Capítulo 6

Soluções de Aprendizado de Máquina para Estimar em Tempo Real a Pose Humana em Aplicações de Saúde

Renan Nascimento (UFDPar), José Everton Fontenele (UFDPar), Rodrigo Baluz (UFPI/UESPI), Rayele Moreira (UFPI/UNINTA), Silmar Teixeira (UFDPar/UFPI), Ariel Teles (IFMA/UFDPar/UFMA)

Abstract

Artificial intelligence applications have been proposed as a solution to several health problems. Machine learning algorithms for computer vision have been used to aid in the diagnosis, monitoring and treatment of problems that affect people's health. This short course aims to introduce the TensorFlow library and models applied to real-time recognition of points in the body joints and segments, which represent the position of the human body, contributing to the development of solutions in the area of health informatics. To exemplify, two TensorFlow-based applications are presented: the first one is proposed to assist in postural and range of motion assessment; the second one is a serious game used in the neurofunctional rehabilitation process in upper limbs for post-stroke patients in domestic settings and rehabilitation clinics.

Resumo

Aplicações de inteligência artificial vêm sendo propostas como solução para diversos problemas na área da saúde. Algoritmos de aprendizado de máquina para visão computacional vêm sendo usados para auxílio no diagnóstico, monitoramento e tratamento de problemas que afetam a saúde de pessoas. Este minicurso visa introduzir a biblioteca TensorFlow e modelos aplicados ao reconhecimento em tempo real de pontos nas articulações e segmentos corporais, que representam a posição do corpo humano, contribuindo para o desenvolvimento de soluções na área de informática em saúde. Para exemplificar, são apresentadas duas aplicações que fazem uso do TensorFlow: uma para auxiliar na avaliação postural e da amplitude de movimento; a outra, um jogo sério usada no processo de reabilitação neurofuncional em membros superiores para pacientes pós-AVC em ambientes domésticos e clínicas de reabilitação.

6.1. Introdução

O corpo humano é organizado em sistemas formados por estruturas de variadas complexidades. Dentre eles, os sistemas esquelético e articular, que são interdependentes. O primeiro é construído por ossos de tamanhos variados que se conectam, dando origem ao que conhecemos como articulações do corpo, que por sua vez, formam o sistema articular. Ossos e articulações possuem formatos variados e irregulares, dependendo da região corporal a qual pertencem. Do ponto de vista funcional, os ossos funcionam como alavancas e as articulações como eixos, ajudando na mobilidade do corpo humano [Graaff and Marshall 2003]. Anatomicamente, ossos e articulações são usados como ponto de referência em processos avaliativos, tais como é feito na avaliação antropométrica, método por meio do qual se faz a medição do tamanho do corpo ou de suas partes individualmente. Esse processo é tradicionalmente feito usando uma fita métrica para mensurar o tamanho dos segmentos (e.g., braço, perna, tronco), usando saliências/proeminências ósseas como pontos anatômicos de referência [Sabharwal and Kumar 2008].

Além da antropometria, a identificação de pontos e segmentos anatômicos é importante para outros processos que envolvem estimativa da pose humana, tais como a avaliação postural, que permite identificar alterações no alinhamento corporal e a avaliação goniométrica, técnica de avaliação da amplitude de movimento articular. Apesar do método clínico direto usando fita métrica ser considerado de fácil aplicação, ele depende da identificação manual desses pontos de referência e essa avaliação pode ficar sujeita a vieses, dependendo da experiência do examinador. Além dessas avaliações, a estimativa de pose humana também pode ser útil em processos de avaliação dinâmica, como avaliação da marcha e em processos terapêuticos que visem ganho de mobilidade, como na reabilitação motora. Além dos métodos manuais de avaliação que envolvem inspeção, palpação e mensuração com a fita métrica, os métodos baseados em modelos computacionais têm sido empregados como ferramentas de avaliação, tais como a fotogrametria, a qual pode fazer uso de Visão Computacional (VC).

A interação entre homem e máquinas faz parte do cotidiano das pessoas no mundo industrializado e essa relação tem evoluído de tal forma que hoje já falamos de máquinas que conseguem enxergar e fazer coisas tão bem ou melhor que o ser humano. Hoje temos dispositivos com capacidade de processamento cada vez maior e com funções variadas que vão além daquelas que dispunham na sua concepção original. A interação homem-máquina atingiu um nível no qual é possível ser guiado por elas por lugares desconhecidos sem se perder, ser orientado sobre condições climáticas, fazer reconhecimento facial de animais e humanos, entre outras funções. Essa habilidade das máquinas enxergarem, processarem e interpretarem imagens é conhecida como VC [Forsyth and Ponce 2002].

No caso da fotogrametria, que é um método computadorizado que pode ser usado para estimar postura, amplitude de movimento e medidas antropométricas, a base é o processamento de uma fotografia do paciente, realizadas na vista anterior, posterior e laterais. Métodos como esse alavancaram o uso da VC para além do processamento de imagens médicas (e.g., tomografia, Raio-X, ressonância magnética), passando a ser usada como recurso para estimativa e avaliação dinâmica do ser humano. Além da fotogrametria, o uso de VC pode ser também aplicado na técnica de gameterapia para a reabi-

litação de pacientes com sequelas motoras após Acidente Vascular Cerebral (AVC). A gameterapia se apresenta como um método auxiliar que utiliza cenários interativos dos jogos sérios para potencializar a humanização do tratamento e a experiência do paciente durante a execução dos exercícios necessários para a recuperação das funções motoras [Domínguez-Téllez et al. 2020].

Avanços computacionais têm permitido utilizar máquinas em tarefas variadas que incluem diferentes níveis de complexidade. Boa parte desses avanços estão associados ao uso de algoritmos de aprendizado de máquina (do inglês, *Machine Learning* - ML) que processam e interpretam datasets de forma inteligente. Baseado no processamento de datasets de uma amostra, algoritmos treinados identificam padrões existentes e, a partir deles, fazem inferências à medida que novos dados são incorporados ao sistema [Aneja et al. 2019]. Esses algoritmos ampliaram ainda mais o alcance dos modelos de VC, permitindo interpretar diferentes características de imagens e dados, extraindo deles informações que permitam compreendê-las, interpretá-las e aplicá-las a algum propósito. A possibilidade de processar grande conjuntos de dados e extrair deles informações úteis, para alcançar um dado objetivo de forma mais rápida e precisa, fazem com que esses algoritmos sejam usados em diferentes aplicações de saúde.

A partir do uso de algoritmos de ML, processos de identificação dos pontos e segmentos anatômicos passam a ocorrer de forma automatizada. Essa automatização pode favorecer o desenvolvimento de instrumentos e métodos terapêuticos dinâmicos que podem facilitar a recuperação e adesão dos pacientes ao tratamento. Nesse contexto, o objetivo deste capítulo é introduzir a biblioteca *TensorFlow* (TF) e suas extensões aplicadas ao reconhecimento em tempo real de estruturas e movimento do corpo humano, além de apresentar algumas de suas possíveis contribuições para as pesquisas na área de informática em saúde.

Este capítulo está organizado da seguinte maneira. A Seção 6.2 apresenta a biblioteca TF. Na Seção 6.3, são apresentados duas extensões de VC do TF, o *PoseNet* e o *MediaPipe*, para a estimação da pose humana em tempo real, enquanto a Seção 6.4 ilustra exemplos de aplicações de saúde que fazem uso dessas extensões. Por fim, a Seção 6.5 conclui o capítulo com nossas considerações.

6.2. A Biblioteca *TensorFlow*

O TF foi desenvolvido originalmente por pesquisadores e engenheiros que trabalham na equipe do *Google Brain*, dentro da organização *Machine Intelligence Research* da Google, para realizar pesquisas em ML[1]. É uma ferramenta poderosa para treinamento em larga escala, pois utiliza eficientemente centenas de servidores, os quais possuem unidades de processamento gráfico (do inglês, *Graphics Processing Units* - GPUs) e unidade de processamento de tensor (do inglês, *Tensor Processing Units* - TPUs) para treinamento e execução de modelos treinados [Abadi et al. 2016]. As GPUs são processadores dedicados ao processamento e renderização de gráficos em tempo real. Já as TPUs são unidades de processamento de algoritmos de Inteligência Artificial (IA) e Redes Neurais Convolucionais (do inglês, *Convolutional Neural Networks* - CNNs).

[1] `https://github.com/tensorflow/tensorflow`

Em particular, o TF é especializado em Redes Neurais Artificiais (do inglês, *Artificial Neural Networks* - ANNs). Uma ANN é uma ferramenta de inteligência computacional inspirada no sistema neural humano [Basheer and Hajmeer 2000], que visa replicar o processo de comunicação entre neurônios, gerando um modelo de estruturas interconectadas capazes de realizar cálculos para processamento de dados e representação de conhecimento. Uma CNN é um algoritmo de aprendizado profundo que pode captar uma entrada, atribuir importância (i.e., pesos e vieses que podem ser aprendidos) a vários aspectos/objetos do elemento de entrada, e ser capaz de diferenciar ou classificar um do outro.

6.2.1. *Google Colaboratory*

O *Google Colaboratory*, ou simplesmente *Colab*, é uma plataforma executada em nuvem e funciona em tempo real, contendo servidores com GPUs e TPUs. A plataforma foi desenvolvida para tornar mais fácil o trabalho de estudantes, cientistas de dados e pesquisadores da área de IA. A interface do *Colab* apresenta resultados de processamento dos algoritmos em tempo real, o que torna a experiência do usuário (i.e., do desenvolvedor) mais dinâmica. Além disso, ele permite que códigos possam ser compartilhados e os usuários possam trabalhar colaborativamente ao mesmo tempo [Google 2019]. O *Colab* é baseado no *Jupyter Notebook* [Carneiro et al. 2018].

O *Colab* é uma plataforma de ambiente interativo e vem configurada por padrão com a linguagem *Python*. Além disso, ele já vem com as principais bibliotecas úteis para ML prontas para uso, como o TF, o *Pandas*, e o *Matplotlib* [Google 2019]. O ambiente de desenvolvimento do *Colab* é composto por notebooks (i.e., um documento utilizado para o desenvolvimento de algoritmos na plataforma com a extensão .ipynb). Um notebook é composto por lista de células. As células de código são onde se pode escrever e executar códigos fonte, além de visualizar as saídas, que mostram os resultados após a execução. Os resultados mais comuns são textos, tabelas e gráficos [Carneiro et al. 2018]. As células também podem conter textos explicativos que são formatadas usando uma linguagem de marcação chamada *markdown*.

A partir de uma conta do *Google*, utilizando o *Google Drive* ou acessando diretamente o site do *Colab*, é possível criar um novo notebook, como ilustrado na Figura 6.1. Para criar um novo notebook, primeiro clicamos em novo, depois em Mais->Google Colaboraotory.

Abaixo é mostrado uma célula de código (Figura 6.2) usada para a escrita e execução do código fonte, a qual funciona em tempo de execução e pode ser executado individualmente. A execução do código pode ser feita clicando do botão “Play”. Uma célula de texto (Figura 6.3) é útil para adicionar comentários ao notebook. Para editar um bloco de texto, basta clicar duas vezes na célula.

Abaixo é mostrado dois exemplos de como funcionam as células. No primeiro exemplo (Figura 6.4), são criados dois blocos: o primeiro foi inicializado uma variável **v** recebendo uma soma; no segundo bloco, a função *print()* para exibir o valor da variável, já com o resultado da soma. O outro exemplo (Figura 6.5) é da célula de texto escrita na linguagem de marcação *markdown*. A célula é composta por uma barra de ferramentas para ajudar na edição. Do lado esquerdo a parte de marcação da linguagem e do lado

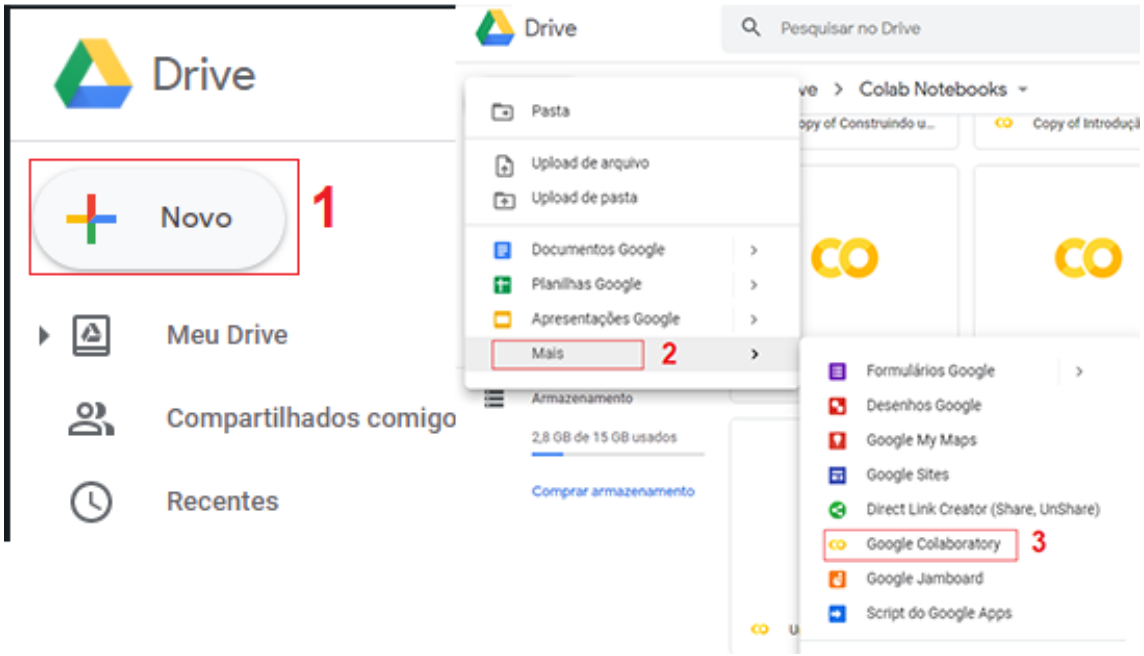

Figura 6.1. Criação de notebook no *Google Colab*.

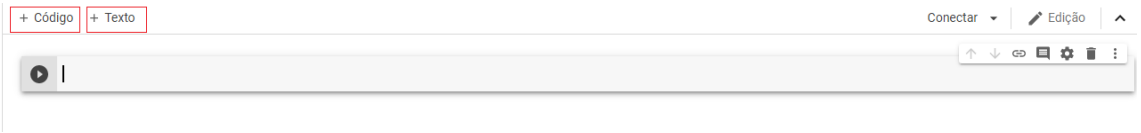

Figura 6.2. Célula para código fonte.

Figura 6.3. Célula para texto usando a linguagem de marcação *markdown*.

direto o resultado.

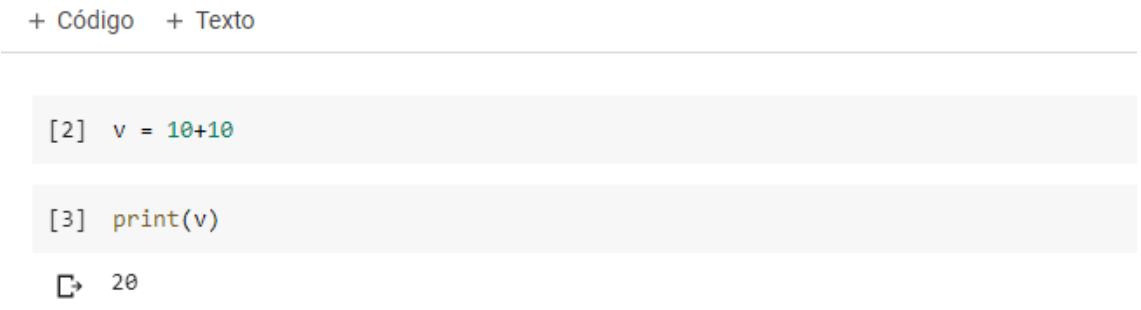

Figura 6.4. Exemplo da célula de código.

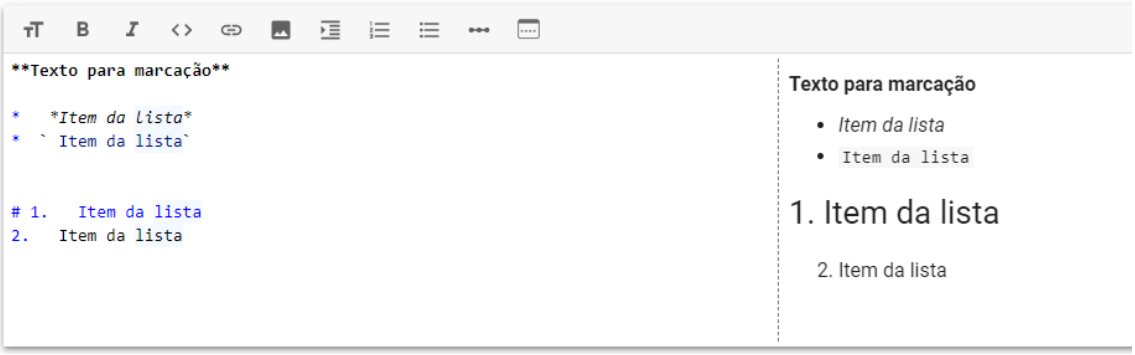

Figura 6.5. Exemplo da célula de texto.

6.2.2. Versões

Até o momento da escrita desse capítulo, o TF está na versão 2.3.0. Nessa seção, serão apresentadas as principais mudanças da versão 1.0 para a 2.0. A primeira versão publicada do TF (v1.0) data de 2018 e já possuía portabilidade para as linguagens *Python*, *JavaScript*, *C++*, *Java*, *Go* e *Swift*, também presentes nas versões mais recentes. Segundo a documentação oficial, a API *Python* é a mais completa, mas as APIs de outras linguagens também podem ser facilmente integradas a projetos e oferecer algumas vantagens de desempenho na execução. Como forma de disseminar a biblioteca, é incentivado à comunidade de desenvolvedores a implementar e manter o suporte para outras linguagens de programação. Dentre elas, tem-se versões em *C#* (i.e., *TensorFlowSharp* e *TensorFlow.NET*), *askell*, *Julia*, *MATLAB*, *R*, *Ruby*, *Rust* e *Scala*.

A versão TF 2.0 veio com uma proposta de simplificar e facilitar seu uso. Uma das principais alterações propostas foi a incorporação do *Keras* (*tf.keras*), a qual é uma API de alto nível do TF. Ela é utilizada para criar e treinar modelos de aprendizado profundo, abordando uma prototipagem rápida e com recursos avançados, tendo como vantagens a facilidade de usar modelos compostos e fácil de entender.

Outras mudanças propostas foram a forma de acesso às variáveis e constantes. Na versão 1.0, a manipulação e atualização de variáveis eram possíveis apenas com a inicialização de uma sessão (*session = tf.Session()*). Então houveram algumas alterações da API, em que muitas classes, funções e tipos de dados foram renomeados ou removidos, e nomes dos argumentos de funções também foram alterados. Por exemplo, na versão 2.0 não existe mais a classe *tf.ConditionalAccumulator*. Houveram também correções no preenchimento automático, recurso agregado à plataforma que completa o código enquanto está digitando. Contudo, todas essas mudanças foram motivadas pela consistência e clareza de sua utilização.

6.2.3. Modelos e Extensões

Um módulo é uma parte autônoma de um grafo do TF, com os respectivos pesos e recursos, que podem ser reutilizados em diferentes tarefas. Dentre os modelos disponíveis, podemos citar os que estão disponíveis nos repositórios *TensorFlow Model Garden*[2],

[2]https://github.com/tensorflow/models

TensroFlow Hub[3] e *TensorFlow.js models*[4]. O *TensorFlow Model Garden* contém os modelos oficiais do TF que usam as APIs de alto nível do TF disponibilizados através do *GitHub*[5]. O *TensorFlow Hub*[6] é um repositório de modelos de ML. Em particular, ele fornece modelos salvos, treinados previamente, que podem ser reutilizados para resolver novas tarefas com menos tempo e menos dados para o treinamento.

Já o *TensorFlow.js models* é um repositório de modelos treinados previamente que foram portados para a linguagem *JavaScript*. Os modelos *TensorFlow.js* são executados no navegador e no ambiente *Node.js*[7]. Alguns dos modelos presentes no repositório são o *PoseNet* para estimativa de pose, o *Coco SSD* para o reconhecimento de objetos, e o *DeepLab V3* para a segmentação semântica, em que a partir de uma imagem é retornado a descrição dos elementos presentes.

6.2.4. Exemplo de Rede Neural Artificial com o *TensorFlow*

Como exemplo, apresentamos o desenvolvimento, treinamento e teste de uma CNN utilizando o TF, disponível em: `https://github.com/RenanFialho-Dev/ercempi_2020`. O *dataset* utilizado para a construção é o *Fashion MNIST*[8], o qual contém um conjunto de treinamento com 60 mil imagens com resolução de 28x28 pixels em escala de cinza, e um conjunto de teste com 10 mil imagens com mesmas características. O dataset possui imagens de roupas e calçados.

Inicialmente, são importadas as bibliotecas TF e *NumPy*. Esta segunda é uma biblioteca para computação numérica, utilizada para operações matemáticas. A partir do TF, é realizada a importação do dataset *MNIST*, o qual é, em seguida, carregado em uma tupla de *arrays* tipados segundo a biblioteca de tipos de dados do *NumPy* (*np.array*). Como ilustrado na Figura 6.6, temos a tupla (*X_treino, Y_treino*) para o conjunto de treinamento com seus rótulos, e *X_teste, Y_teste* para o conjunto de imagens de teste.

```
[14] import numpy as np
     import tensorflow as tf
     from tensorflow.keras.datasets import fashion_mnist
```

Carregando o dataset

```
[11] (X_treino, X_teste), (Y_treino, Y_teste) = fashion_mnist.load_data()
```

Figura 6.6. Bibliotecas importadas e tupla de *arrays* para armazenar o dataset.

Com o dataset carregado, realiza-se a normalização dos dados. O processo de

[3] `https://tfhub.dev/`
[4] `https://www.tensorflow.org/js/models`
[5] `https://github.com/tensorflow/models/tree/master/official`
[6] `https://tfhub.dev/`
[7] Node.js é um interpretador *JavaScript*: `https://nodejs.org/en/`
[8] `https://keras.io/api/datasets/mnist/`

normalização faz com que os dados fiquem dentro de um limite de valores, no nosso caso, entre 0 e 1. Ao realizar esse procedimento, melhoramos a qualidade dos dados, evitando dados ruidosos e garantimos que a nossa CNN irá treinar mais rapidamente. Em seguida, utiliza-se a função *reshape* da biblioteca *NumPy*, a qual realiza uma transposição dos dados para uma nova forma sem que seus dados sejam perdidos. Por exemplo, uma matriz de 4 linhas e 3 colunas após *reshape* ficará com 3 linhas e 4 colunas. Em nosso exemplo, *reshape(-1, 28*28)* retorna uma nova matriz com 60 mil linhas, correspondente a quantidade de imagens que temos, e 784 colunas, que representam os valores de cor em cada pixel, como visto na Figura 6.7.

```
#normalização
X_treino = X_treino/255.0
X_teste = X_teste/255.0

#reshape dos dados
X_treino = X_treino.reshape(-1, 28*28)
X_teste = X_teste.reshape(-1, 28*28)
```

Figura 6.7. Normalização e *reshape* dos dados.

A partir dos dados preparados, construímos a estrutura da nossa CNN. Primeiramente, acessamos a biblioteca *tf.keras* e definimos uma classe do tipo *models* a uma variável, que atribui propriedades de camadas em um objeto com recursos de treinamento e inferência. Seguimos adicionando as configurações para as camadas de entrada, intermediárias (também chamadas de densas, as quais recebem essa definição porque todos os neurônios são conectados aos neurônios da camada seguinte), e camada de saída, utilizando o comando *model.add()*. Cada camada adicionada através da função *tf.keras.layers.Dense()* contará com os hiper-parâmetros (estes são variáveis de configuração contendo os parâmetros que controlam o próprio processo de treinamento): número de unidades/neurônios, função de ativação e tamanho da entrada.

Após a camada de entrada, é adicionada uma camada de *dropout*. Esta é uma técnica de regularização em que definimos aleatoriamente os neurônios de uma camada para zero. Dessa forma, durante o treinamento, esses neurônios não serão atualizados. Como alguma porcentagem de neurônios não será atualizada, todo o processo de treinamento é longo e temos menos chances que a rede aprenda mais sobre o conjunto de treinamento ao invés de aprender a generalizar, chamado também de superajuste (i.e., *overfitting*). Para a camada de saída, o número de unidades/neurônios corresponde a quantidade de classes presentes no dataset (i.e., 10 no *Fashion MNIST*) com uma função de ativação *softmax*, conforme a Figura 6.8.

Com estas configurações, a CNN deve ser compilada, e estará preparada para executar a fase de treinamento e teste. A função *compile* recebe os seguintes parâmetros: um otimizador (*optimizer*), um parâmetro perda (*loss*), e um parâmetro contendo métricas (*metrics)* a serem avaliadas pelo modelo durante o treinamento e teste. O otimizador é um parâmetro que representa uma função de ativação, que será implementando no nosso modelo. Dentro do pacote *keras.optimizers* encontramos as funções que podem ser implementadas dentro do seu conjunto de classes definidas. O parâmetro perda recebe uma

```
[10] #definição do modelo
     model = tf.keras.models.Sequential()
     model.add(tf.keras.layers.Dense(units=32, activation='relu', input_shape=(784, )))
     model.add(tf.keras.layers.Dropout(0.4))
     model.add(tf.keras.layers.Dense(units=64, activation='relu'))
     model.add(tf.keras.layers.Dropout(0.6))
     model.add(tf.keras.layers.Dense(128, activation='relu'))
     model.add(tf.keras.layers.Dense(units=10, activation='softmax'))
```

Figura 6.8. Implementação do modelo de rede do TF.

função do tipo *tf.keras.losses* que visa calcular os erros entre classificações verdadeiras e previstas. Já o parâmetro contento as métricas é utilizado para avaliar o modelo durante o treinamento e o teste, e pode ser dos seguintes tipos: uma sequência de caracteres (nome de uma função interna), uma função ou uma instância *tf.keras.metrics.Metric*.

O treinamento da rede recebe como parâmetros de entrada o conjunto de treinamento (*X_treino*), com suas classes (*Y_treino*) e o número de épocas (*epochs*) para treinar o modelo, como ilustrado na Figura 6.9). Na Figura 6.9 também é mostrado o início do processo de treinamento com a contagem de épocas, tempo, valor de perda e acurácia.

Compilando o Modelo

```
[ ] model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['sparse_categorical_accuracy'])
```

Treinamento

```
model.fit(X_treino, Y_treino, epochs=10)

Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.9707 - sparse_categorical_accuracy: 0.6313
Epoch 2/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7188 - sparse_categorical_accuracy: 0.7365
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6664 - sparse_categorical_accuracy: 0.7596
```

Figura 6.9. Compilação e treinamento do modelo de CNN.

Finalizado o treinamento da CNN, é possível avaliar a acurácia passando os dados de teste. A função *evaluate* recebe como parâmetro o conjunto de teste (*X_Teste*) e o conjunto de classes de teste (*Y_Teste*), e retorna uma tupla contendo o valor de perda (*error score*) e a acurácia para o conjunto fornecido. Na Figura 6.10 é possível ver que a CNN do exemplo obteve uma acurácia de 81%.

```
Teste_perda, Test_acuracia = model.evaluate(X_teste, Y_teste)

313/313 [==============================] - 0s 1ms/step - loss: 0.5603 - sparse_categorical_accuracy: 0.8125
```

```
[8] print("Acurácia para os dados de teste: {}".format(Test_acuracia))

Acurácia para os dados de teste: 0.8125
```

Figura 6.10. Avaliação da CNN.

6.3. Estimando a Pose Humana em Tempo Real

6.3.1. *PoseNet*

O *PoseNet*[9] é um modelo de VC para a estimativa de pose/postura humana a partir de uma imagem ou vídeo. Ele é um dos modelos disponibilizados pela biblioteca *TensorFlow.js*. Sua tarefa não é apresentar ou identificar quem está presente na imagem, mas sim determinar onde está localizado 17 pontos do corpo humano.

O *PoseNet* compreende duas opções de configuração que podem ser usadas: uma para a identificação de pose para uma única pessoa (i.e., *single pose*) e e outra para múltiplas pessoas (i.e., *multiple poses*). Ambas buscam identificar poses através de imagem ou vídeo. A imagem/vídeo é passada para um modelo de CNN já treinado e, ao final do seu processamento, retorna um objeto JSON do tipo *Pose* para uma pessoa, ou um conjunto de *Poses*, para a configuração de múltiplas pessoas.

Um objeto *Pose* compreende os seguintes dados: um valor de acurácia geral (i.e., um *score* que descreve o quanto o modelo tem confiança dos achados) e os pontos-chave (i.e., os *keypoints*). Um *keypoint* é composto de: coordenadas do tipo *position*, acurácia (i.e, *score*) para o ponto identificado, e a descrição (*part*) do ponto localizado. Existem 17 *keypoints* que o algoritmo é capaz de identificar, são eles: nariz, olho esquerdo, olho direito, orelha esquerda, orelha direita, ombro esquerdo, ombro direito, cotovelo esquerdo, cotovelo direito, pulso esquerdo, pulso direito, quadril esquerdo, quadril direito, joelho esquerdo, joelho direito, tornozelo esquerdo, e tornozelo direito.

Inicialmente, é necessário entender os parâmetros de configuração do modelo do *PoseNet*, e como utilizá-los. Ao iniciar um objeto do tipo *PoseNet*, é necessário definir a arquitetura (*architecture*), resolução/passo de saída (*outputStride*), resolução de entrada (*inputResolution*), multiplicador (*multiplier*), quantidade de bytes (*quantBytes*) e *modelUrl*. A seguir estes atributos são descritos.

- Arquitetura: as arquiteturas presentes são *MobileNetV1* e *ResNet*. Ao instanciarmos um modelo com *MobileNetV1*, estaremos usando um modelo mais leve, rápido e, consequentemente, com acurácia menor, pois foi desenvolvido para ser executado em dispositivos móveis. A *ResNet* possui uma acurácia maior e, portanto, mais lenta. Ambas cumprem o seu papel, a escolha delas depende do poder de processamento do dispositivo onde será executado;

- Resolução/Passo de saída: este atributo pode receber os valores 8, 16 e 32. Para a arquitetura do tipo *ResNet*, os conjuntos disponíveis são 16 e 32. Para esse atributo, quanto menor o valor, maior será a resolução de saída e mais preciso será o modelo. Porém, a velocidade de processamento é comprometida;

- Resolução de entrada: esse atributo especifica o tamanho em que a imagem/vídeo é redimensionada antes de ser inserida no modelo *PoseNet*. A resolução mínima e padrão do modelo é de 257 pixels. Quanto maior o valor, mais precisos serão os resultados, porém se torna mais lento, por consumir mais tempo de processamento.

[9] `https://github.com/tensorflow/tfjs-models/tree/master/posenet`

Ao contrário, quando passado um valor menor como parâmetro, aumenta-se a velocidade e se obtém uma menor precisão;

- Multiplicador: atributo usado apenas pela arquitetura *MobileNetV1*. É um valor numérico que pode ser 1.0, 0.75 ou 0.5. Quanto maior o valor, maior o tamanho das camadas de convolução da CNN, e mais preciso é o modelo, se tornando mais lento;

- *quantBytes*: este argumento controla os bytes usados para quantização dos pesos da CNN. As opções disponíveis são 4, 2 e 1 bytes por quantização;

- *modelUrl*: um parâmetro opcional que pode conter um endereço de URL personalizado do modelo. Ele é útil para o desenvolvimento local ou para países que não têm acesso ao modelo hospedado no *Google Cloud Platform*.

Com o entendimento geral do modelo *PoseNet*, podemos nos aprofundar em utilizá-lo. Desenvolvemos aqui um exemplo para estimar a pose de uma única pessoa (i.e., *single pose*) a partir de uma *webcam* em uma página HTML. Ao passar os dados capturados da *webcam* para o modelo *PoseNet*, veremos como resultado um objeto com os pontos-chave retornados.

Inicialmente criamos uma página HTML chamada *index.html* com uma estrutura padrão HTML5 e importamos os módulos *@tensorflow/tfjs*, *@tensorflow-models/posenet* e *camera.js* por meio da *tag script*, como ilustrado na Figura 6.11. Como pode ser visto, o *PoseNet* é um arquivo *JavaScript* contendo as funções que manipulam o vídeo e a implementação do modelo. Este exemplo completo pode ser encontrado em: `https://github.com/RenanFialho-Dev/ercempi_2020`

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs"></script>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/posenet"></script>
<script src="camera.js"></script>
```

Figura 6.11. Importação dos pacotes do TF, *PoseNet* e *camera.js*.

Dentro do arquivo *camera.js*, criamos a função construtora *Camera*. No construtor, criamos um objeto *videoTag* através do acesso ao elemento vídeo presente no HTML, e outro objeto *videoConfig* para guardar as configurações de vídeo. Como boa prática, as funções e métodos que manipulam nossa classe não estão presentes no construtor. Para evitarmos consumo de memória desnecessário, pois quando um objeto é criado, é alocado memória para ele. Ao invés disso, utilizamos o recurso de *prototype*, que é um objeto associado a uma função construtora. Colocando os métodos neste objeto, todas as instâncias usarão as mesmas cópias de cada método.

No nosso *prototype*, implementamos no objeto câmera duas propriedades e uma *promise*. A propriedade *start* contém uma função anônima que faz acesso ao método *MediaDevices.getUserMedia* solicitando ao usuário permissão para usar uma entrada de mídia. No nosso caso, a *webcam* do dispositivo. Com permissão concedida pelo usuário, recebemos um *mediaStream* contendo o vídeo que é repassado a propriedade *srcObject*

da *tag* video. A Figura 6.12 apresenta a implementação da propriedade *start*, usado para inicializar a câmera no navegador.

```
Camera.prototype = {
    start: function () {
        this.ligado = true

        navigator.mediaDevices.getUserMedia(this.videoConfig).then(function (mediaStream) {
            this.videoTag = document.querySelector('video');
            this.videoTag.srcObject = mediaStream;
        }).catch(function (err) {
            console.log("Não foi possível iniciar o vídeo" + err);
        })
    },
```

Figura 6.12. Método *start* para inicializar a câmera no navegador.

A propriedade *stop* também é executada por uma função anônima. Por ela obtemos o fluxo do elemento de vídeo de sua propriedade *srcObject*, acessando a variável *videoTag*. Em seguida, a lista de faixas do fluxo é obtida chamando seu método *getTracks()*. A partir daí, tudo o que resta a fazer é percorrer a lista de trilhas usando o método *forEach()* e chamando o método *stop()* de cada trilha. Por fim, *srcObject* é definido como nulo para interromper o link para o objeto *mediaStream* para que ele possa ser liberado, como ilustrado na Figura 6.13.

```
    stop: function () {
        this.ligado = false
        const stream = this.videoTag.srcObject
        const tracks = stream.getTracks();

        tracks.forEach(function (track) {
            track.stop();
        });

        this.videoTag.srcObject = null;
    },
```

Figura 6.13. Método *stop* para interromper o fluxo do elemento vídeo.

A função *estimateSinglePoseOnVideo* implementa o modelo do *PoseNet* e retorna um objeto *Pose*. Criamos uma variável *configNet*, que é instanciada com as configurações do modelo, a qual pode ser vista na Figura 6.14. Uma vez que a nossa variável está pronta, acessamos a função *estimateSinglePose* passando os parâmetros de *input*, *flipHorizontal* e *decodingMethod*. A entrada (*input*) é o vídeo que estamos captando, o espelhamento (*flipHorizontal*) recebe um valor verdadeiro ou falso (quando verdadeiro, replica o efeito de movimento espelho) e, por último, o método de identificação que usaremos, o *single-person* (Figura 6.14).

Dentro do corpo (*body*) da nossa página, inserimos as *tags* de vídeo e *canvas*. A *tag* vídeo é responsável por exibir o conteúdo captado pela câmera do dispositivo e o *canvas* estará exibindo o mesmo conteúdo que o vídeo. Através dele acrescentamos marcadores dos pontos-chave encontrados pelo modelo do *PoseNet*. Dentro da *tag script*,

```
async estimateSinglePoseOnVideo () {
    if (this.ligado == true){
        this.configNet = await posenet.load({
            architecture: 'MobileNetV1',
            outputStride: 16,
            inputResolution: { width: 300, height: 300 },
            multiplier: 0.75
        });

        await this.configNet.estimateSinglePose(this.videoTag, { flipHorizontal: false,
        decodingMethod: 'single-person' }).then(result => {
            this.resultPoses = result;
        }).catch((err) => {
            console.log('Falha ao analisar', err);
        });

        this.configNet.dispose();
    }
    return this.resultPoses;
}
```

Figura 6.14. Função assíncrona *estimateSinglePoseOnVideo*.

instanciamos 3 variáveis: *canvas* e *vídeo*, que acessam a interface de programação dos elementos que possuem seus nomes, e a variável *ctx* recebe o contexto de desenho da *tag canvas*. Para realizar o acesso à câmera, criamos um objeto *camera* e realizamos a chamada da função *start*.

Para passarmos os dados capturados pela câmera para o *PoseNet*, acessamos o evento *onprogress* da *tag* vídeo, verificamos se o vídeo não está parado ou finalizado, e então fazemos a chamada da função *estimateSinglePoseOnVideo* do objeto *camera* anteriormente instanciado. Obtemos como retorno o objeto *Pose* contendo as informações dos pontos-chave localizados. Em seguida, passamos o objeto *Pose* para a função *poseDetectionFramePoints()*, que percorrerá o *array* de *keypoints* e passará as coordenadas (x,y) para a função *drawKeypoints()* (Figura 6.15), que realizará o desenho dos pontos na tela.

Para sincronizar a imagem captada pela câmera e a representação do *canvas*, adicionamos um evento *camera.addEventListener* ao vídeo enquanto estiver executando. O contexto do *canvas* executa a função *drawImage()*, que é atualizada frequentemente. E como resultado do nosso exemplo temos o vídeo captado pela câmera e uma réplica do vídeo com marcações de pontos identificados pelo modelo de CNN, como visto na Figura 6.16.

6.3.2. *MediaPipe*

O *MediaPipe*[10] é um *framework open-source* para o desenvolvimento de soluções de ML, desenvolvido e mantido pela *Google*. O *MediaPipe* provê modelos treinados que podem ser reutilizados de forma simples, pois visa ajudar pesquisadores e desenvolvedores a criarem suas próprias soluções. Ele disponibiliza modelos que podem ser implementados em diversas plataformas, tais como *Android*, *iOS*, *Windows*, *Linux* e Web.

A arquitetura do *framework* adota os seguintes conceitos [Lugaresi et al. 2019]: grafos (`Graph`), nós (`Calculators`), pacotes (`Packets`) e fluxos (`Streams`). A

[10] `https://github.com/google/mediapipe`

```
function drawKeypoints(ctx, x, y, r, scale = 1) {
    if (x !== 0 && y !== 0) {
        ctx.beginPath();
        ctx.arc(x, y, r, 0, 2 * Math.PI);
        ctx.fillStyle = '#4DB680';
        ctx.fill();;
    }
}

function poseDetectionFramePoints(pose) {
    this.setValuesOnInput(pose);
    pose.keypoints.forEach(value => {
        if (value.score >= 0.75) {
            drawKeypoints(ctx, value.position.x, value.position.y, 5);
        }
    });
}

function getProgressVideo() {
    video.onprogress = () => {
        if (!video.paused && !video.ended) {
            camera.estimateSinglePoseOnVideo().then(resultPoses => {
                poseDetectionFramePoints(resultPoses)
            });
        }
    }
}
```

Figura 6.15. Funções *getProgressVideo()*, *poseDetectionFramePoints()* e *drawKeypoints()*.

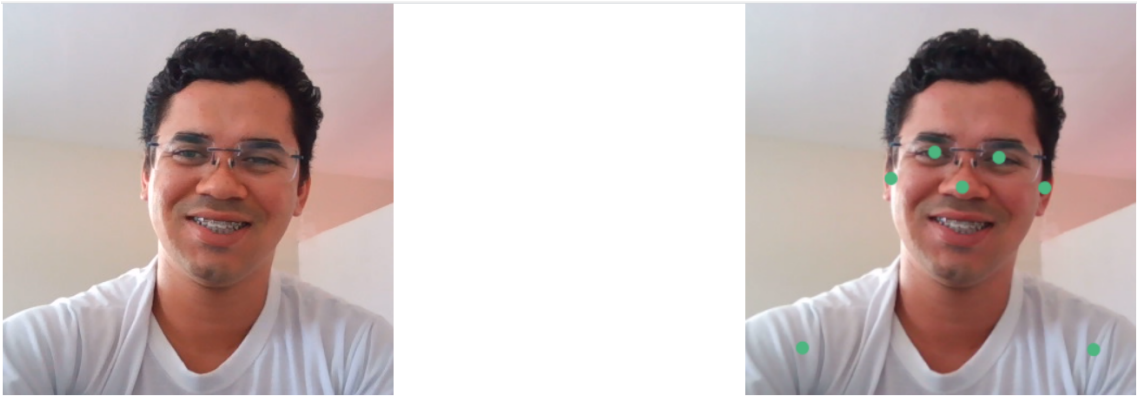

Figura 6.16. Estimativa do *PoseNet* em tempo real.

arquitetura é organizada em uma pipeline de um grafo de nós. No grafo, os nós são conectados por fluxos de dados. Cada fluxo representa uma série temporal de pacotes de dados. Nós e fluxos definem o grafo de fluxo de dados. Os pacotes que fluem pelo grafo são agrupados por seus *timestamps* dentro da série temporal (i.e., eles são as unidades básicas de fluxos de dados). A pipeline é flexível, permitindo inserir ou substituir qualquer nó do grafo, além de customizá-los. Os grafos podem ter qualquer número de entradas e saídas, podendo ramificar ou mesclar os fluxos de dados. Os dados entre os nós fluem de forma bidirecional. Nos nós, é onde boa parte do processamento de um grafo ocorre. Sua função é consumir pacotes e sua interface é responsável por definir o número de portas de entrada e saída para os fluxos, as quais são identificadas por índices.

Dentre os diversos modelos disponíveis no *MediaPipe*[11], existem soluções para a detecção de face (*Face Detection*), malha de rosto (*Face Mesh*), mãos (*Hands Detection*), segmentação de cabelo (*Hair Segmentation*), detecção de objetos (*Object Detection*), dentre outros. Em particular, o *MediaPipe Hands* (MH)[12] é uma solução de rastreamento das mãos e dedos de seres humanos. Nele, foram usadas técnicas de ML para inferir 21 pontos-chave da mão a partir de um único quadro. O método utilizado pelo *MediaPipe* consegue um bom desempenho em tempo real funcionando em *smartphones*. Ele utiliza dois modelos de ML: o primeiro é aplicado à imagem completa e delimita uma caixa ao redor da mão, portanto é utilizado para detectar a mão; já o segundo modelo trabalha somente na região delimitada previamente para identificar pontos-chave.

Nesta seção daremos ênfase ao modelo MH. A seguir é mostrado um exemplo prático de construção de uma página Web para a detecção de uma das mãos. Este exemplo pode ser acessado em: `https://github.com/RenanFialho-Dev/ercempi_2020`. Por meio de uma *webcam* acessada por uma página HTML, os dados captados pela *webcam* são passados para o modelo *HandPose*. Como resultado, tem-se 21 pontos-chave. Para esse exemplo, foi criado uma página HTML5 e importadas as bibliotecas *TensorFlow* e *HandPose* por meio da tag *script* (Figura 6.17).

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs"></script>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/handpose"></script>
<script src="camera.js"></script>
```

Figura 6.17. Importando as bibliotecas tensorflow, handpose e camera

No *prototype*, implementamos no objeto câmera duas propriedades (*start* e *stop*) e uma função assíncrona para estimar a pose da mão chamada *estimateHandsPoseOnVideo*. Para isso, utilize os códigos já explicados anteriormente para as Figuras 6.12 e 6.13.

A função *estimateHandsPoseOnVideo* é responsável por estimar a pose da mão. A variável *modelo* foi criada pare receber o objeto retornado por essa função, o qual contém as configurações do modelo do *MediaPipe*. Após a variável *modelo* obter o objeto, podemos usar a função *estimateHands*, passando os parâmetros *input* e *flipHorizontal*. O *input* é responsável por fazer a captura do video e *flipHorizontal* recebe um valor booleano responsável por espelhar o video no sentindo horizontal (Figura 6.18).

No *body* da página, foram inseridas as *tags video* e *canvas*. A *tag video* exibe o conteúdo capturado pela câmera do dispositivo e o *canvas* exibe os *frames* do vídeo com as marcações dos pontos-chave identificados. Na *tag script*, foram criadas 3 constates: *canvas*, *video* e *ctx*. As duas primeiras são responsáveis por capturar e renderizar. A variável *ctx* é utilizada para desenhar na *tag canvas*. Os índices dos dedos são guardados no objeto *fingerLookupIndices*, contendo 25 posições, onde esses pontos são armazenados em *arrays* representando cada dedo (o índice 0 é responsável pela palma da mão), como visto na Figura 6.19.

[11] `https://google.github.io/mediapipe/solutions/solutions.html`
[12] `https://google.github.io/mediapipe/solutions/hands`

```
async estimateHandsPoseOnVideo() {
    const model = await handpose.load();
    await model.estimateHands(this.videoTag, { flipHorizontal: false }).then(result => {
        this.resultPoses = result;
    }).catch((err) => {
        console.log('Falha ao analisar', err);
    });
    return this.resultPoses;
}
```

Figura 6.18. Função *estimateHandsPoseOnVideo*.

```
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext('2d');
const video = document.getElementById("video");
let fingerLookupIndices = {
  thumb: [0, 1, 2, 3, 4],
  indexFinger: [0, 5, 6, 7, 8],
  middleFinger: [0, 9, 10, 11, 12],
  ringFinger: [0, 13, 14, 15, 16],
  pinky: [0, 17, 18, 19, 20]
};
```

Figura 6.19. Declaração de variáveis para uso do modelo.

O evento *onprogress* é responsável por capturar os dados da câmera e passar para o *HandPose*. A função *estimateHandsPoseOnVideo* é estanciada pelo objeto *camera* para obter os pontos-chave. Em seguida é verificada se a mão aparece na câmera. Caso verdadeiro, é passado o *array* dos pontos para a função *drawKeypoints*, responsável por desenhar os pontos-chave (Figura 6.20).

A função *drawKeypoints(keypoints)* (Figura 6.21) executa um *loop* para percorrer o *array* dos pontos-chave, passando as coordenadas (x, y) para a função *drawPoint* (Figura 6.22), a qual desenha um único ponto por vez. A medida que o *loop* é executado, os pontos são desenhados na tela. Em seguida é executado um outro *loop* para desenhar as linhas entre os pontos. A constante *points* recebe o mesmo *array* de pontos-chave usados para desenhar a linha e é passada para a função *drawPath*, responsável por desenhar uma única linha (Figura 6.23). O resultado do exemplo é um vídeo capturado pela câmera e exibido na *tag video* e os *frames* sendo exibidos no *canvas* com a marcação dos

```
function getProgressVideo() {
  video.onprogress = () => {
    camera.estimateHandsPoseOnVideo().then(resultPoses => {
      if (resultPoses.length > 0) {
        drawKeypoints(resultPoses[0].landmarks)
      }
    });
  }
}
```

Figura 6.20. Função *getProgressVideo()*.

pontos-chave identificados pelo *HandPose*, como visto na Figura 6.24.

```
function drawKeypoints(keypoints) {
  const keypointsArray = keypoints;

  for (let i = 0; i < keypointsArray.length; i++) {
    const y = keypointsArray[i][0];
    const x = keypointsArray[i][1];
    drawPoint(ctx, x - 2, y - 2, 3);
  }
  // mapeamento das linha
  const fingers = Object.keys(fingerLookupIndices);
  for (let i = 0; i < fingers.length; i++) {
    const finger = fingers[i];
    const points = fingerLookupIndices[finger].map(idx => keypoints[idx]);
    drawPath(ctx, points, false);
  }
}
```

Figura 6.21. Função para mapear pontos-chave.

6.4. Aplicações na Saúde

Algoritmos de ML têm sido aplicados na saúde como alternativa para melhorar a acurácia de métodos de triagem, diagnósticos e terapêuticos, usados no monitoramento

```
function drawPoint(ctx, y, x, r) {
  ctx.beginPath();
  ctx.arc(x, y, r, 0, 2 * Math.PI);
  ctx.fillStyle = '#FFFFFF';
  ctx.fill();
}
```

Figura 6.22. Função responsável por desenhar um único ponto.

```
function drawPath(ctx, points, closePath) {
  const region = new Path2D();
  region.moveTo(points[0][0], points[0][1]);
  for (let i = 1; i < points.length; i++) {
    const point = points[i];
    region.lineTo(point[0], point[1]);
  }

  if (closePath) {
    region.closePath();
  }
  ctx.stroke(region);
}
```

Figura 6.23. Função responsável por desenhar uma única linha.

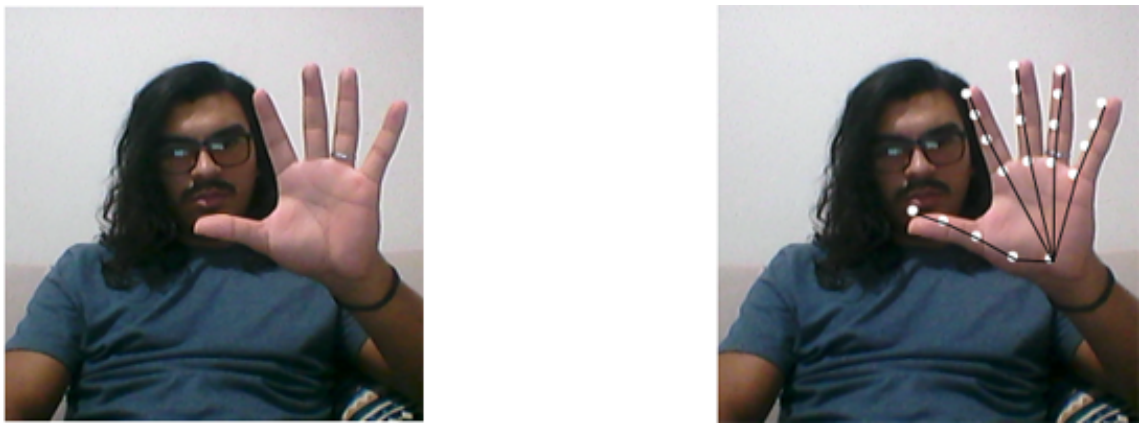

Figura 6.24. Mapeamento da mão em tempo real usando o *HandPose*.

e recuperação da saúde humana. A seguir são apresentados e discutidos exemplos de soluções de ML para estimar em tempo real a pose humana em aplicações de saúde.

6.4.1. Avaliação Postural

No contexto da avaliação dinâmica da posição do corpo humano, algoritmos de ML possibilitam detectar articulações do corpo humano e a partir delas estimar a posição e movimento humano [Voulodimos et al. 2018]. Esse recurso já foi aplicado, por exemplo, para estimar o posicionamento de membros de bebês prematuros em unidades de terapia intensiva e, a partir disso, avaliar o estado de saúde deles [Moccia et al. 2019]. Além disso, ele já foi testado no monitoramento semi-automatizado da posição do tronco superior de pacientes adultos em leitos clínicos [Chen et al. 2018] e em outras situações que podem ser de interesse clínico, por exemplo, o monitoramento das posições do corpo durante o sono [Liu et al. 2019].

Como indicado acima, a estimativa de pose humana baseada em algoritmos de aprendizado de máquina pode auxiliar no monitoramento de situações relacionadas à saúde humana, mas sua aplicação ainda pode ser ampliada para otimizar outros métodos de avaliação em saúde já existentes. Um exemplo que pode ser citado é sua aplicação ao processo de avaliação postural do corpo humano em pé. No contexto destacado aqui, enfatizamos não apenas o rastreamento dos movimentos do corpo e seus variados segmentos, mas a busca por desalinhamentos da coluna vertebral, considerando possíveis inclinações que possam ser sugestivas para a ocorrência de desvios nos planos sagital e coronal, como escoliose, hiperlordose e hipercifose.

A identificação de articulações corporais é um dos passos iniciais envolvidos no processo de avaliação postural realizado por profissionais de saúde. Os métodos tradicionalmente usados para avaliação postural ainda exigem a marcação manual usando marcadores reflexivos posicionados nas articulações. Nem mesmo aplicações para *smartphone* utilizadas com esse propósito se mostraram capazes de realizar essa marcação de forma automatizada. Nesse sentido, uma vez que algoritmos de ML já se mostram capazes de fazer essa identificação de modo automático, então podem ser aplicados ao processo de avaliação da postura, otimizando a marcação de pontos anatômicos de referência [Moreira et al. 2020]. A integração entre algoritmos de ML e recursos disponíveis nos smartphones permite o desenvolvimento de aplicações, como o que vem sendo feito por pesquisadores do Laboratório de Neuroinovação Tecnológica & Mapeamento Cerebral (NitLab) da Universidade Federal do Delta do Parnaíba (UFDPar): um protótipo de aplicação móvel para avaliação postural que permite identificar e conectar pontos anatômicos através de linhas e, assim, determinar o tamanho dos segmentos corporais que influencia na ocorrência de desvios posturais, como ilustrado na Figura 6.25, em que é utilizado o *PoseNet*.

6.4.2. Avaliação Goniométrica

Algoritmos de ML que permitem rastrear o posicionamento do corpo humano baseado na identificação das articulações corporais também podem ser aplicados como modelos inteligentes para avaliação goniométrica, ou seja, para avaliar a amplitude dos movimentos. Amplitude de movimento refere-se à quantidade de movimento realizado

Figura 6.25. Exemplo de aplicação de ML para reconhecimento e demarcação anatômica.

por um determinado segmento do corpo, por exemplo, o cotovelo, quadril e joelho, em torno de seu eixo articular. Essa amplitude pode variar em graus e pode ser afetada por condições patológicas que afetam as próprias articulações ou tecidos adjacentes como músculos, tendões, fáscias e tecido neural.

Nesse contexto, não basta movimentar-se, é necessário que os movimentos do corpo ocorram dentro de um amplitude funcional, ou seja, com quantidade suficiente para que o ser humano consiga realizar suas atividades do dia a dia. Considerando então a importância da amplitude de movimento para o desempenho funcional do ser humano, também faz-se necessário empregar métodos de avaliação que permitam quantificar esses movimentos.

Um método clínico tradicionalmente usado para esse fim é a inspeção visual, por meio da qual um avaliador solicita ao paciente que eleve o segmento avaliado no sentido do movimento de interesse (flexão, extensão, abdução adução ou rotação), e analisa se o segmento se move ou não dentro de uma amplitude funcional. Nesse modelo avaliativo, não é possível quantificar os graus de movimento, mas como alternativa para essa limitação, foi desenvolvido o goniômetro universal. Trata-se de uma régua com um eixo de 360 graus que é posicionando no centro da articulação e, a medida que o segmento se move, uma das hastes dessa régua é levada pelo examinador, no sentido do movimento realizado [Gogia et al. 1987, Carvalho et al. 2012].

Assim como na avaliação postural, para a avaliação goniométrica é necessário palpar a articulação e identificar o segmento móvel e usá-los como ponto de referência para o

posicionamento do eixo e das hastes fixa e móvel do goniômetro. Essa avaliação manual pode ser comprometida em função do deslocamento do goniômetro durante o movimento angular do segmento corporal avaliado. Nesse contexto, algoritmos de ML podem ser usado em modelos de avaliação goniométrica automatizada para otimizar esse processo de identificação articular, reduzindo riscos de viés produzidos durante a avaliação manual [Moreira et al. 2020].

6.4.3. Reabilitação Neurofuncional

Os acidentes vasculares cerebrais (AVC) são a terceira principal causa de incapacidade no mundo [Johnson et al. 2016, Raffin and Hummel 2018], associada a baixa qualidade de vida. Existe um número considerável de indivíduos com problemas neurológicos que vivem com alguma deficiência [Baumann et al. 2011] e destes, 80% apresentam deficit motor, provocado, por exemplo por uma hemiparesia que reduz a capacidade cinético-funcional do paciente [Scherbakov et al. 2013]. Apenas metade dos pacientes que sobrevivem ao AVC alcançam a recuperação funcional de seu membro superior parético [Lee et al. 2012], o que afeta seriamente o autocuidado e a participação na sociedade.

A recuperação da função motora, especialmente a função do membro superior, é um é um fator-chave para determinar o nível de independência funcional após AVC [Veerbeek et al. 2011]. Recuperar ou melhorar a capacidade de movimentar segmentos corporais ativamente facilita o desempenho das atividades de vida diária do paciente. Considerando que a maioria dessas atividades envolve o uso dos membros superiores, é crucial desenvolver mecanismos terapêuticos que facilitem sua recuperação e consequente uso funcional, mesmo após o AVC [Choo et al. 2015]

Nesse propósito, profissionais da saúde especializados na reabilitação motora pós-AVC têm utilizado diversas técnicas a fim de tentar acelerar a recuperação do paciente. Dentre estas, a gameterapia [Domínguez-Téllez et al. 2020], um método que utiliza ambientes de videogames (i.e., jogos sérios) no auxílio à reabilitação funcional dos pacientes. Alguns desses jogos sérios apresentam ambientes de gamificação [Ferreira et al. 2014] voltados especialmente para a reabilitação dos membros superiores de pacientes com sequelas motoras após AVC [Fathima et al. 2018]. Entretanto, as tecnologias computacionais associadas a gameterapia se baseiam no uso de sensores, dispositivos vestíveis, braços robóticos ou plataformas proprietárias de videogames, como *Wii*, *Kinect*, *Xbox*, as quais apresentam um elevado custo de aquisição, tornando o uso não popularizado como tecnologias sociais.

Aplicações baseadas em jogos sérios e realidade virtual tornam o programa de treinamento menos monótono, favorecendo o engajamento do paciente [Ayed et al. 2019, Chen et al. 2019]. Baseado nisso, o uso da gameterapia para a reabilitação motora deve ser incentivado, tornando-se rotina não apenas nas clínicas de reabilitação, mas também no ambiente domiciliar. Para que isso se torne viável é necessário oferecer aplicações mais acessíveis aos profissionais, bem como ao próprio paciente. Neste contexto, o TF se apresenta como uma solução robusta que permite aos desenvolvedores de software, treinar e implementar modelos de ML que podem ser usados na gameterapia. Entre as vantagens de se trabalhar com esta biblioteca, é a possibilidade de integração em várias plataformas e ambientes web. Em particular, o *PoseNet* é aplicável a diversas situações que envolvem

posicionamento e rastreamento da posição do corpo. Ao integrar o *PoseNet* às tecnologias utilizadas por navegadores web, torna-se possível implementar um ambiente favorável a gameterapia.

Um protótipo de uma plataforma web tem sido desenvolvido por pesquisadores do NitLab da UFDPar, o qual usa o TF e o *PoseNet* para o controle dos movimentos do paciente em uma interface de jogo sério, como ilustrado na figura 6.26. O contexto do jogo envolve o livre movimento dos membros superiores, aqui representados pelo elemento visual alvo. A dinâmica no jogo visa o paciente estourar os balões exibidos na tela levando o alvo ao encontro do balão. Ao iniciar o jogo, a câmera captura a imagem do usuário e a partir da movimentação do braço, a posição do alvo no cenário do jogo é alterada.

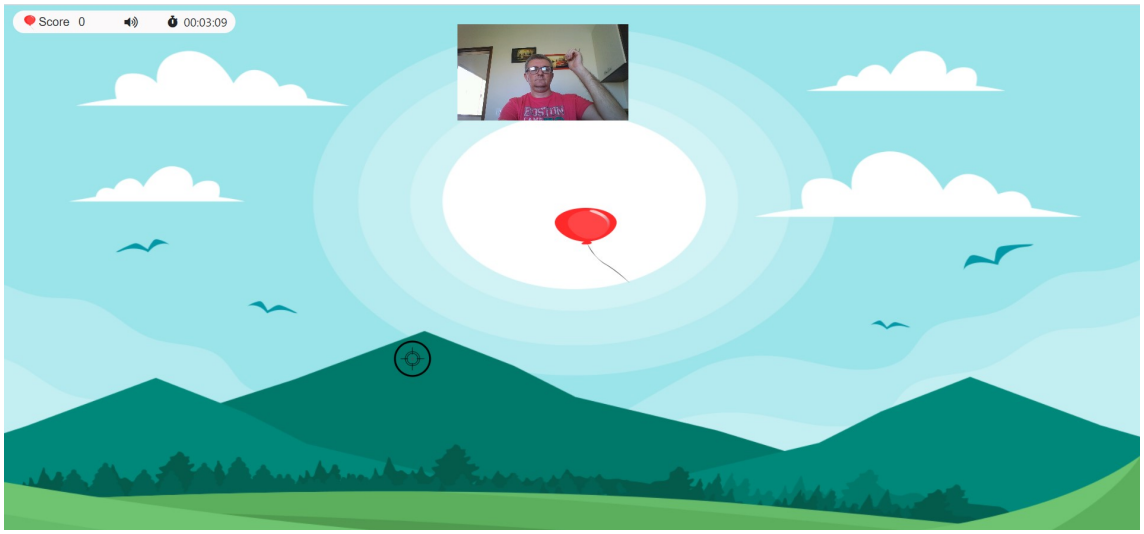

Figura 6.26. Interface de um Jogo Sério para Reabilitação Motora - Gameterapia.

Para o protótipo inicial, estamos usando apenas a posição 9 no *array* de pontos-chave do *PoseNet*, que representa a posição *leftWrist* (pulso esquerdo). Na estrutura do alvo e do balão, no cenário do jogo, existe um elemento colisor, que cria um retângulo os envolvendo. Esse colisor é responsável por retornar uma ação na dinâmica do jogo (um estouro do balão) quando os dois objetos entram em contato.

6.5. Considerações Finais

Este capítulo apresentou a biblioteca *TensorFlow* e suas aplicações no reconhecimento em tempo real de estruturas e movimento do corpo humano, contribuindo para as pesquisas na área de informática em saúde. Mais especificamente, os modelos de ML para visão computacional *PoseNet* e *MediaPipe* foram apresentados, os quais foram desenvolvidos para a estimativa de pose do corpo humano a partir de uma imagem ou vídeo. Para exemplificar soluções de ML para estimar em tempo real a pose humana em aplicações de saúde, as quais fazem uso das tecnologias mostradas, este capítulo também apresentou algumas pesquisas em desenvolvimento no NitLab da UFDPar. Inicialmente, uma solução para auxiliar no processo de avaliação postural e goniométrica. Outra, um jogo sério em desenvolvimento que deverá ser utilizado no processo de reabilitação neurofuncional

de membros superiores para pacientes pós-AVC em ambientes domésticos e clínicas de reabilitação.

A tecnologia está presente em diversos setores da economia, com inovações e soluções usadas no cotidiano da sociedade para aprimorar o modo de vida das pessoas. Como visto neste capítulo, a inteligência artificial aplicada à saúde está cada vez mais presente, com novos procedimentos e técnicas, seja na identificação e prevenção de doenças, no desenvolvimento de tratamentos para casos complexos, como no aumento da eficiência dos serviços de saúde. Estas inovações atingem todos os processos, desde o registro do paciente ao monitoramento de dados, dos testes laboratoriais aos aparelhos que possibilitam que o próprio paciente faça seu monitoramento.

Consentimento Informado

Todas as figuras com fotos de pessoas utilizadas neste capítulo foram autorizadas para fins de pesquisa.

Referências

[Abadi et al. 2016] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA. USENIX Association.

[Aneja et al. 2019] Aneja, S., Chang, E., and Omuro, A. (2019). Applications of artificial intelligence in neuro-oncology. *Current Opinion in Neurology*, 32:1.

[Ayed et al. 2019] Ayed, I., Ghazel, A., Jaume-i Capó, A., Moyà-Alcover, G., Varona, J., and Martínez-Bueso, P. (2019). Vision-based serious games and virtual reality systems for motor rehabilitation: A review geared toward a research methodology. *International journal of medical informatics*, 131:103909.

[Basheer and Hajmeer 2000] Basheer, I. A. and Hajmeer, M. (2000). Artificial neural networks: Fundamentals, computing, design, and application. *Journal of Microbiological Methods*, 43(1):3–31.

[Baumann et al. 2011] Baumann, M., Lurbe-Puerto, K., Alzahouri, K., and Aïach, P. (2011). Increased residual disability among poststroke survivors and the repercussions for the lives of informal caregivers. *Topics in stroke rehabilitation*, 18(2):162–171.

[Carneiro et al. 2018] Carneiro, T., Medeiros Da Nobrega, R. V., Nepomuceno, T., Bian, G.-B., De Albuquerque, V. H. C., and Filho, P. P. R. (2018). Performance Analysis of Google Colaboratory as a Tool for Accelerating Deep Learning Applications. *IEEE Access*, 6:61677–61685.

[Carvalho et al. 2012] Carvalho, R. M. F. d., Mazzer, N., and Barbieri, C. H. (2012). Analysis of the reliability and reproducibility of goniometry compared to hand photogrammetry. *Acta Ortopédica Brasileira*, 20:139–149.

[Chen et al. 2018] Chen, K., Gabriel, P., Alasfour, A., Gong, C., Doyle, W. K., Devinsky, O., Friedman, D., Dugan, P., Melloni, L., Thesen, T., Gonda, D., Sattar, S., Wang, S., and Gilja, V. (2018). Patient-specific pose estimation in clinical environments. *IEEE Journal of Translational Engineering in Health and Medicine*, 6:1–11.

[Chen et al. 2019] Chen, Y., Abel, K. T., Janecek, J. T., Chen, Y., Zheng, K., and Cramer, S. C. (2019). Home-based technologies for stroke rehabilitation: a systematic review. *International journal of medical informatics*, 123:11–22.

[Choo et al. 2015] Choo, P. L., Gallagher, H. L., Morris, J., Pomeroy, V. M., and Van Wijck, F. (2015). Correlations between arm motor behavior and brain function following bilateral arm training after stroke: a systematic review. *Brain and behavior*, 5(12):e00411.

[Domínguez-Téllez et al. 2020] Domínguez-Téllez, P., Moral-Muñoz, J. A., Salazar, A., Casado-Fernández, E., and Lucena-Antón, D. (2020). Game-based virtual reality interventions to improve upper limb motor function and quality of life after stroke: Systematic review and meta-analysis. *Games for Health Journal*, 9(1):1–10.

[Fathima et al. 2018] Fathima, S., Shankar, S., and Thajudeen, A. A. (2018). Activities of daily living rehab game play system with augmented reality based gamification therapy for automation of post stroke upper limb rehabilitation. *Journal of Computational and Theoretical Nanoscience*, 15(5):1445–1451.

[Ferreira et al. 2014] Ferreira, C., Guimarães, V., Santos, A., and Sousa, I. (2014). Gamification of stroke rehabilitation exercises using a smartphone. In *Proceedings of the 8th International Conference on Pervasive Computing Technologies for Healthcare*, pages 282–285.

[Forsyth and Ponce 2002] Forsyth, D. A. and Ponce, J. (2002). *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference.

[Gogia et al. 1987] Gogia, P. P., Braatz, J. H., Rose, S. J., and Norton, B. J. (1987). Reliability and validity of goniometric measurements at the knee. *Physical Therapy*, 67(2):192–195.

[Google 2019] Google (2019). Welcome To Colaboratory.

[Graaff and Marshall 2003] Graaff, V. D. and Marshall, K. (2003). *Anatomia humana*, volume 6.

[Johnson et al. 2016] Johnson, W., Onuma, O., Owolabi, M., and Sachdev, S. (2016). Stroke: a global response is needed. *Bulletin of the World Health Organization*, 94(9):634.

[Lee et al. 2012] Lee, M. M., Cho, H.-y., and Song, C. H. (2012). The mirror therapy program enhances upper-limb motor recovery and motor function in acute stroke patients. *American journal of physical medicine & rehabilitation*, 91(8):689–700.

[Liu et al. 2019] Liu, S., Yin, Y., and Ostadabbas, S. (2019). In-bed pose estimation: Deep learning with shallow dataset. *IEEE Journal of Translational Engineering in Health and Medicine*, 7.

[Lugaresi et al. 2019] Lugaresi, C., Tang, J., Nash, H., McClanahan, C., Uboweja, E., Hays, M., Zhang, F., Chang, C.-L., Yong, M. G., Lee, J., Chang, W.-T., Hua, W., Georg, M., and Grundmann, M. (2019). Mediapipe: A framework for building perception pipelines.

[Moccia et al. 2019] Moccia, S., Migliorelli, L., Carnielli, V., and Frontoni, E. (2019). Preterm infants' pose estimation with spatio-temporal features. *IEEE Transactions on Biomedical Engineering*, pages 1–1.

[Moreira et al. 2020] Moreira, R., Teles, A., Fialho, R., [dos Santos], T. C. P., Vasconcelos, S. S., [de Sá], I. C., Bastos, V. H., Silva, F., and Teixeira, S. (2020). Can human posture and range of motion be measured automatically by smart mobile applications? *Medical Hypotheses*, 142:109741.

[Raffin and Hummel 2018] Raffin, E. and Hummel, F. C. (2018). Restoring motor functions after stroke: multiple approaches and opportunities. *The Neuroscientist*, 24(4):400–416.

[Sabharwal and Kumar 2008] Sabharwal, S. and Kumar, A. (2008). Methods for assessing leg length discrepancy. *Clinical Orthopaedics and Related Research*, 466(12):2910–2922.

[Scherbakov et al. 2013] Scherbakov, N., Von Haehling, S., Anker, S. D., Dirnagl, U., and Doehner, W. (2013). Stroke induced sarcopenia: muscle wasting and disability after stroke. *International journal of cardiology*, 170(2):89–94.

[Veerbeek et al. 2011] Veerbeek, J. M., Kwakkel, G., van Wegen, E. E., Ket, J. C., and Heymans, M. W. (2011). Early prediction of outcome of activities of daily living after stroke: a systematic review. *Stroke*, 42(5):1482–1488.

[Voulodimos et al. 2018] Voulodimos, A., Doulamis, N., Doulamis, A., and Protopapadakis, E. (2018). Deep learning for computer vision: A brief review. *Computational Intelligence and Neuroscience*, 2018:1–13.