PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS NÚCLEO DE EDUCAÇÃO A DISTÂNCIA

Pós-graduação Lato Sensu em Inteligência Artificial e Aprendizado de Máquina

Alan Henrique Ferreira

REDES NEURAIS CONVOLUCIONAIS APLICADAS AO CONTROLE DO MÍLDIO NA CULTURA DE UVAS

Petrópolis, 09 de Agosto de 2021

Alan Henrique Ferreira

REDES NEURAIS CONVOLUCIONAIS APLICADAS AO CONTROLE DO MÍLDIO NA CULTURA DE UVAS

Trabalho apresentado ao Curso de Especialização em Inteligência Artificial e Aprendizado de Máquina, como requisito para obtenção do título de especialista.

Petrópolis, 09 de Agosto de 2021

Sumário

1 Introdução

A computação visual vem sendo aplicada em diversas áreas de conhecimento devido ao imenso conjunto de atividades desempenhadas a partir da visão humana e também devido ao baixo custo na operacionalização de algumas soluções que muitas das vezes utilizam uma infraestrutura preexistente. Favorecida pela vasta oferta de poder computacional no advento da computação em nuvem, de novas tecnologias de *hardware* e sistemas embarcado, muitas atividades antes realizadas por humanos puderam ser automatizadas com um razoável nível de confiança. Assim como na indústria, na medicina e entretenimento, a agricultura também vem buscando absorver parte dessa tecnologia para apoiar seu processo produtivo, principalmente na área de agricultura de precisão onde o tratamento químico é feito de forma pontual e em quantidades adequadas.

No cultivo da uva por exemplo, regiões tropicais como o Brasil acabam enfrentado um grande desafio no controle de fungos. O míldio, doença causada pelo fungo *Plasmopara vitícola,* é uma das ameaças constantes na cultura da uva em regiões com alta umidade e temperaturas de moderadas a quentes. Um dos danos mais comuns causados pelo míldio é a queda prematura das folhas da videira. Essa condição acaba resultando em um desenvolvimento anormal do fruto que pode sofrer uma perda de produção de até 75% se as plantas não forem pulverizadas e podadas logo no início da disseminação (Cultiva, (2001)).

Este trabalho tem por objetivo, a experimentação de modelos preditivos utilizando técnicas de Visão Computacional a fim de auxiliar o monitoramento e controle do míldio. A partir dos resultados fornecidos pelo modelo, o agricultor teria condições de identificar com precisão os locais mais afetados pela doença a fim de tomar medidas de contenção no avanço e disseminação da mesma.

1.1 Organização do Trabalho

O trabalho foi estruturado da seguinte maneira:

No Capítulo 2, serão apresentados um conjunto de definições acerca dos conceitos e técnicas utilizadas na área de Visão Computacional. O objetivo é oferecer uma visão mais generalista das técnicas empregadas e introduzir alguns fundamentos teóricos necessários para uma interpretação mais profunda acerca dos resultados obtidos por este trabalho.

O Capítulo 3 apresenta as ferramentas e abordagens utilizadas para implementação dos modelos propostos, como foram concebidos os dados necessários para o treinamento, qual a estratégia definida para a etapa de experimentação dos modelos, escolha e análise de métricas e quais as ações tomadas para a melhoria dos resultados.

O Capítulo 4 discute sobre a arquitetura implementada para os testes do modelo em campo assim como algumas sugestões de operacionalização do sistema.

Encerrando este trabalho, no Capítulo 5 se encontram algumas conclusões sobre a metodologia utilizada, os resultados obtidos e as dificuldades encontradas.

2 Visão Computacional e Inteligência Artificial

2.1 Introdução à Visão Computacional

Podemos definir Visão Computacional como a ciência que estuda e desenvolve ferramentas que proporcionem as máquinas a capacidade de enxergar e extrair características do meio, através de imagens capturadas por diferentes tipos de sensores e dispositivos (Brown, 1982).

Assim como câmeras fotográficas, aparelhos de raios-x e outras tecnologias desenvolvidas nos primórdios da era contemporânea, o principal objetivo da Visão Computacional sempre foi tentar imitar a natureza humana. Pensava-se no passado que logo seria possível criar uma máquina capaz de reproduzir a visão humana de forma completa, mas a principal barreira para tal propósito sempre foi a rasa compreensão de como as imagens são interpretadas pelo cérebro humano.

O grande salto na área de Visão Computacional foi justamente alavancado por pesquisas científicas que buscavam entender o funcionamento do cérebro e sua relação com a visão. Além da neurobiologia, diversos outros campos de estudo contribuem para evolução da Visão Computacional. A Inteligência Artificial é um dos campos de estudo que frequentemente colaboram para esse desenvolvimento tecnológico, principalmente na indústria de dispositivos autônomos como carros e robôs.

Além da Inteligência Artificial existem diversos outros campos de estudo como Processamento de Imagens, Reconhecimento de Padrões que também contribuem fortemente para a área de Visão Computacional (Barelli, 2018). Abaixo um grafo contendo os campos de estudo de maior relevância na área de Visão Computacional:

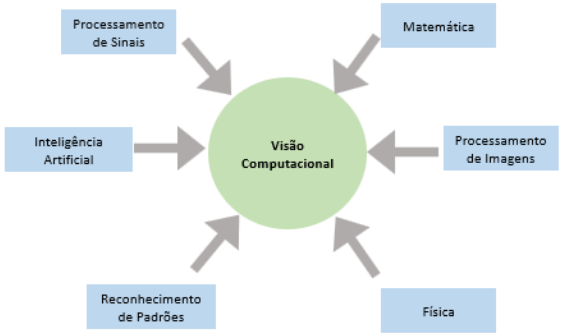

Figura 2-1Campos de estudo em visão computacional

2.2 Fluxo de trabalho na Visão Computacional

Mesmo sendo aplicados em abordagens diversas, os sistemas de Visão Computacional geralmente apresentam um fluxo padrão. A figura a seguir apresenta esse fluxo:

Figura 2-2 Fluxo de sistema baseado em Visão Computacional

A primeira etapa de um sistema de Visão Computacional é a aquisição da imagem. Nela, sensores eletrônicos são utilizados para obter e digitalizar a imagem. A imagem digitalizada é composta por *pixels*, que representam o menor elemento que compõem uma imagem digital. Cada *pixel* carrega um valor que representa um tom dentro de uma escala que varia de 0 a 255. Para o caso de imagens bidimensionais em tons de cinza, existe apenas um grupo de *pixels* distribuídos espacialmente em um plano 2D. Para uma imagem em cores, tipicamente existem 3 ou 4 conjuntos de *pixels*, cada um contendo a tonalidade de uma das cores. Para o caso de um espaço de cores RBG (que faz parte dos experimentos deste estudo), existem 3 cores primarias: R-Red, Blue-Azul e G-Green. A soma de tonalidades dessas três camadas forma uma imagem colorida.

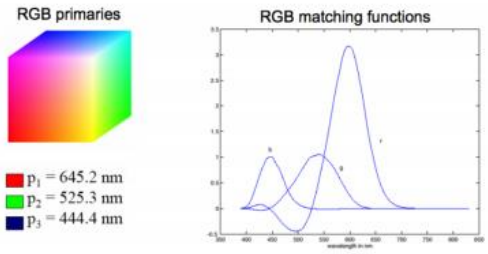

Figura 2-3Representação das cores primárias RBG e suas funções

Umas das modelagens matemáticas possíveis para uma imagem digital, bidimensional e em tons de cinza seria um tensor de *rank* 2 mais conhecido como matriz. Nessas matrizes as linhas e colunas representam as coordenadas espaciais (x,y) de cada *pixe*l e os elementos desta matriz carregam as informações de tonalidade de cada ponto. Para o caso de uma imagem bidimensional colorida, a representação matemática mais apropriada é um tensor de *rank* 3 (x,y,c). Neste tensor a última dimensão representa os canais de cores da imagem

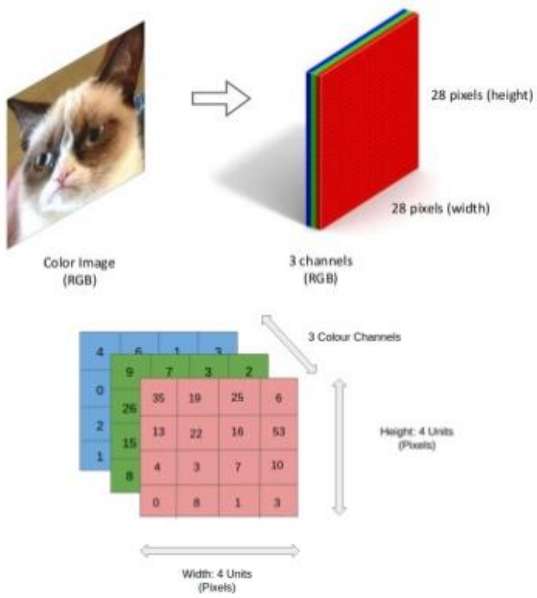

Figura 2-4 Representação de imagens RGB como tensor rank-3

A segunda etapa consiste no pré-processamento da imagem. Após definirmos o objeto de interesse algumas técnicas de pré-processamento podem ser aplicadas para destacar bordas, formas geométricas, e outros elementos da imagem que auxiliam a identificação e extração de características pelo sistema de Visão Computacional. Na figura 2.5 a direita, um grupo de bolas verdes e azuis distribuídas em um plano 2D poderia ser representado por círculos coloridos sem a necessidade do plano de fundo e a ideia de profundidade conforme vemos na imagem da esquerda:

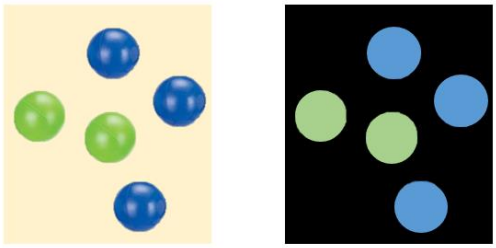

Figura 2-5 Etapa de pré-processamento

Na etapa de segmentação, objetos que possuem características semelhantes poderiam ser segmentados do resto da imagem com o propósito de facilitar ainda mais a extração de característica. Ainda analisando a figura 2-5, suponha que estejamos implementando um sistema que analise bolas na cor verde. Para isso essa imagem poderia ser novamente submetida a uma outra técnica de pré-processamento, a fim de mantermos na imagem final somente bolas na cor verde. A figura a seguir ilustra esse procedimento:

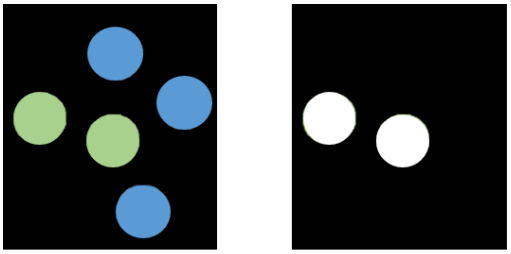

Figura 2-6 Etapa de Segmentação

Por fim, na última etapa ocorre o processamento de alto nível, onde o objetivo é reconhecer atributos do objeto de interesse. Para esse exemplo poderíamos identificar o diâmetro de cada figura. Neste momento também seria possível identificar se o objeto é um círculo ou quadrado a partir dos seus atributos.

2.3 Sistemas Lineares e Filtros Convolucionais

O fato de uma imagem ser representada por um matriz ou tensor, nos abre uma gama de possibilidades matemáticas para realização de pré-processamento, segmentação e extração de características das imagens. Uma delas é a representação da imagem como uma função $f : \mathcal{R}^2 \rightarrow \mathcal{R}$ conforme a imagem abaixo:

$$\begin{bmatrix} \ddots & & \vdots & & \iddots \\ & f[-1,1] & f[0,1] & f[1,1] & \\ \dots & f[-1,0] & f[0,0] & f[0,1] & \dots \\ & f[-1,-1] & f[0,-1] & f[1,-1] & \dots \\ \iddots & & \vdots & & \ddots \end{bmatrix}$$

Figura 2-7 Imagem representada como função discreta

Aqui os valores $f[n,m]$ representam a intensidade de um pixel na posição $[m,n]$. Observe que estamos usando colchetes ao invés de parêntese para fazer referência as funções discretas. Por exemplo, a função f retorna à intensidade em tons de cinza de um *pixel* em uma imagem localizada horizontalmente entre a posição a e b e verticalmente entre as posições c e d.

$$f: [a,b] \text{ X } [c,d] \rightarrow [0{,}255]$$

O grupo de valores $[a,b]$ X $[c,d]$ é conhecido como domínio de suporte e contém todos os valores que são entradas válidas da função f, enquanto $[0{,}255]$ é o range definido para as possíveis saídas.

Uma imagem também pode ser tratada como uma função de mapeamento do $R2 \rightarrow R3$. Por exemplo em imagens RGB, a intensidade de um *pixel* pode ser escrita como a função u abaixo:

$$u[x,y] = \begin{bmatrix} r[x,y] \\ g[x,y] \\ b[x,y] \end{bmatrix}$$

Onde $r, g, b : [a,b]$ X $[c,d] \rightarrow [0{,}255]$

Deste modo, podemos realizar o pré-processamento, segmentação e extração de características utilizando sistemas lineares (filtros), que são basicamente unidades capazes de converter uma função de entrada $f[n,m]$ em uma função de resposta (ou saída) $u[n,m]$. Para o caso do processamento de imagens, sistemas lineares são capazes de oferecer em sua saída variações da imagem de entrada. A notação S é referenciada como um operador de sistema que mapeia um membro de um conjunto de possibilidades da saída $u[n,m]$ para um membro do conjunto de possibilidades da entrada $f[n,m]$. Quando usamos a notação envolvendo S podemos escrever as equações das seguintes maneiras:

$$S[u] = f$$

$$S\{f[m,n]\} = u[m,n]$$

$$f[m,n] \xrightarrow{S} u[m,n]$$

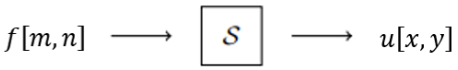

Figura 2-8 Representação gráfica sistema de mapeamento de f para u

Um bom exemplo de filtro é o sistema de médias móveis, este tipo de filtro calcula o valor de um *pixel* como a média dos valores dos seus vizinhos. Matematicamente é possível representar como:

$$u[m,n] = \frac{1}{9} \sum_{i=-1}^{1} \sum_{j=-1}^{1} f[m-i, n-j]$$

Este filtro de média ponderada suaviza as bordas da imagem criando um efeito de borrão.

Um outro sistema muito utilizado como parte da etapa de pré-processamento, segmentação e extração de característica de imagens são os filtros convolucionais.

Filtros convolucionais são uma categoria de sistema que usa as informações dos pixels vizinhos para modificar o pixel a ser filtrado. A operação de convolução é representada pelo símbolo *. No exemplo $f[n,m] * h[n,m]$, temos a função f sendo multiplicada pela função resposta de impulso deslocado

$h[m,n]$. A operação de convolução permite que qualquer sinal de entrada que passe através de um sistema, seja calculado simplesmente considerando a resposta do sistema ao impulso. Na prática o teorema de convolução é usado para implementar filtros no domínio da frequência, no caso de processamento de imagens, filtros convolucionais podem ser implementados como os algoritmos de média ponderada, e neste caso a função $h[m,n]$ será uma matriz conhecida como *kernel*. Filtros convolucionais são ferramentas valiosas para pré-processamento de imagens e como veremos no próximo capitulo, é possível obter filtros convolucionais que explorem os objetos de interesse e suas características a partir de técnicas de aprendizado de máquina.

2.4 Redes Neurais Convolucionais

Uma boa definição para Redes Neurais Convolucionais poderia ser: modelos baseados em algoritmos de aprendizado profundo que são capazes de receber imagens, atribuir importância (pesos e vieses) às características das imagens e identificar seus padrões. Isso faz com que o pré-processamento exigido em uma ConvNet seja muito menor se comparado com outros algoritmos de classificação convencionais (Academy, 2021). Sabemos que por meio de alguns cálculos e experimentação é possível obter filtros convolucionais que possam extrair característica primitivas de uma imagem, contudo esse processo manual seria muito demorado e custoso. ConvNets têm a capacidade de aprender como configurar de forma ótima seus filtros utilizando apenas imagens de exemplo no processo de treinamento da rede.

A estrutura de uma rede convolucional é baseado nos padrões de conectividade de neurônios do Cortex Visual no cérebro humano. Os neurônios individualmente respondem a estímulos apenas em uma região restrita do campo visual conhecida como Campo Receptivo. (Academy, 2021)

Abaixo é possível ter uma visão alto nível da arquitetura de uma ConvNet:

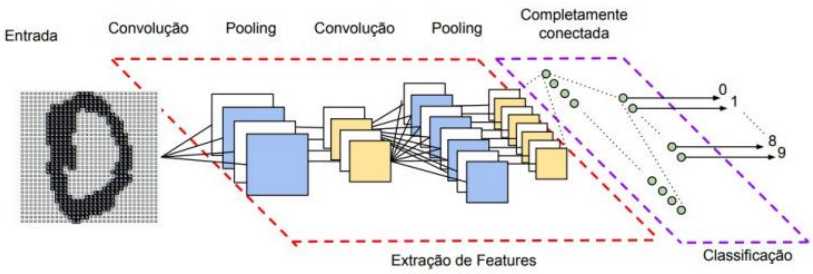

Figura 2-9 Arquitetura de uma rede CNN

Essa rede é composta basicamente por três camadas: convolucional, *pooling* e *fully-connected* (totalmente conectada). Cada camada pode ser interpretada como um volume de dados, isto porque cada uma delas pode ser representada por três dimensões: largura e altura que dizem respeito as informações espaciais da entrada e a profundidade que diz respeito aos níveis ou número de filtros de cada camada.

Nas camadas de convolução, cada nível reúne os neurônios ativados para operações de convolução. Esses níveis, também chamado de mapa de características são responsáveis por extrair características de uma entrada. Isso significa que todos os neurônios de uma camada de convolução detectam exatamente as mesmas características, apenas em locais diferentes da entrada. Essa é a principal diferença entre as camadas convolucionais de uma CNN e as redes neurais tradicionais (*fully-connected*). Na primeira abordagem as conexões entre as camadas ocorrem de forma local, por exemplo, um neurônio da primeira camada convolucional se conecta apenas a um grupo específico de neurônios da camada de entrada, deste modo é realizada a operação de convolução discreta entre saídas de uma camada anterior e um conjunto de filtros da camada de convolução. Cada filtro possui um tamanho que é determinado por um hiperparâmetro chamado *kernel*, este representa o tamanho da "área" de convolução. O volume de saída é determinado por três hiperparâmetros: número de filtros, *stride* e *padding*. O primeiro controla a profundidade do volume de saída e corresponde ao número de filtros que se deseja utilizar na camada. O segundo determina a quantidade de "saltos" que se realiza no volume de entrada para aplicar a convolução discreta. O terceiro determina a quantidade de zeros que se deseja adicionar nas bordas do volume de entrada para que seja possível realizar a operação de convolução reduzindo a perda de informação (Zisserman & Simonvan, 2014).

Na camada de pooling geralmente é utilizada a função *max pooling*, uma função que extrai o valor máximo entre os neurônios de uma determinada região de um mapa de características, reduzindo o volume de amostras da camada anterior. Uma função comumente utilizada na ativação de neurônios destas camadas é denominada Unidades Lineares Retificadas (ReLU), essa é uma função não-linear que exerce um grande impacto no desempenho do treinamento (Agarap, 2018) .

A última camada de pooling geralmente é transformada em um vetor que alimenta uma rede *fully-connected*, onde todos os neurônios da camada de entrada falam com todos os neurônios da camada oculta. Assim como nas camadas convolucionais existe uma função de ativação que insere não linearidade a rede. A camada totalmente conectada de uma CNN cumpre o papel de retornar uma decisão em relação a classificação da imagem como um todo. Na realidade, o que torna possível essa classificação é o mecanismo de análise e extração de características isoladas da imagem, provenientes das camadas anteriores (convolução e pooling) (Deshpande, 2016). Em problemas classificatórios, a úl-

tima camada possui uma função sigmod para classificação binária ou uma função softmax para classificação multi-classe. O treinamento na forma de aprendizado supervisionado ocorre a partir do processo de *Backpropagation*, onde uma função de otimização é aplicada para redução do erro (Nielsen, 2015).

3 Definição do Problema e Solução Proposta

3.1 Definição do Problema e Objetivos

Como já foi dito no capítulo introdutório, devido as condições climáticas do nosso país, o cultivo da uva acaba sendo constantemente afetado por surtos de míldio. Os sintomas desta doença são marcantes nas folhas e galhos da videira, isso porque os primeiros sintomas são manchas de coloração verde-clara seguidas de manchas escuras na cor marrom. Os experimentos demonstrados neste capítulo exploram essas características visuais do míldio a fim de detectar áreas afetadas pela doença utilizando modelos preditivos baseados em Visão Computacional.

Figura 3-1 Folhas infectada pelo Míldio

Podemos observar que este problema pode ser configurado como uma classificação binária e será tratado neste trabalho a partir de uma Rede Neural Convolucional. A partir desta definição, foi possível iniciar a etapa de aquisição de exemplos para o treinamento do modelo. Inicialmente foram selecionadas duzentas fotos, onde uma parte teve sua origem em pesquisas na internet e outra parte foi obtida em campo. O *dataset* reuniu 100 exemplos da folha da uva em seu estágio saudável e 100 exemplos de folhas no seu estágio contaminado.

3.2 Ferramentas para Implementação do Modelo, Arquitetura da Rede e Métricas de Avaliação

Atualmente podemos encontrar diversas ferramentas para implementação de ConvNets, contudo existem duas ferramentas que se destacam no cenário atual: PyTorch e TensorFlow, ambas desenvolvidas na linguagem Python. Considerando todo conhecimento prévio na linguagem Python e no domínio da ferramenta TensorFlow-Keras, os códigos aqui apresentados serão implementados com este conjunto de habilidades.

Os ensaios e experimentos foram realizados em um ambiente virtual Anaconda utilizando a última versão do Jupyter Lab como notebook. O Sistema Operacional e Hardware utilizados foram: Ubuntu 20.04 sobre uma máquina Intel I7 8º geração com 16GB de memória RAM, sem o auxílio de GPU.

A arquitetura inicial da rede proposta foi implementada utilizando o *Sequence Model API* do *framework* Keras conforme imagem abaixo:

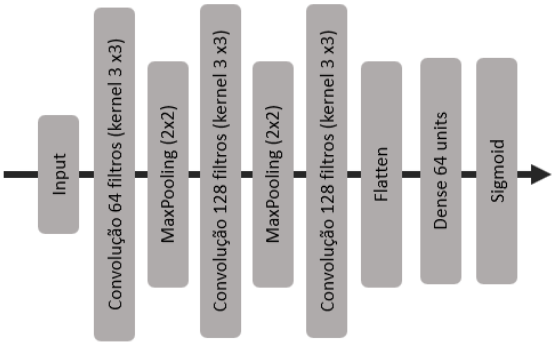

Figura 3-2 Arquitetura 1 proposta (CNN)

Os dados foram divididos em dois conjuntos: Treinamento e Teste. Esta abordagem oferece condições para uma análise mais detalhada dos efeitos do *overfiting* e *bias* para o *dataset* de treinamento (Brownlee, How to Identify Overfitting Machine Learning Models in Scikit-Learn, 2020). Para os experimentos utilizamos uma proporção de 75% de dados de treino e 25% de dados teste.

Como as imagens adquiridas estão em tamanhos diferentes, foi necessário um pré-processamento de *scale* das imagens em formato padrão 32p x 32p. Além disso, como recomenda a literatura, os valores dos canais RGB foram normalizados (Sergey Ioffe, 2015). Para simplificar esse processo foi utilizado técnica de *max-scale* onde todos os valores são divididos por um valor máximo que para o caso de imagens RGB seria o valor 255.

Neste trabalho foi utilizado um método disponibilizado pelo framework Keras chamado *Image Data Generator*. Abaixo um trecho do código onde esse pré-processamento foi implementado:

```
datagen_train=ImageDataGenerator(rescale=1./255)
train_data = datagen_train.flow_from_dataframe(dataframe=train_df, target_size=(32,32),batch_size=8
```

Figura 3-3 Pré-Processamento utilizando Keras

Foram realizados diversos experimentos com a variação dos seguintes hiperparâmentos:

- Taxa de aprendizado da função de otimização
- Número de neurônios por camada
- Número de camadas
- Número de épocas

Após algumas rodadas de experimentação, ficou evidente a necessidade de uma ferramenta que registrasse cada versão dos modelos junto com seus hiperparâmetros para fins de comparação. Além disso, a acurácia (medida inicialmente escolhida para avaliar a performance do modelo) não ofereceu um resultado claro sobre o percentual de erros para falsos positivos: situação onde o modelo classifica uma folha como doente, mas na verdade ela está sadia, ou casos de falsos negativos: quando o modelo acusa que a folha está saudável e na verdade ela está doente. Essa medida pode ser ainda mais enganosa se as classes estão desbalanceadas, ou seja, o número de exemplos positivos para treinamento é consideravelmente diferente do número negativos.

Uma medida possível seria a utilização da Precisão, que pode ser utilizada em uma situação em que os Falsos Positivos são considerados mais prejudiciais que os Falsos Negativos. Por exemplo no mercado de ações, podemos considerar aceitável que o modelo por vezes classifique bons investimentos como maus investimentos (Falso Negativo), isto porque um Falso Positivo nesse cenário significa uma grande perda financeira.

Outra medida possível seria o Recall que pode ser usado em uma situação em que os Falsos Negativos são considerados mais prejudiciais que os Falsos Positivos. Um exemplo seria um modelo que deve a qualquer custo encontrar todos os pacientes doentes, mesmo que classifique alguns saudáveis como doentes (Falso Positivo). Ou seja, o modelo deve ter alto recall, pois classificar pacientes doentes como saudáveis pode caracterizar uma tragédia.

O *F1-Score* é uma média harmônica entre a Precisão e Recall, que busca acompanhar os menores valores diferente de uma média aritmética simples. Ou seja, quando se tem um *F1-Score* baixo, é

um indicativo de que ou a precisão ou o recall está baixo. Por motivos de praticidade, pois podemos observar somente uma métrica ao invés de duas, adotaremos o *F1-Score* como medida de avaliação do modelo.

Para registrar, métricas, hiperparâmetros, arquitetura e o modelo propriamente dito foi utilizado a ferramenta MlFlow.

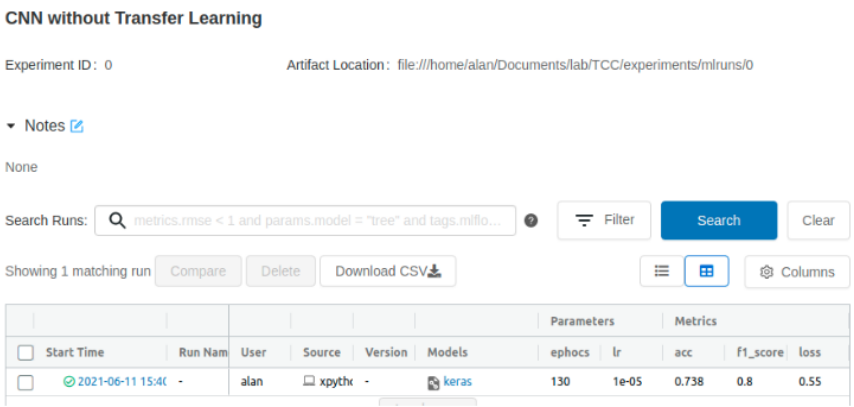

Figura 3-4 Interface web MlFlow

3.3 Discussão dos Resultados

Após realizar o primeiro experimento os resultados encontrados foram:

- Acurácia:0.53
- *F1-Score*: 0.52

Fica evidente que um valor de 0.52 para o *F1-Score* não é um resultado razoável, pois caracteriza um evento quase que aleatório.

Para uma avaliação mais detalhada foi adicionado ao código a matriz de confusão. Essa matriz oferece a soma de verdadeiro positivos e verdadeiros negativos (diagonal principal da matriz), falsos positivos e falsos negativos.

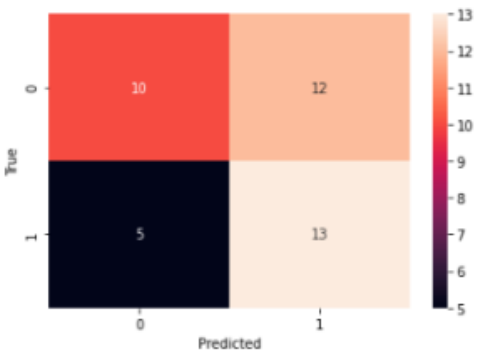

Figura 3-5 Matriz de confusão do experimento inicial

Foi proposto uma segunda arquitetura com a adição de mais camadas convolucionais e camadas *fully-connected*, além do incremento do número de épocas. Contudo, os resultados encontrados tiveram uma variação de apenas 0,03 no *F1-Score*. Retornando a matriz de confusão desse segundo experimento ficou evidente que o modelo tinha dificuldades de identificar folhas saudáveis, pois o número de falsos positivos (onde o modelo estava classificando uma folha saudável como doente) era de aproximadamente 30% (12 a 14 erros).

Para tentar resolver este problema foram inspecionados manualmente todos os exemplos de folhas saudáveis e observou-se que as imagens de folhas saudáveis não caracterizavam muito bem esta condição. Muitas fotos possuíam composições e objetos concorrentes onde era difícil identificar uma folha saudável.

Figura 3-6 Exemplo de imagem inicialmente classificada como positiva

Foi feita uma seleção/recorte de fotos com folhas saudáveis e doentes dentro do *dataset* total de modo a isolar as folhas da videira em cada imagem. Com isso o *dataset* ficou reduzido a 77 fotos de folhas doentes e 51 fotos de folhas saudáveis. Para balancear as classes foram selecionadas mais 30 fotos na internet de folhas saudáveis totalizando 158 fotos para treinamento e teste. Além das novas fotos, foram aplicadas técnicas de *Data Augmentation* utilizando a ferramenta Keras. Na figura abaixo é possível ver que foram aplicadas técnicas de rotação, deslocamento horizontal e vertical, zoom, e rebatimento horizontal com intuito de incrementar os dados de treinamento.

```
path_dataset= './imagens/doenca_fungica_videira/dataset'
datagen_train=ImageDataGenerator(rescale=1./255,
                                 rotation_range=0.2,
                                 width_shift_range=0.2,
                                 height_shift_range=0.2,
                                 zoom_range=0.2,
                                 horizontal_flip=True)
```

Figura 3-7 Data Augmentation com Keras

Após essas mudanças o *F1-Score* obtido aumentou 0,25 alcançando a marca de 80%! A matriz de confusão desta nova abordagem reuniu os seguintes números:

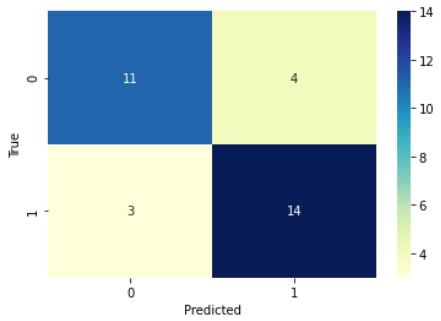

Figura 3-8 Matriz de confusão após seleção de imagens

Além de utilizar as arquiteturas já apresentadas, também foi realizada um experimento utilizando técnicas de transferência de aprendizado. Nesta técnica um modelo que foi previamente treinado em um grande conjunto de dados (normalmente em uma tarefa de classificação de imagem em grande escala) tem seus pesos e arquitetura salvos para uso futuro. Deste modo, é possível reaproveitá-lo adicionando apenas uma camada de saída para se adequar a um propósito específico de classificação (número de classes).

A intuição por trás do aprendizado por transferência para classificação de imagens é que se um modelo for treinado em um conjunto de dados grande e geral o suficiente, ele servirá efetivamente como uma base genérica do mundo visual. Você pode então tirar proveito desses mapas de recursos já treinados, concentrando-se apenas no treinamento da camada de saída com um número reduzido de imagens.

A ferramenta Keras proporciona este recurso de forma nativa a partir de um repositório público chamado Tensorflow hub, nele é possível encontrar centenas de arquiteturas CNN pré-treinadas para extração de características de imagens e classificação. Para nosso experimento utilizamos uma arquitetura conhecida como *EfficientNet* B0. No gráfico abaixo podemos verificar um comparativo das arquiteturas mais populares para classificação de imagens. A Rede efficientNet B0 é a mais eficiente se consideramos os números de parâmetros X acurácia.

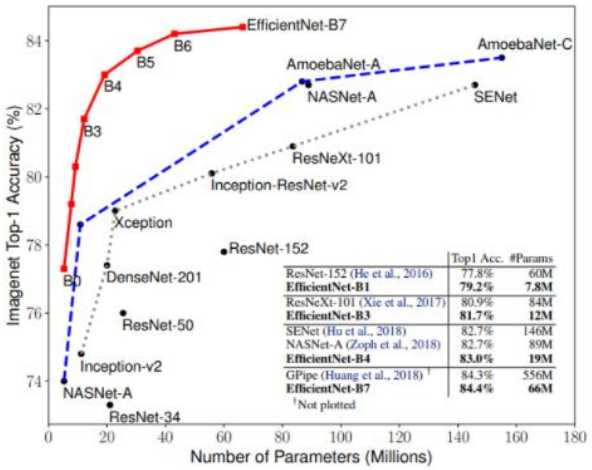

Figura 3-7 Gráfico comparativo de eficiência das arquiteturas CNN

Como não vamos realizar o *Fine Tunning* da rede, ou seja, o treinamento de suas camadas ocultas, precisamos "congelar" essas camadas tornando-as não treináveis, isso é feito a partir do parâmetro "*trainable=False*", conforme o código abaixo:

```
efficientnet_url = "https://tfhub.dev/tensorflow/efficientnet/b0/feature-vector/1"
```

```
def create_model(model_url,image_shape):
  # Download the pretrained model and save it as a Keras layer
  feature_extractor_layer = hub.KerasLayer(model_url,
                                           trainable=False, # freeze the underlying patterns
                                           name='feature_extraction_layer',
                                           input_shape=image_shape) # define the input image shape

  # Create our own model
  model = tf.keras.Sequential([
    feature_extractor_layer, # use the feature extraction layer as the base
    Dense(units=64, activation='relu'),
    Dense(units=64, activation='relu'),
    Dense(1, activation='sigmoid', name='output_layer') # create our own output layer
  ])

  return model
```

Figura 3-9 Implementação da transferência de conhecimento com Keras

Após o treinamento deste modelo por apenas 15 épocas o *F1-Score* se estabilizou em um valor próximo ao experimento anterior.

Abaixo temos um quadro comparativo com os experimentos realizados:

		Camadas				Resultados em Teste	
		Convolucionais	Fully Connected	Épocas	Taxa Aprendizado	Acurácia	F1 Score
Dados não Tratados	Experimento I	3	2	50	0,001	53%	52%
		6	3	150	0,00001	57%	55%
Dados Tratados	Experimento I	6	3	150	0,00001	78%	80%
	Experimento II	N/A	3	15	0,0001	75%	77%

Figura 3-10 Comparação dos Experimentos

Apesar do resultado *do F1-Score* do segundo experimento após o tratamento dos dados ficar um pouco abaixo se comparado com as arquiteturas customizadas no experimento I, é possível perceber o quão poderoso é o recurso de transferência de aprendizado. Possivelmente técnicas de *Fine Tunning* poderiam explorar a capacidade e potencial desta rede, mas considerando que este é um processo muito custoso do ponto de vista computacional, esta técnica não foi avaliada neste trabalho.

4 Testes em Campo e Estratégia de Operacionalização

4.1 Salvando e Disponibilizando um Modelo via API

Uma grande vantagem ao se trabalhar com um *Framework* versátil como o Keras, é que você pode salvar seu modelo como um *arquivo .pb* (protocol buffer) que depois poderia ser utilizado em qualquer ambiente que contenha as bibliotecas necessárias para sua execução. O keras realiza esse procedimento de forma muito simples utilizando a chamada da função *"model_name.save('nome_arquivo')"* . Como estamos utilizando o MLFlow para log e registro dos modelos, este arquivo foi armazenado a partir do comando *"mlflow.keras.log_model(model_1,'model')"* que naturalmente usa o comando *"save"* do keras como base. Por questões de replicabilidade e rapidez do *deploy* do modelo, a fim de disponibilizar uma API que fosse acessível no local de cultivo das videiras, foi implementado um pipeline de dados utilizando um orquestrador de tarefas chamado Jenkins. Esta ferramenta foi construída e configurada dentro de um cluster Kubernetes em uma máquina local com o propósito de executar algumas tarefas conforme a figura abaixo:

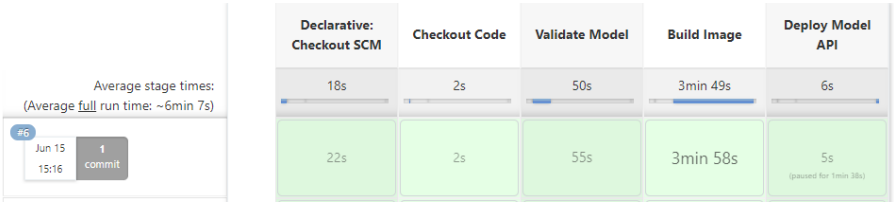

Figura 4-1 Etapas do pipeline na interface web Jenkins

Inicialmente ele faz uma cópia do código fonte da API e dos modelos e seus metadados registrados via Mlflow a partir do repositório remoto Github para um volume criado no cluster (todo código foi versionado utilizando o Git como repositório local). Um segundo *Job* inicia a etapa de validação automática do modelo onde o melhor modelo na fase de experimentação é escolhido a partir do seu *F1-Score* registrado nos arquivos do MlFlow. Após a escolha do melhor modelo, uma cópia é feita na pasta de arquivos da API.

```
+ python validate_model/validateModel.py
Experiment name: CNN without Transfer Learning
Max fscore: 0.7999999999999999
Variation with max fscore (path): experiments/mlruns/0/31802e4299044fa5b13a8337ea8fa526/artifacts
```

Figura 4-2 Etapa de validação no console Jenkins

A terceira etapa do *deploy* é a construção de uma imagem Docker contendo como dependências o *Framework* do modelo e da API. Essa imagem foi criada pelo serviço *Clould Build* GCP e armazenada em um registry privado da mesma nuvem. Por fim o orquestrador faz uma chamada ao *cluster* para a criação de um *service* e um *deploy* kubernetes a partir dos *manifests* previamente inseridos dentro da pasta do projeto da API.

Todo o código da API foi escrito em Python utilizando o *Framework* FastAPI. Esse *Framework* se mostrou muito útil pois possui um recurso *Auto-build* da documentação *Swagger* que poderia ser utilizado para consumir os serviços da API e realizar predições via interface web na área de cultivo. Sendo assim foi possível fazer o upload de uma foto via browser em um aparelho celular que estivesse conectado a mesma rede do *cluster* Kubernetes.

Figura 4-3 Teste em campo utilizando Swagger UI API

Com a utilização do pipeline foi possível disponibilizar para testes em campo um novo experimento em menos de 6 minutos!

4.2 Captura de Dados com Microcontroladores

No tópico 5.1 foi apresentado a possibilidade de testes em campos a partir do browser de um aparelho celular. Apesar de cômodo, a dinâmica proposta é que o agricultor não precise estar na plantação para ser notificado. Para isso serão discutidas algumas soluções de IoT utilizando microcontrolador como o Esp32-Cam para aquisição dos dados e envio automático para as interfaces do modelo preditivo.

Figura 4-4 Microcontrolador Esp32-Cam

O ESP32 é uma série de microcontroladores de baixo custo e baixo consumo de energia com câmera e Wi-Fi integrado. A série ESP32 emprega um microprocessador Tensilica Xtensa LX6 com uma antena RF que amplifica a potência de transmissão e recepção, reduz ruído e gerencia o consumo de energia. A grande vantagem é que a série esp32 pode ser programada via IDE Arduino e é compatível com todas as bibliotecas deste dispositivo. Esse microcontrolador pode ser comprado no Brasil por menos de R$ 50,00, o que o torna um ótimo candidato para aquisição dos dados em campo. Uma

outra grande facilidade é a vasta lista de exemplos e documentação, onde vários projetos para implementação de um WebClient estão disponíveis. Um repositório muito popular é o repositório oficial Arduino: create.arduino.cc/projecthub.

Além da API para consumo do modelo, na dinâmica do projeto também seria necessário um serviço capaz de enviar e-mails no momento em que o modelo detectasse folhas contaminadas na área de cultivo. Com intuito de facilitar a localização do setor contaminado, cada câmera deve enviar uma referência do seu ponto de instalação. O posicionamento da câmera, deve ser definido com base no local de maior probabilidade da doença que normalmente se concentra em áreas com menor circulação de ar. Abaixo pode ser visto um diagrama da arquitetura proposta:

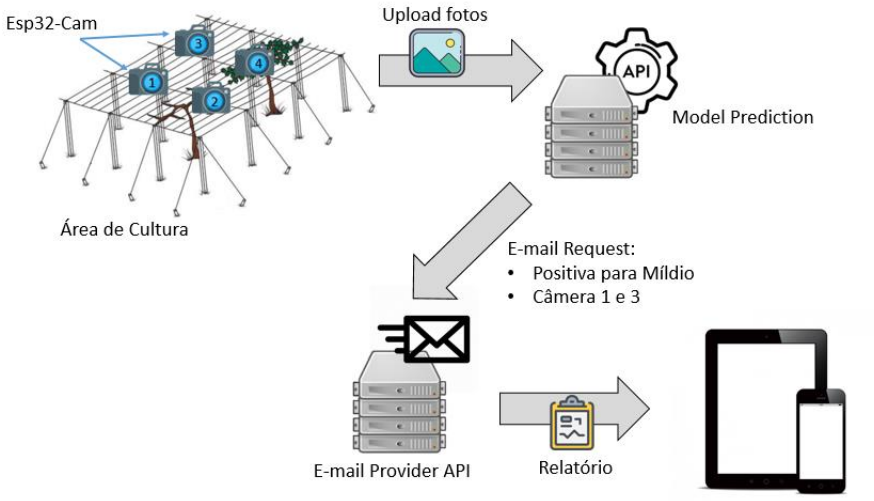

Figura 4-5 Sistema proposto para o controle de míldio

5 Conclusão

A partir dos experimentos realizados na sessão 4.1, foi possível constatar que as Redes Neurais Convolucionais foram capazes de reconhecer padrões e características do míldio com um nível acurácia aceitável. Além disso, os resultados obtidos no capítulo 5 para os testes realizados em campo se mostraram suficientes.

O pipeline proposto no capitulo 5 utilizando Jenkins e Kubernetes facilitaram sobremaneira o *deploy* das APIs e permitiram uma avaliação rápida de cada experimento a partir de novas imagens obtidas em campo.

Por conta da indisponibilidade de grandes vinhedos para experimentação em campo, não foi possível explorar uma maior variabilidade da manifestação do míldio. Todos os experimentos em campo foram realizados em uma vinicultura caseira de aproximadamente $5m^2$. Além disso, o período de observação de um mês e meio durante o inverno (junho e julho) foram pouco representativos no que diz respeito ao comportamento do míldio nas folhas da videira.

Em resumo, apesar de não dispormos de uma grande diversidade de dados durante o treinamento e os testes em campo, é possível afirmar que as aplicações desenvolvidas neste trabalho se mostraram úteis para a monitoração e controle do míldio no cultivo da uva.

6 Apêndice A

6.1 Repositórios do Projeto

Código fonte da API: https://github.com/alantellecom/api-pipeline_vine_fungi_tcc

Código fonte do pipeline de dados: https://github.com/alantellecom/jenkins_config_tcc

Código fonte dos experimentos: https://github.com/alantellecom/experiments_vine_fungi_tcc

7 Referências Bibliográficas

Academy, D. S. (2021). *Deep Learning Book.*

Agarap, A. F. (2018). *Deep Learning using Rectified Linear Units (ReLU).*

Barelli, F. (2018). *Introdução À Visão Computacional - Uma Abordagem Prática Com Python E Opencv.*

Brown, B. e. (1982). *Computer Vision.*

Brownlee, J. (2017). *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*.

Brownlee, J. (2020). How to Identify Overfitting Machine Learning Models in Scikit-Learn. Fonte: https://machinelearningmastery.com/overfitting-machine-learning-models/

Cultiva, R. ((2001)). *Organization.* Acesso em 13 de 01 de 2014, disponível em www.grupocultivar.com.br/artigos/uva-hora-do-ataque-do-mildio

Deshpande, A. (2016). *A Beginner's Guide To Understanding Convolutional Neural Networks.*

Nielsen, M. A. (2015). *Neural Networks and Deep Learning.*

Sergey Ioffe, C. S. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Fonte: https://arxiv.org/abs/1502.03167

Zisserman, & Simonvan. (2014). *Deep inside convolutional networks: Visualising image classification models and saliency maps.*