PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

Trabalho Prático de Processamento de Imagens*

Arthur Henrique Rodrigues Teixeira[1]
Eduardo Braz de Campos[2]
Túlio Nunes Polido Lopes[3]

Resumo

O presente trabalho consiste em desenvolver um software com interface que explora os conceitos teóricos de processamento de imagens. Este software deve abrir imagens TIFF, DICOM e PNG, para que possam ser analisadas e classificadas com a utilização de redes neurais.

*Trabalho Teórico Prático apresentado na disciplina de Projeto e Análise de Algoritmos do curso de Ciência da Computação da Pontifícia Universidade Católica de Minas Gerais sobre Travelling Salesman Problem(TSP).

[1]Aluno do Programa de Graduação em Ciência da Computação, Brasil – arthur.teixeira@sga.pucminas.br.
[2]Aluno do Programa de Graduação em Engenharia da Computação, Brasil – ebcampos@sga.pucminas.br.
[3]Aluno do Programa de Graduação em Ciência da Computação, Brasil – tulio.polido@sga.pucminas.br.

1 INTRODUÇÃO

O desenvolvimento das técnicas de processamento de imagens digitais ocorrido nas últimas décadas propiciou o surgimento de uma grande variedade de aplicações em diversas áreas e aliado a expensão do aprendizado de máquina, em especial às Redes Neurais, propôs-se o desenvolvimento de um software capaz de receber imagens de exames mamográficos e classificá-las de acordo com a escala de densidade BIRADS. Para tal, foram utilizados dois descritores de texturas e um banco de dados com quatrocentas imagens dividias igualmente em quatro grupos de acordo com sua classificação BIRADS e, com o desenvolvimento de uma rede neural multicamada (MLP), criou-se um modelo de classificação de densidade mamária. As imagens selecionadas pelo usuário são reamostradas em relação ao seu número de tons de cinza e por fim a rede neural classifica em qual classe BIRADS a imagem inserida pertence.

2 IMPLEMENTAÇÃO

Nesta seção será abordada a implementação do software. As bibliotecas usadas serão citadas e as principais partes do código serão explicadas.

2.1 Listagem e instalação de bibliotecas

Bibliotecas importadas

```
import tkinter as tk    #Biblioteca para desenvolvimento
                        #de interface
from PIL import Image, ImageTk #Bib para
                               #tratamento de imagens
import os # Bib para acesso aos arquivos do S.O.
import pydicom #Bib para tratamento de imagens DICOM
import cv2 #Bib para Visao Computacional
import mahotas as mt #Bib para proc. de imagens
import matplotlib.pyplot as plt #Bib para
                                #plotagem de graficos
import numpy as np #Bib para processamento numerico
import time #Bib para calculos temporais
from math import ceil, copysign, log10

#Classe e funcoes para Rede Neural
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.neural_network import MLPClassifier
```

A instalação de todas a bibliotecas citadas acima podem ser feitas através do instalador de pacotes do Python, conhecido como PIP. No software descrito nesse artigo, a versão utilizada foi o PIP3, específico para o Python3. No sistema operacional Ubuntu, pode-se instalar o PIP3 através do comando:

Instalação do PIP no Ubuntu

```
$ sudo apt install python3-pip
```

Com o PIP3 instalado, cada uma das bibliotecas citadas podem ser instaladas utilizando o mesmo comando:

Instalação das Bibliotecas

```
$ pip3 install *nome_da_biblioteca*
```

Este comando deve ser feito para cada uma das bibliotecas citadas que não são instaladas por padrão com o Python3.

2.2 Algoritmo

2.2.1 Pré-definições

O software é desenvolvido utilizando a biblioteca Tkinter que é uma biblioteca para desenvolvimento de interfaces. No início do método principal da classe App, são inicializadas as variáveis globais que serão utilizadas pelo algoritmo. Em seguida a tela do software é definida, com a declaração de todos os botões presentes no menu de opções. Então, duas áreas de exibição de imagem são criadas, la, que exibe a imagem aberta pelo usuário, e la2, que exibe o recorte feito pelo usuário.

Resumo do método init(main)

```
...
#variaveis globais
Frame.__init__(self, master)
self.master.title('Trabalho de Processamento de Imagens')
[...]
self.Contraste = True
self.caracteristicas = [self.Entropia, self.Energia,
    self.Homogeneidade, self.Contraste]


#Tela do software
fram = Frame(self)
Button(fram, text="Abrir imagem", command=self.open).pack(side=LEFT)
[...]
Button(fram, text="Treinar classificador",
 command=self.trein_clas,bg='gray').pack(side=LEFT)
fram.pack(side=TOP, fill=BOTH)

#Area em que a imagem ficara presente
self.la = Label(self)
self.la.pack()

#Area que o recorte ficara presente
self.la2 = Label(self)
self.la2.pack(side=BOTTOM)

self.pack()
```

2.2.2 Leitura de imagens

Assim que o usuário seleciona o botão de "Abrir imagem", o método open() é chamado. A primeira verificação feita é se já existe uma imagem aberta no canvas, caso não exista ele procede para a abertura da imagem conforme mostrado a seguir:

Parte da função open()

```
...
if self.filename != "":
    if self.filename.endswith('.png') or self.filename.endswith('.tif
        self.im =  Image.open(self.filename)
        self.img_atual =  Image.open(self.filename)
        self.width = self.im.width
        self.height = self.im.height
        self.chg_image()
...
```

Primeiro, é verificado se o arquivo a ser aberto não é um arquivo vazio. Caso não seja, a próxima verificação é se ele é o tipo PNG ou TIFF. Sendo um deles, a imagem é aberta utilizando a classe Image da biblioteca Pillow em duas variáveis, self.im e self.img_atual. A primeira guarda sempre a imagem no formato original, já a segunda é constantemente atualizada de acordo com as opções de zoom do usuário. Então, as variáveis globais que guardam os valores de pixels na altura e na largura são atualizadas. Finalmente, o método chg_image() é chamado para utilizar essas variáveis globais a fim de atualizar a imagem na tela do software.

Caso a imagem seja do formato DICOM, para abri-la é necessário fazer uma conversão. Essa conversão é feita no método convert_to_png() que utiliza a biblioteca pydicom para abrir a imagem e transformá-la num array de bits. Essa array é, posteriormente, exibido como uma imagem utilizando a biblioteca matplotlib. Então essa imagem é salva com o nome genérico "dicom.png", e, a partir desse momento, o código segue o mesmo fluxo que seguiria caso fosse abrir diretamente uma imagem PNG ou TIFF.

2.2.3 Recorte de imagem

O botão "Selecionar Área"permite ao usuário fazer o recorte de uma área 128x128 pixels na imagem para ser analisada. Ele é controlado por uma função mais complexa que as citadas anteriormente. Inicialmente ele verifica se a imagem selecionada não é vazia, além de conferir se o tamanho dela é maior que 128x128 pixels pra ser possível fazer o recorte.

Caso seja uma imagem válida, são criadas as variáveis de controle topx, topy, botx e boty que definem são responsáveis por guardar os valores de limites da janela 128x128 a ser recortada. E também há a variável rect_id que receberá o valor do retângulo a ser exibido no

software para visualização do usuário. Por fim, há a ligação entre o botão esquerdo do mouse e as funções get_mouse_posn(), que verifica a posição do mouse na tela ao ser pressionado apenas uma vez, e a função confirm_cut(), que recorta a área na posição atual do mouse ao receber dois cliques.

A última verificação a ser feita antes de realizar o recorte, é se há uma imagem atualmente aberta no label de exibição, caso haja, o label é fechado e a imagem é aberta no canvas. Então o algoritmo segue da seguinte forma:

Parte da função select_area()

```
...
aux_img = Image.open(self.filename)
aux_img = aux_img.resize((self.width, self.height))
img = ImageTk.PhotoImage(aux_img)
self.canvas = tk.Canvas(self.la, width=img.width(),
    height=img.height(), borderwidth=0, highlightthickness=0)
self.canvas.pack(expand=True)
self.canvas.img = img
self.canvas.create_image(0, 0, image=img, anchor=tk.NW)
self.temCanvas = True

# Desenha retangulo verde em cima da imagem
rect_id = self.canvas.create_rectangle(topx, topy, botx, boty,
    fill='',outline='LimeGreen', width=2)
...
```

Inicialmente a imagem é aberta numa variável auxiliar e é feito um redimensionamento da mesma para o zoom selecionado anteriormente pelo usuário. Posteriormente essa imagem auxiliar é aberta pelo método PhotoImage da classe ImageTK, para que possa ser exibida pelo software. Um canvas é criado tendo como dimensões, as próprias da imagem aberta e o retângulo é desenhado na cor verde. Caso o usuário clique com o mouse em outra área da imagem, esse retângulo é redesenhado tendo como centro o ponto exato de clique do usuário.

2.2.4 Zoom In/Out

As funções de zoom são de implementação simples. Ambas conferem se há uma imagem aberta. Caso haja, o tamanho original da imagem é salvo em variáveis globais. Então o software verifica se a imagem já passa do limite superior(para zoom in) ou inferior(para zoom out), e apenas se ela não ultrapassar os limites é que o zoom é aplicado. O tamanho da imagem é recalculado e enfim o método chg_image() é chamado para que a imagem seja atualizada da tela.

2.2.5 Leitura do diretório de treinamento

Ler um diretório com as imagens de treinamento é uma opção do programa. Isso é feito pela função ler_dir() listada a seguir:

Parte da função ler_dir()

```
...
folder = filedialog.askdirectory()

for i in range(1,5):
    subFolder = folder + '/' + str(i)
    files = os.listdir(subFolder)

    for arquivo in files:
        img = cv2.imread(subFolder + '/' + arquivo)
        self.imagens.append(img)
...
```

Primeiro é requisitado ao usuário uma pasta para ser verificada. Então o programa tenta abrir essa pasta, listar as subpastas presentes nela e ler cada uma das imagens presentes nessas subpastas utilizando a biblioteca OpenCV. Essas imagens são adicionadas uma por uma a uma lista pertencente à própria classe. Caso haja algum erro nesse caminho, o software considera a pasta selecionada como inválida e então requisita novamente outro diretório.

2.2.6 Cálculos das propriedades de Haralick

Haralick é um dos descritores utilizados no software para a classificação das imagens. Uma função foi implementada para o cálculo das propriedades de Haralick para matrizes de coocorrência circulares com distâncias iguais a 1, 2, 4, 8 e 16. O método recebe uma imagem e uma lista de booleans que seleciona quais características serão analisadas.

Parte inicial da função Haralick()

```
resultado = []
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) #converte para cinza
final = gray/8
final = final.astype(int)
...
```

Inicialmente é criado uma lista vazia que irá conter os valores de Haralick para a imagem recebida. Então a imagem recebida é tratada. Primeiro ela é transformada em uma imagem em escalas de cinza. Posteriormente ocorre um reescalonamento para que os valores fiquem entre 0 e 32, e então esses valores são convertidos para números inteiros.

Loop da função Haralick()

```
...
dist = 1
while dist <= 16:
    features = mt.features.haralick(final,distance=dist)

    if(caracteristicas[0]):
        #somar homogeneidade
        parcial = (features[0][4] + features[1][4] + features[2][4]
        + features[3][4])/4
        resultado.append(parcial)
    if(caracteristicas[1]):
        #somar entropia
        parcial = (features[0][8] + features[1][8] + features[2][8]
        + features[3][8])/4
        resultado.append(parcial)
    if(caracteristicas[2]):
        #somar energia
        parcial = (features[0][0] + features[1][0] + features[2][0]
        + features[3][0])/4
        resultado.append(parcial)
    if(caracteristicas[3]):
        #somar contraste
        parcial = (features[0][1] + features[1][1] + features[2][1]
        + features[3][1])/4
        resultado.append(parcial)

    dist*=2
...
```

A variável de controle do loop é o inteiro dist que se inicia de 1. Ele vai dobrando de valor a cada interação até que atinja o valor final de 16 que é a última distância a ser calculada. A cada interação, todas as características de Haralick são calculadas para uma distância específica na imagem recebida. Isso é feito pela função haralick() da biblioteca Mahotas mencionada anteriormente. Então o loop verifica quais características foram escolhidas pelo usuário para serem analisadas e retorna um vetor com todos os valores selecionados.

2.2.7 Cálculos dos Momentos Invariantes de Hu

Além de Haralick, o software também utiliza momentos invariantes de Hu para classificar as imagens. O início é o mesmo da função Haralick(), porém há um arredondamento dos valores fracionados. Isso é feito porque o cálculo dos momentos da imagem são feitos utilizando números inteiros.

Parte da Função Hu()

```
# Arredonda os valores fracionados
for x in range(len(final)):
    for y in range(len(final[0])):
        final[x][y] = ceil(final[x][y])

moments = cv2.moments(gray) #calcula os momentos da imagem
huMoments = cv2.HuMoments(moments) #calcula os momentos de Hu
...
```

Com a matriz inteira, os momentos da imagem são calculados pela função moments() do OpenCV. E o resultado disso é utilizado para o cálculo dos momentos invariantes de Hu, também utilizando uma função própria do OpenCV. Os valores calculados são reescalados, adicionados a uma lista e retornados à função de origem.

2.2.8 Rede Neural

A rede neural foi construída utilizando a classe MLPClassifier da biblioteca sklearn. Essa classe automatiza a construção de uma rede neural multicamada permitindo ao programador selecionar um conjunto vasto de opções como a quantidade de camadas, a função de ativação, a quantidade de neurônios por camada, uma seed para geração de aleatoriedade entre outras opções.

Inicialmente, como pode ser visto no trecho de código a seguir, verificamos se o vetor de imagens da classe contém a quantidade necessária de imagens para a realização dos testes. Então iniciamos a preparação dos dados para que sejam aplicados à rede neural, onde uma lista com os labels das imagens é criado, enquanto outra lista com os valores de Hu e Haralick são unificados num único vetor a ser analisado pela rede neural.

Início da Rede Neural

```
...
if len(self.imagens) == 400:
    inicio = time.time()
    train_feat = []
    train_labels = []

    #seta o vetor de labels
    for i in range(0,400):
        train_labels.append(int(i/100) + 1)

    #Cria o vetor com os valores a serem analisados
    for imagem in self.imagens:
        val = self.Hu(imagem) + self.Haralick(imagem)
        train_feat.append(val)
...
```

Esses dados são divididos pelas suas classes, para que a divisão entre dados de treinamento e de testes esteja sempre balanceada. Os resultados dessa divisão são aplicados à função train_test_split() que os divide em 4 variáveis. Duas listas contendo 3/4 das imagens para treinamento, sendo a primeira contendo as imagens e a segunda contendo as classes corretas. E duas listas contendo 1/4 restante para testes no mesmo formato.

Divisão dos testes

```
...
#Dividir os dados nas 4 classes
Tclas1,Tclas2,Tclas3,Tclas4 = np.array_split(train_feat,4)
Lclas1,Lclas2,Lclas3,Lclas4 = np.array_split(train_labels,4)

#Balancea os dados em 75% treinamento e 25% testes
feat_train1, feat_test1, label_train1, label_test1
    = train_test_split(Tclas1, Lclas1,test_size=0.25, random_state=1)
feat_train2, feat_test2, label_train2, label_test2
    = train_test_split(Tclas2, Lclas2,test_size=0.25, random_state=1)
feat_train3, feat_test3, label_train3, label_test3
    = train_test_split(Tclas3, Lclas3,test_size=0.25, random_state=1)
feat_train4, feat_test4, label_train4, label_test4
    = train_test_split(Tclas4, Lclas4,test_size=0.25, random_state=1)
...
```

Essas listas são reunidas novamente criando quatro listas finais. Duas para treinamento, contendo os dados e as classes, e duas para testes, contendo as mesmas informações.

União da lista de testes

```
...
#Reunir os dados
feat_train = np.concatenate((feat_train1,feat_train2,
    feat_train3,feat_train4))
feat_test = np.concatenate((feat_test1,feat_test2,
    feat_test3,feat_test4))
label_train = np.concatenate((label_train1,label_train2,
    label_train3,label_train4))
label_test = np.concatenate((label_test1,label_test2,
    label_test3,label_test4))
...
```

O classificador então é criado como demonstrado no trecho a seguir:

Criação da rede neural

```
...
self.mlp = MLPClassifier(solver='lbfgs', random_state=5,
        max_iter=400, hidden_layer_sizes=[200,300])
self.mlp.fit(feat_train, label_train)
y_pred = self.mlp.predict(feat_test)
...
```

Primeiro é criado a rede neural, utilizando o construtor da classe MLPClassifier, onde o solucionador escolhido é o 'lbfgs' por ser mais eficiente em bancos de dados com pouca informação. O número máximo de iterações estabelecido é de quatrocentas iterações, e a quantidade de neurônios por camada é de, respectivamente, duzentos e trezentos.

Esse classificador, então, é preparado para a nossa base de dados utilizando a função fit() que recebe as duas listas de treinamento, sendo uma com as informações das imagens e a outra com as classes dessas respectivas imagens. O teste é aplicado com a função predict() que retorna os valores de classificação para cada imagem de teste.

A partir desse momento a rede neural está pronta para uso. O próximo passo é calcular a matriz de confusão dos testes com a função confusion_matrix() da biblioteca sklearn. Essa função recebe os labels de teste e os labels previstos, retornando os dados cruzados em uma matriz.

Trecho final da rede neural

```
...
# Calcula a matriz de confus o
cnf_matrix = confusion_matrix(label_test, y_pred)
acuracia = self.acuracia(cnf_matrix)
especificidade = self.especificidade(cnf_matrix)
...
```

Com a matriz de confusão, é possível calcular a acurácia do classificador e sua especificidade. Ambas, funções implementadas no próprio código. A acurácia retorna o somatório da diagonal principal da matriz sobre o total de imagens de teste, como mostrado no trecho a seguir:

Função acuracia()

```
...
resp = 0
for i in range(0,4):
    resp += matriz[i][i]
return resp/100
```

A especificidade pega o restante dos valores da matriz e divide sua soma pela quantidade de imagens de treinamento, no caso 300. Esse valor é subtraído de 1 e retornado para a função original.

Função especificidade()

```
...
resp = 0
for i in range(0,4):
    for j in range(0,4):
        if i != j:
            resp+= matriz[i][j]
resp = 1-(resp/300)
return resp
```

2.2.9 Analisando a área recortada

Após o treinamento do classificador de imagens, a área recortada pelo usuário pode ser analisada pelo mesmo. Ao clicar do botão analisar área, a função analisar_area() é chamada e seu primeiro passo é verificar a existência de uma imagem a ser analisada e de um classificador treinado. Em caso de positivo, a exibição da imagem é removida da tela do software.

Parte da função analisar_area()

```
...
cropped = cv2.imread('.crop.png')

inicio = time.time()
val = self.Hu(cropped) + self.Haralick(cropped)
t = time.time() - inicio

val = np.array(val)
prediction = self.mlp.predict(val.reshape(1,-1))[0]

self.printaValores(tempo=t,carac=val,classe=prediction)
...
```

No trecho acima vemos o passo a passo seguido pelo método. Primeiro é feita a leitura da imagem recortada. Então os valores de Hu e Haralick da imagem são calculados e unidos em uma única lista de informações enquanto tempo desse processo é salvo em uma variável para ser exibido posteriormente. O classificador é chamado para fazer a predição da classe da imagem e todas essas informações são repassadas ao método printaValores() que fica encarregado de compilar e formatar todas informações recebidas para exibí-las ao usuário.

3 TESTES E RESULTADOS

Os testes feitos no software tiveram como objetivo principal obter a maior acurácia da rede neural. Neles, experimentamos valores diferentes para o número máximo de iterações, tamanho variável da rede, além de selecionar características diferentes para a análise das imagens.

3.1 Teste 1

No primeiro teste, estabelecemos 5 camadas na rede. Contendo 200 neurônios cada camada. Os pesos das camadas de entrada são 12 e 200 respectivamente. Todos os descritores estavam ativos e o número máximo de iterações utilizado foi 200. Esse teste durou um tempo total de 42,20 segundos, resultando numa acurácia de 60% de acertos.

```
Camadas da rede: 5
Neurônios na camada oculta: [200, 200, 200]
Neurônios na camada de saída: 4
Pesos na camada de entrada: (12, 200)
Pesos na camada oculta: (200, 200)
Acurácia da base de treinamento: 0.60
Acurácia da base de teste: 0.62
Tempo de execução: 42.30876612663269
```

3.2 Teste 2

Neste teste o número de camadas foi reduzido para 4, além de haver também uma redução na quantidade de neurônios por camada. Com isso o esperado seria uma rede mais leve e rápida nas análises sem uma perda de acurácia significativa. O tempo de teste foi reduzido para 28,41 segundos, e ainda assim a acurácia se manteve na casa do 61% o que demonstra uma otimização grande para uma perda pequena na quantidade de acertos.

```
Camadas da rede: 4
Neurônios na camada oculta: [100, 200]
Neurônios na camada de saída: 4
Pesos na camada de entrada: (12, 100)
Pesos na camada oculta: (100, 200)
Acurácia da base de treinamento: 0.61
Acurácia da base de teste: 0.61
Tempo de execução: 28.41601061820984
```

3.3 Teste 3

No Terceiro teste, estabelecemos 5 camadas na rede. Aumentamos a quantidade de neurônios para 300 cada camada. Também aumentamos o peso da camada de entrada para 12 e 300 respectivamente, e na camada oculta para 300 e 300 respectivamente. Como esperado, esse teste demorou 100,87 segundos, um aumento significativo no tempo de execução do treinamento. Teve como resultado o aumento na taxa de acurácia na base de treinamento para 65% de acerto, mas na base de teste essa taxa foi de 60%, sendo igual à do primeiro teste.

```
Camadas da rede: 5
Neurônios na camada oculta: [300, 300, 300]
Neurônios na camada de saída: 4
Pesos na camada de entrada: (12, 300)
Pesos na camada oculta: (300, 300)
Acurácia da base de treinamento: 0.65
Acurácia da base de teste: 0.60
Tempo de execução: 100.86514091491699
```

3.4 Teste 4

No último teste utilizamos novamente 4 camadas, porém aumentamos o número de neurônios em relação ao segundo teste. A quantidade de neurônios nas camadas ocultas foram 100 e 300 respectivamente. Isso levou a um teste de 29,84 segundos, porém com uma acurácia mais alta de 63%, sendo considerado nosso teste mais eficaz.

```
Camadas da rede: 4
Neurônios na camada oculta: [100, 300]
Neurônios na camada de saída: 4
Pesos na camada de entrada: (12, 100)
Pesos na camada oculta: (100, 300)
Acurácia da base de treinamento: 0.61
Acurácia da base de teste: 0.63
Tempo de execução: 29.841143369674683
```

4 CONCLUSÃO

O diagnóstico de câncer de mama através do processamento de imagens é um tema que vem sido muito abordado nos últimos anos, e com o avanço das redes neurais, aumentou-se a eficácia dos resultados obtidos em diversos estudos. O presente trabalho possibilitou a prática no processamento digital de imagens, em especial nas áreas de descrição e classificação de imagens onde foi necessário testar diferentes técnicas e, inclusive, adicioná-las e normalizá-las afim de garantir um resultado coerente com a realidade; para que não impacta-se no resultado do aprendizado de máquina. Os resultados mais promissores encontrados na literatura, notam uma precisão de cerca de 73% de acerto na classificação de imagens mamográficas; no presente trabalho obteve-se uma acurácia média de 61% na base de testes. Portanto, se tratando de uma rede de menor potencial ao comparado com os utilizados na literatura e no mercado e levando em conta a pequena base de dados utilizada, conclui-se que os valores obtidos e os métodos selecionados para descrição da imagens são satisfatórios. Tem-se ainda que, aumentando a base de dados, analisando outras técnicas de aprendizado de máquina e diferentes formas de descrição de texturas é possível obter resultados ainda maiores.

5 ANEXOS

Listagem dos Programas :
main_dev.py - Software de classificação
imagens/ - Diretório com as imagens de treinamento
teste/ - Diretório com imagens de teste
docs/ - Diretório com documentos relativos ao trabalho

REFERÊNCIAS

CLARK, Alex. **Pillow Documentation**. [S.l.]: Pillow Docs, 2020. <https://pillow.readthedocs.io/en/stable/>.

HARALICK, Robert M; SHANMUGAM, Karthikeyan; DINSTEIN, Its' Hak. Textural features for image classification. **IEEE Transactions on systems, man, and cybernetics**, Ieee, n. 6, p. 610–621, 1973.

HUANG, Zhihu; LENG, Jinsong. Analysis of hu's moment invariants on image scaling and rotation. In: IEEE. **2010 2nd International Conference on Computer Engineering and Technology**. [S.l.], 2010. v. 7, p. V7–476.

MATPLOTLIB Documentation. [S.l.]: Matplotlib Organization, 2020. <https://matplotlib.org/3.3.2/contents.html>.

OPENCV Documentation. [S.l.]: OpenCV Docs, 2020. <https://docs.opencv.org/master/d6/d00/tutorial_py_root.html>.

PYDICOM Documentation. [S.l.]: GitHub, 2020. <https://pydicom.github.io/pydicom/stable/old/pydicom_user_guide.html>.

TKINTER Documentation. [S.l.]: Python Organization, 2020. <https://docs.python.org/3/library/tk.html>.