COORDENADAS DE CÂMERA USANDO REDES NEURAIS PROFUNDAS

RODRIGO REIS DA ROCHA

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientador: José Gabriel Rodriguez Carneiro Gomes

RIO DE JANEIRO

NOVEMBRO DE 2022

COORDENADAS DE CÂMERA COM REDES NEURAIS PROFUNDAS

Rodrigo Reis da Rocha

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Orientador: José Gabriel Rodriguez Carneiro Gomes

Aprovado por: Mariane Rembold Petraglia

Charles Bezerra do Prado

RIO DE JANEIRO, RJ – BRASIL

NOVEMBRO DE 2022

Da Rocha, Rodrigo Reis

Coordenadas de Câmera com Rede Neurais Profundas / Rodrigo Reis da Rocha – Rio de Janeiro: UFRJ/COPPE, 2022.

XIII, 40 p.: il.; 29,7 cm.

Orientador: José Gabriel Rodriguez Carneiro Gomes

Dissertação (mestrado) – UFRJ / COPPE / Programa de Engenharia Elétrica, 2022.

Referências Bibliográficas: p. 41 – 44.

1. Redes Neurais. 2. Redes Profundas. 3. Posicionamento de câmera. 4. Homografia. 5. OpenCV. I. José Gabriel Rodriguez Carneiro Gomes. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

A todos que querem um país melhor para todos os brasileiros não importando sua origem ou opinião.

Agradecimentos

A minha família que sempre me apoiou e me deu forças em tudo que faço e que é responsável por eu ser o homem que sou hoje.

Ao professor José Gabriel que demonstrou uma capacidade insuperável de inspiração e orientação durante a confecção deste trabalho.

Ao amigo Ricardo Silva pela ajuda e força ao longo de todo o curso, sendo um grande apoio durante todas as etapas do curso.

A todos os professores do PEE, que dedicaram seu tempo e esforço para ensinar com maestria a alunos sortudos como eu que tem a oportunidade de estudar nesta instituição espetacular da COPPE.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

COORDENADAS DE CAMERA USANDO NEURAIS PROFUNDAS

Rodrigo Rocha

NOVEMBRO DE 2022

Orientador: José Gabriel Rodriguez Carneiro Gomes

Programa: Engenharia Elétrica

Apresenta-se, nesta tese, um estudo da utilização de redes neurais profundas conhecidas aplicadas a detecção de posição de câmera, em forma direta, com apenas uma imagem. É explorada a utilização de extração de características das redes profundas VGG, Resnet e Mobilenet, para treinamento de uma rede densamente conectada gerando parâmetros de câmera. É realizada a comparação entre o método proposto por este trabalho e um utilizando a biblioteca OpenCV, onde, o método proposto atinge um erro médio quadrático de 0,1493 versus 0,1766 obtido pelo método que utiliza OpenCV.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

CAMERA POSITION ESTIMATION WITH DEEP NEURAL NETWORKS

Rodrigo Rocha

NOVEMBER 2022

Advisor: José Gabriel Rodriguez Carneiro Gomes

Department: Electrical Engineering

In this work, we present a study of the use of known deep neural networks applied to the detection of camera positional parameters directly from only one image. The feature extraction of known deep neural networks VGG, Resnet and Mobilenet, is used in the training of a fully connected network to generate the camera parameters. We perform a comparison between the proposed method and one that uses the OpenCV library, in which the methods proposed achieved the result of 0,1493 on mean square error compared to 0,1766 of the OpenCV.

Sumário

Lista de figuras

Lista de Tabelas

1 Introdução

Este trabalho apresenta um estudo da utilização de redes neurais profundas VGG16 (Simonyan *et al*., 2014), Resnet (He *et al*., 2016), Mobilenet (Howard *et al*., 2017) e o processo de geração de parâmetros de câmera baseado em geometria epipolar e homografia do OpenCV (Bradski, 2000) para a extração de parâmetros de posicionamento de uma câmera de um vídeo padrão criado na ferramenta 3-D chamada Blender (Hess, 2010).

Atualmente existem vários trabalhos que utilizam redes neurais para o reconhecimento de padrões em imagens[1] seja para classificação, identificação ou quantificação (regressão), relacionados à acuidade visual. Existem técnicas clássicas conhecidas, utilizadas em SLAM visual (*Spatial Localization and Mapping* – Mapeamento e Localização Especial), que determinam a localização e posição espacial da câmera para a localização espacial dentro do contexto sendo analisado (Taketomi *et al*., 2017). Entretanto, é importante frisar que o presente estudo não utiliza técnicas de SLAM visual.

Foram analisados os resultados do uso de características encontradas pelas redes neurais profundas, já mencionadas, com o intuito de gerar o mapeamento do posicionamento de câmera, isto é, gerar a matriz de rotação e translação, diretamente da imagem captada pela mesma somente com a rede neural. Também foi realizada uma comparação de resultados com os algoritmos utilizados na API (*Application Programming Interface* – Interface de Programaçao de Aplicações) OpenCV para o cálculo de homografias com geometria

[1] (Bruno *et al*., 2021), (Guangqiang *et al*., 2021), (Ahmed *et al*. 1999), (Pedra *et al*. 2013), (Yongtae, 1999)

epipolar. Os resultados foram listados com as conclusões do estudo comparativo entre os dois métodos.

1.1 Objetivo

O objetivo principal deste trabalho é definir um método para obtenção de informações de posicionamento de câmera (matriz de rotação e translação) de forma rápida e precisa somente da análise de uma imagem com redes neurais profundas.

Dentro deste objetivo principal, procuramos estudar a eficácia de redes neurais já em uso e bem-sucedidas em identificação visual como VGG16, Resnet e Mobilenet, para verificar se a extração de características feita por estas redes pode ser igualmente aplicada com sucesso para a identificação de parâmetros extrínsecos de posicionamento de câmera. Os parâmetros extrínsecos de uma câmera não se referem às correlações de distorções causadas por propriedades internas da câmera definidas pela lente ou pelo sensor, que seriam parâmetros intrínsecos, mas informações relacionadas somente ao posicionamento espacial da câmera. Neste trabalho, vamos nos referir, na Seção 2.2, à regressão não-linear dos parâmetros de posicionamento da câmera, não obstante a da regressão não-linear dos parâmetros de calibração da mesma.

E, por fim, procuramos aplicar como referência um método, que não utiliza redes neurais (método com OpenCV e cálculos de homografia baseados em geometria epipolar), para verificar e validar a eficácia do processo inerente às redes neurais profundas.

1.2 Ambiente Computacional Utilizado no Trabalho

Os modelos neste trabalho foram implementados em Python com uso da ferramenta Jupyter Notebook aplicando as bibliotecas Numpy (Harris *et al*., 2020), TensorFlow (Abadi *et al*., 2015), Keras (Chollet *et al*., 2015) e OpenCV (Bradski, 2000). O treinamento, a validação e o teste do modelo, foram realizados em um computador com 16 GB de RAM, placa de vídeo GTX 1050 e processador AMD Phenom (tm) II X4 955 @ 3,2 GHz com 4 núcleos com uma média de 30 minutos para processamento de 30.000 imagens.

A linguagem Python (Rossum, 2009) está entre uma das linguagens de programação mais comuns para inteligência computacional, assim como as bibliotecas Tensorflow e Keras. Neste trabalho foi utilizado o interpretador interativo Jupyter Notebook[2], que é uma ferramenta de programação.

1.3 Motivação

Durante o curso do mestrado da linha de pesquisa em redes neurais estudamos uma parte da grande variação de possibilidades dentro do espaço de conhecimento da inteligência computacional e com isso vemos os avanços técnicos já atingidos por estudiosos da área.

O campo da identificação visual está chegando a resultados aproximados da capacidade humana no que concerne a obtenção de características e de reconhecimento de imagens. Estes resultados impressionantes provavelmente podem ser aplicados a processos antes realizados por algoritmos baseados em geometria projetiva (Hartley e Zisserman, 2004), (Faugeras, 1993) para realizar a localização espacial da câmera.

[2] https://jupyter.org/

Este estudo objetiva a verificação da flexibilidade da aplicação de redes neurais já aplicadas em trabalhos anteriores para o processamento de imagens, que atingiram resultados importantes apresentados nas Seções 2.4, 2.5 e 2.6, e a capacidade de generalização dessas redes neurais.

1.4 Trabalhos Correlatos

Um método robusto de calibração de câmeras estéreo, por meio de técnicas de otimização para a solução dos problemas dos mínimos quadrados, foi apresentado em (Silva, 2003). Este método foi aplicado para correspondência entre pontos homólogos em pares de imagens estéreo com compensação de distorções introduzidas por efeitos de lentes.

Em 2013, com uso de uma rede neural com três camadas, foram obtidos parâmetros posicionais de câmera diretamente para uma aproximação em 3-D (Pedra *et al*., 2013). Os resultados obtidos demonstram vantagens no uso dessa técnica para realização deste processo.

São propostos métodos de obtenção de posição de câmera, em imagens estereoscópicas, utilizando redes neurais em quatro aproximações distintas em (Do, 1999). O estudo aponta que o mapeamento das não linearidades intrínsecas das distorções causadas pelas lentes em imagens capturadas é possível utilizando inteligência computacional. Em métodos baseados em geometria epipolar (Hartley e Zisserman, 2004), (Faugeras, 1993), as distorções precisam ser obtidas anteriormente ao processamento das imagens. Essas distorções, ou não linearidades, podem ser do tipo barril, almofada ou

complexas, dependendo do tipo de lentes, mas como são inerentes às lentes sendo utilizadas normalmente são calculadas somente uma vez.

2 Fundamentos Teóricos

Aqui mostramos algumas informações sobre redes neurais, redes neurais profundas, imagens digitais, modelos utilizados (por modelos utilizados, nós nos referimos às redes neurais profundas especificas, que utilizamos no desenvolvimento deste trabalho, e mencionamos estes modelos específicos na Seção 3.3) e geometria epipolar para embasar o estudo deste trabalho.

A extensão do que é abordado é limitada à informação necessária para a realização do trabalho, embora a maioria destes temas tenha teses ou livros escritos somente sobre eles.

2.1 Estrutura Teórica de Redes Neurais

Desde a criação dos primeiros estudos em redes neurais a criação do modelo perceptron (Rosenblatt, 1958) consistia na imitação do que seria um neurônio do cérebro humano. Neste modelo, vários sinais ou valores (análogos aos sinais elétricos das sinapses dos neurônios do cerebro) são ponderados para gerar uma saída calculada. Na Equação (2.1) x_i corresponde a cada um dos sinais de entrada, w_i corresponde aos pesos treinados e y à saida calculada. As entradas são aplicadas à rede neural. A partir das entradas, as saídas da rede neural são calculadas. As diferenças entre as saídas calculadas e as saídas esperadas compõem um vetor de erros. O vetor de erros é utilizado para ajustar os pesos.

$$y = \sum_{i=1}^{n}(x_i * w_i) \tag{2.1}$$

As propriedades básicas de qualquer rede neural são as entradas, os pesos e os neurônios com suas funções de ativação. As entradas simulam um sinal de estímulo, os pesos são treinados para alguma finalidade específica e as funções de ativação moderam o valor das entradas com os pesos. Dependendo da função de ativação o comportamento da saída pode variar e dentro das possíveis funções estão a tangente hiperbólica, ReLU, linear, sigmóide, entre outras.

Vários métodos foram desenvolvidos para chegarmos ao estágio atual como o algoritmo *backpropagation* (Kelley, 1960) (Rumelhart *et al*., 1986), a regra da cadeia, para calcular as derivadas e conseguirmos o treinamento em redes neurais, os algoritmos de treinamento como o ADAM (nome derivado de *Adaptative Moment Estimation* ou Estimativa Adaptativa usando Momento) para treinamento adaptativo, técnicas para evitar *overfitting* como *dropout* ou normalização. A normalização consiste em dividir todos os valores pelo valor máximo do conjunto de valores, fazendo com que todos os valores fiquem entre zero e um. Todos estes avanços vêm conseguindo atingir ótimos patamares em treinamentos e resultados de redes.

Existem atualmente três tipos distintos de treinamentos de redes neurais. No treinamento supervisionado, temos exemplos com resultados conhecidos para o treinamento da rede. Estes exemplos são então calculados pela rede neural e a saída comparada com as saídas conhecidas. A diferença encontrada é então utilizada para ajuste dos pesos. No treinamento não supervisionado, os alvos dos dados do treinamento não são conhecidos. Então são aplicadas técnicas como, por exemplo, de agrupamentos para determinar grupos de classes. E no aprendizado por reforço, a rede aprende baseada em reforços positivos ou negativos. Neste trabalho estamos avaliando redes com treinamento supervisionado somente.

2.2 Redes Neurais Profundas

Com a evolução das redes neurais surgiram as redes neurais profundas, que empregam técnicas para análise de problemas não lineares, de ordem superior e de tamanho superior. Problemas como processamento de imagem são particularmente mais fáceis de processar utilizando técnicas como convolução. Esses problemas não lineares necessitam de mais camadas nas redes neurais para tratar a complexidade do que está sendo processado. Assim, as redes neurais profundas são assim denominadas pela grande quantidade de camadas.

As redes neurais convolucionais (do inglês, CNNs – *Convolutional Neural Network*) começaram a ser utilizadas em 1998 (LeCun *et al*, 1998). A convolução é uma operação matemática que captura a variação de uma função em relação a outra. No caso de aplicação de convoluções em filtros, sobre imagens, podem ser capturadas características dentro dessas imagens. As características por sua vez permitem encontrar informações relevantes. Nas imagens processadas, os filtros encontram linhas horizontais, verticais, curvas e círculos, nas camadas mais baixas de filtragem.

Nas camadas mais altas, os filtros encontram até mesmo propriedades como olhos, rodas e outras informações de mais alto nível. Na área de estudo de redes profundas utilizadas neste trabalho utilizamos três tipos de redes já bem conhecidas para o processamento de imagens e com seus pesos já pré treinados. Esses pesos já levaram a um bom resultado no reconhecimento de imagens na base de dados chamada ImageNet (Deng *et al*., 2009). Essas redes são a VGG16 (Simonyan *et al*., 2014), que atingiu 92,7% de acurácia na ImageNet, Resnet (He *et al*., 2016), venceu em primeiro lugar no concurso ImageNet de detecção e localização, e Mobilenet (Howard *et al*., 2017) e vão ser mais detalhadas na Seção 2.4, Seção 2.5 e Seção 2.6 respectivamente.

As redes Resnet e Mobilenet serão utilizadas neste trabalho em duas versões distintas. Assim, isso significa que um total de cinco versões de redes neurais profundas são testadas neste trabalho. Os modelos empregados no trabalho são VGG16, Resnet, Resnet V2, Mobilenet e Mobilenet V2. Neste trabalho, instanciamos os modelos que estão disponíveis na API Keras (Chollet *et al*., 2015). Instanciar um modelo significa carregá-lo na memória com base na descrição que é dada na API. Nesta API, é possível escolher a forma de instanciar de um modelo, de modo que ele já venha previamente treinado no banco de dados ImageNet (Deng *et al*., 2009). Esta forma de instanciar o modelo é a que usamos neste trabalho.

Esses modelos foram utilizados para a extração de características para treinar uma rede neural densamente conectada posterior a obtenção de características, visto que obtiveram bons resultados de classificação e reconhecimento de padrões em imagens. A rede neural criada neste trabalho faz a regressão para a geração das informações de posição de câmera.

Neste trabalho, utilizamos as camadas convolucionais, dos modelos citados no parágrafo anterior, para extrair características de imagens contendo alvos para calibração de câmeras. As camadas densamente conectadas são eliminadas e substituídas por camadas densamente conectadas novas, cujos pesos sinápticos são inicializados com valores aleatórios. Essas camadas densamente conectadas são treinadas, por meio do otimizador ADAM conforme explicado na Seção 3.3, de forma que as características sirvam para a regressão não-linear dos parâmetros de posicionamento de câmera.

O processo de carregar pesos já treinados com uma base de dados é chamado aprendizado por transferência (ou *transfer learning*). Esse processo possibilita encurtar o tempo de treinamento total já que uma parte grande do processo já foi realizado.

2.3 Imagens Digitais

Para falarmos de processamento de imagens também precisamos explicar como as imagens são guardadas digitalmente no computador. É preciso explicar os conceitos de quantização (quando se agrupa os valores reais de cores de uma imagem em uma lista finita de cores), resolução, canais de cor, e como ocorre o armazenamento da informação.

Numerar quantas corres existem seria impossível, já que a quantidade de cores que existem é infinita, assim como são infinitos os números entre zero e um. Então precisamos fixar uma quantidade de cores e agrupá-las em cores próximas de acordo com uma palheta de cores (quantização). Dependendo da quantidade de cores analisadas o valor utilizado para representar essas cores fica maior. Isso consequentemente interfere no tamanho final do armazenamento da imagem. Antigamente, nos computadores, foram utilizadas palhetas de cores até mesmo de 8 cores fixas. Este padrão não é praticado há anos e hoje em dia se utiliza, tipicamente, um byte em cada canal de cor (ou seja, três bytes, ou 24 bits, considerando canais de cor vermelho, verde e azul, o que totaliza 2^24 = 16.777.216 cores possíveis) para representar a quantidade de cores.

Dentro de cada imagem existem pontos descritos como pixels, que são pontos independentes de cor dentro da imagem, e a quantidade de pixels de uma imagem é a resolução da imagem. Quanto maior a quantidade pixels maior a resolução da imagem e mais informação (ou detalhe da imagem) pode ser armazenada.

Quando armazenamos digitalmente uma imagem é comum separar em diferentes canais de cor a informação para melhor organização e quanto maior a quantidade de canais mais informação sobre a cor da imagem é armazenada. Um padrão amplamente adotado é o RGB (*Red Green Blue* – Vermelho Verde Azul), mas a escala de cinza também é muito utilizada onde uma sequência de níveis de cinza é definida para o armazenamento da

informação da imagem. O padrão RGB guarda os canais de cor vermelho, verde e azul, que são as cores mais percebidas pela visão humana e com isso em cada pixel é guardada a informação para a intensidade de cada cor.

Assim temos as imagens digitais, que variam de tamanho dependendo da quantidade de canais, quantidade de cores, resolução. Por exemplo uma imagem que utiliza 1 byte para definir cada cor com três canais e resolução de 224x224 pixels ocuparia 150.528 bytes de armazenamento no computador.

2.4 Modelo VGG16

Introduzido em abril de 2015, o modelo VGG sob o título de "*Very Deep Convolutional Networks for Large-Scale Image Recognition*" (Simonyan *et al*., 2014) (redes neurais convolucionais muito profundas para reconhecimento de imagens em larga escala) atingiu o *top five* no teste de acurácia daquele ano com 92,7% na base de dados ImageNet (Deng *et al*., 2009). Este modelo tem esse nome porque tem 16 camadas de parâmetros treináveis e iremos utilizar os valores destes parâmetros já treinados. O artigo que apresentou esta rede neural realizou um total de seis variações de camadas. A versão de 16 camadas será aplicada, neste trabalho, como uma das bases da extração de características.

A utilização desta rede neural profunda na forma de 16 camadas treináveis é bem simples com a API (*Application Programming Interface* – Interface de Programação de Aplicação) Keras (Chollet *et al*., 2015). Podem ser carregados os pesos treinados com a base de dados ImageNet e com essa rede neural extrair características das imagens de calibração. Os vetores de característica, extraídos das imagens de calibração, podem então ser utilizados para o treinamento da rede densamente conectada.

A rede neural VGG16 neste formato possui duas camadas convolucionais iniciais com 64 filtros aplicados em uma imagem de 224x224 pixels com três canais de cor. Ao final, 512 características são geradas, representadas por 512 mapas 7x7. Essas são então processadas por uma camada de média global para chegar a 512 valores. A camada de média global foi adicionada para diminuir a quantidade de parâmetros de saída de 25.088 para somente 512, extraindo a média de cada um dos 512 filtros 7x7. Isso diminui a quantidade de informação da saída da rede, mas tenta preservar a informação relevante.

2.5 Modelo Resnet

O modelo Resnet (He *et al*., 2016) "*Deep Residual Learning for Image Recognition"* (aprendizado residual profundo para reconhecimento de imagens) foi apresentado em 2015 como uma alternativa para modelos anteriores, que eram custosos (demandavam muitos dados e/ou tempo) de treinar como VGG16, procurava resolver o problema do desvanecimento do gradiente (este processo acontece porque quanto mais profunda for a rede neural menor o gradiente de ajuste se torna e consequentemente mais lento é o treinamento) e apresenta uma capacidade de treinar uma rede de mais de 150 camadas que é uma das mais profundas até então. O gradiente é mais facilmente retropropagado, das camadas mais profundas em direção às camadas mais próximas da entrada devido ao processo de propagação residual. O processo de propagação residual consiste em diminuir saída de um bloco de camadas do gradiente de entrada dessas camadas transmitindo, assim, o gradiente e não somente o erro, essas ligações são chamadas conexões de identidade, e com isso propagam o gradiente para camadas mais baixas diminuindo o problema do desvanecimento de gradiente.

Este modelo se beneficia de uma rede profunda (152 camadas com pesos treináveis) para otimizar o resultado e atingir melhores ganhos na ImageNet atingindo um erro de 3,57% e primeiro lugar na tarefa de detecção e classificação do ILSVRC 2015. A principal característica adicionada por rede foi o processo de propagação residual possibilitando o treinamento de camadas mais profundas do que se tinha atingido até aquele momento. O artigo apresenta cinco configurações distintas da rede. Os autores atingem resultados melhores, que a rede neural sem os atalhos de propagação de residual, a partir da rede com profundidade de 34 camadas treináveis e redes neurais com mais camadas.

A arquitetura da Resnet utilizada neste trabalho foi a de 50 camadas. A implementação dessa rede neural foi utilizada com os pesos treinados para o conjunto de imagens ImageNet, que atingiu uma acurácia melhor que a arquitetura de 34 camadas. A Resnet neste formato tem o processamento mais rápido que nas arquiteturas de 101 e 152 camadas, também apresentadas no trablaho original da Resnet, por conter menos parâmetros.

As diferenças entre as duas versões de Resnet vistas nesse trabalho mudam o comportamento e o resultado da rede neural. A Resnet V1 aplica uma camada ReLU após a adição da conexão de propagação residual (conexão de identidade), assim adicionando uma não linearidade à rede neural. Já a Resnet V2 aplica uma normalização e uma camada ReLU na conexão identidade antes da multiplicação. A Figura 2.3 ilustra essa diferença.

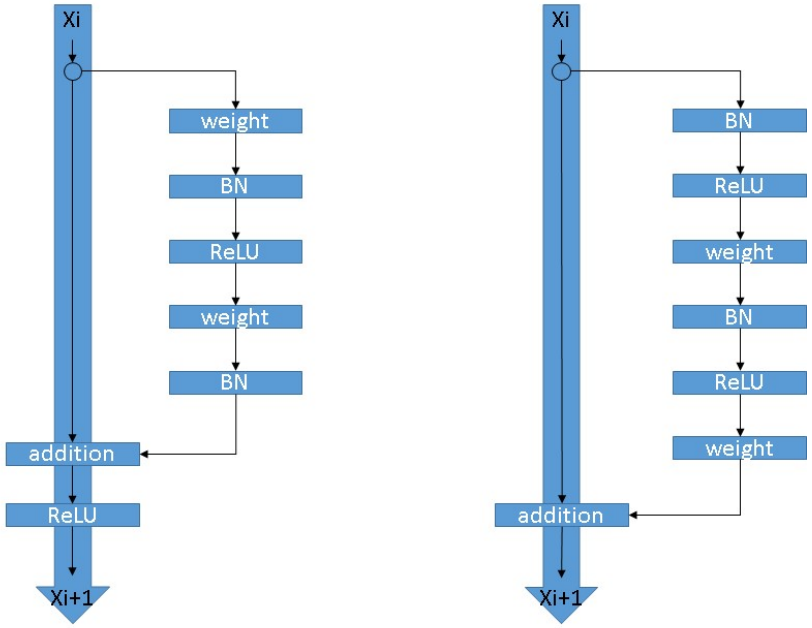

Figura 2.1 Ilustração[3] da diferença entre as versões 1 e 2 da rede neural Resnet (imagem adaptada com base em imagens do site https://cv-tricks.com/keras/understand-implement-resnets)

Essa alteração cria uma alteração no resultado encontrado, diferenciando as redes não só na arquitetura, como no valor final dos pesos treinados e também no resultado obtido. Esse ponto é tratado no Capítulo 4, no final da discussão referente aos resultados da Tabela 4.2.

2.6 Modelo Mobilenet

O trabalho Mobilenets (Howard *et al.*, 2017) traz uma classe de redes focadas em velocidade de processamento para dispositivos móveis ou com alguma forma de limitação de

[3] Imagem obtida do site.

processamento, como microcontroladores ou sensores remotos. Foram especificados dois hiperparâmetros, que são utilizados para diminuir a complexidade computacional (quantidade de parâmetros treináveis) da rede.

A parametrização da quantidade de camadas e da quantidade de filtros é feita por meio da multiplicação destes pelos fatores multiplicativos N e M. Essa otimização aumenta ou diminui a quantidade de camadas e a quantidade de filtros de características. Com esses fatores multiplicadores é possível fazer uma escolha em relação a quantidade de processamento e a acurácia do modelo. De um ponto de vista matemático estes parâmetros mantêm a consistência do modelo.

Os parâmetros de alteração da quantidade de camadas e filtros (N e M), chamados hiperparâmetros do modelo, configuram uma variação na quantidade de pesos treináveis realizando uma aglutinação ou expansão no modelo, modificando o tamanho da matriz de pesos treináveis com a diminuição ou aumento da quantidade final de valores que serão utilizados para realizar o cálculo da rede neural. Essa variação na quantidade de parâmetros pode, assim, ajustar o algoritmo ao poder de processamento do dispositivo.

A arquitetura deste modelo utilizada neste trabalho é a provida pela API Keras carregada com os pesos treinados na base de imagens ImageNet. O modelo é utilizado em duas versões cada uma com suas características. A ideia por trás da Mobilenet V1 é trocar convoluções custosas computacionalmente por convoluções menores e mais baratas em termos de processamento. Com isso, essa rede neural pode ser executada por dispositivos como celulares ou com limitações em poder de processamento. Já na Mobilenet V2 foi introduzido o mesmo conceito de conexões residuais e com isso foi obtido uma redução ainda maior dos tensores. Com tensores menores, que sofreriam do problema de desvanecimento de gradiente sem a conexão residual, foi incluída uma expansão com uma camada 1x1 na

entrada do processo e diminuído o número de canais das convoluções. Na Figura 2.4 são ilustradas as diferenças nos blocos das duas versões da rede neural.

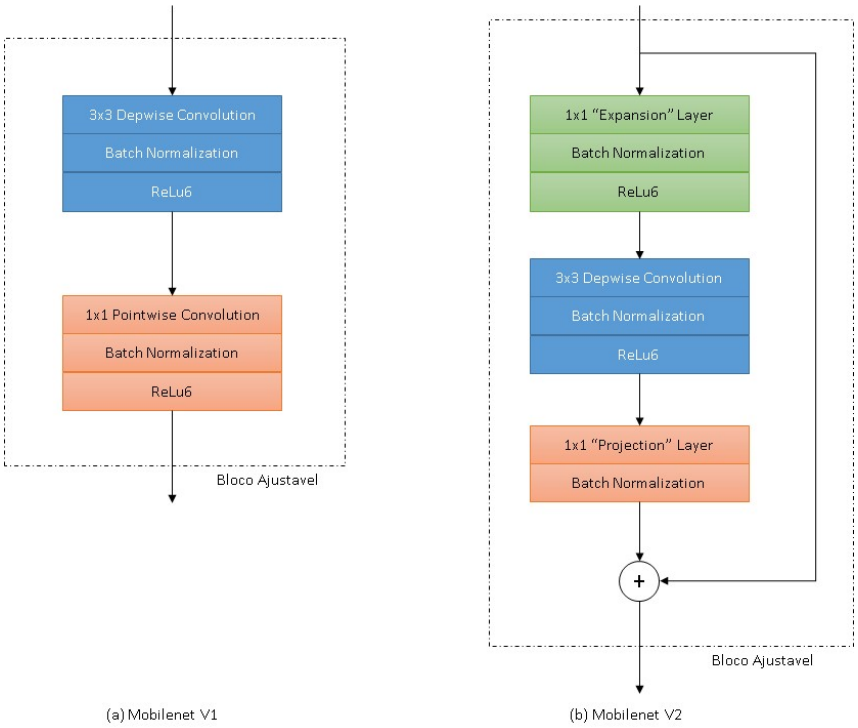

Figura 2.2 Ilustração dos blocos ajustáveis da Mobilenet V1 e V2 (imagem adaptada com base em imagens do site https://machinethink.net/blog/mobilenet-v2/)

2.7 Dados do Estudo

Na Figura 2.3, mostramos exemplos de imagens de calibração utilizadas para o treinamento de rede neural, das quais foram extraídas as características pelas redes neurais VGG16, Resnet e Mobilenet para o posterior tratamento por uma rede densamente conectada.

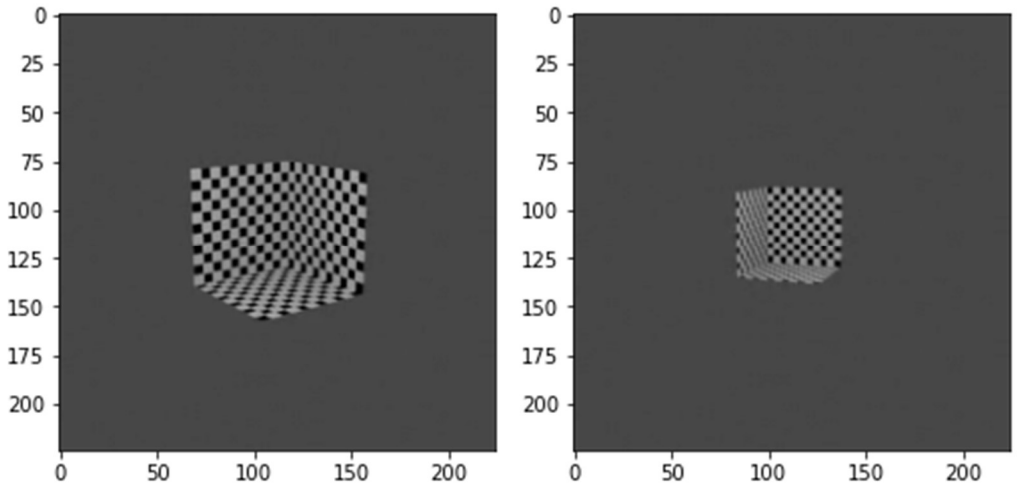

Figura 2.3 Exemplos de imagens utilizadas no trabalho.

Estas imagens foram geradas através do Software Blender, com a criação de um modelo 3D triplanar e modificando a posição da câmera, em um semi-circulo, apontada em direção de seu centro. No Capítulo 3 estão descritos os procedimentos para a confecção deste modelo.

Juntamente com as imagens geradas pelo modelo no Blender foram extraídos os parâmetros-base de movimentação quadro-a-quadro da posição de translação, em relação ao centro da imagem, e rotação da câmera. Essa informação é usada na definição dos alvos para o treinamento (*Ground Truth*) e testes dos resultados dos modelos criados.

2.8 Geometria Epipolar e Homografia

Estes termos, utilizados frequentemente no trabalho, se referem à geometria para a descrição de relações geométricas em visões de câmeras estéreo, onde estas câmeras veem diferentes projeções 2-D de uma mesma imagem 3-D. Essas projeções levam a restrições, que são as relações entre pontos dessas imagens. A correspondência entre os pontos dessas

imagens, que identificam os mesmos pontos no modelo 3-D, são chamadas de homografia, ou seja, pontos de uma mesma referência nas duas imagens são correlacionados por uma matriz de homografia.

O cálculo do deslocamento de uma câmera, que está capturando imagens de um mesmo modelo, é calculado pela multiplicação da posição da câmera pela matriz de transformação. Essa relação geométrica entre as imagens é obtida criando um sistema de equações com oito ou mais pontos equivalentes entre as imagens. Este sistema de equações, então, é utilizado para o cálculo das incógnitas da relação de transformação. A relação entre as coordenadas antes e depois do movimento da câmera é demonstrada na Equação (2.2) considerando c1 = (x, y, z) e c2 = (x', y', z'), como a origem da câmera e a posição após a movimentação.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} h11 & h12 & h13 \\ h21 & h22 & h23 \\ h31 & h32 & h33 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \tag{2.2}$$

A Figura 2.4 demonstra a relação entre as imagens do mesmo ponto x_1, visto por duas câmeras em posições diferentes. São necessários pelo menos oito pontos de correspondências para remover as ambiguidades, pois para C_1 os pontos que C_2 enxerga, x_1, x_2 e x_3, são os mesmos. Esse cálculo é realizado pela API OpenCV durante o processo de cálculo do deslocamento entre imagens.

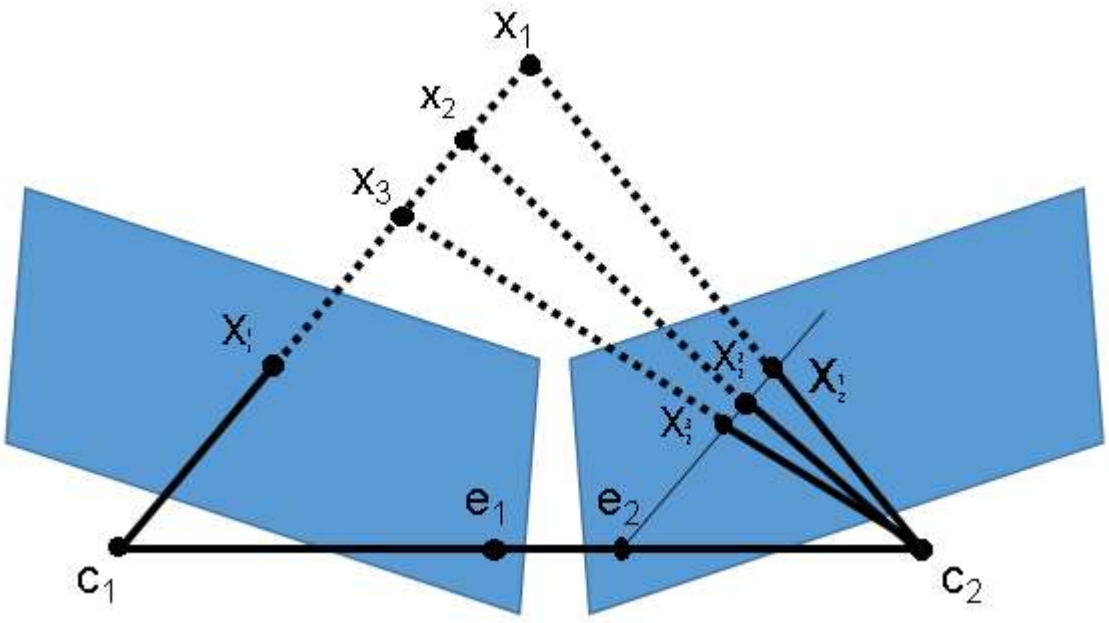

Figura 2.4 Geometria epipolar. Essa figura foi adaptada de (Faugeras, 1993)

São eleitos oito ou mais pontos nas duas imagens, que correspondem ao mesmo ponto no modelo 3D. Então são montadas oito equações, uma para cada par de pontos onde um ponto é igual ao outro modificado pela transformação homográfica. Isso gera um sistema determinado ou sobredeterminado (quando existem mais de oito pontos correspondentes e com isso mais equações que incógnitas).

3 Metodologia

Este trabalho consiste na geração de um modelo 3-D na ferramenta Blender e na análise de um vídeo em ângulos diferenciados, obtidos com a movimentação da câmera, desse modelo. Os parâmetros extraídos correspondem à posição de câmera. É realizada também a comparação posterior entre técnicas para um melhor embasamento quantitativo.

Para iniciar, veremos o método como o modelo foi criado e como os dados de informação base de posição da câmera foram extraídos do Blender. Posteriormente, veremos os métodos que foram aplicados para a geração de parâmetros de comparação. Depois vamos explorar como foi realizada a extração de características utilizando redes neurais VGG16, Resnet e Mobilenet. Veremos, então, o treinamento da rede neural densamente conectada e, após isso, veremos um procedimento utilizando a API do OpenCV (Bradski, 2000).

3.1 Blender

Para a geração de um vídeo padrão, para a posterior análise, foi criado o modelo 3-D na ferramenta Blender (Hess, 2010). Este modelo compreende os três eixos (X, Y, Z) para facilitar a análise em apenas uma imagem. Este processo envolveu comandos de criação de linhas, planos e aplicação de texturas sólidas para criação de uma base quadriculada formando um objeto 3-D. Posteriormente este objeto foi utilizado como alvo da câmera se movimentando como na Figura 3.1, criando imagens como aquelas mostradas na Figura 2.3.

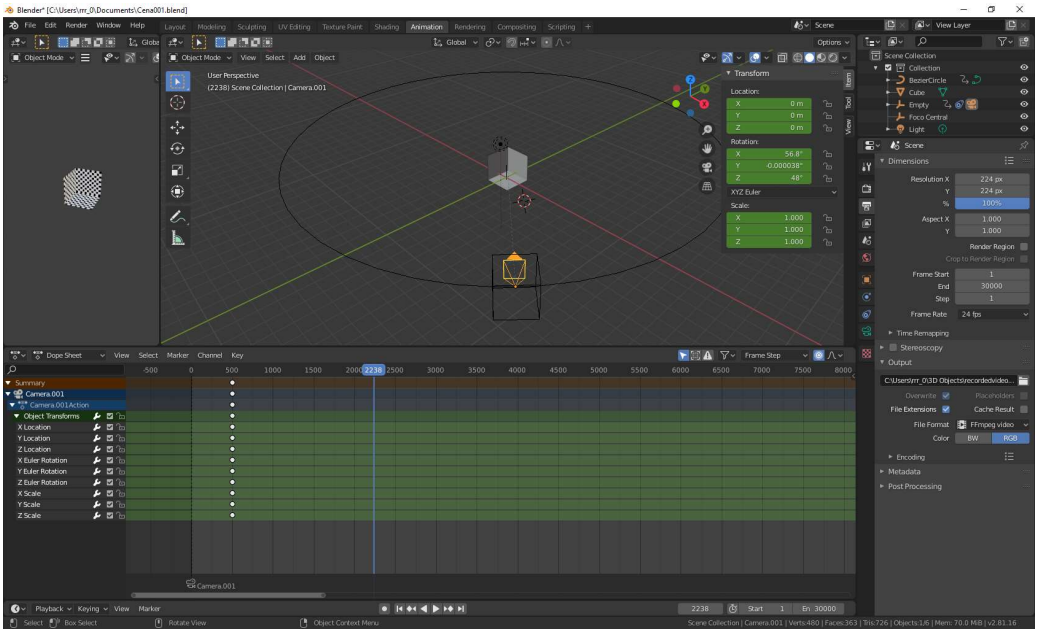

Figura 3.1 Criação do modelo no Blender.

Utilizando a variação de posição de câmera foi criada uma animação do movimento simulando uma movimentação em semi-circulo. Com o código em Python apresentado na Seção 1 do Apêndice A foram gerados os valores reais de posição de câmera pelo Blender criando, assim, os valores de base (*ground truth*) para o estudo. Foi necessário estudar a documentação relacionada aos objetos disponíveis, sendo estes objetos classes com funções manipulativas do ambiente, em Python, para realizar o *script* de extração de parâmetros de câmera do Blender.

Apesar da complexidade envolvida em criar um modelo virtual e aprender as ferramentas de construção de objetos do Blender, este procedimento se tornou muito vantajoso pela assertividade na extração de parâmetros de valores base. Assim, a obtenção de parâmetros não será afetada por erros humanos ou aproximações manuais, que ocorrem quando tentamos medir coisas no mundo real.

Os objetos principais utilizados da biblioteca python do Blender foram Scene, que realiza todas as manipulações relacionadas a cena, e Camera, que realiza as manipulações relacionadas a movimentação, posicionamento e controle, de câmera. Estes objetos residem no pacote “bpy” ondem residem todos os objetos manipulativos do Blender no python.

3.2 Extração de Características nas Redes Neurais

A partir do vídeo criado, como base para o estudo, foram geradas as características aplicadas ao trabalho com as redes neurais VGG16, Resnet e Mobilenet. Estas duas últimas foram usadas em duas versões, através da API (*Application Programming Interface* – Interface de programação de aplicação) Keras e descartando a camada de classificação. Foram utilizados os pesos pré-treinados no banco de dados ImageNet.

Para a extração de características das imagens de calibração, por meio das cinco redes neurais, foi utilizada a API Keras, conforme o código em Python mostrado na Seção 2 do Apêndice A. Neste processo a placa de vídeo realizou o processamento de cada quadro de imagem, para extrair mapas de características como o mostrado na parte direita da Figura 3.2.

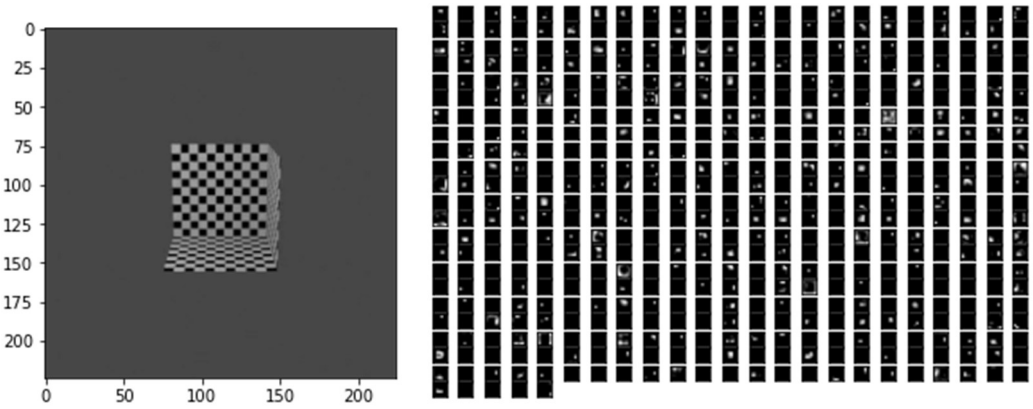

Figura 3.2 Mapa de 512 características extraído da imagem pelas camadas convolucionais da rede neural VGG16.

Essas características foram agrupadas em uma camada de média global (*global average pooling*) e os 512 (esse número de características vale para a rede neural VGG16; para as redes neurais Resnet e Mobilenet, as dimensões dos vetores de características são dadas na Tabela 3.1) valores médios das características foram guardados em arquivos, sendo que um vetor é gerado por imagem para cada rede neural profunda empregada. Esses arquivos são empregados no treinamento, teste e validação, da rede densamente conectada para regressão, que gera os valores estimados de parâmetros de posição de câmera.

Ao carregar as redes neurais, dentre as várias opções, existe a opção de remoção da camada de classificação e a opção de adicionar uma camada de média global no final destas redes neurais. A camada de média global, quando é adicionada, agrupa os valores de saída dos filtros, representando a média de cada um dos filtros. Assim, o número de dimensões da saída não será tão grande e a manipulação desses valores fica mais fácil. Por exemplo, para a Resnet são 2048 valores. Na Mobilenet V1 e na Mobilenet V2 são 1024 e 1280, respectivamente.

3.3 Treinamento da Rede Neural para Regressão de Parâmetros de Calibração

Ao fim da extração das características, realizada pelas redes neurais VGG16, Resnet e Mobilenet, uma rede neural densamente conectada é treinada com as características obtidas. Este treinamento utiliza os vetores-alvo de localização espacial da câmera gerados pelo Blender. A rede neural densamente conectada processa, então, a informação para a regressão, gerando valores aproximados das coordenadas correspondentes após o treinamento, de três ou seis parâmetros de calibração. Quanto ao número de neurônios na camada escondida, que é a camada imediatamente anterior à camada de saída, foram investigadas três opções: 64, 128 e 256 neurônios.

Tabela 3.1 Dimensões das saídas das últimas camadas das redes neurais profundas utilizadas.

Redes utilizadas	**Características**	**Parametros de Média Gobal**
VGG16	7x7x512	512
Resnet 50	7x7x2048	2048
Resnet 50 V2	7x7x2048	2048
Mobilenet	7x7x1024	1024
Mobilenet V2	7x7x1280	1280

Como pode ser visto na Seção 3 do Apêndice A, um programa em Python foi criado para carregar, treinar e validar uma rede neural densamente conectada. Os arquivos com vetores de características, gerados no processamento das redes neurais profundas pela

última camada convolucional, e seus respectivos alvos (*ground truth)* são usados neste treinamento. Cada linha da Tabela 3.1 leva a um resultado de treino distinto. Esta tabela lista a quantidade de filtros de características de saída e quantos parâmetros restam na camada de média global.

Na rede neural densamente conectada, a função de ativação utilizada, para os neurônios da camada escondida, foi a tangente hiperbólica e a função de ativação da camada de saída foi a função linear. Assim, a saída de média global das redes neurais profundas, treinadas no banco de imagens ImageNet, foi aplicada à entrada da rede neural densamente conectada para o treinamento, teste e validação. Os vetores-alvo contendo parâmetros de posicionamento da câmera são utilizados em duas formas alternativas: i) somente os três parâmetros de rotação, que são os três ângulos que descrevem a orientação de um corpo rígido em um espaço euclideano tridimensional (também conhecidos como ângulos de Euler), e ii) os três parâmetros de rotação mencionados, mais três parâmetros relacionados à translação (deslocamento) da câmera em relação à origem, totalizando seis dimensões para os vetores-alvo nesta segunda alternativa.

3.4 OpenCV

Adotando a API de manipulação visual OpenCV (Bradski, 2000), foi criado um processo de cálculo de homografia. Esse processo usa geometria epipolar para calcular o deslocamento em relação a uma imagem central. Assim, são estimados todos os parâmetros de deslocamento de todas as posições de câmera adicionando a posição da imagem central ao deslocamento encontrado pelo cálculo de homografia do OpenCV. A título de esclarecimento, o trabalho não utiliza qualquer processo de SLAM visual, apesar do código

utilizado para o cálculo deste deslocamento ser baseado no código do Capítulo 7 do livro *Slambook* 2 (Gao, *et al*. 2017).

Na Seção 1 do Apêndice B, está a listagem do código C++ utilizando a API para estimar os deslocamentos de câmera. Essas estimativas de deslocamento se relacionam com as informações de rotação e translação, da câmera, em cada imagem. Esses parâmetros, quando adicionados aos dados de base da imagem central, compõem os deslocamentos de câmera.

Foi necessário empregar as Equações (3.1), (3.2) e (3.3) (Mallick, 2016), para encontrar os parâmetros de rotação baseados em ângulos de Euler, pois a API do OpenCV provia somente a informação em função da matriz de rotação completa com nove parâmetros. O cálculo foi testado empiricamente utilizando os valores extraídos dos valores-alvo (*ground truth*), que são exatos com base na natureza digital do processo de sua geração (Mallick, 2016).

$$Rx = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\,(\theta x) & -sin\,(\theta x) \\ 0 & sin\,(\theta x) & cos\,(\theta x) \end{bmatrix} \tag{3.1}$$

$$Ry = \begin{bmatrix} cos\,(\theta y) & 0 & sin\,(\theta y) \\ 0 & 1 & 0 \\ -sin\,(\theta y) & 0 & cos\,(\theta y) \end{bmatrix} \tag{3.2}$$

$$Rz = \begin{bmatrix} cos\,(\theta z) & -sin\,(\theta z) & 0 \\ sin\,(\theta z) & cos\,(\theta z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

Como pode ser visto na Figura 3.3, as transformações foram realizadas e registradas em um arquivo. Este arquivo foi utilizado na comparação de resultados entre o processo com redes neurais profundas e o processo mediante homografias do OpenCV.

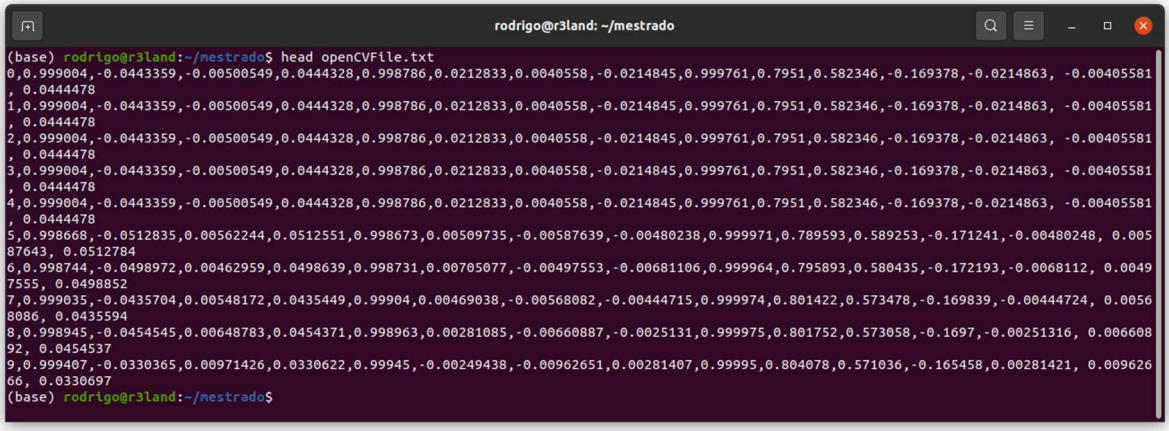

Figura 3.3 Homografia OpenCV.

4 Resultados

Neste capítulo serão apresentados os resultados encontrados no trabalho. Estes resultados representam a implementação do método de processamento de imagens utilizando redes neurais profundas e o cálculo de deslocamento pelo OpenCV. É realizada a comparação entre estes métodos incluindo os treinamentos, validação e testes realizados.

4.1 Dados

As imagens utilizadas neste trabalho foram geradas com a ferramenta de modelagem 3-D chamada Blender. Com um objeto tri-planar ao centro do plano projetivo, um vídeo foi criado contendo 30000 quadros de imagens distintas. Os parâmetros de posição da câmera são variados, de modo que a distância da câmera ao centro da imagem é modificada de 1 a 30 metros. A câmera se movimentou seguindo um caminho em um padrão semicircular de zero a noventa graus, ao redor do objeto central, em zig-zag, e apontando para o centro da imagem.

Destas 30000 imagens geradas, foram separadas 27000 imagens para treinamento da rede neural densamente conectada, 2000 imagens para a validação e 1000 imagens para teste. Mesmo com imagens oriundas do mesmo vídeo, a cada 500 imagens o caminho circular percorrido pela câmera tem alterações nos valores de altura e distância do centro da imagem. As alterações ocorrem em altura e amplitude em relação ao centro da imagem, gerando coordenadas com valores distintos. Sendo assim, os conjuntos de treino, validação e teste têm valores com distância maior que 1 metro entre si.

4.2 Resultados Obtidos

É realizada uma comparação entre os métodos diferentes (rede neurais profundas e OpenCV) de regressão/estimação de informação posicional da câmera na análise do vídeo de teste. São consideradas variações no número de neurônios, na camada escondida da rede neural densamente conectada, e na dimensão do vetor de entrada, para cada modelo de rede neural, como mostrado na Tabela 4.1. Na análise de parâmetros foram verificados, primeiramente, os ângulos de rotação somente. Isso significa que as saídas e os alvos têm três dimensões. Em um experimento posterior, as três coordenadas de translação foram anexadas às saídas e aos alvos. Nesse caso, as saídas e os alvos das redes neurais e do estimador baseado no OpenCV têm seis dimensões. Os resultados de treino e validação mostrados nas Figuras 4.1 até 4.4 se referem a este segundo caso, em que os vetores com alvos de calibração têm seis dimensões

Os treinamentos da rede neural densamente conectada atingiram, com cerca de quinze épocas, o patamar próximo ao erro mínimo como mostrado nas Figuras 4.1 até 4.4. As Figuras 4.2 e 4.4 mostram as curvas de erro de treino e de validação em escala logarítmica, para facilitar a visualização das diferenças entre os resultados. O ponto de parada escolhido foi obtido após testes com múltiplas variações de parâmetros no treinamento. Ele foi escolhido de modo a evitar a perda da capacidade de generalização da rede.

Foram utilizados para visualização os valores de MAE (*Mean Absolute Error* – Erro Médio Absoluto) e MSE (*Mean Squared Error* – Erro Médio Quadrático), que representam erros relacionadso à distância real. Por exemplo, um valor de erro absoluto igual a 3 corresponde a 3 centímetros de diferença, em média, entre as coordenadas estimadas e as coordenadas reais. O erro médio quadrático é mais utilizado como base para a atualização

dos parâmetros (pesos e *biases*) da rede neural. Na etapa de retropropagação de erros (*backpropagation*), quando os pesos e *biases* são atualizados na direção contrária à do

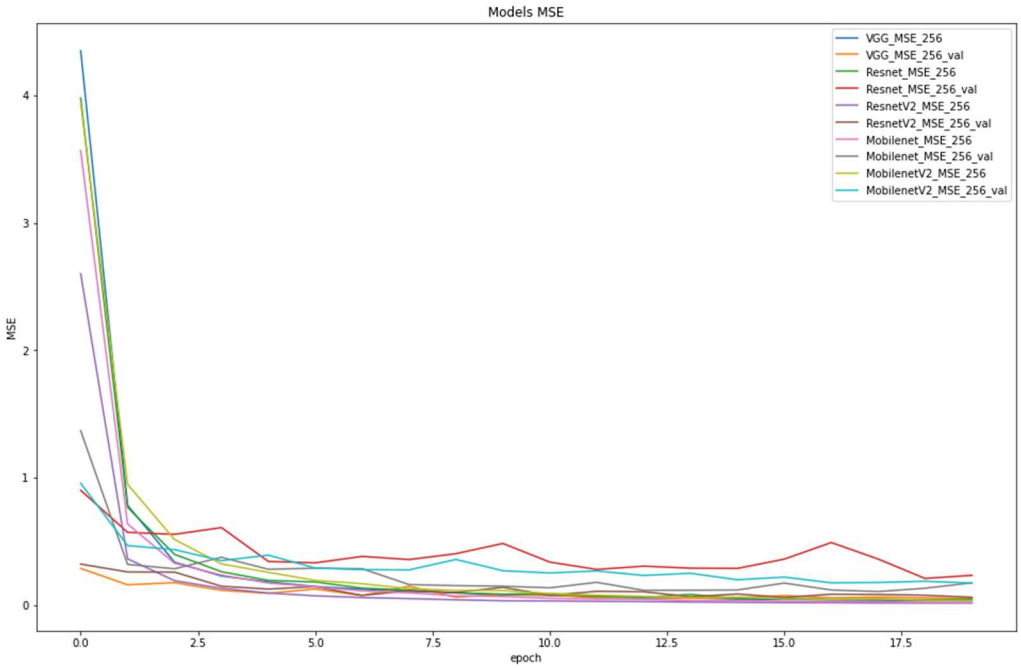

Figura 4.1 MSE de treino e MSE de validação, de cinco redes neurais com 256 neurônios na camada escondida, durante treinamento, em função do número de épocas.

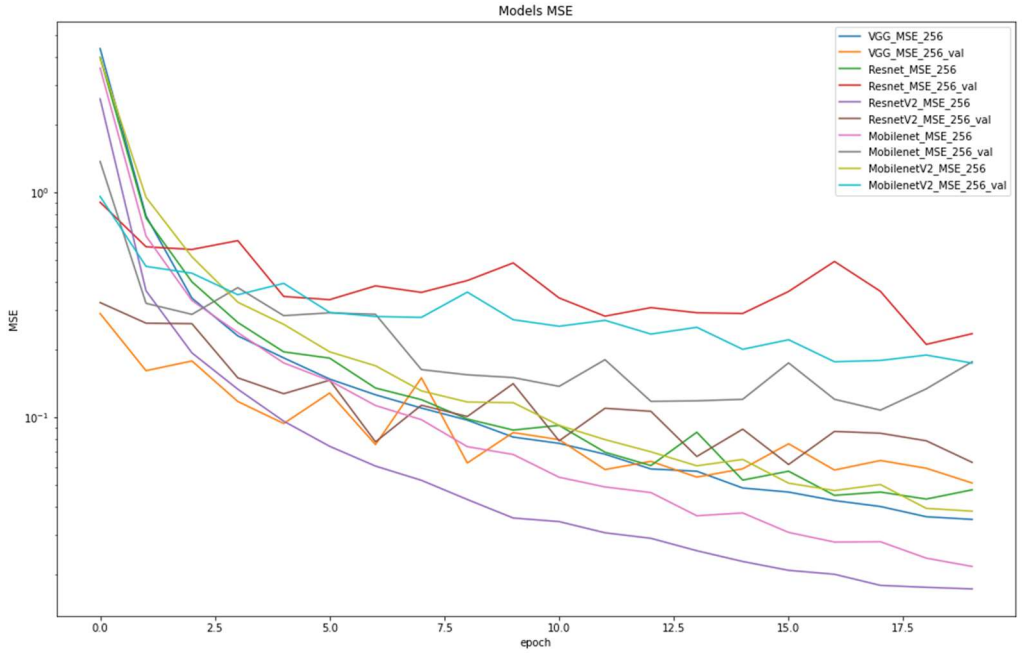

Figura 4.2 MSE de treino e MSE de validação, de cinco redes neurais com 256 neurônios na camada escondida, durante treinamento, em escala logarítmica.

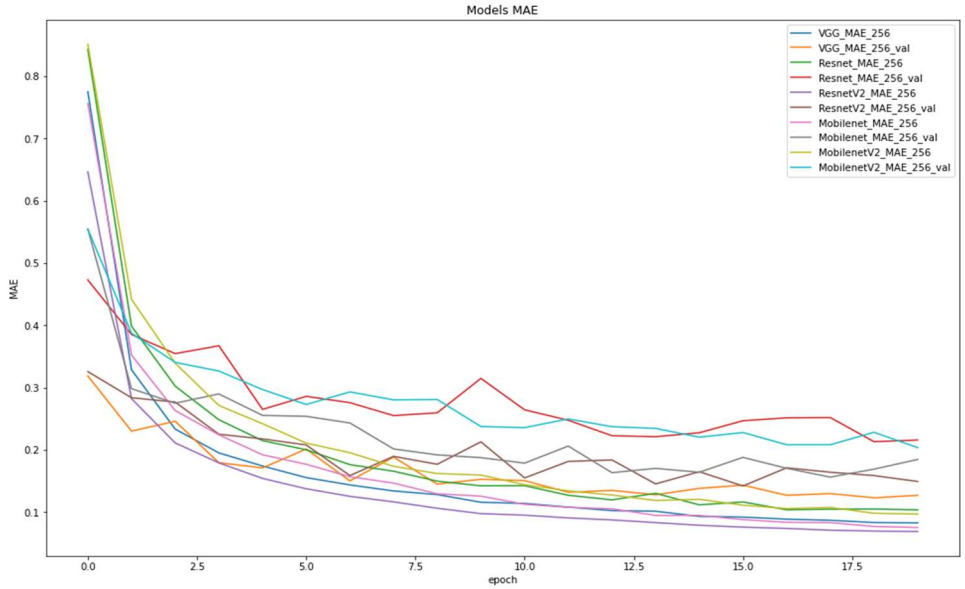

Figura 4.3 MAE de treino e MAE de validação, de cinco redes neurais com 256 neurônios na camada escondida, durante treinamento, em função do número de épocas. O MAE representa o erro, em centímetros, entre as estimativas e os valores reais das coordenadas de rotação e translação da câmera.

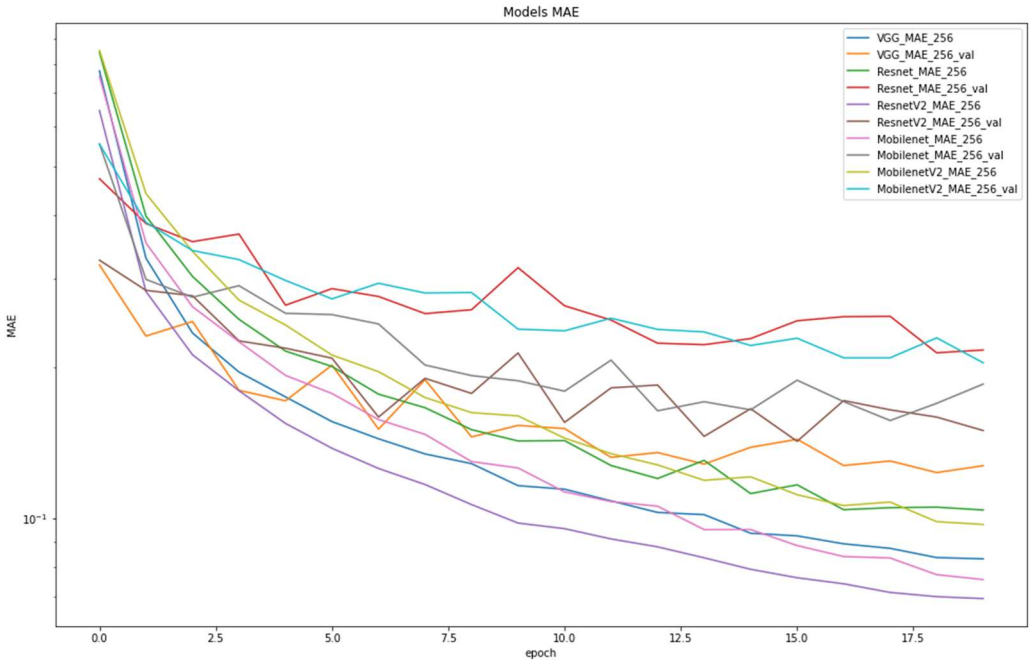

Figura 4.4 MAE de treino e MAE de validação, de cinco redes neurais com 256 neurônios na camada escondida, durante treinamento, em escala logarítmica, para melhor comparação entre os métodos utilizados.

gradiente da função-custo (ou função perda, do inglês *loss function*), é o gradiente da função MSE em relação aos pesos e *biases* que é utilizado. Isso atribui importância maior aos erros maiores. Por isso, durante o treinamento, o valor de erro utilizado para ajuste dos pesos foi o MSE.

Os gráficos foram construídos com as curvas de treinamento e validação para a visualização da perda de generalização conforme o treinamento evolui. Pode ser visualizado nos gráficos que a diferença entre o treinamento e a validação começa a ficar maior, indicando essa perda de generalização. Os resultados de validação são os mais importantes nos gráficos, pois refletem o uso da rede treinada com dados não vistos anteriormente.

Com relação aos gráficos das Figuras 4.1 a 4.4, vale salientar que, na validação, a curva obtida com base em características extraídas por VGG16 (em laranja) se alterna com a curva obtida com Resnet V2 (em marrom). Esses são os melhores resultados de validação obtidos. As curvas de validação em vermelho, azul claro e cinza (Resnet, Mobilenet V2 e Mobilenet, respectivamente) correspondem a resultados de validação piores. Isso pode ser visto em detalhes na Figura 4.2, e também na Figura 4.4. A representação em escala logarítmica facilita a visualização das diferenças entre os valores de MSE (Figura 4.2) ou de MAE (Figura 4.4).

Ao final do treinamento, o erro de validação (MSE ou MAE) indicado pelas curvas VGG16_*_256_val (em laranja) está menor que o erro de validação indicado pelas curvas ResnetV2_*_256_val (em marrom). O símbolo * pode se referir a MSE ou a MAE, conforme estejamos olhando a Figura 4.2 ou a Figura 4.4. Com base neste resultado, seria esperado que as redes neurais densamente conectadas treinadas com base em características VGG16 se mantivessem melhores, em termos de resultados de teste, que aquelas treinadas com

base em características Resnet V2. No entanto, dentro do conjunto de dados utilizados para teste, os melhores resultados foram alcançados pela rede neural densamente conectada treinada com base em características Resnet V2. Provavelmente, flutuações estatísticas entre os conjuntos de dados, de validação e de teste, explicariam essa inversão do melhor resultado por ocasião do teste. É provavél que as soluções obtidas com base em características VGG16 e Resnet V2 estejam, na prática, empatadas.

Na Tabela 4.1, estão detalhadas todas as variações de redes neurais profundas, neurônios na rede densamente conectada e saídas mapeadas. Não foram utilizadas mais variações na camada escondida por limitações técnicas do ambiente de processamento. A limitação de memória da placa de vídeo no ambiente em questão não permitiu o processamento com mais neurônios. Mesmo assim, consideramos provável que mais neurônios poderiam melhorar a precisão da aproximação da posição da câmera. Pelos indicadores do treinamento, quanto mais neurônios foram adicionados melhor foi o resultado obtido. Para a rede neural densamente conectada baseada em características VGG16, que foi a única que foi possível rodar com 512 neurônios, o resultado de teste correspondeu a erros menores. Ainda não é possível afirmar, sem os devidos testes, se isso seria verdadeiro para as outras redes neurais profundas ou mesmo para mais neurônios que 512.

Na Tabela 4.2, os resultados da avaliação de resultados dentro do conjunto de imagens de teste para as cinco redes podem ser vistos. São mostrados os valores de MSE e de MAE, nos casos de alvos com três ou com seis dimensões. Estes resultados foram obtidos utilizando o conjunto de testes, após o término do treinamento.

Os valores em negrito na Tabela 4.2 refletem os melhores resultados encontrados pelas redes neurais densamente conectadas baseadas em características VGG16 e Resnet V2. Como pode ser visto, o uso de características Resnet V2 proporciona um MSE 16% menor que o MSE obtido com base em características VGG16. Não foi feita a comparação

entre o tempo de processamento das duas redes, mas é esperado que a VGG16, com 14,714,688 parâmetros, termine mais rapidamente que a Resnet V2, com 23,564,800 parâmetros, por ter menos

Tabela 4.1 Redes neurais treinadas.

Rede Entrada	Entrada	Camada Escondida	Saída
VGG16	512	64	3
VGG16	512	128	3
VGG16	512	256	3
VGG16	512	64	6
VGG16	512	128	6
VGG16	512	256	6
Resnet	2048	64	3
Resnet	2048	128	3
Resnet	2048	256	3
Resnet	2048	64	6
Resnet	2048	128	6
Resnet	2048	256	6
Resnet V2	2048	64	3
Resnet V2	2048	128	3
Resnet V2	2048	256	3
Resnet V2	2048	64	6
Resnet V2	2048	128	6
Resnet V2	2048	256	6
Mobilenet	1024	64	3
Mobilenet	1024	128	3
Mobilenet	1024	256	3
Mobilenet	1024	64	6
Mobilenet	1024	128	6
Mobilenet	1024	256	6
Mobilenet V2	1280	64	3
Mobilenet V2	1280	128	3
Mobilenet V2	1280	256	3
Mobilenet V2	1280	64	6
Mobilenet V2	1280	128	6
Mobilenet V2	1280	256	6

Tabela 4.2 Resultados obtidos dentro do conjunto de dados para teste, em termos de erro médio absoluto (MAE) e erro médio quadrático (MSE). Os quatro números em negrito na coluna mais à direita indicam os dois melhores resultados que foram obtidos quando os alvos de calibração têm seis dimensões. Os dois números em negrito na coluna mais central indicam o melhor resultado que foi obtido quando os alvos de calibração têm três dimensões.

	VGG16					
N Neuron	64	128	256	64	128	256
Param	3			6		
MSE	0,6223	0,5331	0,4441	0,4427	0,3424	**0,1768**
MAE	0,5858	0,5225	0,4466	0,3127	0,2791	**0,2042**
	Resnet 50					
N Neuron	64	128	256	64	128	256
Param	3			6		
MSE	1,0014	0,7712	0,5291	0,7704	0,4794	0,2502
MAE	0,7312	0,6107	0,4882	0,4179	0,3112	0,2412
	Resnet 50 V2					
N Neuron	64	128	256	64	128	256
Param	3			6		
MSE	0,7278	0,3889	**0,2879**	0,2964	0,1691	**0,1493**
MAE	0,6253	0,4266	**0,3969**	0,2669	0,2125	**0,2031**
	Mobilenet					
N Neuron	64	128	256	64	128	256
Param	3			6		
MSE	1,6234	1,1669	0,9072	1,6608	0,6238	0,5913
MAE	0,7345	0,5691	0,5094	0,4874	0,2879	0,2969
	Mobilenet V2					
N Neuron	64	128	256	64	128	256
Param	3			6		
MSE	1,2799	1,4788	1,1600	0,6443	0,6477	0,5978
MAE	0,7060	0,6625	0,6470	0,3661	0,3355	0,3311

parâmetros para processar. Também não foi considerado o tempo de treinamento destas redes profundas, já que as mesmas foram utilizadas com seus pesos pré-treinados no banco de dados Imagenet. Como relação aos resultados obtidos com base em características extraídas por redes neurais profundas Resnet V1 e Resnet V2, e também extraídas por Mobilenet V1 e Mobilenet V2, pode-se verificar que os valores de MAE e MSE obtidos são diferentes. Isso poderia ser esperado, como consequência das diferenças entre as estruturas destas redes neurais profundas.

Na Tabela 4.3, são mostrados os valores de MSE e MAE obtidos no processo de estimação de deslocamentos utilizando o OpenCV.

Tabela 4.3 Valores MSE e MAE alcançados pelos algoritmos de cálculo de homografias sobre as imagens de teste, a partir da implementação disponível na biblioteca OpenCV. Na coluna marcada com "3", os alvos, e as suas estimativas, são vetores contendo somente três parâmetros de rotação. Na coluna marcada com "6", os alvos de calibração e suas estimativas têm seis dimensões, porque foram incluídos os três parâmetros de translação da câmera.

	OpenCV	
Param	3	6
MSE	**0,1766**	0,3453
MAE	**0,3107**	0,3220

Na Tabela 4.4, podemos ver o resultado comparativo entre o melhor resultado da utilização de características, extraídas por redes neurais convolucionais profundas, e da utilização do OpenCV. O valor de MSE igual a 0,1493 que é mostrado na Tabela 4.4 corresponde ao mesmo resultado, mostrado na Tabela 4.2, para uma rede neural densamente conectada com 256 neurônios, prevendo alvos de seis dimensões e usando características Resnet V2.

No trabalho foram utilizadas duas versões de conjuntos de alvo (com três e seis parâmetros) e os resultados obtidos refletem essa diferença. Com um gradiente mais rico em informação, associado à presença de mais parâmetros de saída no treinamento das redes neurais, o resultado das redes com alvo de seis parâmetros correspondeu a um erro mais baixo que com três parâmetros de alvo, porque essa quantidade maior de informação permitiu à rede neural densamente conectada se estruturar melhor. Essa diferença não aconteceu no processo do OpenCV, que é um processo baseado em solução de sistema de equações e não em treinamento.

Tabela 4.4 Comparação entre os melhores resultados obtidos com rede neural baseada em características Resnet V2 e alvos de calibração com seis dimensões, conforme a Tabela 4.2, e obtidos com OpenCV e alvos de calibração com três dimensões, conforme a Tabela 4.3.

	OpenCV	Resnet 50 V2
MSE	0,1766	**0,1493**

Estes resultados foram atingidos com o processamento de múltiplas variações de redes neurais profundas e a execução de um processo utilizando a API do OpenCV, para a extração de parâmetros de posição de câmera. A Tabela 4.4 foi constituída com os menores de MSE entre estes métodos para uma comparação quantitativa. A seguir serão apresentadas as conclusões deste trabalho.

5 Conclusões

Neste trabalho foi definido um método de extração de coordenadas de câmera com redes neurais profundas. Estas redes neurais foram carregadas com os pesos treinados na base de dados ImageNet configurando o processo de *transfer learning*. Estas, então, realizaram o processamento de imagens para a extração de características, que, posteriormente, foram usadas para treinar uma rede densamente conectada para estimar parâmetros de rotação e translação da câmera com apenas uma imagem.

Com a utilização da rede neural densamente conectada e usando características extraídas pelas redes neurais convolucionais profundas, o processamento de imagens para a obtenção de parâmetros de localização obteve uma melhora no resultado quando se considera rotação e translação (isto é, estimação de seis parâmetros) em mais de 36% para o MAE (de 0,3220 para 0,2031) e mais de 15% para o MSE (de 0,1766 para 0,1493). Esse resultado foi obtido com a rede neural Resnet V2 e uma rede densamente conectada com 256 neurônios treinada para predição dos alvos de calibração a partir das características.

Em alguns trabalhos já são utilizadas redes neurais para os processos de VSLAM (*Visual Simultaneous Localization and Mapping*) como em “LIFT-SLAM” (Hudson *et al*., 2021) ou “A deep-learning real-time visual SLAM system based on multi-task feature extraction network and self-supervised feature points” (Guangqiang *et al*., 2021).

Como trabalhos futuros podemos explorar mais a utilização de redes neurais específicas para aprimorar o resultado e aplicar mais diretamente esse uso em um processo de SLAM visual.

A utilização de redes neurais convolucionais pré-treinadas no banco de dados ImageNet resultou em uma melhora em relação ao processo de cálculo de deslocamento baseado em homografia já padronizado pela API OpenCV. É possível que utilizar redes

neurais profundas treinadas para realizar o mapeamento e a geração direta de coordenadas leve a um resultado ainda melhor.

Bibliografia

(Abadi *et al.*, 2015) M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, https://www.tensorflow.org/, 2015.

(Ahmed, 1999) M. T. Ahmed, E. Hemayed, A. A. Farag, Neurocalibration: A Neural Network That Can Tell Camera Calibration Parameters, CVIP Lab. University of Louisville, 1999.

(Bradski, 2000) G. Bradski, The OpenCV Library. Dr. Dobb's Journal of Software Tools, 2000.

(Bruno *et al.*, 2021) H. M. S. Bruno, E L. Colombini, LIFT-SLAM: A Deep-Learning Feature-Based Monocular Visual SLAM Method, Neurocomputing, Volume 455, 2021,

(Chollet *et al.*, 2015) F. Chollet, & others. Keras. https://keras.io, 2015.

(Do, 1999) Y. Do, Application of Neural Networks for Stereo-Camera Calibration, School of Computer and Communication Engineering, Taegu University, 1999.

(Deng *et al*., 2009) J. Deng, W. Dong, R. Socher, L. Li., K. Li, L. Fei-Fei, ImageNet: A large-scale hierarchical image database. IEEE conference on computer vision and pattern recognition, 2009.

(Faugeras, 1993) O. Faugeras, Three-Dimensional Computer Vision: A Geometric Viewpoint, MIT Press, 1993.

(Guangqiang *et al*., 2020) L. Guangqiang, Y. Lei, F. Shumin, A Deep-Learning Real-Time Visual SLAM System Based on Multi-Task Feature Extraction Network and Self-Supervised Feature Points, Measurement, Volume 168, 2020

(Gao, 2017) X. Gao, T. Zhang, Y. Liu, Q. Yan. Lectures on Visual SLAM: From Theory to Practice. Publishing House of Electronics Industry, 2017.

(He *et al*., 2016) K. He, X. Zhang, S. Ren, J. Sun. Deep Residual Learning for Image Recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016.

(Hess, 2010) R. Hess. Blender Foundations: The Essential Guide to Learning Blender 2.6, Focal Press, 2010.

(Howard *et al*., 2017) A. G. Howard, M. Zhu, B, Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, Computer Vision and Pattern Recognition, 2017.

(Harris *et al*., 2020) C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, T. E. Oliphant. Array programming with NumPy. Nature, 2020.

(Hartley e Zisserman, 2004) R. Hartley e A. Zisserman. Multiple View Geometry. Cambridge University Press, Segunda Edição, 2004.

(Kelley, 1960) H. J. Kelley, Gradient theory of optimal flight paths. Ars Journal, 1960.

(LeCun, 1989) Y. LeCun; B. Boser; J. S. Denker; D. Henderson; R. E. Howard; W. Hubbard; L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. Periódico Neural Computation, IEEE, 1989.

(Mallick, 2016) S. Mallick, Rotation Matrix to Euler Angles, https://learnopencv.com/rotation-matrix-to-euler-angles/, 2016.

(Pedra *et al*., 2013) A. V. B. M. Pedra, M. Mendonça, M. A. F. Finocchio, L. V. R. Arruda, J. E. C. Castanho, Camera Calibration Using Detection and Neural Networks, International Federation of Automatic Control, 2013.

(Rosenblatt, 1958) F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. Cornell Aeronautical Laboratory, 1958.

(Rossum, 2009) V. Rossum, G. & D. Jr, F.L. *Python 3 Reference Manual*, Scotts Valley, CA: CreateSpace, 2009.

(Rumelhart *et al.*, 1986) D. Rumelhart, G. Hinton e R. Williams. Learning representations by back-propagating errors. Nature 323, pp. 533-586, 1986.

(Simonyan et. al., 2014) K. Simonyan, A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, ICLR, 2014.

(Silva, 2003) L. C. Silva. Método Robusto para a Calibração de Câmeras em Estereofotogrametria. Tese de Doutorado, UFRJ/COPPE/PEE, 2003.

(Taketomi *et al.*, 2017) T. Taketomi, H. Uchiyama, S. Ikeda. Visual SLAM algorithms: a survey from 2010 to 2016. IPSJ Transactions on Computer Vision and Applications, 2017.

Anexo A – Código em Python

A. 1 Extração de parâmetros no Blender

```
import bpy
import os
paramsFile = "groundTruth.txt"
scn = bpy.context.scene
fileParam = open(paramsFile, "w")
increment =0
for f in range(0, 30000):
    scn.frame_set(f)
    mat = scn.camera.matrix_world
    camRotationParams=scn.camera.rotation_euler
    fileParam.write("%d,
%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s\n"   %   (   increment,
mat[0][0], mat[0][1], mat[0][2], mat[1][0], mat[1][1], mat[1][2],
mat[2][0],    mat[2][1],    mat[2][2],    mat[0][3],    mat[1][3],
mat[2][3],camRotationParams[0],camRotationParams[1],camRotationParam
s[2]))
    increment+=1
```

A. 2 – Extração de *features* das redes neurais

```
from keras.models import Sequential
from keras.layers import Dense
from keras import metrics
import numpy as np
import csv
import cv2
import random
import tensorflow as tf


gpus = tf.config.experimental.list_physical_devices('GPU')

for gpu in gpus:
```

```
    tf.config.experimental.set_memory_growth(gpu, True)


#definindo a semente de geração de números aleatórios.
random.seed(19801001)

basePath="C:/Users/rrr_0/Documents/bkp_mestrado/mestrado/"

data_input_file= basePath + 'groundtruth.txt'


#Escolhe entre os tipos de redes
for choose in [1,2,3,4,5]:

    video_path = basePath + "recordedvideo224.mkv"
    vidObj  =  cv2.VideoCapture(video_path)  #Abre  o  vídeo  para
processamento

    #Tamanho padrão aguardado pelas redes
    img_height = 224
    img_width = 224

    wfn='_WeightsFile.txt'

    if (choose==1):
        from keras.applications.vgg16 import VGG16
        from keras.applications.vgg16 import preprocess_input
        graphPrefix="VGG"
        model   =   VGG16(input_shape=(224,224,3),include_top=False,
pooling="avg",weights="imagenet")

    if (choose==2):
        from keras.applications.resnet import ResNet50
        from keras.applications.resnet import preprocess_input
        graphPrefix="Resnet"
        model  =  ResNet50(input_shape=(224,224,3),include_top=False,
pooling="avg", weights="imagenet")

    if (choose==3):
        from keras.applications.resnet_v2 import ResNet50V2
        from keras.applications.resnet_v2 import preprocess_input
        graphPrefix="ResnetV2"
        model                                                    =
ResNet50V2(input_shape=(224,224,3),include_top=False,  pooling="avg",
weights="imagenet")

    if (choose==4):
        from keras.applications.mobilenet import MobileNet
        from keras.applications.mobilenet import preprocess_input
```

```
        graphPrefix="Mobilenet"
        model = MobileNet(input_shape=(224,224,3),include_top=False,
pooling="avg", weights="imagenet")

    if (choose==5):
        from keras.applications.mobilenet_v2 import MobileNetV2
        from keras.applications.mobilenet_v2 import preprocess_input
        graphPrefix="MobilenetV2"
        model                                                    =
MobileNetV2(input_shape=(224,224,3),include_top=False,
pooling="avg", weights="imagenet")

    weightsFileName = basePath + graphPrefix + wfn
    print(weightsFileName)
    lines=[]
    count=0
    weightsFile = open(weightsFileName,"w")
    while vidObj.isOpened():
        success, image = vidObj.read()
        if success:
            #image  =  cv2.resize(image,  (img_height,  img_width))
#adequa o tamanho da imagem
            image = image.reshape((1, image.shape[0], image.shape[1],
image.shape[2]))
            preproc_image = preprocess_input(image)
            predict_result = model.predict(preproc_image)
            temp_parameter=[]
            for item in predict_result[0]:
                temp_parameter.append(item)
            #line_write=str(count)+","+",".join(str(v)   for   v   in
temp_parameter)
            line_write=",".join(str(v) for v in temp_parameter)
            lines.append(line_write)
            if (count%600) == 0:
                print (count)
                for line in lines:
                    weightsFile.write(line)
                    weightsFile.write("\n")
                lines=[]
                weightsFile.flush()
            if count==30000:
                print("breaking on 30k")
                break
        else:
            print ("breaking on fail:" + str(count))
            break
        count+=1
    for line in lines:
        weightsFile.write(line)
        weightsFile.write("\n")
```

```
    lines=[]
    weightsFile.close()
```

A. 3 – Treinamento rede neural densamente conectada

```
from keras.models import Sequential
from keras.layers import Dense
from tensorflow import keras
from keras import metrics
import numpy as np
import matplotlib.pyplot as plt
import csv
import cv2
import random
import tensorboard
import tensorflow as tf
from datetime import datetime

### configuration Steps
gpus = tf.config.experimental.list_physical_devices('GPU')

for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
###

#definindo a semente de geração de números aleatórios.
#random.seed(1980100119810212)
random.seed(19801001)

#basePath = "/home/rodrigo/mestrado/"
basePath="C:/Users/rrr_0/Documents/bkp_mestrado/mestrado/"


plt.rcParams["figure.figsize"] = (16.5, 10.5)

wfn='_WeightsFile.txt'

##definindo pasta de log para tensorboard
logdir = "logs/scalars/" + datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = keras.callbacks.TensorBoard(log_dir=logdir)
graphPrefix=""

MSE=True

train_size=3000

for output_count in [6]:
#choose 3 or 6
#output_count=6
```

```
    if output_count == 6:
        data_input_file=basePath + 'dataconfiguration.txt'
    else:
        data_input_file=basePath + 'dataconfiguration_less.txt'


    #Load training data
    train_result=[]
    line_count = 0
    with open(data_input_file) as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
            train_result.append([float(i) for i in row])
            line_count = line_count +1
#    tf.data.experimental.make_csv_dataset(csv_file)
    #print("linecount:%s" % line_count)
    #Filtering out the fields that will be used

    #loading Training data
    #dataset = tf.contrib.data.CsvDataset("")


    train_output=np.array(train_result[:27000])
    validation_output=np.array(train_result[27001:28000])
    test_output=np.array(train_result[28001:29999])

    for choose in [1,2,3,4,5]:
    #choose between 1 and 5
    #choose=3

        for neuron_qtd in [256]:#512,256,128,64]:
        #choose neuron qtd
        #neuron_qtd = 256

            #Modelo da saida
            model_out = Sequential()

            print("choose:", choose, " neuron_qtd:", neuron_qtd, "
output_count:", output_count)

            if (choose==1):
                weightsFileName = basePath + 'vgg16WeightsFile.txt'
                model_out.add(Dense(neuron_qtd,       input_dim=512,
activation='tanh'))
                model_out.add(Dense(output_count,
input_dim=neuron_qtd, activation='linear'))
                graphPrefix="VGG"

            if (choose==2):
```

```
                weightsFileName         =         basePath         +
'resnet50WeightsFile.txt'
                model_out.add(Dense(neuron_qtd,      input_dim=2048,
activation='tanh'))
                model_out.add(Dense(output_count,
input_dim=neuron_qtd, activation='linear'))
                graphPrefix="Resnet"

            if (choose==3):
                weightsFileName         =         basePath         +
'resnet50V2WeightsFile.txt'
                model_out.add(Dense(neuron_qtd,      input_dim=2048,
activation='tanh'))
                model_out.add(Dense(output_count,
input_dim=neuron_qtd, activation='linear'))
                graphPrefix="ResnetV2"

            if (choose==4):
                weightsFileName         =         basePath         +
'mobilenetWeightsFile.txt'
                model_out.add(Dense(neuron_qtd,      input_dim=1024,
activation='tanh'))
                model_out.add(Dense(output_count,
input_dim=neuron_qtd, activation='linear'))
                graphPrefix="Mobilenet"

            if (choose==5):
                weightsFileName         =         basePath         +
'mobilenetV2WeightsFile.txt'
                model_out.add(Dense(neuron_qtd,      input_dim=1280,
activation='tanh'))
                model_out.add(Dense(output_count,
input_dim=neuron_qtd, activation='linear'))
                graphPrefix="MobilenetV2"

            weightsFileName = basePath + graphPrefix + wfn


optimization_method=keras.optimizers.Adam(learning_rate=0.0008,
                                                    beta_1=0.8,
                                                    beta_2=0.999)

#optimization_method=keras.optimizers.SGD(learning_rate=0.001)
            #optimization_method=keras.optimizers.SGD()

#optimization_method=tf.keras.optimizers.Adamax(learning_rate=0.0001
)

#optimization_method=keras.optimizers.SGD(learning_rate=0.2)
```

```
            #model_out.compile(optimizer='adam',         loss='mse',
metrics=['mae'])
            model_out.compile(optimizer=optimization_method,
loss='mse', metrics=['mae'])
            #model_out.compile(optimizer='adam',         loss='mse',
metrics=['mae'])

            #load params file
            features_extracted=[]
            line_count = 0
            with open(weightsFileName) as csv_file:
                print("weightsFileName:",weightsFileName)
                csv_reader = csv.reader(csv_file, delimiter=',')
                for row in csv_reader:
                    features_extracted.append([float(i)   for   i   in
row])
                    line_count = line_count +1
            #print("linecount:%s" % line_count)

            #transformar os valores para numpy array
            input_data=np.array(features_extracted[:27000])

validation_data=np.array(features_extracted[27001:28000])
            test_data=np.array(features_extracted[28001:29999])


            history = model_out.fit(input_data
                                    ,train_output
                                    ,epochs=20
                                    ,workers=4
                                    #,batch_size=train_size
                                    ,verbose=0
                                    ,shuffle=True
                                    #,validation_split  =  0.2  #Usar
validation split ou validation data 0.2 equivale a 20%

,validation_data=(validation_data, validation_output)

,callbacks=[tensorboard_callback]
                                   )

            line_label=graphPrefix+"_MSE_"+str(neuron_qtd)


            plt.figure(1)
            plt.plot(history.history['loss'], label=line_label)

            plt.figure(2)
            plt.semilogy(history.history['loss'], label=line_label)
```

```
            line_label=graphPrefix+"_MSE_"+str(neuron_qtd)+"_val"

            plt.figure(1)
            plt.plot(history.history['val_loss'], label=line_label)

            plt.figure(2)
            plt.semilogy(history.history['val_loss'],
label=line_label)


            line_label=graphPrefix+"_MAE_"+str(neuron_qtd)

            plt.figure(3)
            plt.plot(history.history['mae'], label=line_label)

            plt.figure(4)
            plt.semilogy(history.history['mae'], label=line_label)

            line_label=graphPrefix+"_MAE_"+str(neuron_qtd)+"_val"

            plt.figure(3)
            plt.plot(history.history['val_mae'], label=line_label)

            plt.figure(4)
            plt.semilogy(history.history['val_mae'],
label=line_label)

            print("Average          train          loss:          ",
str(np.average(history.history['loss'])).replace(".", ","))
            print("Average          train           mae:          ",
str(np.average(history.history['mae'])).replace(".", ","))

            history_eval      =      model_out.evaluate(x=test_data,
y=test_output, verbose=1)
            train_result.append([float(i) for i in row])

            print("Average          test          loss:          ",
str(history_eval[0]).replace(".", ","))
            print("Average          test           mae:          ",
str(history_eval[1]).replace(".", ","))

            print(history_eval)


##MSE
plt.figure(1)
plt.title('Models MSE')
plt.ylabel('MSE')
plt.xlabel('epoch')
plt.legend()
```

```
plt.figure(2)
plt.title('Models MSE')
plt.ylabel('MSE')
plt.xlabel('epoch')
plt.legend()

##MAE
plt.figure(3)
plt.title('Models MAE')
plt.ylabel('MAE')
plt.xlabel('epoch')
plt.legend()

plt.figure(4)
plt.title('Models MAE')
plt.ylabel('MAE')
plt.xlabel('epoch')
plt.legend()

plt.show()
```

A. 4 – Comparação de resultados OpenCV com *GroundTruth*

```
fileParamsOpenCV = "dataFileFormated.txt"
fileGroundTruth = "groundTruth.txt"


if __name__ == "__main__":
   loadFloats = list()
   with open(fileParamsOpenCV) as dataFile:
      for line in dataFile:
         line = line.strip()
         loadFloats.append(list(map(float, line.split(","))))
   groundTruth = list()
   somaGroundTruth=list()
   for index in range(15):
      somaGroundTruth.append(float(0))
   with open(fileGroundTruth) as dataFile:
      for line in dataFile:
         line = list(map(float, line.split(",")))
```

```
            groundTruth.append(line)
            for index in range(15):
               somaGroundTruth[index]+=line[index+1]
      gtIndex = 0
      compareCount=0
      difStatistics=list()
      listSize=len(groundTruth)
      list2size = len(loadFloats)
      #print (somaGroundTruth)

      somaLoss = list()
      for index in range(15):
         somaLoss.append(float(0))

      for loadFloatsLine in loadFloats:
         if loadFloatsLine[0] == groundTruth[gtIndex][0]:
            rowDif=list()
            for index in range(1,16):
               rowDif.append((groundTruth[gtIndex][index]-
loadFloatsLine[index])**2)
               somaLoss[index-1]   +=   abs(groundTruth[gtIndex][index]-
loadFloatsLine[index]);
            difStatistics.append(rowDif)
            gtIndex+=1
         else:
            print("ERROR!!!!!")
            exit(0)

      soma=list()
      for index in range(15):
         soma.append(float(0))
      for lineList in difStatistics:
         for index in range(15):
```

```
      soma[index] += float(lineList[index])

somaMAE =  1.0/ len(difStatistics) * soma[9];
somaMAE += 1.0/ len(difStatistics) * soma[10];
somaMAE += 1.0/ len(difStatistics) * soma[11];
somaMAE += 1.0/ len(difStatistics) * soma[12];
somaMAE += 1.0/ len(difStatistics) * soma[13];
somaMAE += 1.0/ len(difStatistics) * soma[14];
somaMAE = somaMAE / 6.0;

print("output MAE6 params:", somaMAE)

somaMAE = 1.0/ len(difStatistics) * soma[12];
somaMAE += 1.0/ len(difStatistics) * soma[13];
somaMAE += 1.0/ len(difStatistics) * soma[14];
somaMAE = somaMAE / 3.0;

print("output MAE3 params:", somaMAE)

lossCalc = 1.0/ len(difStatistics) *somaLoss[9];
lossCalc += 1.0/ len(difStatistics) *somaLoss[10];
lossCalc += 1.0/ len(difStatistics) *somaLoss[11];
lossCalc += 1.0/ len(difStatistics) *somaLoss[12];
lossCalc += 1.0/ len(difStatistics) *somaLoss[13];
lossCalc += 1.0/ len(difStatistics) *somaLoss[14];

print("loss 6 params: ", lossCalc/6.0)

lossCalc = 1.0/ len(difStatistics) *somaLoss[12];
lossCalc += 1.0/ len(difStatistics) *somaLoss[13];
lossCalc += 1.0/ len(difStatistics) *somaLoss[14];

print("loss 3 params: ", lossCalc/3.0)
```

Anexo B - Código em C++

B. 1 - Geração parâmetros OpenCV

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include <fstream>
#define DESTINATION_FILE_PATH "/home/rodrigo/mestrado/dataFile.txt"
using namespace std;
using namespace cv;

Vec3f rotationMatrixToEulerAngles(Mat &R);

void find_feature_matches(
 const Mat &img_1, const Mat &img_2,
 std::vector<KeyPoint> &keypoints_1,
 std::vector<KeyPoint> &keypoints_2,
 std::vector<DMatch> &matches);

void pose_estimation_2d2d(
 std::vector<KeyPoint> keypoints_1,
 std::vector<KeyPoint> keypoints_2,
 std::vector<DMatch> matches,
 Mat &R, Mat &t);

Point2d pixel2cam(const Point2d &p, const Mat &K);
```

```
int main(int argc, char **argv) {

 VideoCapture cap("/home/rodrigo/mestrado/recordedvideo224.mkv");

 cout << "is open: " << cap.isOpened() << endl;

 Mat img_1;
 cap.set(cv::CAP_PROP_POS_FRAMES, 250); //get frame 250 to start
comparison
 cap.read(img_1);//this will be static and compared to all others
 cap.set(cv::CAP_PROP_POS_FRAMES, 0);
 Mat img_2;
 while (cap.isOpened()) {
     cap.read(img_2);
     assert(img_1.data && img_2.data && "Can not load images!");

     vector<KeyPoint> keypoints_1, keypoints_2;
     vector<DMatch> matches;
     find_feature_matches(img_1, img_2, keypoints_1, keypoints_2,
matches);

     if (matches.size() < 7){
         continue;
     }

     Mat R, t;
     pose_estimation_2d2d(keypoints_1, keypoints_2, matches, R, t);

     Mat t_x =
             (Mat_<double>(3, 3) << 0, -t.at<double>(2, 0),
t.at<double>(1, 0),
                     t.at<double>(2, 0), 0, -t.at<double>(0, 0),
```

```
                    -t.at<double>(1, 0), t.at<double>(0, 0), 0);

    cout << "t^R=" << endl << t_x * R << endl;

    Mat K = (Mat_<double>(3, 3) << 520.9, 0, 325.1, 0, 521.0, 249.7,
0, 0, 1);
    for (DMatch m: matches) {
        Point2d pt1 = pixel2cam(keypoints_1[m.queryIdx].pt, K);
        Mat y1 = (Mat_<double>(3, 1) << pt1.x, pt1.y, 1);
        Point2d pt2 = pixel2cam(keypoints_2[m.trainIdx].pt, K);
        Mat y2 = (Mat_<double>(3, 1) << pt2.x, pt2.y, 1);
        Mat d = y2.t() * t_x * R * y1;
        cout << "epipolar constraint = " << d << endl;
    }
 }
 return 0;
}

void find_feature_matches(const Mat &img_1, const Mat &img_2,
                          std::vector<KeyPoint> &keypoints_1,
                          std::vector<KeyPoint> &keypoints_2,
                          std::vector<DMatch> &matches) {
 Mat descriptors_1, descriptors_2;
 // used in OpenCV3
 Ptr<FeatureDetector> detector = ORB::create();
 Ptr<DescriptorExtractor> descriptor = ORB::create();
 // use this if you are in OpenCV2
 // Ptr<FeatureDetector> detector = FeatureDetector::create ( "ORB"
);
 // Ptr<DescriptorExtractor> descriptor = DescriptorExtractor::create
( "ORB" );
 Ptr<DescriptorMatcher>                    matcher                    =
DescriptorMatcher::create("BruteForce-Hamming");
```

```
 detector->detect(img_1, keypoints_1);
 detector->detect(img_2, keypoints_2);

 descriptor->compute(img_1, keypoints_1, descriptors_1);
 descriptor->compute(img_2, keypoints_2, descriptors_2);

 vector<DMatch> match;
 //BFMatcher matcher ( NORM_HAMMING );
 matcher->match(descriptors_1, descriptors_2, match);

 double min_dist = 10000, max_dist = 0;

 for (int i = 0; i < descriptors_1.rows; i++) {
   double dist = match[i].distance;
   if (dist < min_dist) min_dist = dist;
   if (dist > max_dist) max_dist = dist;
 }

 printf("-- Max dist : %f \n", max_dist);
 printf("-- Min dist : %f \n", min_dist);

 for (int i = 0; i < descriptors_1.rows; i++) {
   if (match[i].distance <= max(2 * min_dist, 30.0)) {
     matches.push_back(match[i]);
   }
 }
}

Point2d pixel2cam(const Point2d &p, const Mat &K) {
 return Point2d
   (
     (p.x - K.at<double>(0, 2)) / K.at<double>(0, 0),
     (p.y - K.at<double>(1, 2)) / K.at<double>(1, 1)
```

```
   );
}
#define    CV_FM_8POINT   2
static int increment = 0;
void pose_estimation_2d2d(std::vector<KeyPoint> keypoints_1,
                          std::vector<KeyPoint> keypoints_2,
                          std::vector<DMatch> matches,
                          Mat &R, Mat &t) {
 Mat K = (Mat_<double>(3, 3) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0,
0, 1);

 vector<Point2f> points1;
 vector<Point2f> points2;

 for (int i = 0; i < (int) matches.size(); i++) {
   points1.push_back(keypoints_1[matches[i].queryIdx].pt);
   points2.push_back(keypoints_2[matches[i].trainIdx].pt);
 }

 Mat fundamental_matrix;
 fundamental_matrix    =    findFundamentalMat(points1,    points2,
CV_FM_8POINT);
 cout << "fundamental_matrix is " << endl << fundamental_matrix <<
endl;

 Point2d principal_point(325.1, 249.7);
 double focal_length = 50;//521;
 Mat essential_matrix;
 essential_matrix = findEssentialMat(points1, points2, focal_length,
principal_point);
 cout << "essential_matrix is " << endl << essential_matrix << endl;

 Mat homography_matrix;
```

```
 homography_matrix = findHomography(points1, points2, RANSAC, 3);
 cout << "homography_matrix is " << endl << homography_matrix << endl;

 recoverPose(essential_matrix, points1, points2, R, t, focal_length,
principal_point);
 cout << "R is " << endl << R << endl;
 Vec3f vecRot =  rotationMatrixToEulerAngles(R);
 ofstream dataFile;
   dataFile.open (DESTINATION_FILE_PATH, ios::app);
   dataFile << increment << ",";
   increment++;
   dataFile << R.at<double>(0,0) << ",";
   dataFile << R.at<double>(0,1) << ",";
   dataFile << R.at<double>(0,2) << ",";
   dataFile << R.at<double>(1,0) << ",";
   dataFile << R.at<double>(1,1) << ",";
   dataFile << R.at<double>(1,2) << ",";
   dataFile << R.at<double>(2,0) << ",";
   dataFile << R.at<double>(2,1) << ",";
   dataFile << R.at<double>(2,2) << ",";
   dataFile << t.at<double>(0) << ",";
   dataFile << t.at<double>(1) << ",";
   dataFile << t.at<double>(2) << ",";
   dataFile << vecRot.val[0] << ", " << vecRot.val[1] << ", " <<
vecRot.val[2] << endl;
   dataFile.close();

 cout << "t is " << endl << t << endl;

}

// Checks if a matrix is a valid rotation matrix.
bool isRotationMatrix(Mat &R)
```

```
{
   Mat Rt;
   transpose(R, Rt);
   Mat shouldBeIdentity = Rt * R;
   Mat I = Mat::eye(3,3, shouldBeIdentity.type());

   return  norm(I, shouldBeIdentity) < 1e-6;

}

// Calculates rotation matrix to euler angles
// The result is the same as MATLAB except the order
// of the euler angles ( x and z are swapped ).
Vec3f rotationMatrixToEulerAngles(Mat &R)
{

   assert(isRotationMatrix(R));

   float   sy   =   sqrt(R.at<double>(0,0)   *   R.at<double>(0,0)   +
R.at<double>(1,0) * R.at<double>(1,0) );

   bool singular = sy < 1e-6; // If

   float x, y, z;
   if (!singular)
   {
       x = atan2(R.at<double>(2,1) , R.at<double>(2,2));
       y = atan2(-R.at<double>(2,0), sy);
       z = atan2(R.at<double>(1,0), R.at<double>(0,0));
   }
   else
   {
       x = atan2(-R.at<double>(1,2), R.at<double>(1,1));
```

```
        y = atan2(-R.at<double>(2,0), sy);
        z = 0;
    }
    return Vec3f(x, y, z);
}
```