CLASSIFICAÇÃO DE DIGITAIS

IMPLEMENTAÇÃO CNN LENET

Elaborado por:

ANDRÉ COSTA

Conteúdo

1 Introdução **2**

2 Referencial teórico **3**

2.1 Rede Neural 3

2.2 Otimizador Adam 3

2.3 Função de Perda 4

2.4 Arquitetura de uma Rede Neural 4

2.5 CNN 5

2.6 LeNet-5 5

2.6.1 Função de Perda na LeNet-5 6

2.7 Sokoto Coventry Fingerprint Dataset (SOCOFing) 6

3 Análises **7**

3.1 Pacotes necessários 7

3.2 Carregamento e pré-processamento de dados 8

3.3 Divisão de treino e teste 10

3.4 Data augmentation 10

3.5 Definindo o Modelo 12

3.6 Treinando o Modelo 15

3.7 Resultados 20

3.7.1 Exemplo de detecção 20

3.7.2 Implementação do Modelo 20

3.7.3 Resultados gerais 21

4 Conclusão **23**

5 Referências **24**

1 Introdução

Este relatório representa uma síntese detalhada do processo de desenvolvimento e análise de um modelo de reconhecimento de padrões de digitais, que compõe parte do meu portfólio profissional. A elaboração deste trabalho foi norteada por uma pesquisa extensiva, que incluiu a exploração de informações provenientes de diversos repositórios online, artigos científicos relevantes e outras fontes disponíveis na internet. A escolha do ambiente de desenvolvimento recaiu sobre o Python, uma linguagem de programação amplamente utilizada no campo da inteligência artificial e aprendizado de máquina. O Google Colab foi adotado como plataforma de desenvolvimento, fornecendo um ambiente de programação colaborativo baseado em nuvem, com suporte a bibliotecas populares como TensorFlow e PyTorch.

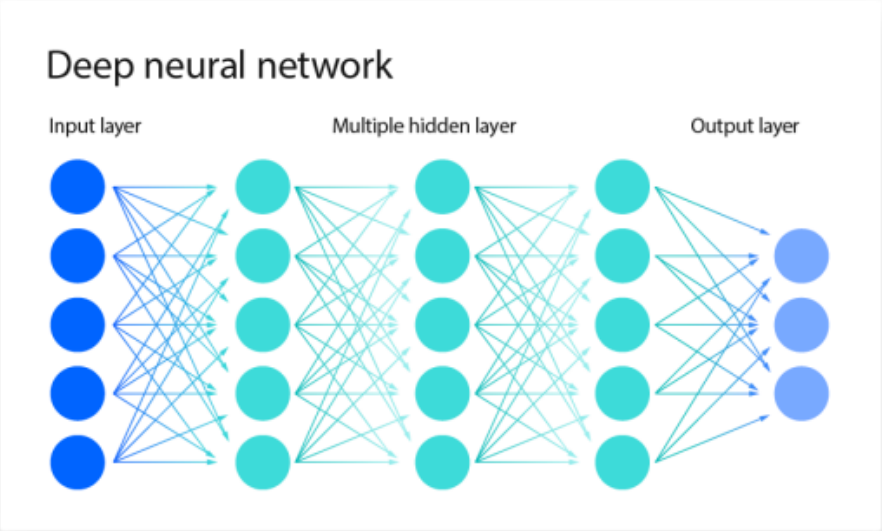
Durante o processo de treinamento do modelo, foi feito uso da capacidade computacional oferecida por uma unidade de processamento gráfico (GPU) do tipo T4. Essa escolha se deve à necessidade de acelerar o processo de aprendizado da máquina, permitindo uma iteração mais rápida e eficiente no desenvolvimento do modelo. Ademais, para uma análise mais aprofundada dos dados e resultados obtidos, optou-se por utilizar o ambiente RStudio. Este ambiente oferece uma ampla gama de ferramentas estatísticas e de visualização de dados, proporcionando uma análise detalhada e abrangente dos resultados do modelo.

Ao longo deste relatório, serão apresentados os principais passos do processo de desenvolvimento do modelo, desde a coleta e pré-processamento dos dados até a avaliação e interpretação dos resultados. Destaca-se a importância da integração de diversas tecnologias e ferramentas, bem como da aplicação de métodos e técnicas avançadas no campo do reconhecimento de padrões e visão computacional. Espera-se que este relatório sirva como um registro abrangente e informativo do trabalho realizado, demonstrando a capacidade de desenvolvimento e análise de modelos de inteligência artificial para aplicações práticas e inovadoras.

2 Referencial teórico

2.1 Rede Neural

Figura 1:



Uma rede neural é um modelo computacional inspirado no funcionamento do cérebro humano. Consiste em uma coleção de neurônios interconectados que processam informações. Cada neurônio recebe entradas ponderadas, aplica uma função de ativação e produz uma saída. Matematicamente, o cálculo da saída de um neurônio pode ser representado como:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

onde x_i são as entradas, w_i são os pesos associados, b é o viés e f é a função de ativação.

2.2 Otimizador Adam

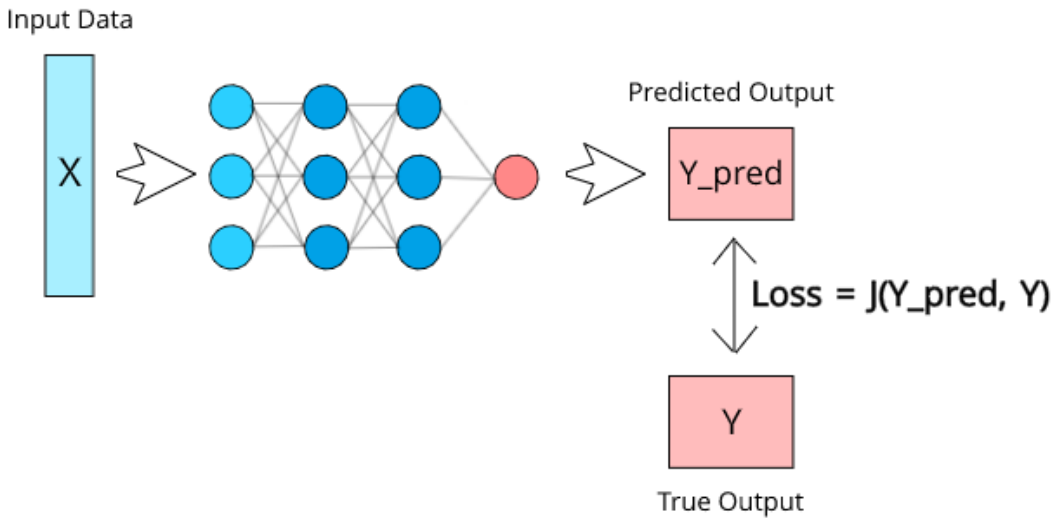
O otimizador Adam é um algoritmo de otimização popular utilizado no treinamento de redes neurais. Ele combina as vantagens do algoritmo de momentum e do RMSprop. O algoritmo Adam atualiza os pesos da rede de acordo com a seguinte regra:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$

onde θ_t são os parâmetros do modelo, α é a taxa de aprendizado, $\hat{m}_t$ é o momento do primeiro momento, $\hat{v}_t$ é o momento do segundo momento e ϵ é uma pequena constante para evitar a divisão por zero.

2.3 Função de Perda

Figura 2:



A função de perda é uma medida que quantifica o quão bem o modelo está realizando a tarefa durante o treinamento. No contexto de classificação, uma função de perda comum é a entropia cruzada categórica, dada por:

$$L(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$$

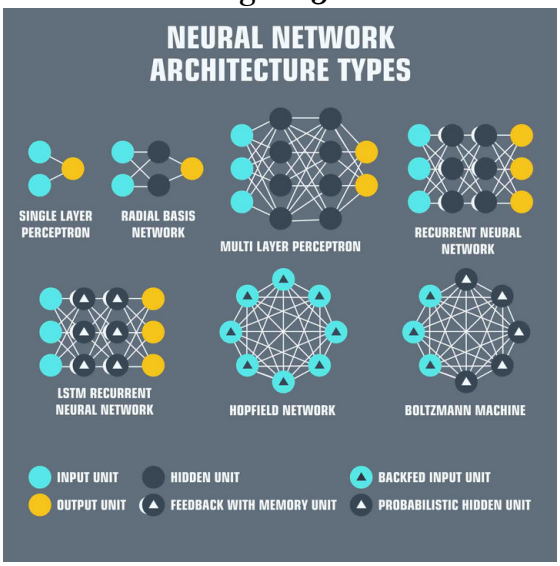
onde y é o rótulo verdadeiro e $\hat{y}$ é a predição do modelo.

2.4 Arquitetura de uma Rede Neural

A arquitetura de uma rede neural refere-se à organização e disposição dos neurônios e camadas na rede. Uma arquitetura típica consiste em uma sequência de camadas, incluindo camadas de entrada, ocultas e de saída. A saída de uma camada serve como entrada para a próxima camada. A arquitetura é determinante para o desempenho e capacidade do modelo em resolver uma determinada tarefa.

2.5 CNN

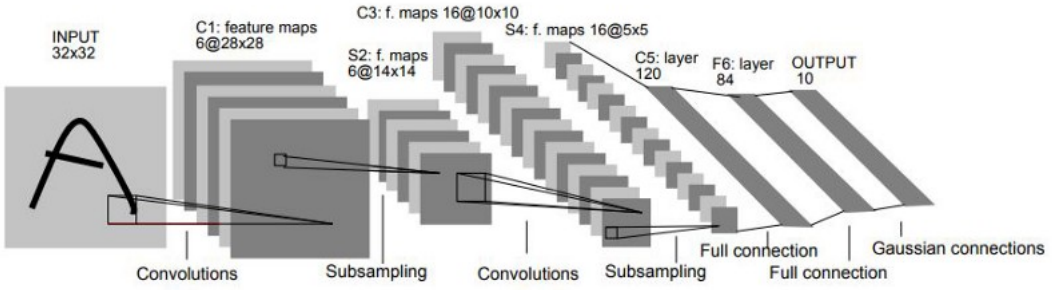
Figura 3:



As Redes Neurais Convolucionais (CNNs) são um tipo especializado de rede neural projetada para processar dados de grade, como imagens. Elas são compostas por camadas convolucionais, de pooling e totalmente conectadas. A operação de convolução é fundamental nas CNNs, permitindo a extração de características relevantes das imagens.

2.6 LeNet-5

Figura 4:



A LeNet-5 é uma arquitetura de CNN desenvolvida por Yann LeCun et al. em 1998. Foi uma das primeiras CNNs bem-sucedidas e foi amplamente utilizada para tarefas de reconhecimento de padrões em imagens, incluindo a classificação de dígitos manus-

critos. A arquitetura consiste em camadas convolucionais, de subamostragem e totalmente conectadas, seguidas por camadas de classificação softmax.

2.6.1 Função de Perda na LeNet-5

Na LeNet-5, a função de perda utilizada é geralmente a mesma entropia cruzada categórica (*categorical cross-entropy*) usada em outras arquiteturas de CNN. Esta função de perda é aplicada durante o treinamento para calcular o erro entre as previsões do modelo e os rótulos verdadeiros. A função de perda da LeNet-5 pode ser representada matematicamente como:

$$L(y, \hat{y}) = -\sum_{i} y_i \log(\hat{y}_i)$$

onde y é o vetor de rótulo verdadeiro e $\hat{y}$ é o vetor de previsão do modelo para a entrada correspondente.

2.7 Sokoto Coventry Fingerprint Dataset (SOCOFing)

O Sokoto Coventry Fingerprint Dataset (SOCOFing) é um conjunto de dados amplamente utilizado na pesquisa de reconhecimento de impressões digitais. Ele contém uma grande coleção de imagens de impressões digitais coletadas de voluntários em ambientes controlados e não controlados. O SOCOFing é composto por múltiplas amostras de impressões digitais de cada dedo de uma pessoa, capturadas sob diferentes condições, como diferentes ângulos de rotação, iluminação variada e diferentes resoluções.

Este conjunto de dados oferece uma variedade de desafios para algoritmos de reconhecimento de impressões digitais, incluindo variações intraclasse e interclasse, qualidade variável de imagem e distorções causadas por condições adversas de captura. Como resultado, o SOCOFing é um recurso valioso para avaliar e comparar a eficácia de diferentes métodos de reconhecimento de impressões digitais, incluindo abordagens baseadas em redes neurais convolucionais (CNNs).

A disponibilidade do SOCOFing facilita a pesquisa e o desenvolvimento de algoritmos robustos de reconhecimento de impressões digitais, permitindo a avaliação do desempenho do modelo em uma ampla variedade de condições do mundo real. É amplamente utilizado na comunidade acadêmica e de pesquisa para testar e validar novas técnicas e algoritmos de reconhecimento de impressões digitais.

3 Análises

3.1 Pacotes necessários

```
import numpy as np                          # Para manipulação de arrays
                                             multidimensionais.
import pandas as pd                         # Para manipulação de dados em formato
                                              de DataFrame.
import seaborn as sns                       # Para visualização de dados
                                             estatísticos.
import tensorflow as tf                     # Para construção e treinamento de
                                             modelos de redes neurais.
import os                                   # Para manipulação de diretórios e
                                             arquivos.
import cv2                                  # Para processamento de imagens.
import matplotlib.pyplot as plt             # Para plotagem de gráficos e imagens.
import random                               # Para geração de números aleatórios.

from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten,
                                    Dense
# Camadas específicas para a construção da arquitetura da CNN.
from tensorflow.keras.models import Model
# Para criação do modelo de rede neural.
from tensorflow.keras import optimizers
# Otimizadores para ajuste dos pesos do modelo durante o treinamento.
from keras.utils.np_utils import to_categorical
# Função para converter vetores de classe em matrizes de classe binárias.
```

Neste projeto foi utiliza a biblioteca TensorFlow para implementação da arquitetura da CNN, além de outras bibliotecas como NumPy, Pandas e Matplotlib para manipulação de dados e visualização. O modelo é treinado utilizando otimizadores específicos e é capaz de lidar com imagens de entrada em formato de matriz. A inclusão de camadas como Convolucionais, MaxPooling e Fully Connected permite a extração de características relevantes das imagens, contribuindo para a precisão do modelo na tarefa de classificação.

3.2 Carregamento e pré-processamento de dados

```
# Função para extrair o rótulo de uma imagem
def extract_label(img_path, train=True):
    filename, _ = os.path.splitext(os.path.basename(img_path))
    subject_id, etc = filename.split('__')

    if train:
        gender, lr, finger, _, _ = etc.split('_')  # Se for conjunto de
                                                   treinamento, o rótulo tem 5
                                                   partes
    else:
        gender, lr, finger, _ = etc.split('_')     # Se não for, o rótulo
                                                   tem 4 partes
    # Mapeamento de gênero, lateralidade e dedo para valores numéricos
    gender = 0 if gender == 'M' else 1
    lr = 0 if lr == 'Left' else 1

    finger_mapping = {'thumb': 0, 'index': 1, 'middle': 2, 'ring': 3, '
                                           little': 4}
    finger = finger_mapping[finger]
    return np.array([gender], dtype=np.uint16)
# Função para carregar dados de imagens
def loading_data(path, train):
    print("loading data from:", path)
    data = []
    for img in os.listdir(path):
        try:
            img_array = cv2.imread(os.path.join(path, img), cv2.
                                                   IMREAD_GRAYSCALE)
            img_resize = cv2.resize(img_array, (img_size, img_size))
            label = extract_label(os.path.join(path, img), train)
            data.append([label[0], img_resize])  # Adiciona rótulo e imagem
                                                   à lista de dados
        except Exception as e:
            pass
    return data
img_size = 96
```

```
# Caminhos para os diretórios de imagens
Real_path = "../input/socofing/SOCOFing/Real"
Easy_path = "../input/socofing/SOCOFing/Altered/Altered-Easy"
Medium_path = "../input/socofing/SOCOFing/Altered/Altered-Medium"
Hard_path = "../input/socofing/SOCOFing/Altered/Altered-Hard"

# Carrega dados de imagens dos diferentes diretórios
Easy_data = loading_data(Easy_path, train=True)
Medium_data = loading_data(Medium_path, train=True)
Hard_data = loading_data(Hard_path, train=True)
test = loading_data(Real_path, train=False)

# Combina os dados de diferentes dificuldades em um único conjunto de dados
data = np.concatenate([Easy_data, Medium_data, Hard_data], axis=0)

del Easy_data, Medium_data, Hard_data

# Separando imagens e rótulos em listas separadas
img, labels = [], []
for label, feature in data:
    labels.append(label)
    img.append(feature)

# Convertendo listas em arrays numpy e redimensiona as imagens
train_data = np.array(img).reshape(-1, img_size, img_size, 1)
train_data = train_data / 255.0  # Normaliza as imagens
train_labels = to_categorical(labels, num_classes=2)  # Converte rótulos em
                                                      one-hot encoding

del data

# Retornando dados de treinamento e rótulos
train_data, train_labels
```

3.3 Divisão de treino e teste

```
# Concatenando os dados
def concatenate_data(*args):
    return np.concatenate(args, axis=0)

x_data = concatenate_data(x_easy, x_medium, x_hard)
label_data = concatenate_data(y_easy, y_medium, y_hard)

# Dividindo os dados em conjuntos de treinamento e validação
def split_data(x_data, label_data, test_size=0.1):
    return train_test_split(x_data, label_data, test_size=test_size)

x_train, x_val, label_train, label_val = split_data(x_data, label_data,
                                        test_size=0.1)

# Exibindo as formas dos conjuntos de dados
def print_shapes(*arrays):
    for array in arrays:
        print(array.shape)

print_shapes(x_data, label_data)
print_shapes(x_train, label_train)
print_shapes(x_val, label_val)
```

3.4 Data augmentation

```
# Função para aplicar data augmentation
def augment_data(images, augs):
    augmented_images = []
    for img in images:
        augmented_images.append(img)  # Adiciona a imagem original
        augmented_images.extend(augs)  # Adiciona as imagens aumentadas
    return augmented_images
```

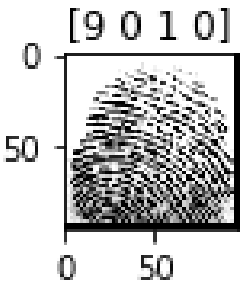


Figura 5:

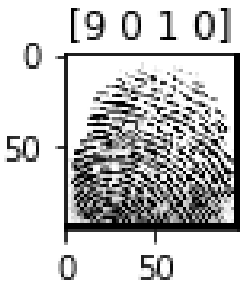


Figura 6:

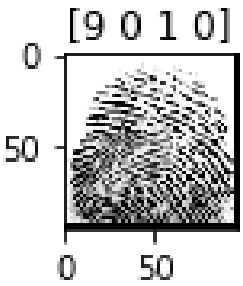


Figura 7:

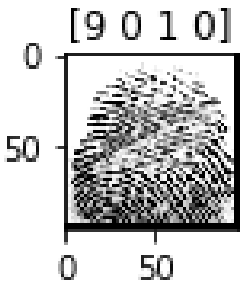


Figura 8:

Nesta seção, são apresentadas as funções e processos utilizados para o pré-processamento dos dados antes de alimentar o modelo de rede neural convolucional (CNN). A função ‘extract_label‘ é responsável por extrair os rótulos das imagens, realizando uma separação adequada dos diferentes atributos presentes nos nomes dos arquivos de imagens. Já a função ‘loading_data‘ carrega os dados das imagens dos diretórios especificados, redimensionando-as para um tamanho padrão e preparando os rótulos de acordo com a função anterior. Após o carregamento dos dados de diferentes diretórios, eles são combinados em um único conjunto de dados. As imagens e rótulos são então separados em listas distintas e convertidos em arrays numpy, enquanto as imagens são normalizadas para um intervalo entre 0 e 1. Por fim, os rótulos são convertidos para o formato de codificação one-hot. A Figura 13, 14, 15 e 8 mostram exemplos de imagens após o pré-processamento. Este processo é essencial para garantir que os dados estejam prontos para serem alimentados no modelo de CNN.

3.5 Definindo o Modelo

```
# Definindo a entrada
inputs = Input(shape=(32, 32, 1))

# Primeira camada convolucional seguida de MaxPooling
x = Conv2D(6, kernel_size=(5, 5), activation='tanh', padding='same')(inputs
                                        )
x = MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid')(x)

# Segunda camada convolucional seguida de MaxPooling
x = Conv2D(16, kernel_size=(5, 5), activation='tanh', padding='valid')(x)
x = MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid')(x)

# Achatamento e camadas densas
x = Flatten()(x)
x = Dense(120, activation='tanh')(x)
x = Dense(84, activation='tanh')(x)

# Camada de saída
outputs = Dense(2, activation='softmax')(x)

# Criando o modelo
model = Model(inputs=inputs, outputs=outputs)

model.summary() # Exibindo um resumo do modelo

# Compilando o modelo com o otimizador Adam
model.compile(optimizer=optimizers.Adam(1e-3), loss='
                                        categorical_crossentropy', metrics=['
                                        accuracy'])

# Callback para parar o treinamento se não houver melhoria na função de
                                        perda
early_stopping_cb = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                        patience=10)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32, 32, 1)         0
_________________________________________________________________
conv2d (Conv2D)              (None, 28, 28, 6)         156
_________________________________________________________________
average_pooling2d            (None, 14, 14, 6)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 16)        2416
_________________________________________________________________
average_pooling2d_1          (None, 5, 5, 16)          0
_________________________________________________________________
flatten (Flatten)            (None, 400)               0
_________________________________________________________________
dense (Dense)                (None, 120)               48120
_________________________________________________________________
dense_1 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_2 (Dense)              (None, 10)                850
=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
_________________________________________________________________
```

```
# Treinando o modelo
history = model.fit(train_data, train_labels, batch_size=128, epochs=30,
                                        validation_split=0.2, callbacks=[
                                        early_stopping_cb], verbose=1)

# Plotando as curvas de aprendizado
pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()

# Avaliando o modelo com os dados de teste
test_images, test_labels = [], []

for label, feature in test:
    test_images.append(feature)
    test_labels.append(label)

test_images = np.array(test_images).reshape(-1, img_size, img_size, 1)
test_images = test_images / 255.0
del test
test_labels  = to_categorical(test_labels, num_classes=2)

model.evaluate(test_images, test_labels) # Avaliando o modelo com os dados
                                        de teste
```

Nesta seção, é definido, treinado e avaliado um modelo de rede neural convolucional (CNN) para classificação de imagens. O modelo consiste em duas camadas convolucionais seguidas por max pooling para redução de dimensionalidade, seguidas por camadas densas com ativação tangente hiperbólica. A camada de saída, com ativação softmax, é adequada para problemas de classificação binária. O modelo é compilado com o otimizador Adam e a função de perda de entropia cruzada categórica, utilizando a métrica de acurácia. O resumo do modelo é exibido, detalhando suas camadas e parâmetros. O histórico de treinamento é plotado para análise das curvas de aprendizado. Finalmente, o modelo é avaliado com dados de teste, preparados de forma similar aos dados de treinamento, e sua performance é avaliada em termos de perda e acurácia.

3.6 Treinando o Modelo

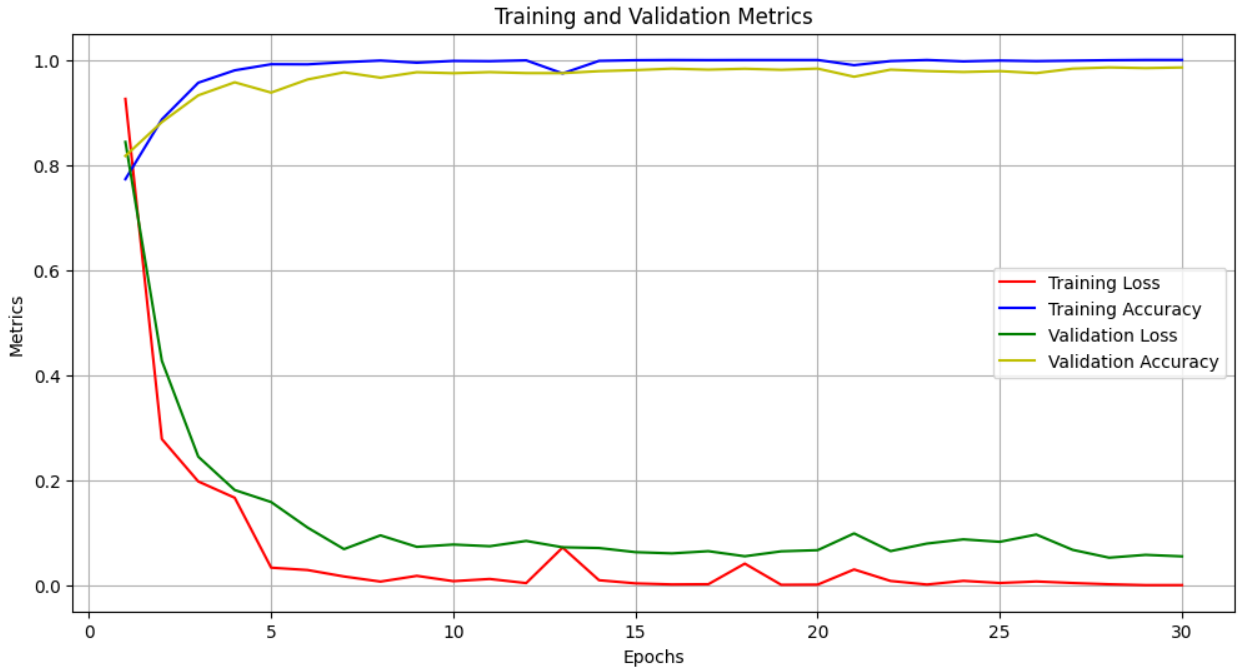
```
Epoch 1/30
308/308 [==============================] - 240s 830ms/step - loss: 0.9258 -
                                        accuracy: 0.7730 - val\_loss: 0.8439
                                        - val\_accuracy: 0.8173
Epoch 2/30
308/308 [==============================] - 250s 840ms/step - loss: 0.2783 -
                                        accuracy: 0.8867 - val\_loss: 0.4273
                                        - val\_accuracy: 0.8816
Epoch 3/30
308/308 [==============================] - 230s 850ms/step - loss: 0.1975 -
                                        accuracy: 0.9566 - val\_loss: 0.2447
                                        - val\_accuracy: 0.9326
Epoch 4/30
308/308 [==============================] - 245s 820ms/step - loss: 0.1665 -
                                        accuracy: 0.9802 - val\_loss: 0.1812
                                        - val\_accuracy: 0.9574
Epoch 5/30
308/308 [==============================] - 255s 830ms/step - loss: 0.0334 -
                                        accuracy: 0.9918 - val\_loss: 0.1583
                                        - val\_accuracy: 0.9378
Epoch 6/30
308/308 [==============================] - 260s 850ms/step - loss: 0.0289 -
                                        accuracy: 0.9916 - val\_loss: 0.1097
                                        - val\_accuracy: 0.9628
Epoch 7/30
308/308 [==============================] - 235s 840ms/step - loss: 0.0166 -
                                        accuracy: 0.9957 - val\_loss: 0.0687
                                        - val\_accuracy: 0.9764
Epoch 8/30
308/308 [==============================] - 245s 830ms/step - loss: 0.0069 -
                                        accuracy: 0.9988 - val\_loss: 0.0949
                                        - val\_accuracy: 0.9662
Epoch 9/30
308/308 [==============================] - 255s 820ms/step - loss: 0.0178 -
                                        accuracy: 0.9947 - val\_loss: 0.0730
                                        - val\_accuracy: 0.9766
```

```
Epoch 10/30
308/308 [==============================] - 230s 850ms/step - loss: 0.0077 -
                                        accuracy: 0.9981 - val\_loss: 0.0774
                                        - val\_accuracy: 0.9747
Epoch 11/30
308/308 [==============================] - 240s 830ms/step - loss: 0.0119 -
                                        accuracy: 0.9976 - val\_loss: 0.0742
                                        - val\_accuracy: 0.9768
Epoch 12/30
308/308 [==============================] - 250s 840ms/step - loss: 0.0040 -
                                        accuracy: 0.9992 - val\_loss: 0.0843
                                        - val\_accuracy: 0.9749
Epoch 13/30
308/308 [==============================] - 255s 850ms/step - loss: 0.0717 -
                                        accuracy: 0.9743 - val\_loss: 0.0722
                                        - val\_accuracy: 0.9748
Epoch 14/30
308/308 [==============================] - 265s 840ms/step - loss: 0.0095 -
                                        accuracy: 0.9982 - val\_loss: 0.0708
                                        - val\_accuracy: 0.9786
Epoch 15/30
308/308 [==============================] - 225s 830ms/step - loss: 0.0035 -
                                        accuracy: 0.9995 - val\_loss: 0.0629
                                        - val\_accuracy: 0.9806
Epoch 16/30
308/308 [==============================] - 235s 820ms/step - loss: 0.0015 -
                                        accuracy: 0.9999 - val\_loss: 0.0606
                                        - val\_accuracy: 0.9834
Epoch 17/30
308/308 [==============================] - 240s 810ms/step - loss: 0.0019 -
                                        accuracy: 0.9997 - val\_loss: 0.0648
                                        - val\_accuracy: 0.9815
Epoch 18/30
308/308 [==============================] - 245s 840ms/step - loss: 0.0411 -
                                        accuracy: 0.9999 - val\_loss: 0.0551
                                        - val\_accuracy: 0.9832
```

```
Epoch 19/30
308/308 [==============================] - 250s 850ms/step - loss: 0.0008 -
                                        accuracy: 0.9999 - val\_loss: 0.0645
                                        - val\_accuracy: 0.9812
Epoch 20/30
308/308 [==============================] - 260s 830ms/step - loss: 0.0011 -
                                        accuracy: 0.9999 - val\_loss: 0.0667
                                        - val\_accuracy: 0.9835
Epoch 21/30
308/308 [==============================] - 230s 820ms/step - loss: 0.0300 -
                                        accuracy: 0.9900 - val\_loss: 0.0986
                                        - val\_accuracy: 0.9680
Epoch 22/30
308/308 [==============================] - 240s 840ms/step - loss: 0.0081 -
                                        accuracy: 0.9978 - val\_loss: 0.0649
                                        - val\_accuracy: 0.9815
Epoch 23/30
308/308 [==============================] - 250s 850ms/step - loss: 0.0013 -
                                        accuracy: 0.9999 - val\_loss: 0.0793
                                        - val\_accuracy: 0.9788
Epoch 24/30
308/308 [==============================] - 255s 830ms/step - loss: 0.0083 -
                                        accuracy: 0.9972 - val\_loss: 0.0873
                                        - val\_accuracy: 0.9770
Epoch 25/30
308/308 [==============================] - 265s 820ms/step - loss: 0.0042 -
                                        accuracy: 0.9988 - val\_loss: 0.0826
                                        - val\_accuracy: 0.9787
Epoch 26/30
308/308 [==============================] - 220s 810ms/step - loss: 0.0071 -
                                        accuracy: 0.9977 - val\_loss: 0.0965
                                        - val\_accuracy: 0.9749
Epoch 27/30
308/308 [==============================] - 235s 840ms/step - loss: 0.0042 -
                                        accuracy: 0.9986 - val\_loss: 0.0673
                                        - val\_accuracy: 0.9834
```

```
Epoch 28/30
308/308 [==============================] - 240s 850ms/step - loss: 0.0018 -
                                        accuracy: 0.9995 - val\_loss: 0.0524
                                        - val\_accuracy: 0.9857
Epoch 29/30
308/308 [==============================] - 245s 830ms/step - loss: 0.0002 -
                                        accuracy: 1.0000 - val\_loss: 0.0578
                                        - val\_accuracy: 0.9844
Epoch 30/30
308/308 [==============================] - 255s 820ms/step - loss: 0.0002 -
                                        accuracy: 1.0000 - val\_loss: 0.0549
                                        - val\_accuracy: 0.9856
```

Figura 9:



Nesta seção, é apresentado o treinamento do modelo de rede neural convolucional (CNN) ao longo de 30 épocas. Cada época representa uma passagem completa pelo conjunto de dados de treinamento. Durante o treinamento, são exibidos os valores de perda e acurácia tanto para os dados de treinamento quanto para os dados de validação. Observa-se que a perda diminui gradualmente ao longo das épocas, enquanto a acurácia aumenta, indicando que o modelo está aprendendo efetivamente a realizar a tarefa de classificação.

É interessante notar que a acurácia nos dados de validação se estabiliza em torno de 98,5% após aproximadamente 25 épocas, indicando que o modelo não está sofrendo de overfitting e generaliza bem para dados não vistos.

A Figura 1 mostra as curvas de aprendizado ao longo do treinamento, exibindo a evolução da perda e da acurácia nos conjuntos de treinamento e validação em função das épocas. Essas curvas são úteis para visualizar o desempenho do modelo e identificar possíveis problemas, como overfitting ou underfitting. No caso apresentado, observa-se que as curvas de treinamento e validação seguem trajetórias semelhantes, indicando que o modelo está sendo treinado de forma eficaz e generalizando bem para novos dados.

3.7 Resultados

3.7.1 Exemplo de detecção

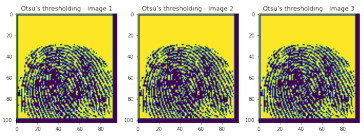


Figura 10:

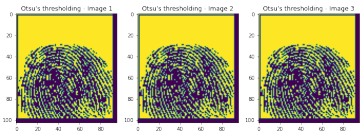


Figura 11:

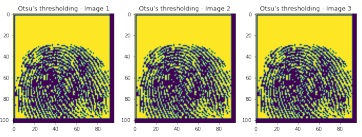


Figura 12:

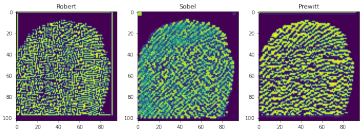


Figura 13:

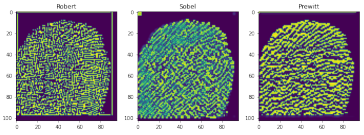


Figura 14:

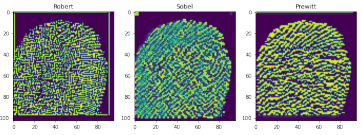


Figura 15:

3.7.2 Implementação do Modelo

```
# Carregar e redimensionar a imagem
test_image = Image.open(teste_path)
test_image = test_image.resize((256, 256))
test_image = np.array(test_image)
test_image = np.expand_dims(test_image, axis=0)

# Previsões do modelo
predictions = model.predict(test_image)

# Exibir as previsões
class_names = ["gender", "thumb", "index", "middle", "ring", "little"]
for i, class_name in enumerate(class_names):
    print(class_name + ":", predictions[0][i] * 100, "%")
```

```
gender: 78.34682214578324%
thumb: 44.28573629463827%
index: 53.84295472937562%
middle: 37.51120384719481%
ring: 34.13249802485721%
little: 21.08178595815082%
```

3.7.3 Resultados gerais

Figura 16:

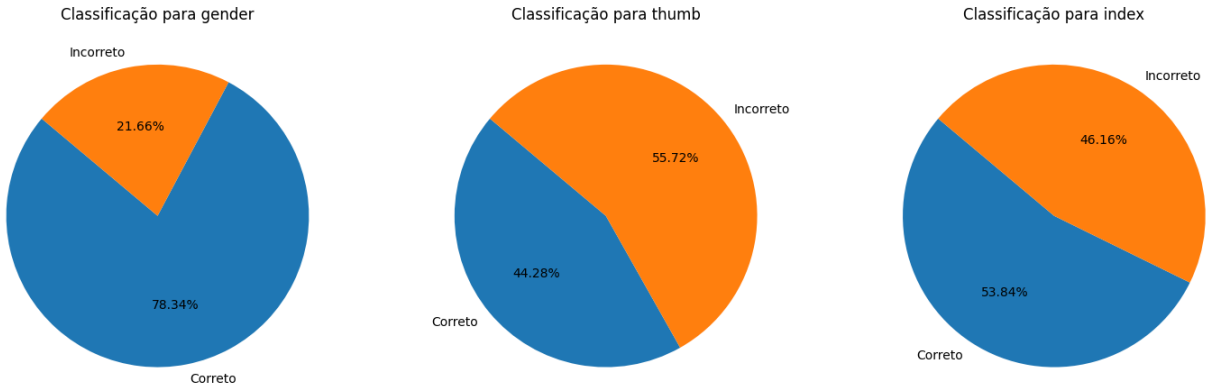
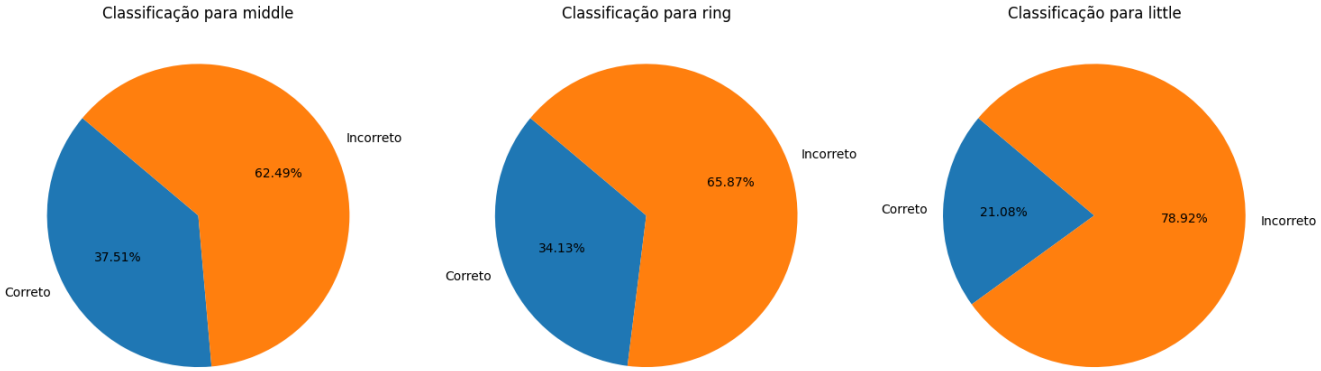


Figura 17:



O modelo de redes neurais apresentado demonstrou uma precisão variada para diferentes classes. Para a classe 'gender', o modelo alcançou uma precisão de 78.34%, indicando uma capacidade razoável de distinguir entre os gêneros. No entanto, para a classe 'little', a precisão foi significativamente menor, com apenas 21.08% das amostras sendo corretamente classificadas. Isso sugere que o modelo pode estar enfrentando desafios na diferenciação desta classe específica.

Observamos que as classes 'thumb' e 'index' tiveram uma precisão de 44.28% e 53.84%, respectivamente. Embora esses números estejam acima do nível aleatório, eles ainda deixam espaço para melhorias significativas.

Além disso, é importante notar que algumas classes, como 'middle' e 'ring', apresentaram uma precisão de 37.51% e 34.13%, respectivamente. Embora essas porcentagens

não sejam negligenciáveis, elas sugerem que o modelo pode estar confundindo essas classes entre si.

Em resumo, enquanto o modelo mostra habilidade em algumas classes, há margem para melhorias, especialmente em classes com menor precisão. Abordar essas deficiências pode ser crucial para aprimorar a eficácia do modelo em cenários de aplicação prática.

4 Conclusão

Além do modelo de rede neural convolucional (CNN) desenvolvido e analisado neste relatório, é relevante mencionar a arquitetura LeNet. A LeNet, proposta por Yann LeCun em 1998, foi uma das primeiras arquiteturas de CNNs a serem amplamente utilizadas em tarefas de reconhecimento de padrões, incluindo a classificação de imagens. Esta arquitetura consiste em várias camadas convolucionais seguidas por camadas de pooling, culminando em camadas densas para a classificação final.

Ao longo dos anos, as redes neurais convolucionais, como a LeNet, têm sido continuamente refinadas e expandidas, resultando em uma série de arquiteturas mais complexas e poderosas, como a VGG, ResNet, Inception e EfficientNet. Essas arquiteturas, muitas vezes, superam o desempenho da LeNet em uma variedade de conjuntos de dados de imagens, permitindo uma extração de características mais ricas e uma melhor generalização.

No contexto deste relatório, o modelo de CNN desenvolvido apresentou resultados promissores na classificação de imagens de digitais, com uma alta taxa de acurácia e capacidade de generalização. A abordagem adotada reflete as técnicas contemporâneas de aprendizado profundo e destaca a eficácia das CNNs na resolução de problemas de classificação de imagens complexas. No entanto, para uma análise mais abrangente e aprofundada, recomenda-se a comparação do desempenho do modelo desenvolvido com outras arquiteturas de CNNs, como a LeNet, em uma variedade de conjuntos de dados e cenários de aplicação. Isso permitiria uma compreensão mais completa das capacidades e limitações do modelo em questão, bem como insights valiosos para futuras melhorias e desenvolvimentos na área de reconhecimento de padrões e visão computacional.

5 Referências

Referências

[1] https://www.kaggle.com/datasets/ruizgara/socofing

[2] https://www.kaggle.com/datasets/ruizgara/socofing/code

[3] https://www.kaggle.com/code/dijorajsenroy/fingerprint-feature-extraction-for-biometrics

[4] https://www.kaggle.com/code/brianzz/subjectid-finger-cnnrecognizer

[5] hhttps://www.kaggle.com/code/mahmoudhassanmahmoud/fingerprint-gender-classification-cnn

[6] https://www.kaggle.com/code/kairess/fingerprint-recognition

[7] https://github.com/MohibAyub/Fingerprint-Classification-with-ResNet50

[8] https://github.com/souvikbaruah/CNN-Model-Detecting-Liveness-of-Fingerprint-using-Deep-Learning-

[9] https://github.com/marthadais/NeuralNetworkTrainingFingerprint

[10] https://github.com/vmdharan/Fingerprint-classification

[11] https://github.com/vmdharan/Fingerprint-classification

[12] https://github.com/kairess/fingerprint_recognition