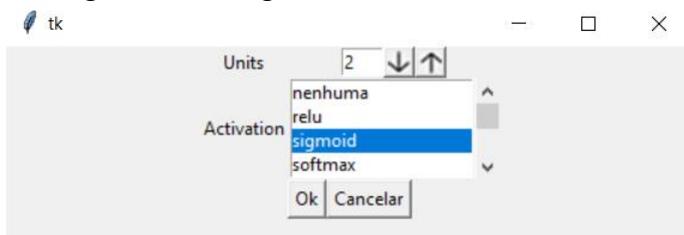
Fonte: o Autor.

Figura 101 – Primeira camada densa.



Fonte: o Autor.

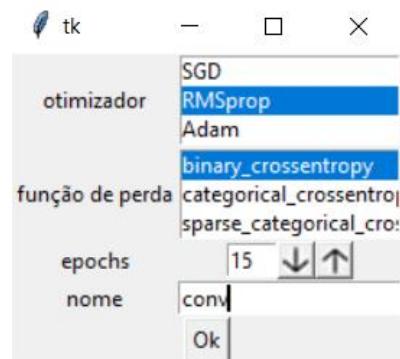
Figura 102 – Segunda e última camada densa.



Fonte: o Autor.

A camada retificadora não foi ilustrada pois não possuem opções de configuração.

Figura 103 – Menu de compilação da rede conv.



Fonte: o Autor

Figura 104 – Etapa de treinamento da rede conv.

```

Epoch 1/15
60/60 [=====] - 49s 812ms/step - loss: 5.7407 - accuracy: 0.8083
Epoch 2/15
60/60 [=====] - 78s 1s/step - loss: 0.2488 - accuracy: 0.9800
Epoch 3/15
60/60 [=====] - 78s 1s/step - loss: 0.2990 - accuracy: 0.9783
Epoch 4/15
60/60 [=====] - 78s 1s/step - loss: 0.7506 - accuracy: 0.9725
Epoch 5/15
60/60 [=====] - 98s 2s/step - loss: 2.4299e-04 - accuracy: 1.0000
Epoch 6/15
60/60 [=====] - 116s 2s/step - loss: 0.2085 - accuracy: 0.9908
Epoch 7/15
60/60 [=====] - 117s 2s/step - loss: 5.3779e-04 - accuracy: 1.0000
Epoch 8/15
60/60 [=====] - 127s 2s/step - loss: 0.4552 - accuracy: 0.9858
Epoch 9/15
60/60 [=====] - 105s 2s/step - loss: 8.5256e-05 - accuracy: 1.0000
Epoch 10/15
60/60 [=====] - 136s 2s/step - loss: 3.8771 - accuracy: 0.9417
Epoch 11/15
60/60 [=====] - 136s 2s/step - loss: 0.0586 - accuracy: 0.9942
Epoch 12/15
60/60 [=====] - 129s 2s/step - loss: 6.8220e-06 - accuracy: 1.0000
Epoch 13/15
60/60 [=====] - 137s 2s/step - loss: 5.8753e-07 - accuracy: 1.0000
Epoch 14/15
60/60 [=====] - 137s 2s/step - loss: 9.8430e-08 - accuracy: 1.0000
Epoch 15/15
60/60 [=====] - 150s 3s/step - loss: 1.6625e-08 - accuracy: 1.0000

```

Fonte: o Autor.

Figura 105 – Comandos usados para plotar a precisão e a perda do treino.

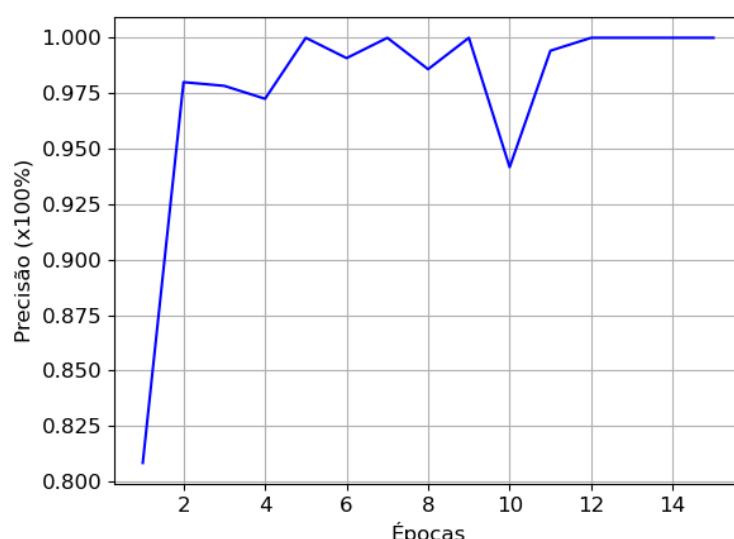
```

>>> from matplotlib import pyplot as plt
>>> from criador import plot
>>> registro = conv.history.history
>>> registro.keys()
dict_keys(['loss', 'accuracy'])
>>> loss = registro['loss']
>>> acc = registro['accuracy']
>>> epochs = range(1, len(acc)+1)
>>> plot(epochs, acc, 'Épocas', 'Precisão (x100%)', '', True, 'b')
>>> plot(epochs, loss, 'Épocas', 'Perda Computada', '', True, 'r')

```

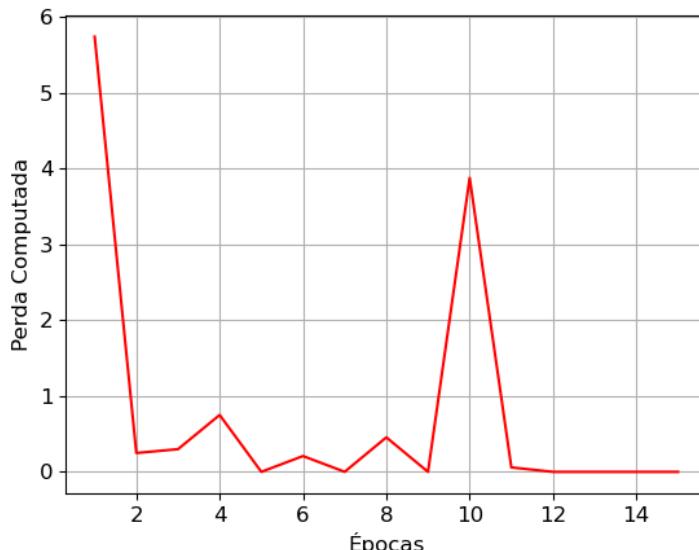
Fonte: o Autor.

Figura 106 - Precisão da rede conv através das épocas.



Fonte: o Autor.

Figura 107 – Perda computada da rede conv através das épocas.



Fonte: o Autor.

Analizando os resultados, nota-se o fenômeno chamado overfitting entre as épocas 5 e 12, onde a rede obtém resultados inferiores após ter treinado mais vezes. Após a época 12 o resultado se estabiliza em 100%, o que é um indício de que a rede está bem calibrada.

Usando o mesmo código das imagens de treino, foram geradas outras 300 imagens para um teste na rede. A figura a seguir ilustram os comandos usados e o resultado da avaliação.

Figura 108 – Comandos usados para obter o caminho do conjunto de testes.

```
>>> from pathlib import Path
>>> path = Path('.')
>>> path.resolve()
WindowsPath('C:/Users/binho/Desktop/Pythono/Interface')
>>> testepath = path/'datasets//geometricas//teste'
>>> testepath.resolve()
WindowsPath('C:/Users/binho/Desktop/Pythono/Interface/datasets/geometricas/teste')
```

Fonte: o Autor.

Figura 109 – Processo de avaliação do conjunto de testes.

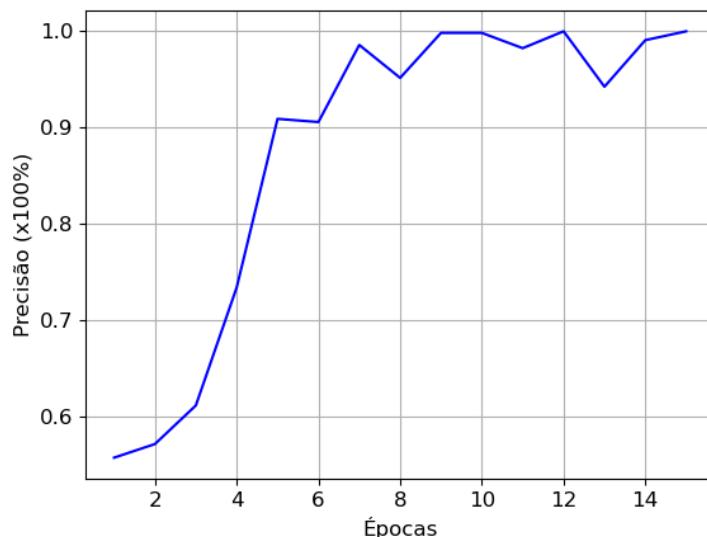
```
>>> gen = conv.generator
>>> testegen = gen.flow_from_directory(testepath, target_size=conv.input_size, batch_size=300)
Found 300 images belonging to 2 classes.
>>> conv.model.evaluate(testegen, batch_size=300, steps=1)
1/1 [=====] - 0s 961us/step - loss: 1.2252 - accuracy: 0.9100
[1.22524094581604, 0.910000262260437]
```

Fonte: o Autor.

A rede alcançou uma precisão de 91%, ou seja, errou em 27 das previsões. Isso mostra que mesmo alcançando os 100% sob o conjunto de treino por quatro épocas seguidas, a rede ainda está sob efeito de overfitting, ou seja, ela não aprendeu a identificar quadrados e triângulos genéricos, e sim os quadrados e triângulos do conjunto de testes.

A fim de resolver o problema foi alterada a função de ativação da primeira camada convolutiva, mudando de *relu* para *softplus*, sob a análise de que a rede estaria tendo problemas em encontrar as informações das figuras geométricas pois a cor preta, que é a cor dessas figuras, é representada por zeros nos canais, então a função poderia estar dificultando o aprendizado.

Figura 110 – Precisão durante o processo de treinamento da nova rede.



Fonte: o Autor.

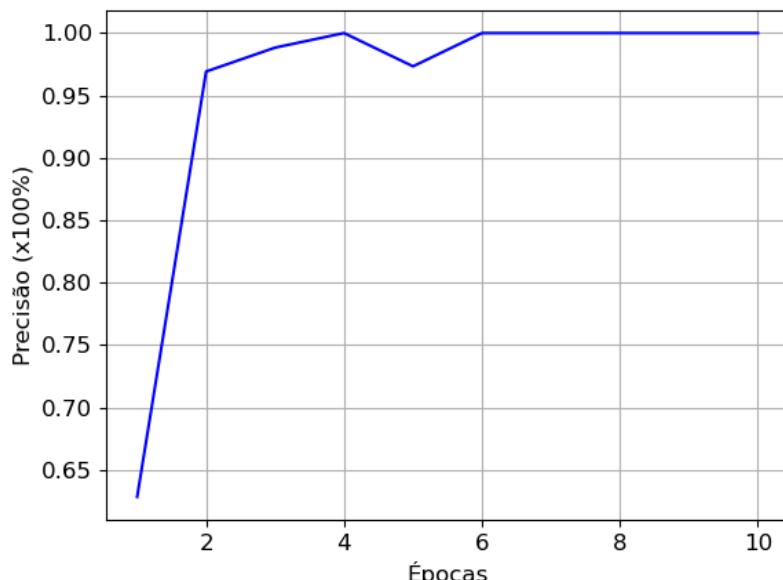
Figura 111 – Resultado da avaliação da rede sob o conjunto de testes.

```
>>> conv.model.evaluate(testegen, batch_size=30, steps=10)
10/10 [=====] - 1s 91ms/step - loss: 2.9673 - accuracy: 0.7967
[2.967339277267456, 0.79666668176651]
```

Fonte: o Autor.

A rede continuou com o mesmo problema, inclusive piorou de rendimento. Foi modificado o tamanho da janela de convolução que antes era de 5 pixels (Figuras 98 e 99), para 3 pixels, além do retorno da modificação de função de ativação da primeira camada convolutiva para *relu*. As figuras a seguir ilustram o resultado obtido:

Figura 112 – Precisão da rede com janelas de tamanho 3 através das épocas.

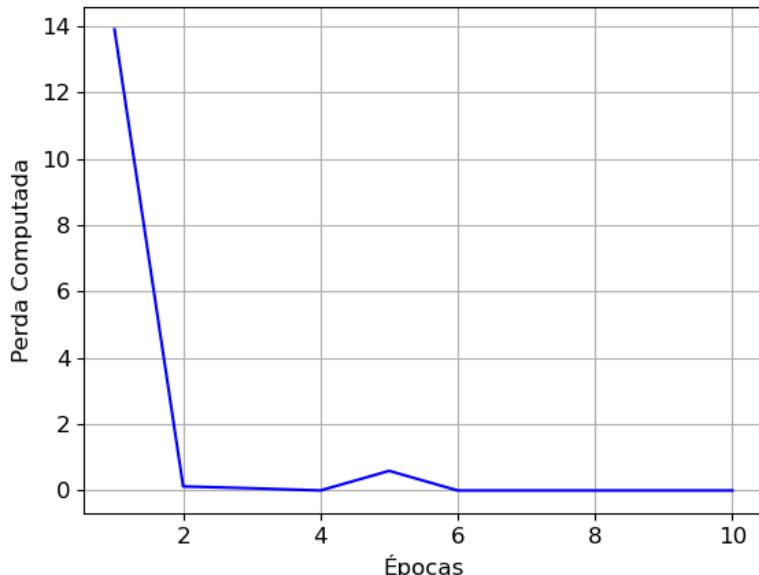


Fonte: o Autor.

O fenômeno de overfitting ainda foi identificado, porém a rede conseguiu se estabilizar nas duas últimas épocas. A perda computada reflete a precisão pois pode-se dizer que ambas

possuem uma relação de proporcionalidade inversa. Se uma rede erra muito, quer dizer que os resultados obtidos estão longe do valor desejado, pois é a média dos cálculos das funções de perda e suas respectivas derivadas parciais aplicadas peso a peso pelo algoritmo de *backpropagation* por todas as camadas da rede.

Figura 113 – Perda computada pela rede com janelas de tamanho 3 através das épocas.



Fonte: o Autor.

A rede ainda apresentou *overfitting*, como pode ser observado na época 5, porém foi mais estável que o anterior. O resultado da avaliação sob o conjunto de treino é ilustrado na Figura X31.

Figura 114 – Resultado da avaliação da rede sob o conjunto de treino.

```
>>> testegen = gen.flow_from_directory(testepath, target_size=conv.input_size, batch_size=30)
Found 300 images belonging to 2 classes.
>>> conv.model.evaluate(testegen, batch_size=30, steps=10)
10/10 [=====] - 1s 77ms/step - loss: 1.8904 - accuracy: 0.8600
[1.8903532028198242, 0.8600000143051147]
```

Fonte: o Autor.

A rede ainda assim falhou em 14% dos casos. Foi então reconfigurada a parte da rede responsável pela classificação, inserindo mais uma camada densa, além da inserção de uma camada *maxpooling* entre as duas camadas convolutivas e mudar o número de filtros da segunda camada convolutiva de 32 para 64, a fim de dar mais liberdade à rede em relação aos padrões que ela pode encontrar. A mudança pode ser feita remontando a rede no software ou modificando diretamente no código.

Figura 115 – Código da rede antes das mudanças.

```
30 #seção do modelo
31 model = models.Sequential()
32 model.add(layers.Conv2D(input_shape=input_shape,filters=32,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
33 model.add(layers.Conv2D(filters=32,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
34 model.add(layers.MaxPooling2D(pool_size=2,strides=1))
35 model.add(layers.Flatten())
36 model.add(layers.Dense(units=256,activation='relu'))
37 model.add(layers.Dense(units=2,activation='sigmoid'))
38 model.compile(optimizer='RMSprop',loss='binary_crossentropy',metrics=['accuracy'])
39 history = model.fit(traingen, steps_per_epoch = timesteps,epochs = 15)
```

Fonte: o Autor.

Figura 116 – Código da rede depois das mudanças.

```

32 #seção do modelo
33 model = models.Sequential()
34 model.add(layers.Conv2D(input_shape=input_shape,filters=32,kernel_size=5,strides=1,dilation_rate=1,activation='relu'))
35 ● model.add(layers.MaxPooling2D(pool_size=2,strides=1))
36 model.add(layers.Conv2D(filters=64,kernel_size=5,strides=1,dilation_rate=1,activation='relu'))
37 model.add(layers.MaxPooling2D(pool_size=2,strides=1))
38 model.add(layers.Flatten())
39 model.add(layers.Dense(units=256,activation='relu'))
40 ● model.add(layers.Dense(units=128,activation='relu'))
41 model.add(layers.Dense(units=2,activation='sigmoid'))
42 model.compile(optimizer='RMSprop',loss='binary_crossentropy',metrics=['accuracy'])
43 history = model.fit(traingen, steps_per_epoch = timesteps,epochs = 10)
44

```

Fonte: o Autor.

Eis os resultados:

Figura 117 – Processo de treinamento da segunda rede conv.

```

Epoch 1/10
120/120 [=====] - 314s 3s/step - loss: 5.0383 - accuracy: 0.6825
Epoch 2/10
120/120 [=====] - 312s 3s/step - loss: 0.1747 - accuracy: 0.9642
Epoch 3/10
120/120 [=====] - 316s 3s/step - loss: 0.1672 - accuracy: 0.9883
Epoch 4/10
120/120 [=====] - 313s 3s/step - loss: 0.0060 - accuracy: 0.9983
Epoch 5/10
120/120 [=====] - 313s 3s/step - loss: 0.1662 - accuracy: 0.9892
Epoch 6/10
120/120 [=====] - 316s 3s/step - loss: 2.7476e-05 - accuracy: 1.0000
Epoch 7/10
120/120 [=====] - 314s 3s/step - loss: 0.6423 - accuracy: 0.9783
Epoch 8/10
120/120 [=====] - 313s 3s/step - loss: 0.2717 - accuracy: 0.9817
Epoch 9/10
120/120 [=====] - 313s 3s/step - loss: 0.0372 - accuracy: 0.9933
Epoch 10/10
120/120 [=====] - 319s 3s/step - loss: 0.1692 - accuracy: 0.9883

```

Fonte: o Autor.

Figura 118 – Avaliação do conjunto de testes da segunda rede conv.

```

>>> gen = conv.generator
>>> from pathlib import Path
>>> testepath = Path('./datasets/geometricas/teste')
>>> testegen = gen.flow_from_directory(testepath,target_size=conv.input_size, batch_size=30)
Found 300 images belonging to 2 classes.
>>> conv.model.evaluate(testegen, batch_size = 30, steps=10)
10/10 [=====] - 5s 539ms/step - loss: 1.0257 - accuracy: 0.8800
[1.0256725549697876, 0.8799999952316284]

```

Fonte: o Autor.

O resultado ainda é pouco confiável. A solução proposta foi a geração de mais 1800 imagens para o conjunto de dados, totalizando um conjunto final de 3000 imagens. O código utilizado para a geração é o mesmo das Figuras 83 e 84. As figuras a seguir mostram a etapa de treinamento e o resultado da avaliação sob o conjunto de testes utilizando a mesma rede das figuras 116 e 117, porém agora com 3000 imagens.

Figura 119 – Processo de treinamento da rede com 3000 imagens.

```
Epoch 1/10
300/300 [=====] - 782s 3s/step - loss: 3.2270 - accuracy: 0.8530
Epoch 2/10
300/300 [=====] - 784s 3s/step - loss: 0.2880 - accuracy: 0.9860
Epoch 3/10
300/300 [=====] - 794s 3s/step - loss: 0.0177 - accuracy: 0.9970
Epoch 4/10
300/300 [=====] - 786s 3s/step - loss: 0.3968 - accuracy: 0.9897
Epoch 5/10
300/300 [=====] - 784s 3s/step - loss: 0.2092 - accuracy: 0.9940
Epoch 6/10
300/300 [=====] - 781s 3s/step - loss: 0.0225 - accuracy: 0.9990
Epoch 7/10
300/300 [=====] - 792s 3s/step - loss: 0.4044 - accuracy: 0.9870
Epoch 8/10
300/300 [=====] - 786s 3s/step - loss: 0.1056 - accuracy: 0.9957
Epoch 9/10
300/300 [=====] - 783s 3s/step - loss: 0.1433 - accuracy: 0.9977
Epoch 10/10
300/300 [=====] - 794s 3s/step - loss: 0.0410 - accuracy: 0.9987
```

Fonte: o Autor.

Figura 120 – Avaliação da rede com 3000 imagens sob o conjunto de testes.

```
>>> gen = conv.generator
>>> from pathlib import Path
>>> testepath = Path('./datasets/geometricas/teste')
>>> testepath.resolve()
WindowsPath('C:/Users/binho/Desktop/Pythono/Interface/datasets/geometricas/teste')
>>> testegen = gen.flow_from_directory(testepath, target_size=conv.input_size, batch_size = 30)
Found 300 images belonging to 2 classes.
>>> conv.model.evaluate(testegen, batch_size=30, steps=10)
10/10 [=====] - 5s 540ms/step - loss: 0.0669 - accuracy: 0.9800
[0.0668923407793045, 0.9800000190734863]
```

Fonte: o Autor.

Apesar de não alcançar 100% de precisão sob o conjunto de treino em nenhum momento, a rede obteve 98% de precisão na avaliação sob o conjunto de testes. Isso mostra que o overfitting neste caso é bem menor que nos anteriores, pois os resultados obtidos no teste e no conjunto são similares, ou seja, a rede foi capaz de generalizar melhor os casos. Nota-se também a diferença de tempo levado para o processamento, que aparecem nos números 782s, 784s, ..., que é a medição em segundos para cada época. Para a rede com 1200 os tempos ficaram no intervalo de 300 a 400 segundos, metade dos atuais. A figura a seguir mostra o sumário da rede.

Figura 121 – Sumário da rede.

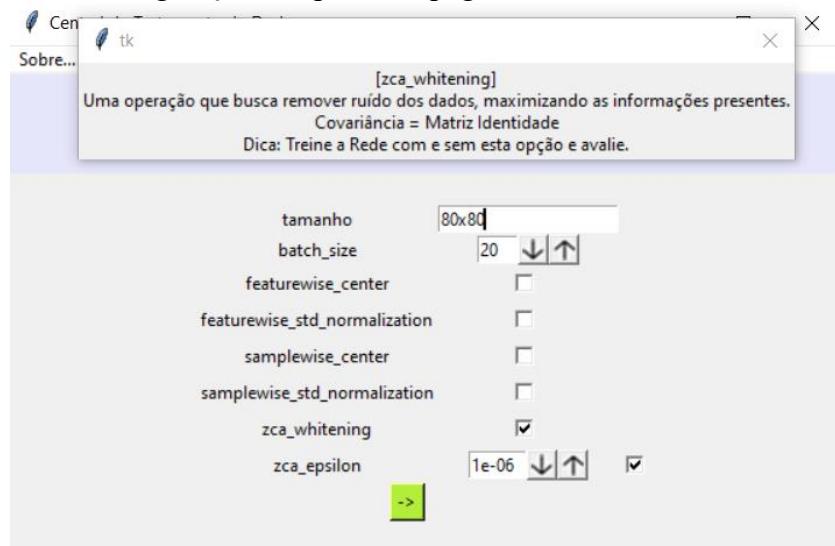
```
>>> conv.model.summary()
Model: "sequential"

Layer (type)                 Output Shape              Param #
=====
conv2d (Conv2D)             (None, 124, 124, 32)      2432
max_pooling2d (MaxPooling2D) (None, 123, 123, 32)      0
conv2d_1 (Conv2D)            (None, 119, 119, 64)     51264
max_pooling2d_1 (MaxPooling2D) (None, 118, 118, 64)      0
flatten (Flatten)           (None, 891136)          0
dense (Dense)               (None, 256)                228131072
dense_1 (Dense)              (None, 128)                32896
dense_2 (Dense)              (None, 2)                  258
=====
Total params: 228,217,922
Trainable params: 228,217,922
Non-trainable params: 0
```

Fonte: o Autor.

Com a finalidade de alcançar uma performance de 100% foi reconfigurado o processo de tratamento dos dados. A figura a seguir ilustra as novas configurações na primeira página do menu de tratamento de dados do software. A segunda página é a mesma da figura X7.

Figura 122 – Configurações da primeira página do menu de tratamento de dados.



Fonte: o Autor.

Foi alterado o tamanho para uma melhora no tempo de processamento, removida a *samplewise\_std\_normalization* e adicionada a operação *zca\_whitening*, que como define a caixa de texto da figura 122, é um processamento que busca remover ruídos nos dados.

Figura 123 – Etapa de treinamento da rede final.

```
Epoch 1/8
300/300 [=====] - 446s 1s/step - loss: 1.4764 - accuracy: 0.9020
Epoch 2/8
300/300 [=====] - 457s 2s/step - loss: 0.2369 - accuracy: 0.9927
Epoch 3/8
300/300 [=====] - 447s 1s/step - loss: 0.1022 - accuracy: 0.9927
Epoch 4/8
300/300 [=====] - 446s 1s/step - loss: 0.0800 - accuracy: 0.9927
Epoch 5/8
300/300 [=====] - 444s 1s/step - loss: 7.4842e-09 - accuracy: 1.0000
Epoch 6/8
300/300 [=====] - 446s 1s/step - loss: 6.3145e-11 - accuracy: 1.0000
Epoch 7/8
300/300 [=====] - 447s 1s/step - loss: 1.9479e-11 - accuracy: 1.0000
Epoch 8/8
300/300 [=====] - 446s 1s/step - loss: 2.9397e-12 - accuracy: 1.0000
```

Fonte: o Autor.

Já é possível notar nesta etapa a ausência de overfitting na rede. Em momento algum a perda computada em uma época foi maior do que a anterior, tampouco a precisão foi menor. A Figura a seguir ilustra a avaliação sob o mesmo conjunto de testes usado nas redes anteriores.

Figura 124 – Avaliação sob o conjunto de testes da rede final.

```
>>> from pathlib import Path
>>> testepath = Path('./datasets/geometricas/teste')
>>> gen = conv.generator
>>> testegen = gen.flow_from_directory(testepath, target_size=conv.input_size, batch_size=30)
Found 300 images belonging to 2 classes.
>>> conv.model.evaluate(testegen, batch_size=30, steps=10)
10/10 [=====] - 18s 2s/step - loss: 0.0012 - accuracy: 1.0000
[0.001199397025629878, 1.0]
```

Fonte: o Autor.

Isso prova que, além da diferenciação de cores, a rede se mostra útil em tarefas de identificação de estruturas que podem estar presentes em qualquer lugar da imagem através do uso de camadas convolutivas.

### 5.3 Diagnósticos Pulmonares

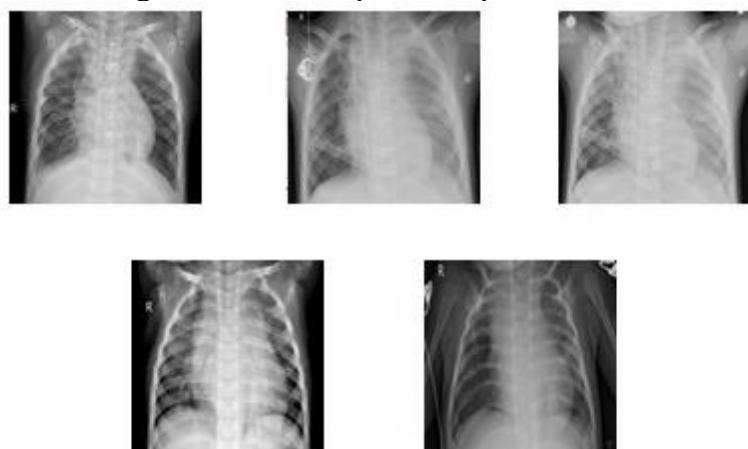
A fim de validar o uso de Redes Neurais Profundas em tarefas relacionadas à medicina, foi realizado o treinamento de uma rede com o objetivo de classificar imagens de raios-x de tórax em amostras que indicam pneumonia ou não. Se trata de um problema de classificação binária.

#### 5.3.1 Os dados

O conjunto de dados consiste em 5232 imagens de raios-x de tórax de crianças para a etapa de treinamento, sendo 1349 de saudáveis e 3883 com pneumonia, e outras 624 imagens com 234 saudáveis e 390 doentes para o conjunto de testes, todas coletadas e classificadas pelos pesquisadores Daniel Kermany, Michael Goldbaum e Kang Zhang [4].

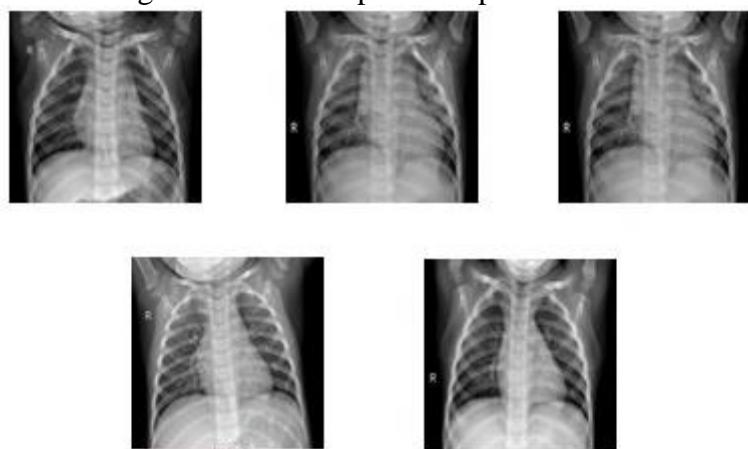
As imagens não têm um padrão de tamanho, portanto foram todas redimensionadas para 226x226 pixels e no processo alguns arquivos estavam corrompidos, sobrando 5212 imagens para o treinamento, sendo 1339 saudáveis e 3873 doentes. As figuras a seguir ilustram algumas das imagens.

Figura 125 – Exemplos com pneumonia.



Fonte: o Autor.

Figura 126 – Exemplos sem pneumonia.

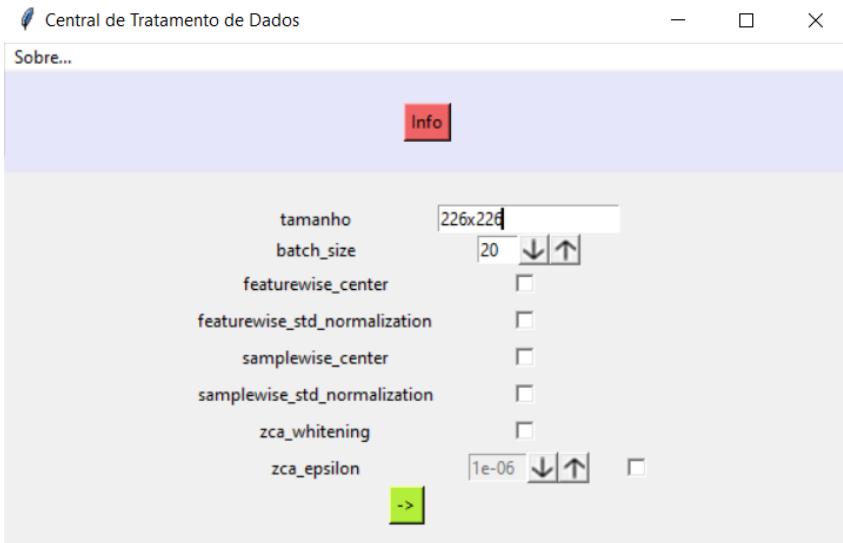


Fonte: o Autor.

### 5.3.2 As Redes

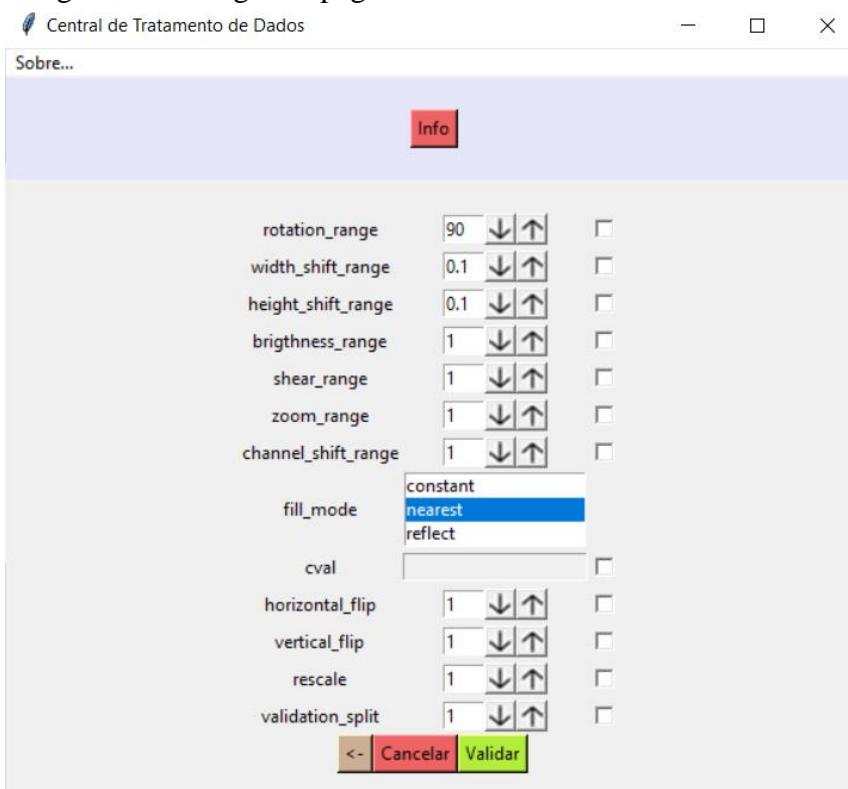
A primeira tentativa não realizou nenhuma operação na etapa de tratamento de dados além do redimensionamento e usou lotes de tamanho 20, ou seja, houve  $5212 / 20 = 260$  iterações (alterações nos pesos) por época. As figuras a seguir ilustram os menus de configuração do tratamento de dados, do modelo, o código e o sumário da rede.

Figura 127 – Primeira página do menu de tratamento de dados.



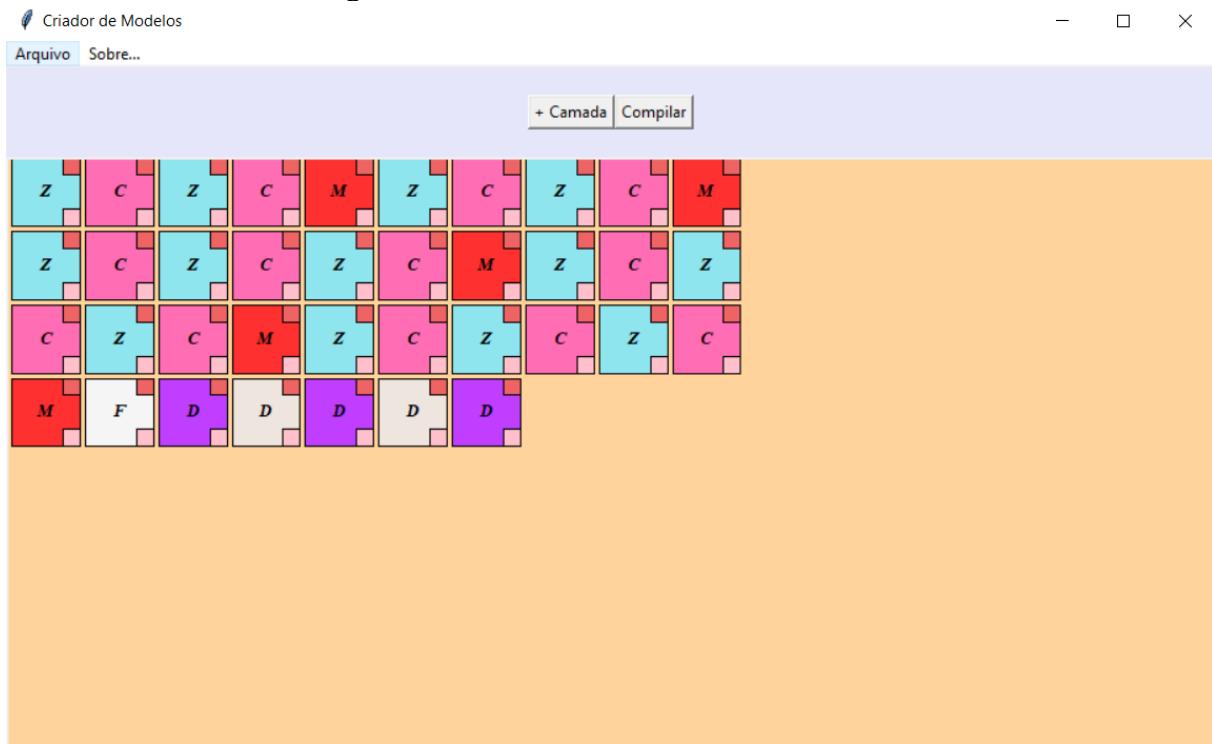
Fonte: o Autor.

Figura 128 – Segunda página do menu de tratamento de dados.



Fonte: o Autor.

Figura 129 – Modelo da rede e suas camadas.



Fonte: o Autor.

Figura 130 – Primeira parte do código gerado para a rede.

```

1  import tensorflow as tf
2  import numpy as np
3  from keras import layers
4  from keras import models
5  from keras.preprocessing import image
6  from pathlib import Path
7  import csv
8  from PIL import Image
9  from keras.utils import to_categorical
10 import os
11 #seção da escolha dos dados
12 path='C:/Users/binho/Desktop/Pythono/Interface/datasets/chest_xray/treino'
13 path = Path(path)
14 labels = 'inferred'
15 itenes = list(path.rglob('*.*'))
16 features = len(itenes)
17 input_size = (226,226,3)
18 #seção do tratamento de dados
19 generator = image.ImageDataGenerator(fill_mode='nearest')
20 batch_size = 20
21 input_size = (226,226)
22 timesteps = int(features/batch_size)
23 for x in range(0,len(itenes)):
24     itenes[x] = Image.open(itenes[x])
25     itenes[x] = itenes[x].resize(size=input_size)
26     itenes[x] = np.asarray(itenes[x])
27 itenes = np.asarray(itenes)
28 y = itenes.shape
29 if(len(y)==3):
30     itenes = np.reshape(itenes, (y[0],y[1],y[2],1))
31 generator.fit(itenes)
32 traingen = generator.flow_from_directory(path, target_size=input_size,batch_size=batch_size)
33 #seção do modelo da arquitetura
34 model = models.Sequential()
35 model.add(layers.ZeroPadding2D(input_shape=(226,226,3), padding=1))
36 model.add(layers.Conv2D(filters=64,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
37 model.add(layers.ZeroPadding2D(1))
38 model.add(layers.Conv2D(filters=64,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
39 model.add(layers.MaxPooling2D(pool_size=2,strides=2))
40 model.add(layers.ZeroPadding2D(1))
41 model.add(layers.Conv2D(filters=128,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
42 model.add(layers.ZeroPadding2D(1))

```

Fonte: o Autor.

Figura 131 – Segunda parte do código gerado para a rede.

```

42 model.add(layers.ZeroPadding2D(1))
43 model.add(layers.Conv2D(filters=128,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
44 model.add(layers.MaxPooling2D(pool_size=2,strides=2))
45 model.add(layers.ZeroPadding2D(1))
46 model.add(layers.Conv2D(filters=256,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
47 model.add(layers.ZeroPadding2D(1))
48 model.add(layers.Conv2D(filters=256,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
49 model.add(layers.ZeroPadding2D(1))
50 model.add(layers.Conv2D(filters=256,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
51 model.add(layers.MaxPooling2D(pool_size=2,strides=2))
52 model.add(layers.ZeroPadding2D(1))
53 model.add(layers.Conv2D(filters=512,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
54 model.add(layers.ZeroPadding2D(1))
55 model.add(layers.Conv2D(filters=512,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
56 model.add(layers.ZeroPadding2D(1))
57 model.add(layers.Conv2D(filters=512,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
58 model.add(layers.MaxPooling2D(pool_size=2,strides=2))
59 model.add(layers.ZeroPadding2D(1))
60 model.add(layers.Conv2D(filters=512,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
61 model.add(layers.ZeroPadding2D(1))
62 model.add(layers.Conv2D(filters=512,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
63 model.add(layers.ZeroPadding2D(1))
64 model.add(layers.Conv2D(filters=512,kernel_size=3,strides=1,dilation_rate=1,activation='relu'))
65 model.add(layers.MaxPooling2D(pool_size=2,strides=2))
66 model.add(layers.Flatten())
67 model.add(layers.Dense(units=4096,activation='relu'))
68 model.add(layers.Dropout(0.5))
69 model.add(layers.Dense(units=4096,activation='relu'))
70 model.add(layers.Dropout(0.5))
71 model.add(layers.Dense(units=2,activation='softmax'))
72 model.compile(optimizer='Adam',loss='categorical_crossentropy',metrics=['accuracy'])
73 model.summary()
74 history = model.fit(traingen, steps_per_epoch = timesteps, epochs = 5)

```

Fonte: o Autor.

Figura 132 – Primeira parte do sumário da rede.

Model: "sequential"		
Layer (type)	Output Shape	Param #
zero_padding2d (ZeroPadding2D)	(None, 228, 228, 3)	0
conv2d (Conv2D)	(None, 226, 226, 64)	1792
zero_padding2d_1 (ZeroPadding2D)	(None, 228, 228, 64)	0
conv2d_1 (Conv2D)	(None, 226, 226, 64)	36928
max_pooling2d (MaxPooling2D)	(None, 113, 113, 64)	0
zero_padding2d_2 (ZeroPadding2D)	(None, 115, 115, 64)	0
conv2d_2 (Conv2D)	(None, 113, 113, 128)	73856
zero_padding2d_3 (ZeroPadding2D)	(None, 115, 115, 128)	0
conv2d_3 (Conv2D)	(None, 113, 113, 128)	147584
max_pooling2d_1 (MaxPooling2D)	(None, 56, 56, 128)	0
zero_padding2d_4 (ZeroPadding2D)	(None, 58, 58, 128)	0
conv2d_4 (Conv2D)	(None, 56, 56, 256)	295168
zero_padding2d_5 (ZeroPadding2D)	(None, 58, 58, 256)	0
conv2d_5 (Conv2D)	(None, 56, 56, 256)	590080
zero_padding2d_6 (ZeroPadding2D)	(None, 58, 58, 256)	0
conv2d_6 (Conv2D)	(None, 56, 56, 256)	590080
max_pooling2d_2 (MaxPooling2D)	(None, 28, 28, 256)	0
zero_padding2d_7 (ZeroPadding2D)	(None, 30, 30, 256)	0
conv2d_7 (Conv2D)	(None, 28, 28, 512)	1180160
zero_padding2d_8 (ZeroPadding2D)	(None, 30, 30, 512)	0

Fonte: o Autor.

Figura 133 – Segunda parte do sumário da rede.

conv2d_8 (Conv2D)	(None, 28, 28, 512)	2359808
zero_padding2d_9 (ZeroPaddin	(None, 30, 30, 512)	0
conv2d_9 (Conv2D)	(None, 28, 28, 512)	2359808
max_pooling2d_3 (MaxPooling2	(None, 14, 14, 512)	0
zero_padding2d_10 (ZeroPaddi	(None, 16, 16, 512)	0
conv2d_10 (Conv2D)	(None, 14, 14, 512)	2359808
zero_padding2d_11 (ZeroPaddi	(None, 16, 16, 512)	0
conv2d_11 (Conv2D)	(None, 14, 14, 512)	2359808
zero_padding2d_12 (ZeroPaddi	(None, 16, 16, 512)	0
conv2d_12 (Conv2D)	(None, 14, 14, 512)	2359808
max_pooling2d_4 (MaxPooling2	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 4096)	102764544
dropout (Dropout)	(None, 4096)	0
dense_1 (Dense)	(None, 4096)	16781312
dropout_1 (Dropout)	(None, 4096)	0
dense_2 (Dense)	(None, 2)	8194
=====		
Total params:	134,268,738	
Trainable params:	134,268,738	
Non-trainable params:	0	

Fonte: o Autor.

Ao todo são 30 camadas, dentre elas 13 convolutivas, 13 ZeroPadding, 5 MaxPooling, 3 densas, 1 retificadora e 2 dropouts. A rede possui 134.268.738 parâmetros treináveis. As configurações usadas em cada camada podem ser visualizadas através do código. A escolha das camadas foi baseada no modelo VGG16, criado em 2014 por Karen Simonyan e Andrew Zisserman, que ultrapassou os melhores resultados obtidos até então em competições de reconhecimento de imagens de 2012 e 2013 [1]. As figuras a seguir ilustram a etapa de treinamento e avaliação sob o conjunto de testes:

Figura 134 – Etapa de treinamento da primeira rede.

```
Epoch 1/5
260/260 [=====] - 4625s 18s/step - loss: 20.0390 - accuracy: 0.7354
Epoch 2/5
260/260 [=====] - 4627s 18s/step - loss: 0.4854 - accuracy: 0.7924
Epoch 3/5
260/260 [=====] - 4580s 18s/step - loss: 0.3361 - accuracy: 0.8555
Epoch 4/5
260/260 [=====] - 4575s 18s/step - loss: 0.4471 - accuracy: 0.8080
Epoch 5/5
260/260 [=====] - 4559s 18s/step - loss: 0.5725 - accuracy: 0.7436
```

Fonte: o Autor.

Figura 135 – Avaliação da primeira rede sob o conjunto de testes.

```
>>> from pathlib import Path
>>> testepath = Path('./datasets/chest_xray/teste')
>>> gen = raiox.generator
>>> testegen = gen.flow_from_directory(testepath,target_size=raiox.input_size, batch_size=10)
Found 624 images belonging to 2 classes.
>>> testegen = gen.flow_from_directory(testepath,target_size=raiox.input_size, batch_size=4)
Found 624 images belonging to 2 classes.
>>> raiox.model.evaluate(testegen, batch_size=4, steps=624/4)
156/156 [=====] - 161s 1s/step - loss: 0.7322 - accuracy: 0.6250
[0.7322285771369934, 0.625]
>>>
```

Fonte: o Autor.

Aqui é possível observar o overfitting a partir da terceira camada, além da queda de rendimento no conjunto de testes. Nota-se também o tempo necessário para o processamento, que no exemplo de formas geométricas ficou em torno dos 400s por época, enquanto nesta rede este tempo foi para 4500s, aproximadamente 1 hora e 15 minutos por época em um computador com um processador Intel i7 de sétima geração, com 2.7 GHz de frequência base de trabalho.

A fim de encontrar melhores resultados foi realizado um novo treinamento, porém com lotes de tamanho 4 e a opção *samplewise\_center* do tratamento de dados ativada, que centraliza os valores dos pixels da imagem em 0. As figuras a seguir ilustram os resultados:

Figura 136 – Etapa de treinamento da segunda rede.

```
Epoch 1/5
1303/1303 [=====] - 5793s 4s/step - loss: 1.0167 - accuracy: 0.7433
Epoch 2/5
1303/1303 [=====] - 5751s 4s/step - loss: 0.5735 - accuracy: 0.7431
Epoch 3/5
1303/1303 [=====] - 5766s 4s/step - loss: 0.5734 - accuracy: 0.7431
Epoch 4/5
1303/1303 [=====] - 5762s 4s/step - loss: 0.5727 - accuracy: 0.7431
Epoch 5/5
1303/1303 [=====] - 6247s 5s/step - loss: 0.5727 - accuracy: 0.7431
```

Fonte: o Autor.

Figura 137 – Avaliação da segunda rede sob o conjunto de testes.

```
>>> raiox2.model.evaluate(testegen, batch_size=4, steps = 624/4)
156/156 [=====] - 180s 1s/step - loss: 0.6849 - accuracy: 0.6250
[0.6849161386489868, 0.625]
>>> from pathlib import Path
>>> testepath = Path('./datasets/chest_xray/teste')
>>> gen = raiox2.generator
>>> testegen = gen.flow_from_directory(testepath, target_size=raiox2.input_size, batch_size=4)
Found 624 images belonging to 2 classes.
>>> raiox2.model.evaluate(testegen, batch_size=4, steps = 624/4)
156/156 [=====] - 179s 1s/step - loss: 0.6849 - accuracy: 0.6250
[0.6849159002304077, 0.625]
```

Fonte: o Autor.

Não houve nenhuma melhoria significativa no resultado obtido, e a rede aparentemente deixou de aprender já na segunda época, repetindo a performance no restante do processo. Ambas as avaliações sob o conjunto de testes obtiveram a mesma precisão, 62.5%.

Em um estudo mais detalhado [5], os mesmos pesquisadores obtiveram resultados acima de 90% para ambos os conjuntos, o de teste e o de treinamento, em um ensaio com mais de 100 épocas. Devido à capacidade limitada de processamento não foram realizados mais testes.

## 5.4 Diagnósticos de câncer em tecidos histopatológicos

Foi realizado o treinamento de uma rede com o objetivo de classificar imagens de tecidos histopatológicos em amostras que indicam metástase ou não. Se trata de outro problema de classificação binária.

### 5.4.1 Os dados

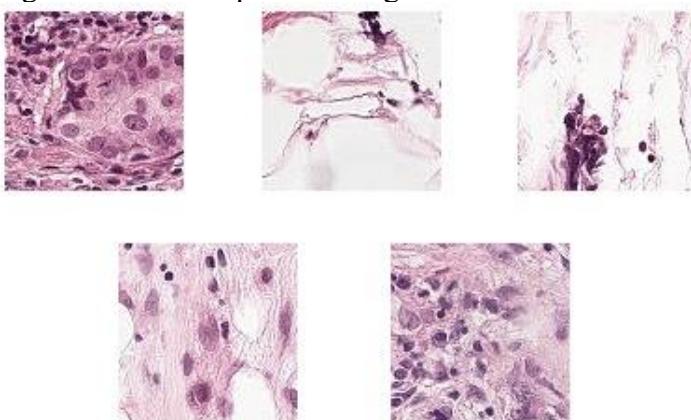
O termo **Histopatologia**, ou **Histologia Patológica**, se refere ao exame microscópico de tecidos biológicos capaz de observar o aparecimento de células cancerosas em amostras finas de tecidos doentes. Ao contrário da Citologia, onde as amostras são analisadas sem serem processadas, a Histopatologia exige um dos dois seguintes procedimentos: a fixação química na qual as amostras de tecido são imersas em um banho de parafina ou cera, por um período de 12 a 16 horas, permitindo que o tecido seja cortado em seções de dois a sete micrômetros para o exame, ou o segundo que é o corte congelado, onde as amostras de tecido são congeladas e também cortadas em fatias finas.

Os dados usados neste experimento fazem parte dos conjuntos de dados PCam e Camelyon16. As imagens dos nós linfáticos foram adquiridas de 399 pacientes diferentes pacientes de dois diferentes hospitais holandeses: RUMC[sigla] e UMCU[sigla], com duas máquinas diferentes. Todas as classificações foram feitas sob a supervisão de patologistas experientes. A classificação foi primeiramente feita por dois estudantes (um de cada hospital) e depois revisadas, uma a uma, pelos especialistas. [3]

A classificação busca encontrar gânglios sentinelas, também chamados de linfonodos sentinelas, que são os prováveis primeiros órgãos invadidos por um câncer em metástase. Os linfonodos são pequenos órgãos perfurados por canais que fazem parte da rede linfática, atuando na defesa do organismo e produção de anticorpos. Nesse sentido, os linfonodos sentinelas são aqueles que indicam que uma metástase está acontecendo, pois serão os primeiros a serem afetados. [2]

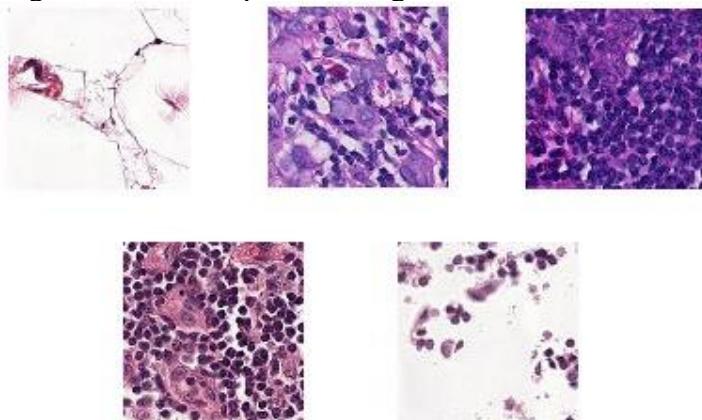
O conjunto total de imagens está disponível no portal Kaggle, que é uma plataforma online onde pessoas do mundo todo podem enviar implementações de redes e conjuntos de dados para treinamento, além de competir entre si. O conjunto de imagens de tecido histopatológico consiste em 277.483 imagens de 32x32 pixels, cada uma acompanhada de um rótulo ‘0’ ou ‘1’, onde 0 significa que não há naquela amostra nenhum pixel que apresente um tecido doente e ‘1’ que há ao menos um pixel onde o câncer está presente. São 188.366 exemplos saudáveis e 89.117 exemplos doentes, que estão divididos em duas partições: uma para o treinamento da rede e outra para os devidos testes. A partição para treinamento é composta por 220.025 imagens, com 70.000 delas rotuladas como ‘1’ e outras 150.025 rotuladas como ‘0’, e a partição para teste será composta por 57.458 imagens, com 19.117 delas rotuladas como ‘1’ e outras 38.341 rotuladas como ‘0’.

Figura 138 – exemplos de imagens rotuladas com ‘1’.



Fonte: o Autor.

Figura 139 – exemplos de imagens rotuladas com ‘0’.



Fonte: o Autor.

#### 5.4.2 A Rede

Devido ao grande volume de dados (mais de 220000 imagens para treino), a estimativa de tempo para cada época era de aproximadamente 11 horas, então foi realizado um treinamento com apenas 10000 das imagens, sendo 5000 saudáveis e 5000 doentes. A rede usada foi a mesma da tarefa de diagnósticos de pneumonia. As figuras a seguir ilustram o processo de treinamento e avaliação de outras 2000 imagens como conjunto de testes.

Figura 140 – Etapa de treinamento da rede para detecção de metástase.

```
Epoch 1/5
500/500 [=====] - 2208s 4s/step - loss: 10.8713 - accuracy: 0.5033
Epoch 2/5
500/500 [=====] - 1978s 4s/step - loss: 0.6934 - accuracy: 0.5052
Epoch 3/5
500/500 [=====] - 2208s 4s/step - loss: 0.6940 - accuracy: 0.4944
Epoch 4/5
500/500 [=====] - 2246s 4s/step - loss: 0.6934 - accuracy: 0.5061
Epoch 5/5
500/500 [=====] - 2209s 4s/step - loss: 0.6932 - accuracy: 0.5089
```

Fonte: o Autor.

Figura 141 – Avaliação sob o conjunto de testes da rede para detecção de metástase.

```
>>> from pathlib import Path
>>> testepath = Path('./datasets/cancer10k/teste')
>>> testepath.resolve()
WindowsPath('C:/Users/binho/Desktop/Pythono/Interface/datasets/cancer10k/teste')
>>> gen = cancer.generator
>>> testegen = gen.flow_from_directory(testepath, target_size=cancer.input_size, batch_size = 20)
Found 4000 images belonging to 2 classes.
>>> cancer.model.evaluate(testegen)
200/200 [=====] - 301s 2s/step - loss: 0.6933 - accuracy: 0.5000
[0.693293571472168, 0.5]
>>>
```

Fonte: o Autor.

Como era de se esperar após a redução no conjunto de dados em 95%, não foram obtidos resultados significantes. Porém, no site de competições Kaggle muitos usuários alcançaram precisões maiores que 90% em suas redes, disponíveis no link <https://www.kaggle.com/c/histopathologic-cancer-detection>, acesso em 14/11/2020.

## 6 CONCLUSÕES

### 6.1 Tarefas Médicas

O Overfitting é a principal razão de não-aplicabilidade das Redes Neurais Profundas em ambientes reais de hospitais e clínicas, pois ele indica que esses dados não podem ser interpretados como a mais pura verdade, já que está comparando-o com os exemplos que conhece, e não classificando com certeza. Outros estudos dos casos usados nesta pesquisa mostram eficiências de até 98 %, porém em casos médicos tomar uma decisão a partir de uma análise que tem uma margem de erro de 2% não é admissível, pois se tratam de vidas. Apesar de um médico ainda não ser capaz de confiar nas Redes e tomar suas classificações como absolutas, o uso das mesmas pode agilizar muitos processos, podendo ser usadas como uma etapa de pré-processamento de dados que separaria as amostras que provavelmente seriam relevantes das que não são após uma certa quantidade de aquisições de imagens de Raio-X ou amostras de tecidos de linfonodos, porém ainda se faria necessária a confirmação humana de todas as classificações da rede.

### 6.2 Tarefas Simples

Foi demonstrado nos dois primeiros exemplos que a rede é capaz de classificar com precisão dados em categorizações mais simples, o que é um bom indício para seu uso em tarefas cada vez mais complexas. O uso de RNPs pode trazer muitos avanços para a humanidade, e o estudo de suas capacidades deve ser algo a ser incentivado pelas próximas décadas.

### 6.3 O software

O software desenvolvido se mostrou funcional, capaz de gerar códigos para RNPs sem problemas. Não foi incorporado ao software a etapa de avaliação da rede, tampouco configurações mais avançadas como o congelamento de camadas específicas, uso de pesos pré-determinados por outras redes, a construção não sequencial de modelos e a avaliação de outros tipos de dados além de imagens, porém serve como uma prova de conceito que tal abstração das RNPs é possível e pode ser de grande utilidade para o futuro da tecnologia.

O software pode ser melhorado implementando todas as funções acima descritas, ou ser reconstruído sob outra plataforma, como uma plataforma web, de forma que fique ainda mais acessível e dinâmico.

Todo o código desenvolvido se encontra disponível em um repositório GitHub que pode ser acessado através do link [https://github.com/fabiorx1/iniciacao\\_cientifica](https://github.com/fabiorx1/iniciacao_cientifica).

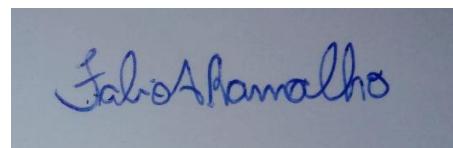
## REFERÊNCIAS BIBLIOGRÁFICAS

- [1] François Chollet (2018). "Deep Learning with Python." Manning Publications Co. ISBN: 9781617294433.
- [2] Mannu, G.S.; Navi, A.; Morgan, A.; Mirza, S.M.; Down, S.K.; Farooq, N.; Burger, A.; Hussien, M.I. (2012). "Sentinel lymph node biopsy before mastectomy and immediate breast reconstruction may predict post-mastectomy radiotherapy, reduce delayed complications and improve the choice of reconstruction". International Journal of Surgery. 10 (5): 259–64. doi:10.1016/j.ijsu.2012.04.010. [PMID 22525383](#).
- [3] Ehteshami Bejnordi et al. "Diagnostic Assessment of Deep Learning Algorithms for Detection of Lymph Node Metastases in Women With Breast Cancer." JAMA: The Journal of the American Medical Association, 318(22), 2199–2210.
- [4] Kermany, Daniel; Zhang, Kang; Goldbaum, Michael (2018), "Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images for Classification", Mendeley Data, V2, doi: 10.17632/rscbjbr9sj.2
- [5] Kermany, Daniel; Zhang, Kang; Goldbaum, Michael; Cai, Wenjia (2018), "Identifying Medical Diagnoses and Treatable by Image-Based Deep Learning", Cell, Volume 172, Issue 5, P1122-1131.E9, doi: 10.1016/j.cell.2018.02.010

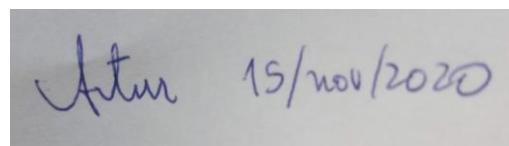
## APÊNDICE A – ASSINATURA DO ALUNO E DO ORIENTADOR

Este relatório apresenta o registro da pesquisa intitulada “Modelagem matemática para elucidar o potencial de redução de dose de Inibidores de Tirosina-Quinase no tratamento de Leucemia Mieloide Crônica”, desenvolvida no período de 05-09-2019 a 15-11-2020, sob a orientação do Prof. Artur César Fassoni, tendo em vista as orientações estipuladas pelo Edital Nº 004/2019.

Itajubá, 15 de novembro de 2020.



Fabio Augusto Ramalho



Artur César Fassoni