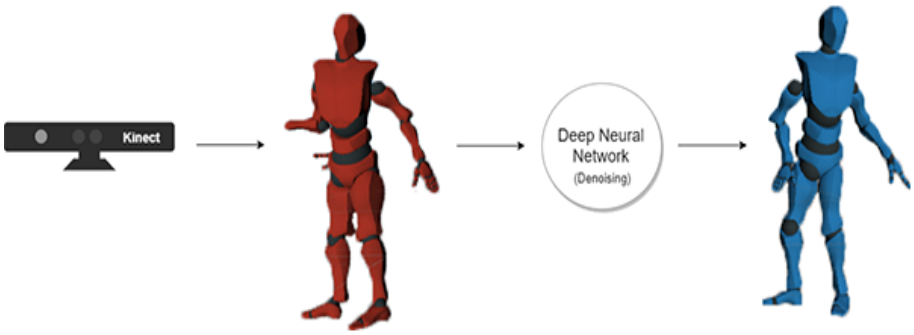
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores



Captura de movimento em tempo real para Realidade Mista

André Correia Gonçalves

Mestrado

Projecto Final para obtenção do Grau de Mestre
em Engenharia de Redes de Comunicação e Multimédia

Orientadores : Professor Doutor Rui Jesus
Professor Doutor Pedro Mendes Jorge

Júri:

Presidente: Professor Doutor Carlos Jorge de Sousa Gonçalves

Vogais: Professor Doutor Jacinto Carlos Peixoto do Nascimento
Professor Doutor Pedro Miguel Torres Mendes Jorge

7, 2020

Resumo

A captura de movimento é uma técnica fundamental na área dos videojogos e cinematografia para animar uma personagem virtual de forma realista e num curto espaço tempo. Através de um só sensor óptico é possível captar os movimentos do jogador para interagir com o mundo virtual, no entanto pode haver ruído durante a captura e partes do corpo ficarem ocluídas, reduzindo a experiência do utilizador.

Este projecto propõem uma solução para corrigir erros na captura de movimento do sensor Microsoft Kinect, através de redes neuronais profundas (*deep neural networks*), treinadas em um conjunto de poses disponibilizadas pela Carnegie Mellon Graphics (CMU) previamente processadas. É também implementado um filtro temporal que suaviza um movimento dado por um conjunto de poses devolvidas pela rede neuronal.

O sistema é implementado utilizando a linguagem de programação Python, em conjunto com a API Tensorflow para suportar as técnicas de aprendizagem automática e o motor de jogo Unity para visualizar e interagir com os esqueletos obtidos.

Os resultados foram avaliados através de um conjunto de métricas e um questionário com 12 participantes.

Palavras-chave: Captura de Movimento, Microsoft Kinect, Aprendizagem Automática, Redes Neuronais, Denoising, Realidade Mista.

Abstract

Motion capture is fundamental technique in video games development and film production to animate a virtual character based on actor's movements, creating more realistic animations in a short time. It is possible to detect the player's movements through one optical sensor to interact with the virtual world, however there may be some noise and certain parts of the body become occluded, decreasing the user experience.

This project present a solution to correct the motion capture errors from Microsoft Kinect sensor, through a deep neural network, trained with a pre-processed dataset of poses offered by Carnegie Mellon Graphics (CMU). A temporal filter is implemented to smooth the movement, given by a set of poses returned by the deep neural network.

This system is implemented in Python with the Tensorflow API which supports the machine learning techniques and the Unity game engine to visualize and interact with the obtained skeletons.

The results were evaluated with a set of metrics and a questionnaire with 12 participants.

Keywords: Motion Capture, Microsoft Kinect, Machine Learning, Neural Networks, Denoising, Mixed Reality.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

CMU Carnegie Mellon University. 2
ResNet *Residual Network*. 8
RNN *Recurrent neural network*. 14
SDK *Software Development Kit*. 19
ASF Acclaim Skeleton File. 22, 23
AMC Acclaim Motion Capture data. 22, 23

Glossário

gimbal lock Perda de um grau de liberdade no espaço tridimensional. Ocorre quando dois eixos entram numa configuração paralela, bloqueando o sistema de modo a que só consiga girar num espaço bidimensional. 27

jitter Pequenas agitações nos sinais que os torna instáveis. 31

tuning Processo de troca dos hiper-parâmetros da rede neuronal com o objectivo de encontrar o mínimo global. 41

1

Introdução

Na área do entretenimento, como videojogos e cinematográfica, os movimentos da personagem virtual são um factor importante para que o conteúdo seja apelativo. As personagens são animadas utilizando ferramentas de edição 3D, como o Blender [5] ou Autodesk Maya [4], através da alteração da pose do esqueleto ligado ao modelo. Existem técnicas de interpolação de *frames* que suavizam os movimentos do esqueleto e facilitam o processo de animação, no entanto é difícil replicar movimentos reais, como os do ser humano.

No caso dos videojogos que se aproximam da realidade, criar todas as animações das personagens manualmente requer muito esforço e tempo. Actualmente são utilizados sistemas que captam o movimento do actor e o transferem para a personagem virtual. Desta forma consegue-se gerar um movimento mais próximo da realidade, num curto espaço de tempo, independentemente da sua complexidade. Esta tecnologia introduz uma nova interacção com os videojogos a partir dos movimentos do jogador e sem a necessidade de controladores. Assim, a captura de movimento é um processo importante não só para gerar movimentos reais e específicos de forma a "dar vida" a personagens virtuais, como também permite uma maior imersão nas aplicações de realidade virtual/aumentada.

Existem dois principais sistemas de captura [11]: O sistema óptico [32], baseado em uma ou mais câmaras que captam a informação do esqueleto através de infravermelhos ou informação de um conjunto de marcadores bem posicionados no actor. Deste sistema destacam-se Microsoft Kinect [10], Intel RealSense [14]

e LeapMotion [9]. O sistema não-óptico consiste num conjunto de dispositivos, compostos por giroscópios, posicionados nas juntas do actor para captarem a posição e a orientação de cada uma. Deste sistema destacam-se Xsens [20], Rokoko [16] e Shadow [19].

Embora o sistema não-óptico ofereça melhor qualidade, tanto na precisão da captura dos movimentos bem como a mobilidade (não está limitado a uma certa área), as componentes de hardware podem ser dispendiosas. Ao contrário, os sistemas ópticos podem-se basear apenas numa câmara, no entanto ficam sujeitos a erros, que deverão ser corrigidos manualmente o que implica esforço e tempo.

Existe uma elevada procura por sistemas que façam automaticamente a correcção dos movimentos captados com o mínimo de hardware possível, por exemplo um só sensor Kinect, de forma a reduzir o custo do sistema.

Actualmente são usadas redes neuronais profundas (*deep neural network*) para alcançar os resultados do "estado da arte" em diversos problemas de computação gráfica, incluindo problemas de eliminação de ruído (*denoising*) [36]. Dois artigos que implementam este sistema são os de Daniel et al. [30] e [34] que segundo os autores, as redes conseguem detectar o ruído e prever a melhor pose do esqueleto bem como determinar se essa pose é possível para o ser humano.

Este trabalho descreve uma solução para corrigir erros de captura da pose do corpo humano provocados pelo sensor Kinect. Nomeadamente, deslocamentos das juntas e erros provocados por oclusões. Através de um estudo realizado com este sensor, concluiu-se que 16% das poses capturadas tem oclusões (apresentado no capítulo 5), surgindo a necessidade de criar um sistema que atenue este problema.

O trabalho desenvolvido segue a abordagem semelhante à de Daniel et al. [30] adaptada ao problema especifico. As redes neuronais exigem uma elevada quantidade de dados de treino, para tal utilizou-se o conjunto de movimentos capturados pela Carnegie Mellon University (CMU) [6], adaptado de forma a que a estrutura do esqueleto seja semelhante ao da Kinect.

Este documento está organizado da seguinte forma:

Capítulo 2- Apresenta um conjunto de trabalhos que propõem uma solução para a correcção de ruído na captura de movimento do corpo humano, através da utilização de filtros e técnicas de aprendizagem automática.

Capítulo 3- São apresentados os conceitos abrangidos ao longo do documento.

Capítulo 4- São descritos os dados que serão utilizados neste trabalho. Nomeadamente os dados da Microsoft Kinect e o conjunto de movimentos da CMU.

Capítulo 5- Apresenta a solução proposta, juntamente com as ferramentas para a sua implementação.

Capítulo 6- Descreve a implementação das várias componentes apresentadas no capítulo anterior, incluindo o desenvolvimento e treino da rede neuronal.

Capítulo 7- São apresentados e discutidos os resultados realizados através de um conjunto de métricas e uma avaliação perceptual para validar o modelo proposto.

Capítulo 8- Apresenta as conclusões e indica algumas direcções para trabalho futuro.

2

Trabalho Relacionado

As propostas para combater o ruído na captura de movimento baseiam-se em técnicas que introduzem dois tipos de prior [30]. O **prior temporal** explora o facto que as juntas devem respeitar as leis da física e não podem saltar instantaneamente para um nova posição. O artigo de Aristidou et al. [21] é um exemplo de uma proposta que se baseia neste prior, através da implementação de um filtro de Kalman. O **prior baseado em pose** (*pose-based*) afirma que o esqueleto só pode tomar poses que sejam possíveis para o ser humano. Os sistemas que codificam este prior utilizam técnicas de aprendizagem automática e dependem de uma grande quantidade de dados.

Actualmente são utilizadas redes neuronais profundas com o conceito de *denoising* para reduzir o ruído de um conjunto de dados, principalmente no âmbito das imagens [36]. Estas redes têm a vantagem de conseguirem codificar os dois priores, passando uma sequência de poses à rede, como apresenta o artigo de Daniel et al. [34]. No entanto, existe outra abordagem proposta por Daniel et al. [30] onde a rede neuronal processa as poses individualmente, aplicando no final um filtro temporal de forma a suavizar o movimento.

De seguida são apresentados dois artigos que utilizam técnicas aprendizagem automática e os dois mencionados acima de aprendizagem profunda.

Aristidou et al. [22] propõe um método que detecta os erros das juntas e corrige-os com base em movimentos semelhantes. As rotações das juntas são agrupadas numa matriz designada por "*motion-texture*" em que cada linha representa uma

junta e cada coluna um instante temporal. A sequência do movimento é dividida em sub-grupos, chamados de "*Motion-words*", apresentados em rectângulos na figura 2.1. Para cada *motion-word* encontram-se os K vizinhos mais próximos (KNN) e através da sua média determina-se a "*mean-motion-texture*. Subtraindo esta última matriz à original dá origem à matriz "*movement digression map*", apresentado na figura 2.1, onde é possível detectar rapidamente os erros e substituir pela média da respetiva *motion-word*.

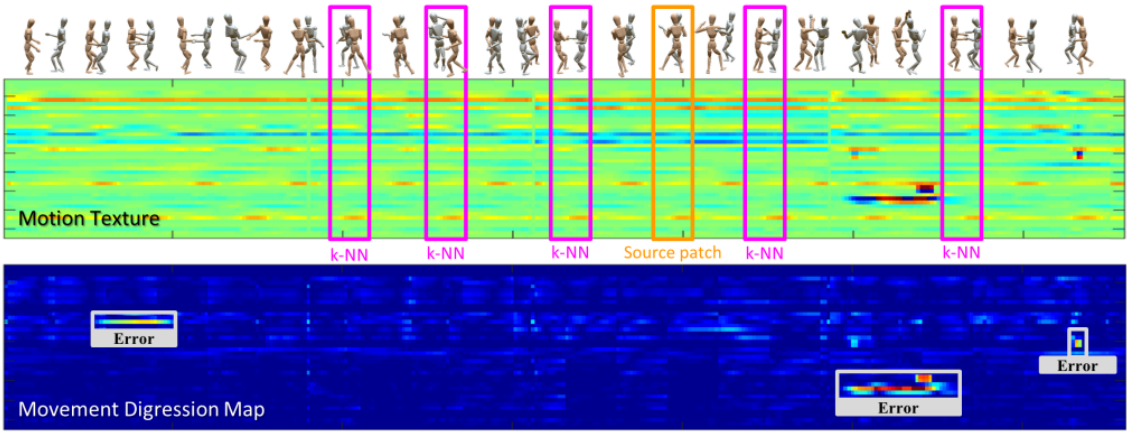


Figura 2.1: Matriz de representação das juntas retirada do artigo publicado por Aristidou et al. [22]

Apesar deste método conseguir detectar e corrigir erros em movimentos simples e complexos, os esqueletos a comparar devem ter uma estrutura semelhante. Tal não acontece no modelo proposto, pois é feita uma transformação e normalização das poses antes de serem processadas pela rede. O tamanho da sequência é outro problema, que pode não ser suficiente para determinar uma boa média de cada *motion-word*. Por último, está dependente de movimentos semelhantes por isso necessita de um grande número de amostras tornando o processo de procura muito lento, ao contrário da rede neuronal que apesar de levar tempo na fase de treino, é rápida nas previsões.

A técnica PCA é utilizada no artigo publicado por Chai et al. [23], com o objectivo de produzir o movimento completo do actor a partir de um pequeno conjunto de marcadores e duas câmaras sincronizadas. Sendo impossível fornecer um movimento preciso apenas com 6 a 9 marcadores, é utilizada uma base de dados com movimentos captados através de 40 a 50 marcadores. O sistema está dividido em 3 blocos ilustrados na figura 2.2. O bloco "*Motion performance*" transforma a informação das duas câmaras em sinais de baixa dimensão. De seguida, o bloco

"*Online local modeling*" aplica a técnica PCA às poses da base de dados de forma a reduzir a sua dimensionalidade para 6 a 9 marcadores e determina as "K" poses mais próximas utilizando o método KNN. Por último, o bloco "*Online motion synthesis*" reconstrói as poses utilizando o modelo linear treinado e as poses sintetizadas anteriormente.

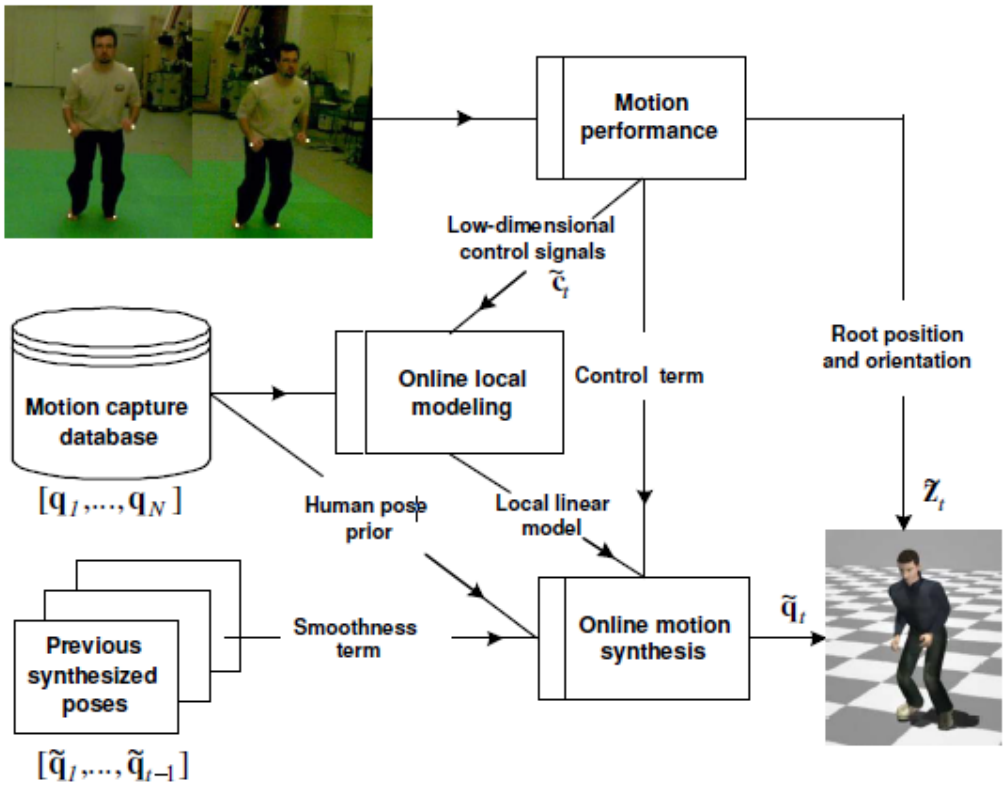


Figura 2.2: Diagrama do sistema retirado de Chai et al. [23]

O método PCA produz um *manifold* (conjunto de poses possíveis) que é usado para remover erros. Sendo um método linear, não se adapta bem ao aumento dos dados de treino o que é fundamental para reconstruir as poses com precisão.

Outro artigo relevante foi o publicado por Daniel et al. [34]. Este utiliza um tipo de rede neuronal designado por autoencoder que tem o objectivo semelhante ao método PCA, aprender a representar um conjunto de dados reduzindo a sua dimensão e com isto gerar uma amostra próxima do que foi introduzido. Este tipo de rede foi muito utilizado nos problemas de redução de ruído, principalmente em imagens. O artigo referenciado descreve uma abordagem que inclui o prior temporal. Para tal é feito um pré-processamento que divide as sequências em subconjuntos de 160 poses e normaliza o comprimentos das juntas. A rede recebe as 160 poses, cada uma com 63 graus de liberdade como ilustra a figura 2.3.

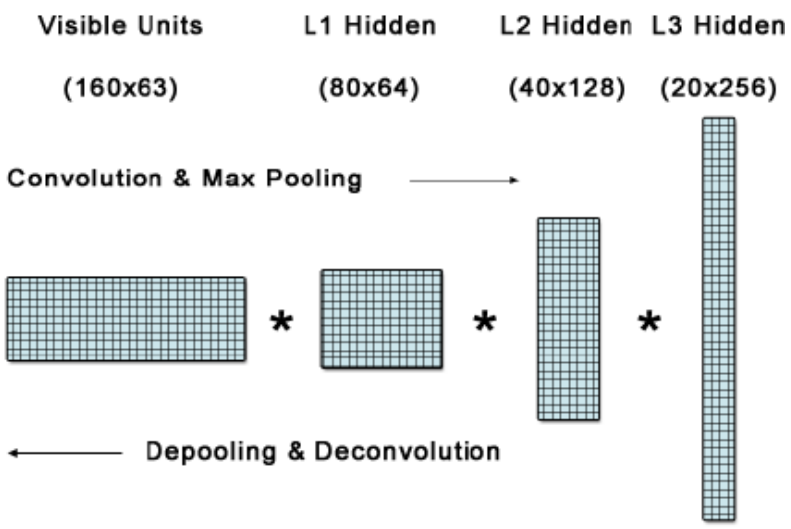


Figura 2.3: Estrutura do autoencoder utilizado no artigo de Daniel et al. [34]

Os autores afirmam que o sistema é ainda capaz de fazer a interpolação dos movimentos e que apresenta melhores resultados que outros métodos como o PCA. A principal diferença do modelo aqui proposto é a codificação do prior temporal na rede, que foi substituída pelo filtro de "Savitzky–Golay" e que apesar de não ser possível fazer a interpolação das poses, remove o *jittered* causado pela rede.

O artigo que mais se destaca na resolução deste problema foi publicado por Daniel et al. [30]. Este artigo baseia-se no treino de uma rede neuronal profunda designada de *Residual Network* (ResNet) e num pré-processamento dos dados da CMU, incluindo os marcadores usados na captura e as juntas obtidas. A altura do esqueleto é normalizada de forma a que não seja preciso lidar com esqueletos de diferentes tamanhos. Para cada pose obtêm um ponto de referência para representar as juntas, a partir da técnica de alinhamento por "*rigid body*", na qual é escolhido um conjunto de marcadores à volta do tronco do esqueleto e calcula-se a média dos pontos relativamente a uma junta escolhida, normalmente pertencente à coluna. O ponto médio calculado é o "*rigid body*" do esqueleto e ponto de referência.

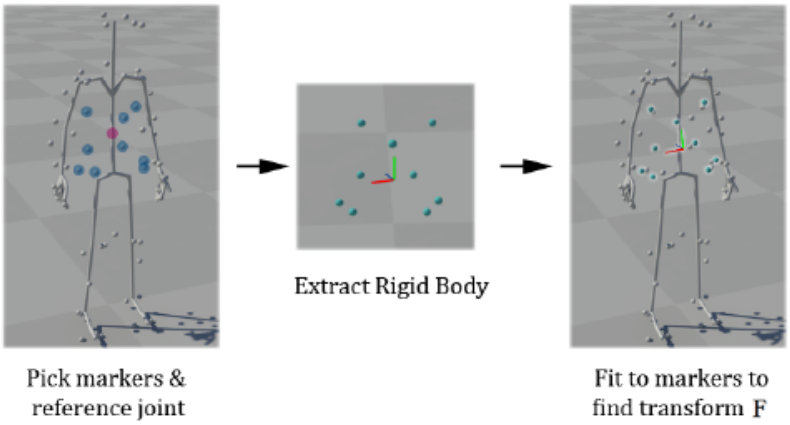


Figura 2.4: Técnica de alinhamento por "*rigid body*" retirado de Daniel et al. [30]

Depois de converter todas as poses para o sistema de coordenadas local, são calculadas três tipos de estatísticas dos dados para serem utilizadas no treino:

- A média e desvio padrão das juntas $y^{\mu} \in \mathbb{R}^{j*3*4}$, $y^{\sigma} \in \mathbb{R}^{j*3*4}$.
- A média e covariância das configurações dos marcadores $z^{\mu} \in \mathbb{R}^{m*j*3}$, $z^{\Sigma} \in \mathbb{R}^{(m*j*3)*(m*j*3)}$.
- A média e o desvio padrão da posição global dos marcadores, $x^{\mu} \in \mathbb{R}^{m*3}$, $x^{\sigma} \in \mathbb{R}^{m*3}$, calculados com o algoritmo "*Linear Blend Skinning*" que relaciona a distância de um marcador a cada junta do esqueleto.

Para corromper os dados, foi implementada uma função que causa oclusões e deslocamentos aleatórios nos marcadores. Segundo os resultados apresentados pelos autores, essa função aumenta significativamente o desempenho da rede, face a outras funções de ruído como o ruído gaussiano, o ruído uniforme e o *dropout*.

No treino é gerada uma configuração de marcadores "Z" a partir de z^{μ} e z^{Σ}. Dada a pose "Y" constrói-se a posição global dos marcadores "X", como demonstra a figura 2.5, de seguida corrompo-se esses marcadores utilizando a função de ruído implementada, aplica-se a normalização "Z-score" e introduz-se os dados na rede produzindo a pose "$\hat{Y}$". Esta pose é comparada à original "Y" e o erro resultante é usado na atualização dos pesos da rede neuronal.

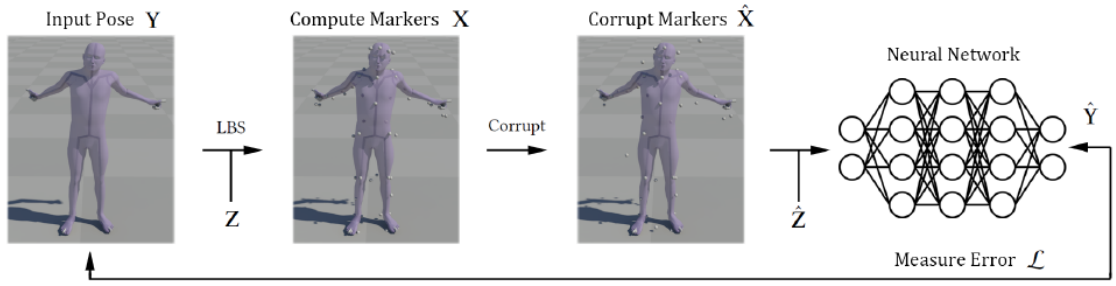


Figura 2.5: Treino da rede neuronal implementado em Daniel et al. [30]

A rede processa as poses individualmente, como tal pode ocasionalmente introduzir *jitter* nos movimentos. Para combater esse problema introduziram o filtro "Savitzky-Golay" à saída da rede, aplicando a cada componente da junta.

Neste documento, pretende-se corrigir as poses devolvidas pelo sensor Kinect. Sendo que os dados devolvidos por este sensor não são idênticos aos da CMU, as soluções desenvolvidas nos artigos acima não se adequam a este caso específico. Será utilizado um método semelhante ao último artigo apresentado, com o mesmo tipo de rede neuronal profunda mas com diferentes *inputs* pois não existe informação dos marcadores. O *dataset* da CMU e todo o pré-processamento será adaptado de forma a corresponder ao esqueleto da Kinect.

3

Conceitos

Este capítulo descreve os conceitos abrangidos ao longo do documento, nomeadamente o significado dos parâmetros das redes neuronais [27] e quais as configurações que mais se adequam ao tipo de problema.

3.1 Redes Neuronais

A rede neuronal artificial é uma técnica de aprendizagem automática baseada nas redes neuronais biológicas com o objectivo de reconhecer padrões. Esta técnica é constituída por um conjunto de unidades ligadas entre si, designadas por **neurónios**, que calculam a soma pesada da informação das suas entradas. Estas unidades organizam-se em **camadas** (*layers*). A figura 3.1 apresenta um exemplo de uma rede neuronal *feedforward* onde as camadas ligam-se exclusivamente à camada seguinte, ou seja, a camada de entrada (*input layer*) está ligada à camada oculta (*hidden layer*) que por sua vez está ligada à camada de saída (*output layer*). A camada oculta pode ser composta por uma ou mais camadas, respeitando a mesma regra para este tipo de redes neuronais.

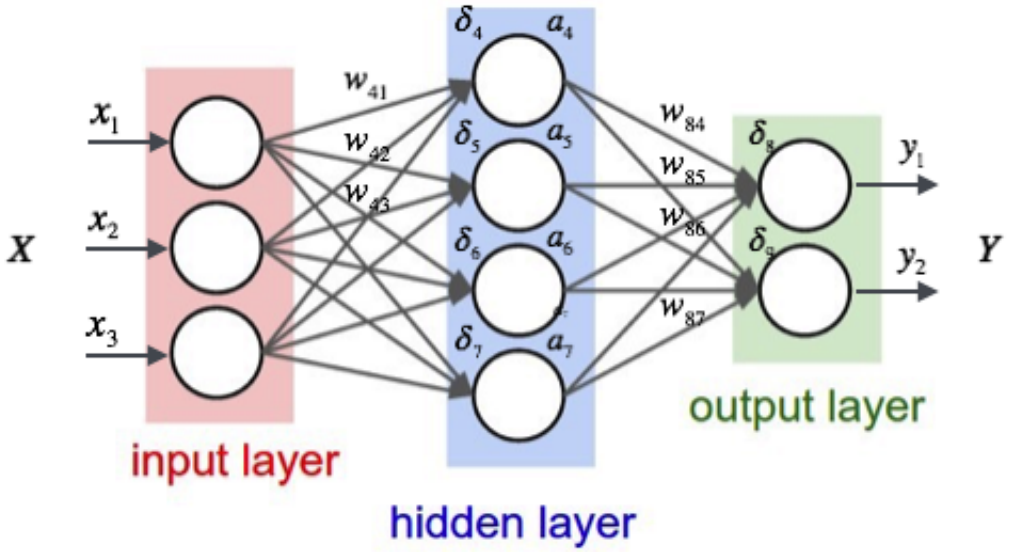


Figura 3.1: Exemplo de uma rede neuronal *feedforward* [15]

A rede aprende através de um processo de treino que envolve amostras de treino, cada uma contendo uma entrada e resultado conhecidos. As amostras são apresentados à rede por meio da camada de entrada, que comunica com as camadas ocultas onde cada neurónio calcula a soma (Z) das entradas pesadas e aplicam a função de activação (f),

$$Z = f(\sum_{i=1}^{n} x_i * w_i) \tag{3.1}$$

onde n representa o número de entradas no neurónio, x_i o valor de uma entrada e w_i o peso correspondente. Por sua vez, a camada oculta comunica da mesma forma com a camada de saída, dando origem ao resultado. A rede actualiza os pesos (w_i) de acordo com um algoritmo de optimização e o erro obtido entre a saída calculada e a saída desejada. Assim, a aprendizagem de uma rede a partir de uma amostra ou conjunto de amostras é o processo de adaptação dos pesos das ligações para que esta apresente à saída os valores mais próximo do desejado.

Depois de receber um número suficiente de exemplos, a rede torna-se capaz de prever resultados de entradas, utilizando as ligações ajustadas a partir do conjunto de treino.

Este método de aprendizagem é baseado no algoritmo Descida do Gradiente [27], capaz de encontrar a solução óptima, através da troca iterativa dos parâmetros (pesos) de forma a minimiza a função do erro. No entanto, sofre do problema da desaparecimento do gradiente que ocorre quando o valor à saída dos neurónios é muito pequeno, diminuindo cada vez mais o gradiente ao ponto de tornar os

pesos inalteráveis. Desta forma o treino não irá alcançar uma boa solução. Este problema pode ser resolvido com a escolha da função de activação apropriada, que faz um determinado mapeamento do valor à saída dos neurónios.

Os hiper-parâmetros são as variáveis que definem a estrutura da rede e como esta é treinada. Para a estrutura da rede deve ser indicado o número de camadas e neurónios dependendo do problema. No caso da camada de entrada, o número de neurónios é igual ao número características de uma amostra. Na camada de saída o número de neurónios depende do tipo de resultado esperado, por exemplo, para um problema de regressão onde se pretende obter um só valor numérico, é necessário apenas um neurónio. Para um problema de classificação, será necessário um neurónio por classe. Quando à camada oculta, terão de ser testadas várias configurações, com diferentes número de camadas e neurónios, tendo em conta que o aumento do número de neurónios pode levar à sobre-aprendizagem (*overfitting*), isto é, a rede consegue prever com precisão as amostras com que foi treinada, mas é incapaz de generalizar para novas amostras.

As sub-secções seguintes descrevem alguns dos hiper-parâmetros e as configurações que mais se adequam ao problema proposto neste trabalho.

3.1.1 Inicialização dos pesos

O algoritmo de treino é um processo iterativo que tem uma fase de inicialização dos pesos e uma fase de actualização. Neste último passo, existe uma diminuição da função do erro, diferença entre a saída calculada e a saída desejada.

Uma das formas de inicialização dos pesos é por um processo aleatório para evitar que os pesos sejam inicializados todos a '0'.A tabela 3.1 indica as os tipos de inicialização para as funções de activação que melhor se adequam.

Inicialização	Função de ativação
Glorot	None, Tanh, Logistic, Softmax
He	ReLu e suas variantes
LeCun	SELU

Tabela 3.1: Inicialização dos pesos adequadas a cada função de activação [27]

3.1.2 Funções de ativação

Um das causas do desaparecimento do gradiente é escolha incorrecta da função de activação. Nas redes com maior complexidade, a função **ReLU** (ver figura 3.2)

é normalmente utilizada devido a não saturar os valores positivos e por ser rápida a calcular o valor mapeado à saída do neurónio, embora sofra do problema "*dying ReLU*" [27] (durante o treino alguns neurónios desligam-se), ou seja, os pesos associados a um neurónio ganham uma certa configuração fazendo com que a soma pesada das entradas origine sempre a mesmo saída 0. A função **LeakyReLU** é uma variante da ReLU que tem como objectivo eliminar o problema anterior. Esta função tem uma ligeira inclinação nos valores negativos, fazendo com que os neurónios adormecidos possam acordar, ou seja, possam recuperar de valores perto de zero. A função **ELU** [25] é outra variante da ReLU que supera todas as outras, em termos de desempenho e no tempo necessário para o treino. Apesar de ser mais lenta a calcular, acaba por compensar devido à rápida convergência da solução óptima. Existe ainda uma variante desta última, designada por **SELU** [28], que segundo os autores, uma rede neuronal composta exclusivamente por *dense layers* com SELU tem a capacidade de se auto normalizar, ou seja, cada *layer* tende a preservar a média 0 e o desvio padrão 1 durante o treino, resolvendo o problema do desaparecimento do gradiente. Para que isto possa acontecer devem-se ser respeitadas as seguintes condições:

1. As características de entrada devem ser padronizadas (média 0 e desvio padrão 1);

2. Cada camada oculta deve ser inicializada com LeCun (distribuição normal);

3. A arquitectura deve ser sequencial. No caso de outras, como por exemplo a ResNet [29] e a *Recurrent neural network* (RNN) [35], a auto normalização não é garantida;

4. Todas as camadas devem ser do tipo *dense* (os neurónios de uma camada ligam-se a todos os neurónios da camada seguinte) ou convolucionais [27] (utilizadas em problemas de análise de imagens).

A figura 3.2 demonstra os gráficos da função ReLU e respectivas variantes.

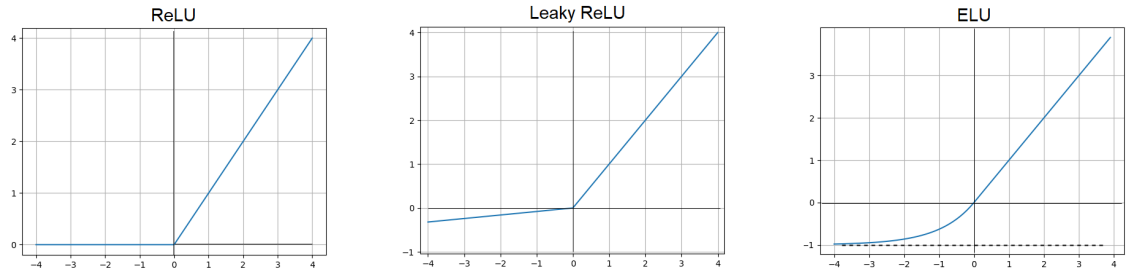


Figura 3.2: Função ReLU e as suas variantes

3.1.3 Algoritmos de optimização

Os algoritmos de optimização utilizados no treino da rede baseiam-se numa função de custo e na sua optimização (diminuição), nomeadamente em algoritmos de descida do seu gradiente. O "**Momentum Optimization**" [33] é um dos algoritmos mais simples que tem um parâmetro adicional designado por "*momentum*". Este faz acelerar o passo da descida até alcançar uma certa velocidade, de forma a chegar mais rápido ao mínimo local. Outro algoritmo é o **Adagrad** [31], que ajusta mais rápido a sua direção para o mínimo local. Este algoritmo tem um desempenho positivo em problemas simples, mas como os passos dados da descida do gradiente (*learning rate*) reduzem demasiado depressa, o treino finaliza antes de chegar à melhor solução. A fim de corrigir este problema foi criado o algoritmo **RMSProp** [1]. Este acumula apenas os gradientes das iterações mais recentes, utiliza o parâmetro adicional "*decay rate*" que decresce exponencialmente o primeiro passo na descida do gradiente.

Um dos algoritmos mais recentes, designado por **Adam** [24], combina os algoritmos *Momentum Optimization* e RMSProp. Adapta automaticamente o *learning rate* podendo esse parâmetro ser descartado durante o processo de troca dos hiperparâmetros, designado por "*tunning*", reduzindo o número de configurações da rede a testar. Este último algoritmo tem duas variantes, **Adamax** e **Nadam** [26]. A primeira pode ou não melhorar o desempenho dependendo do conjunto de dados. A segunda, converge mais rapidamente utilizando a técnica de "Nesterov" e na maior parte dos casos tem um melhor desempenho que o Adam.

4

Conjunto de Dados

Este capítulo introduz as duas estruturas de dados, Kinect e CMU, que são utilizadas neste trabalho. A Kinect capta em tempo real um conjunto de **juntas** que representam um modelo simplificado do corpo humano (esqueleto). O *dataset* da CMU disponibiliza diversos movimentos do corpo humano que devem ser processados de forma a obter os **segmentos** do esqueleto.

4.1 Representação do esqueleto humano

O esqueleto humano é uma estrutura hierárquica composta por ossos e articulações. Em computação gráfica essa estrutura pode ser representada utilizando uma das duas componentes, normalmente designadas por segmentos (ossos) e juntas (articulações). O segmento é um vector com uma determinada direcção e comprimento igual ao do respectivo osso, enquanto a junta é um ponto representado pela sua posição e rotação. A figura 4.1 apresenta um esqueleto artificial com indicação das juntas e segmentos.

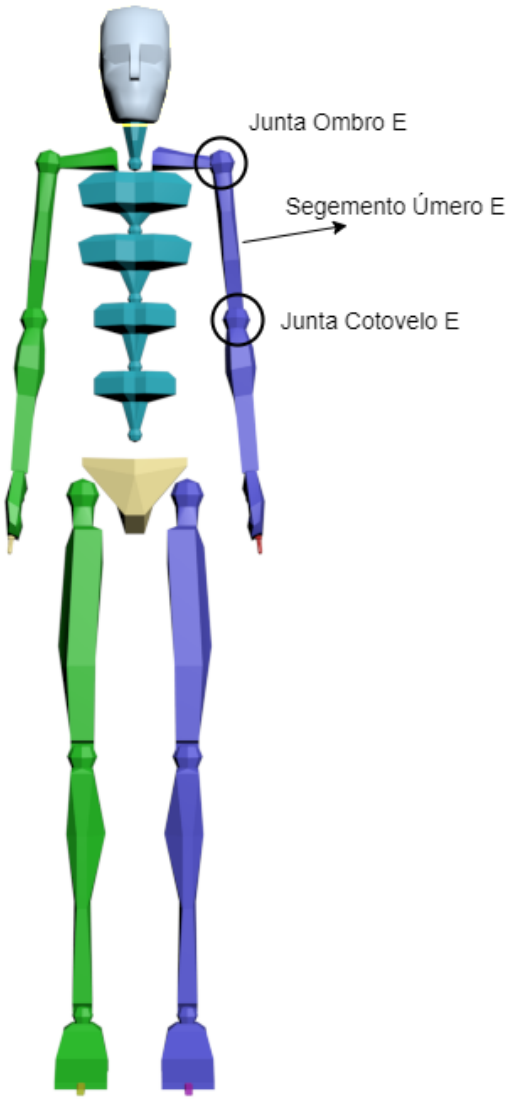


Figura 4.1: Representação de um esqueleto artificial

4.2 Kinect

A Kinect foi um dos primeiros sensores, lançado pela Microsoft, capaz de captar o movimento do utilizador para interagir com videojogos e outras aplicações. É composta por uma câmara RGB, um *array* de microfones e um sensor de profundidade (ver figura 4.2).

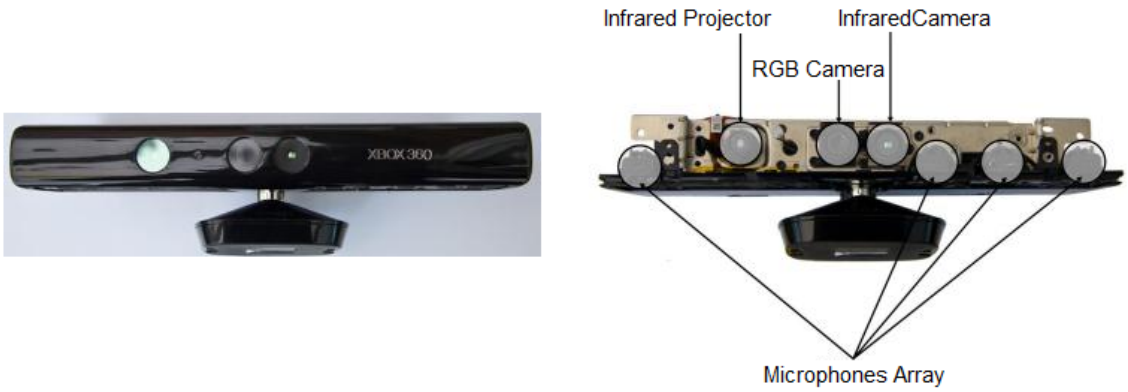


Figura 4.2: Sensor Microsoft Kinect [8]

Embora tenha sido desenvolvido para ser utilizado em jogo da consola Xbox, também tem sido utilizado em projectos de investigação, onde seja necessária informação espacial, não só do esqueleto do actor como do ambiente que rodeia, devido ao fácil acesso e qualidade.

A Microsoft lançou uma segunda versão, designada por "Kinect v2", com melhoramentos no hardware e software, capaz de detectar mais juntas do esqueleto e as suas rotações. A tabela 4.1 descreve as características do hardware das duas versões da Kinect.

	Kinect v1	**Kinect v2**
Color	640x480 30fps	1920x1080 30fps
Depth	320x240 30fps	512x424 30fps
Range	0.8~4.0m	0.5~4.5m
Angle of View (H/V)	57/43 degrees	70/60 degrees

Tabela 4.1: Características do hardware da Kinect v1 e v2

Apesar da Kinect ter o seu próprio *Software Development Kit* (SDK) (v1 e v2), optou-se por utilizar outro desenvolvido pela 3DiVi, designado por Nuitrack, compatível com as duas versões da Kinect e outros sensores de profundidade, nomeadamente, Orbbec Astra S, Asus Xtion Pro, Asus Xtion 2 e Intel RealSense.

Em alternativa, o "OpenNI" é outro SDK compatível com diversos sensores. No entanto, apresenta uma documentação com menos qualidade e não são fornecidos exemplos específicos para o Unity ao contrário da documentação do Nuitrack.

4.2.1 Nuitrack

Este SDK estabelece uma Interface de Programação de Aplicações (*Application Programming Interface* - API) para comunicar com os sensores 3D de modo a representar o esqueleto e seguir os movimentos do corpo humano capturados pelo sensor. O SDK oferece suporte multi plataforma incluindo Android, Windows e Linux. Para qualquer um dos sensores, o Nuitrack SDK representa sempre o esqueleto do corpo humano com o mesmos conjunto de juntas, tornando possível a troca de sensores a qualquer instante.

A API é uma interface implementada em C++ baseada numa camada *middleware* que providencia a comunicação directa com os sensores (ver figura 4.3). É ainda oferecida uma segunda interface na linguagem C# para o desenvolvimento de aplicações em Unity. A desvantagem deste SDK é tempo limitado de gravação na versão gratuita. Em cada execução da aplicação a Kinect captura o esqueleto durante 3 minutos, passado esse tempo é necessário reiniciar a aplicação.

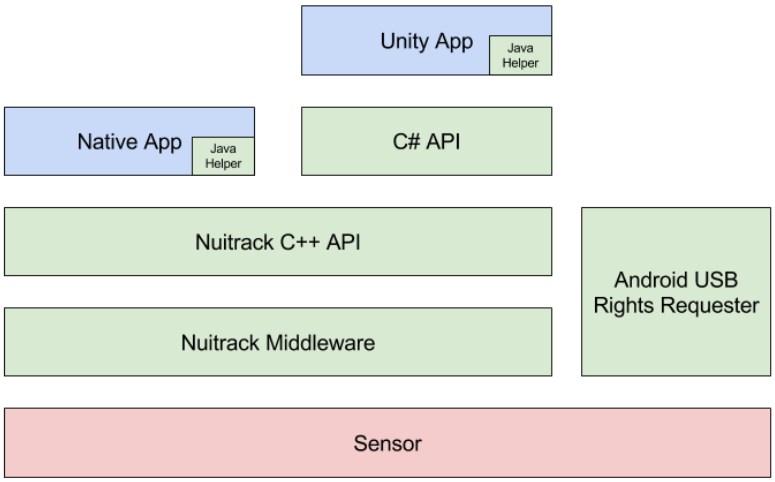


Figura 4.3: Arquitetura do SDK Nuitrack

O Nuitrack representa o corpo humano com um esqueleto constituído por 17 juntas (ver figura 4.4). Para cada uma é dada a sua posição e rotação em relação a um referencial global. O SDK representa cada junta através de uma estrutura composta por vários atributos da junta. No caso da junta estar ocludida, existe um parâmetro (*confidence*) que é colocado a zero, o que permite ter conhecimento prévio do estado da junta.

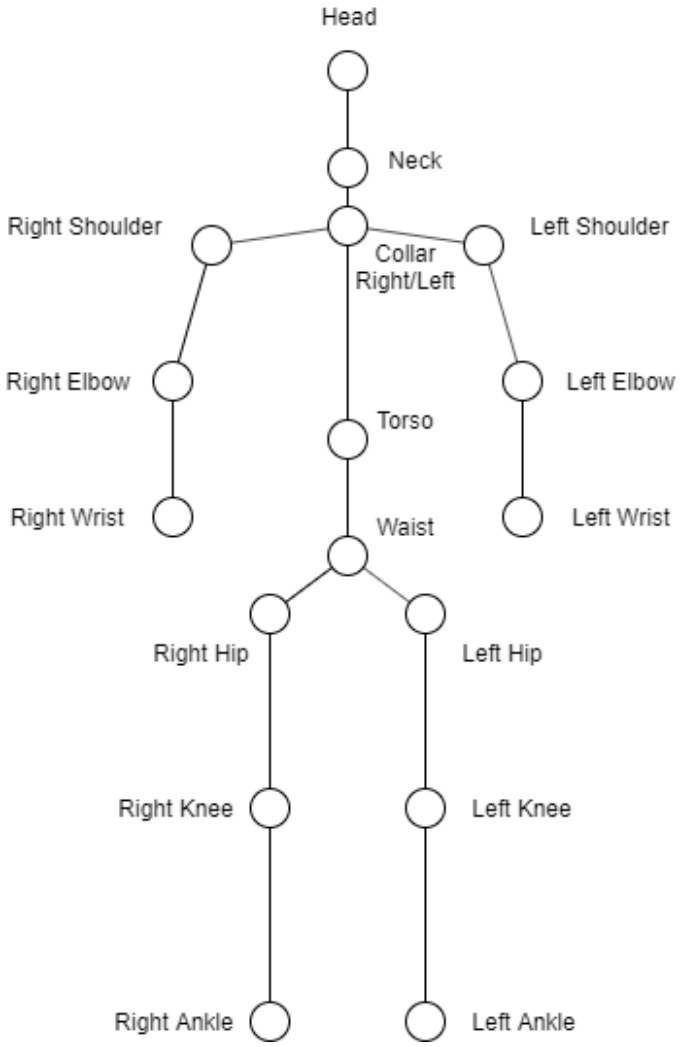


Figura 4.4: Esqueleto Nuitrack

4.3 Dataset da CMU

O *dataset Motion Capture* da CMU [6] é composto por um conjunto de capturas de movimento organizadas por categorias e actores. Cada actor contém um conjunto movimentos com a respectiva descrição.
As capturas são realizadas através de 12 câmaras infravermelhos (Vicon MX-40) com capacidade de capturar imagens a 4 megapixel e 120 Hz, apesar de nem todos os movimentos terem sido capturados a esta frequência.
O actor contém 41 **marcadores** que serão usados para determinar o seu esqueleto durante a gravação (ver figuras 4.6 e 4.7). A posição tridimensional de cada marcador é guardada num ficheiro com a extensão ".c3d". A figura 4.5 apresenta uma configuração possível dos marcadores num actor.

Figura 4.5: Configuração dos marcadores

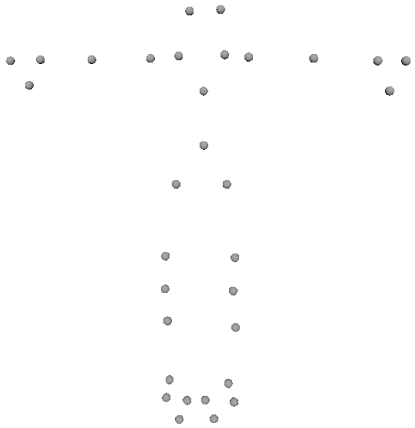


Figura 4.6: Marcadores extraídos

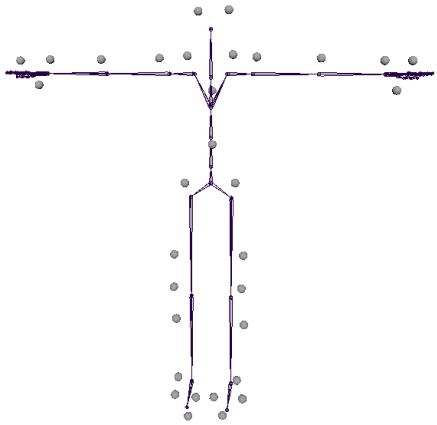


Figura 4.7: Esqueleto do actor

Juntamente com os marcadores, este *dataset* disponibiliza o movimento do **esqueleto** em formato "ASF/AMC" definido pelo grupo "Acclaim" [2]. Este formato é constituído por dois ficheiros, Acclaim Skeleton File (ASF) e Acclaim Motion Capture data (AMC), o primeiro contém informação do esqueleto e o segundo o seu movimento em cada *frame*.

4.3.1 Formato ASF/AMC

Este formato é constituído por dois ficheiros, Acclaim Skeleton File (ASF) e Acclaim Motion Capture data (AMC), ambos gerados em ASCII e com um formato específico, de forma a simplificar a interpretação. O esqueleto é definido por uma hierarquia de segmentos como apresenta a figura 4.8, onde a cintura é a raiz definida num eixo de coordenadas global.

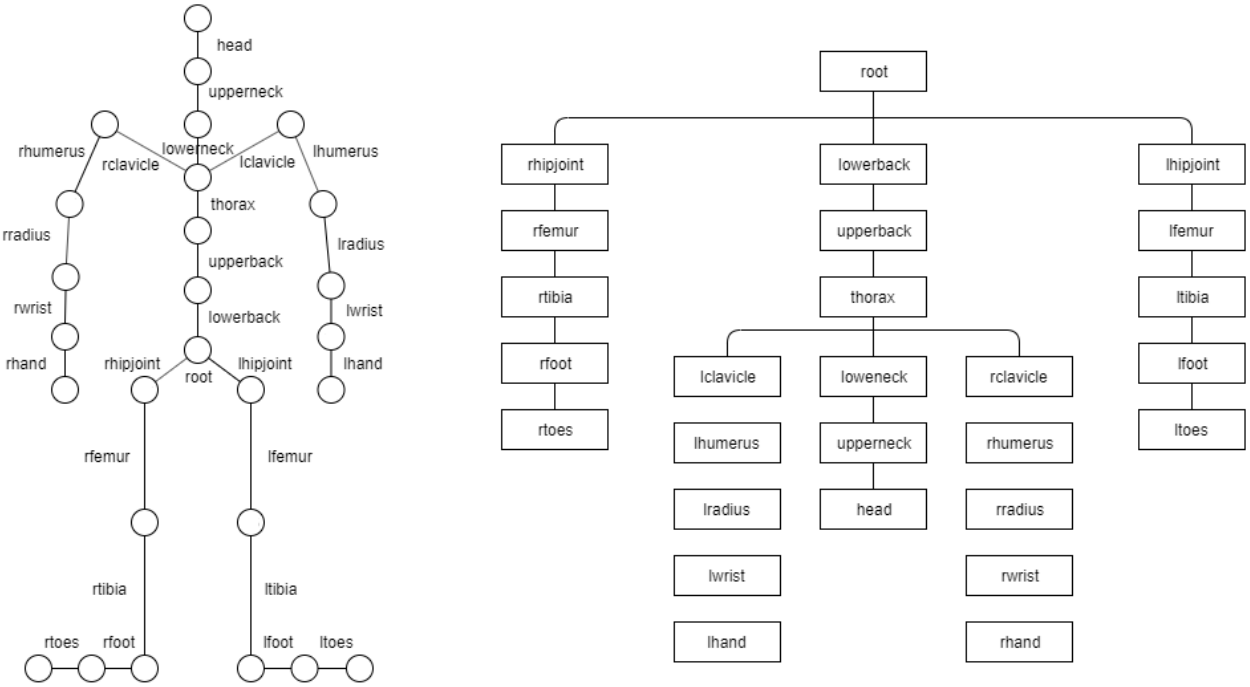


Figura 4.8: Modelo hierárquico do esqueleto do *dataset* da CMU

O ficheiro **ASF** apresenta informação do esqueleto numa determinada pose inicial (ver figura 4.9). Este ficheiro descreve a direcção, comprimento, eixo, graus de liberdade de cada segmento e os limites da rotação. O segmento *root* é uma excepção a qual é indicada apenas a posição e orientação inicial, que se reflecte na posição de todo o esqueleto no espaço global.
Resumindo, os segmentos têm os atributos:

- **Direction**: Vector unitário que aponta no sentido do segmento, num sistema de coordenadas global;
- **Length**: Comprimento do segmento;

- **Axis**: Rotação inicial do segmento relativa ao sistema de coordenadas global;
- **Dof**: Graus de liberdade da rotação (x, y, z);
- **Limits**: Limites, máximos e mínimos, de cada eixo de rotação. Não é usado na interpretação dos dados.

O ficheiro **AMC** contém o movimento para o esqueleto definido no ficheiro ASF. A cada *frame* é descrita a rotação de cada segmento em relação ao respectivo eixo "Axis" definido no ficheiro ASF. Para que o esqueleto se movimente no espaço, é indicada a posição do segmento *root*.

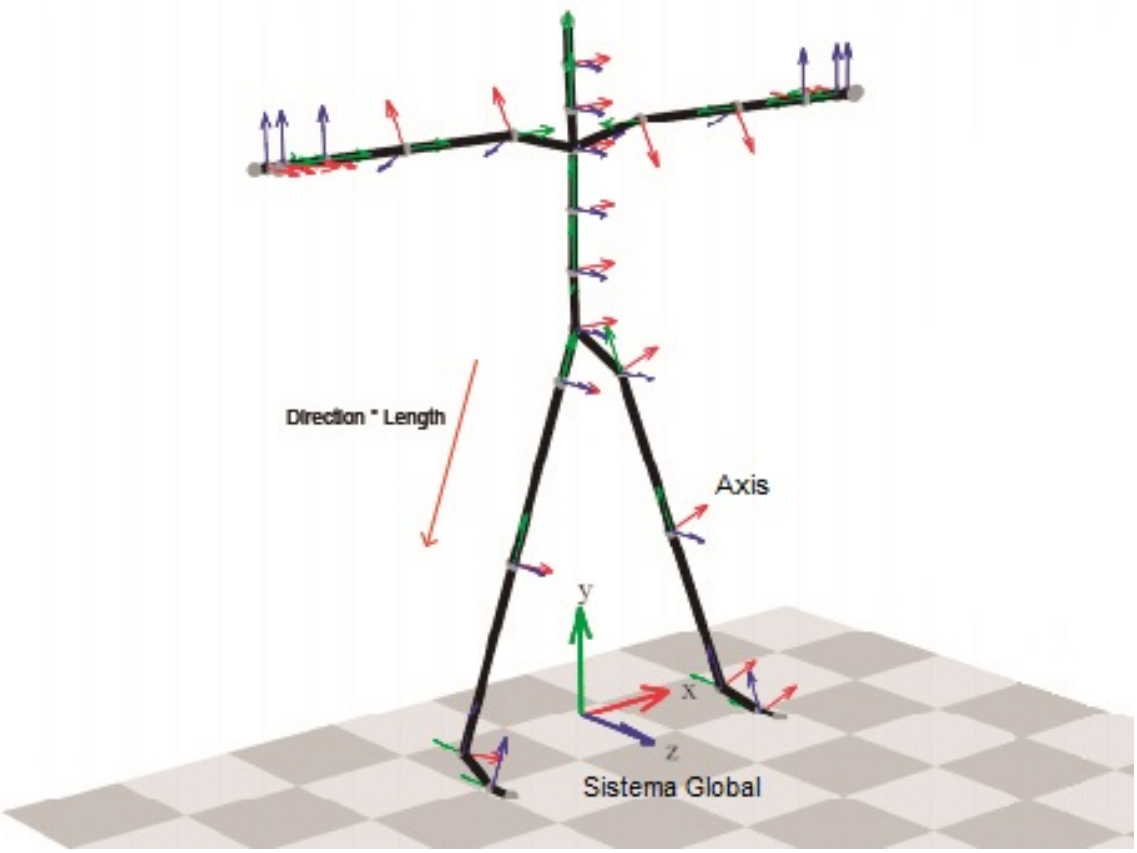


Figura 4.9: Esqueleto ASF

4.3.2 Interpretação dos dados

Esta fase tem como objectivo gerar as poses do esqueleto em cada instante a partir dos ficheiro ASF e AMC. Estas poses são constituídas por segmentos definidos

pela sua rotação e posição global. Ao esqueleto inicial definido pelo ASF são aplicadas as rotações do ficheiro AMC, obtendo a rotação e posição global de cada segmento num dado instante. Para tal definiu-se o objecto "Bone", em Python, com os atributos apresentados na figura 4.10.

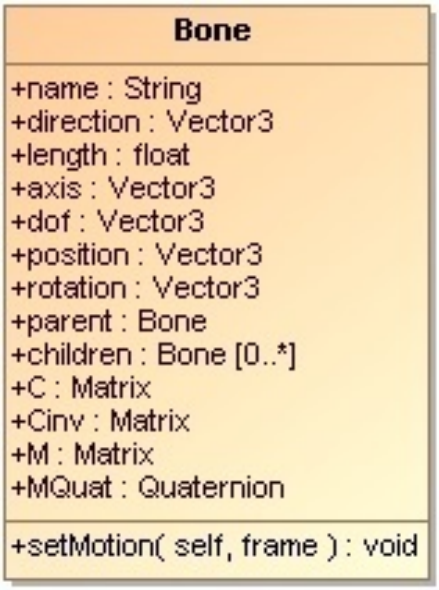


Figura 4.10: Classe "Bone"

Para além dos atributos retirados directamente do ficheiro ASF, existem outros que serão calculados durante a actualização dos movimentos, nomeadamente:

- **position**: posição do extremo do segmento no sistema de coordenadas global;
- **rotation**: rotação actual, proveniente do ficheiro AMC;
- **C**: matriz de rotação composta por "Axis";
- **Cinv**: matriz C inversa;
- **M**: matriz de tranformação global (rotação);
- **MQuat**: rotação global, definido por um quaternião (x,y,z,w), do segmento.

O processo inicia-se com a análise do ficheiro ASF, definindo um total de 31 segmentos, ou seja, 31 objectos do tipo "Bone". A partir deste ficheiro determinam-se as posições iniciais.

A posição do segmento é definida pelo ponto extremo final do segmento, como ilustra a figura 4.11. Tendo conhecimento do vector de direcção e o comprimento do segmento é possível determinar esse ponto somando à posição do segmento ”pai”. Tal ponto é calculado com a soma entre o vector de direcção de magnitude igual ao comprimento do segmento e a posição do segmento ”pai”. Este cálculo é feito segundo a hierarquia de forma a que o segmento ”pai” já tenha sido determinado

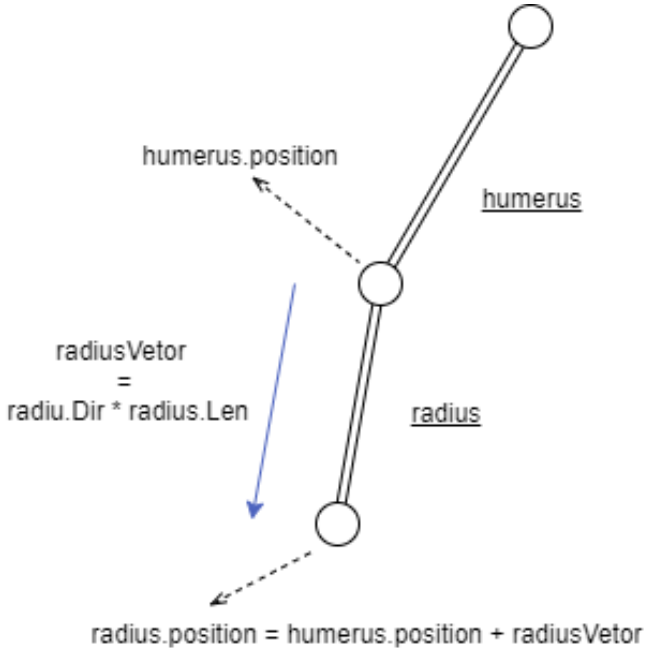


Figura 4.11: Obter a posição do segmento

Depois de calcular as posições de todos os segmentos tem-se a pose inicial do esqueleto ilustrada na figura 4.9.

Para determinar as posições dos segmentos nas próximas *frames* percorre-se a hierarquia do esqueleto começando no elemento *root* e para cada um aplicam-se as respectivas rotações do ficheiro AMC aos vectores de direcção e repete-se o cálculo da posição. Como as rotações do AMC estão representadas em relação aos ”Axis” do ASF, enquanto o vector de direcção vem representado no sistema de coordenadas global, é necessário criar uma matriz de transformação global M que transforme essas rotações para o mesmo sistema de eixos do vector de direcção (ver figura 4.12).

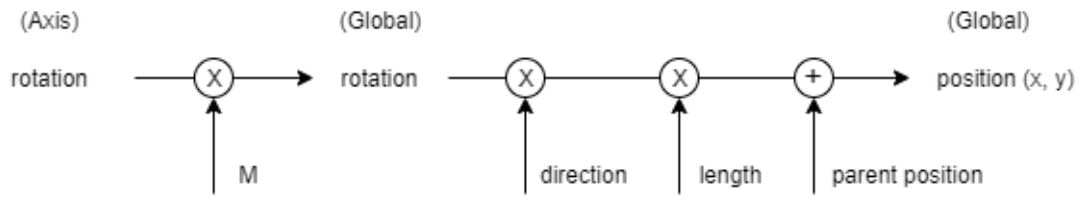


Figura 4.12: Calcular a posição global

A matriz global M é obtida multiplicando a matriz local L à matriz global do elemento "pai". A matriz local é dada pela rotação dos eixos em relação ao sistema global:

$$L = axis * rotation \tag{4.1}$$

De seguida multiplica-se à matriz global do segmento "pai" (M_{parent}), dando origem a M:

$$M = M_{parent} * L \tag{4.2}$$

Sendo a rotação definida por Axis desnecessária, esta é removida multiplicando pelo seu inverso, ficando apenas a mudança causada pela rotação do movimento (ficheiro AMC):

$$M = M * inv(axis) \tag{4.3}$$

A *root* é mais uma vez um caso específico, é o topo da hierarquia e como está directamente ligada ao mundo (sistema de coordenadas global) a sua matriz M é igual à sua matriz local L. Estes cálculos devem ser efectuados percorrendo a hierarquia em profundidade de forma a que o elemento pai do nó corrente já tenha calculado M.
Calculou-se também a rotação global de cada segmento em *quaterniões* para eliminar o problema "*gimbal lock*" causado pelos ângulos de Euler.

No final tem-se um conjunto de poses (esqueleto do corpo humano) constituídas por 31 segmentos, cada um com informação da sua rotação e posição num sistema de coordenadas global.

5

Modelo Proposto

Neste capítulo é inicialmente apresentada uma análise ao problema, juntamente com uma visão geral da abordagem utilizada e as ferramentas para a sua implementação. De seguida são descritas as componentes do modelo proposto em secções individuais.

5.1 Motivação

O sensor Kinect pode ser utilizado para replicar o movimento do corpo humano numa personagem virtual, por exemplo para jogos de realidade virtual/aumentada. No entanto é comum haverem oclusões, ou seja, partes do corpo que a câmara não detecta, reduzindo a experiência do utilizador. Para analisar a ocorrência de oclusões, foi realizado um estudo com 33000 poses capturadas pelo sensor Kinect. Na figura 5.1 é apresentado um histograma com as ocorrências de poses para diversos valores de oclusões. Pode verificar-se que 16% das poses tem oclusões, variando entre 1 a 6 oclusões por pose.

Este problema pode ser atenuado com a utilização de um sistema que consiga detectar as poses ocluídas do corpo humano e prever as poses corretas.

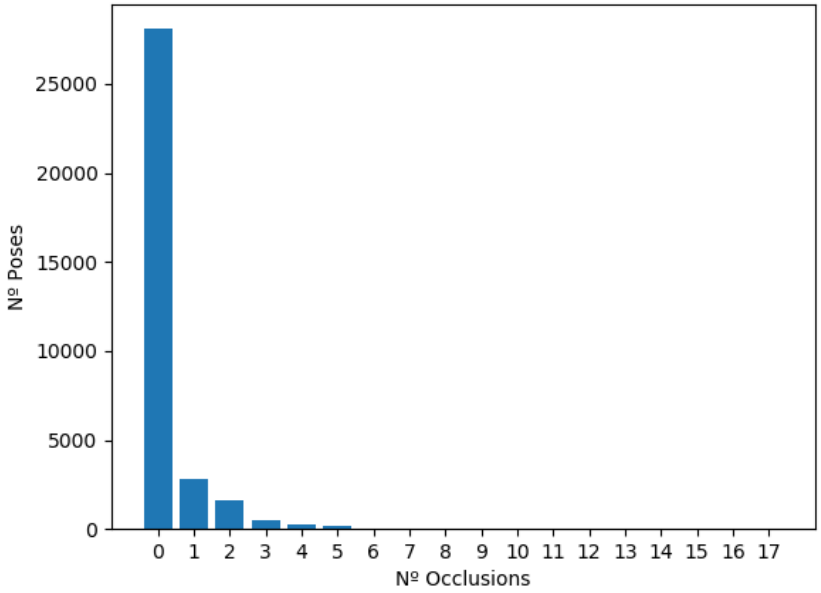


Figura 5.1: Histograma do número de oclusões por pose

5.2 Visão geral da solução

O objectivo deste trabalho é corrigir as poses do corpo humano capturadas pela Kinect recorrendo a técnicas de aprendizagem automática, em particular, às redes neuronais profundas. Por isso, o modelo desenvolvimento é composto por um método de aprendizagem profunda baseada numa ResNet. Este tipo de métodos exige uma grande quantidade de dados de treino. Assim, a rede deve ser treinada com um conjunto de treino idênticos aos dados da Kinect, isto é, um conjunto de 17 juntas bem posicionadas contendo a posição e a rotação. Na pesquisa realizada durante o desenvolvimento deste trabalho, não foi encontrada nenhuma base de dados com movimentos capturados pela Kinect e respectiva meta-informação, adequada aos objectivos deste trabalho. Assim, foi utilizado o conjunto de dados disponibilizados pela CMU. Estes dados apresentam diferenças na forma como são capturados (vários sensores) e também no modelo utilizado para representar o corpo humano (esqueleto). Por isso, o modelo proposto inclui um bloco de adaptação dos dados da CMU aos dados da Kinect.

Foi seguida uma abordagem idêntica à publicada em [30] mas adaptada aos dados da Kinect. A abordagem está dividida num conjunto de blocos, apresentados na figura 5.2, que implementam funcionalidades específicas. Inicia-se com o

processo de transformação do esqueleto de modo a adaptar-se ao da Kinect. De seguida, faz-se um pré-processamento dos dados para que não seja preciso lidar com esqueletos de diferentes tamanhos e transformam-se as coordenadas globais das juntas para o referencial da cintura (junta "Waist"), reduzindo o número de configurações das juntas a uma configuração por pose, ou seja, qualquer que seja a posição do esqueleto no referencial global, as juntas têm sempre o mesmo valor para uma certa pose.

A rede neuronal implementada processa as poses individualmente, ou seja, não tem conhecimento temporal da animação, por isso é provável que surja *jitter* na sequência de poses devolvida. Para combater esse problema introduziu-se o filtro Savitzky–Golay à saída da rede que suaviza o movimento.

Por último, faz-se a conversão inversa do bloco "pré-processamento" de forma a obter as poses nas condições iniciais.

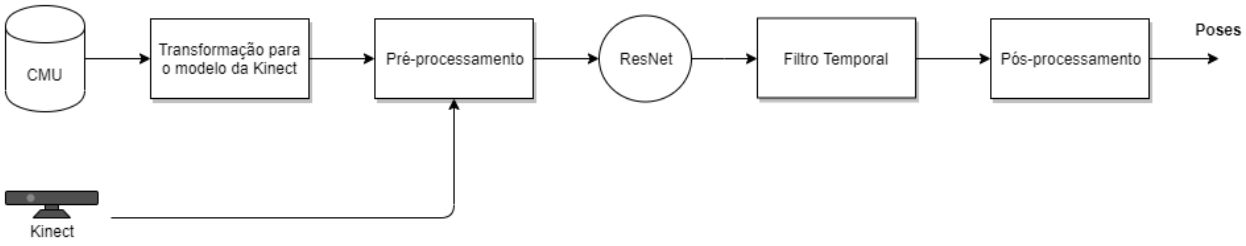


Figura 5.2: Diagrama do modelo proposto

O método proposto é implementado em Python utilizando as bibliotecas para processamento de dados e a API TensorFlow para facilitar a implementação dos métodos de aprendizagem automática utilizados. Esta API incorpora outra designada por "Keras" que oferece uma camada de mais alto nível. Para além de ser amplamente utilizada e de estar bem documentada, tem suporte para a GPU, acelerando significativamente a fase de treino da rede. O TensorFlow suporta outras linguagens de programação como o JavaScript e C#, ideais para desenvolver aplicações no motor de jogo "Unity" que no âmbito deste projecto é utilizado para visualizar as poses de uma forma mais interactiva e detectar problemas que possam ocorrer.

As seguintes secções descrevem detalhadamente os procedimentos de cada bloco da figura 5.2.

5.3 Adaptação do esqueleto da CMU ao modelo da Kinect

Depois de obter um conjunto de poses da base de dados da CMU, de acordo com o capítulo 4, o primeiro passo é converter os **segmentos** para **juntas**, de forma a que o esqueleto seja representado por um conjunto de juntas e não de segmentos. Essa conversão é feita percorrendo a hierarquia de segmentos e para cada um gera-se um novo objecto "Joint" (ver figura 5.4), onde a sua posição e rotação são definidas da seguinte forma:

$$joint.position = bone.position \tag{5.1}$$

$$joint.rotation = childBone.rotation \tag{5.2}$$

A figura 5.3 apresenta a relação entre o segmento úmero e a junta cotovelo. A posição desta junta é definida pela posição do segmento úmero, mas a sua rotação deve afectar a junta seguinte, ou seja, é definida pela rotação do segmento "filho" que neste caso é o segmento rádio.

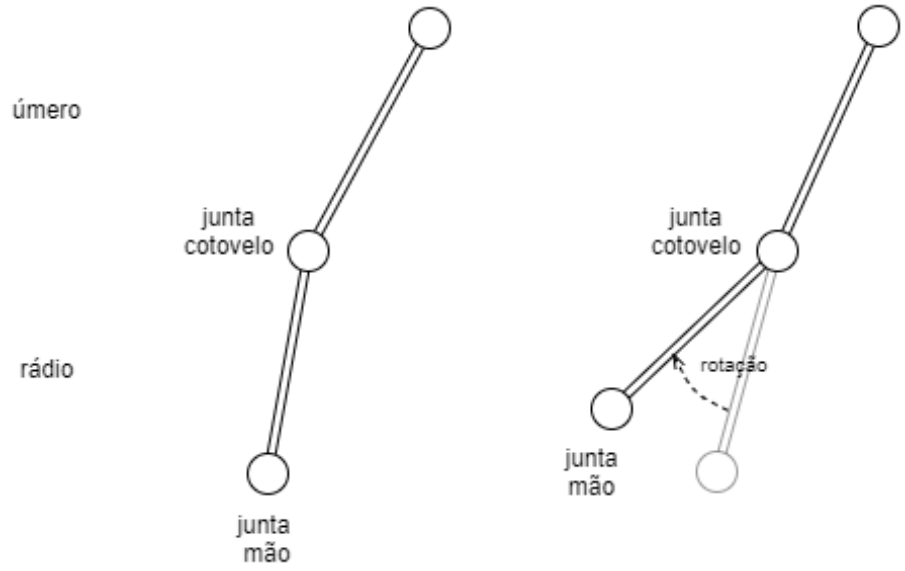


Figura 5.3: Relação entre segmento e junta

De seguida selecciona-se apenas as juntas semelhantes ao esqueleto da Kinect e remove-se as restantes. Na figura 5.5 as juntas numeradas no esqueleto da CMU foram seleccionadas para representar as juntas da Kinect com o respectivo número.

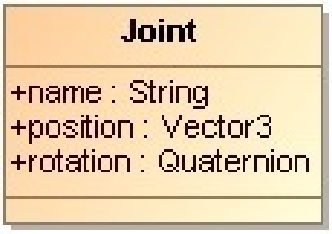


Figura 5.4: Classe "Joint"

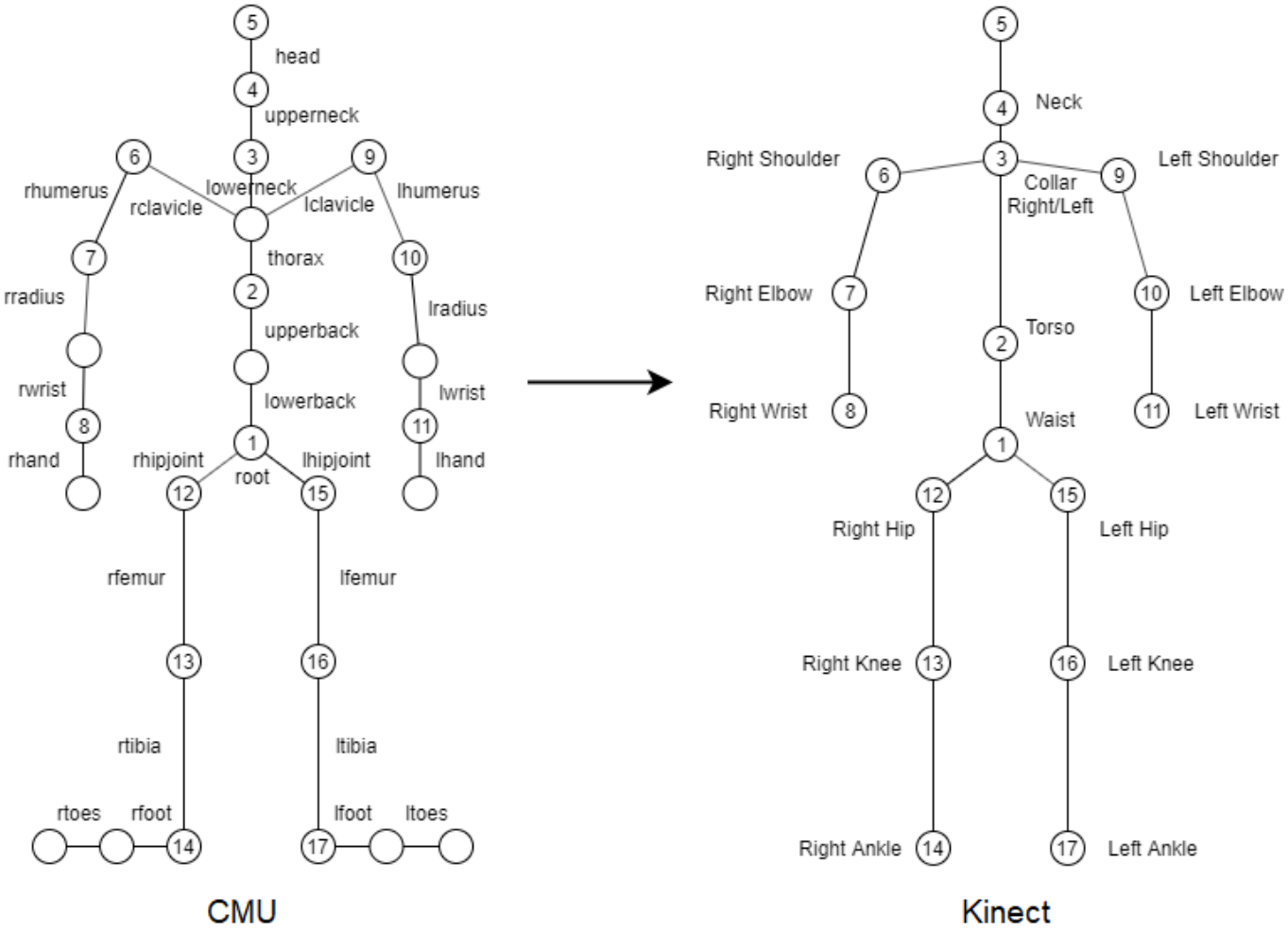


Figura 5.5: Mapeamento esqueleto CMU / esqueleto Kinect

O resultado final da interpretação dos dados é gravado num ficheiro CSV (tabela 5.1) por cada AMC, ou seja, por cada captura. O CSV é constituído pelo nome da junta, as componentes (x, y, z) da posição e a rotação (quaterniões) e o número da *frame*. Em relação aos quaterniões, é usada a biblioteca "transforms3d" que suporta os cálculos desta unidade. Esta biblioteca lida com os quaterniões no

formato (w, x, y, z), diferente da convenção da Kinect e do Unity (x, y, z, w). Para converter ao formato desejado passa-se o canal "w" para último e inverte-se as coordenadas "y" e "z", ficando:

$$[w, x, y, z] \rightarrow [x, -y, -z, w]$$

Joint	Tx	Ty	Tz	Rx	Ry	Rz	Rw	Frame
"nameJoint1"	0	0	0	0	0	0	1	1
. . .								
"nameJoint1"	-49.7	16.0	17.6	0.02	0.91	0.01	0.40	2

Tabela 5.1: Resultado da interpretação do *dataset*

5.4 Pré-processamento dos dados

As poses devem ser representadas num sistema de coordenadas local, sempre com o mesmo ponto de origem, para que não haja distinção entre poses iguais mas com diferentes coordenadas, ou seja, cada pose tem uma única configuração. As funções de pré-processamento convertem as coordenadas globais de uma dada junta para um sistema de coordenadas locais de uma dada origem, neste caso a origem é a cintura do esqueleto, junta "waist". Para cada pose calcula-se a matriz de translação (T) do ponto de origem, neste caso a junta "waist", seguido do vector de rotação Q:

$$T = \begin{bmatrix} 1 & 0 & 0 & Px \\ 0 & 1 & 0 & Py \\ 0 & 0 & 1 & Pz \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

$$Q = \begin{bmatrix} Qx & Qy & Qz & Qw \end{bmatrix} \tag{5.4}$$

A matriz de transformação local é o inverso da matriz global T e a rotação inversa de Q:

$$\begin{aligned} T_{local} &= T^{-1} \\ Q_{local} &= Q^{-1} \end{aligned} \tag{5.5}$$

De seguida multiplica-se a posição de cada junta (J) pela matriz T_{local} e a rotação

por Q_{local} da respectiva pose (N).

$$\begin{aligned} Position_{J,N} &= T_{local,N} \cdot Position_{J,N} \\ Rotation_{J,N} &= Q_{local} \cdot Rotation_{J,N} \end{aligned} \tag{5.6}$$

Com intuito de voltar às coordenadas globais, estas podem ser obtidas através da sua multiplicação pela matriz de translação global T, em função disso são guardadas as matrizes T e rotação Q de cada pose.

As poses são escaladas de forma a uniformizar a altura e não ter que lidar com esqueletos de diferentes tamanho, apenas com diferentes proporções. O factor de escala utilizado é o comprimento médio dos segmentos durante a pose inicial. Para cada pose, percorre-se a hierarquia das juntas e calculam-se as novas posições dada pela função 5.7, onde "$p_{corrente}$" é a posição da junta que se pretende normalizar, "p_{pai}" a posição da junta "pai" e "p_{norm}" a posição corrente normalizada.

$$p_{norm} = p_{pai} + \frac{p_{corrente} - p_{pai}}{meanBoneLen} \tag{5.7}$$

Ou seja, a posição normalizada será o vector que aponta da junta "pai" à junta que se pretende normalizar dividido pelo comprimento médio dos segmentos. A figura 5.6 demonstra um exemplo para a normalização da junta "tronco".

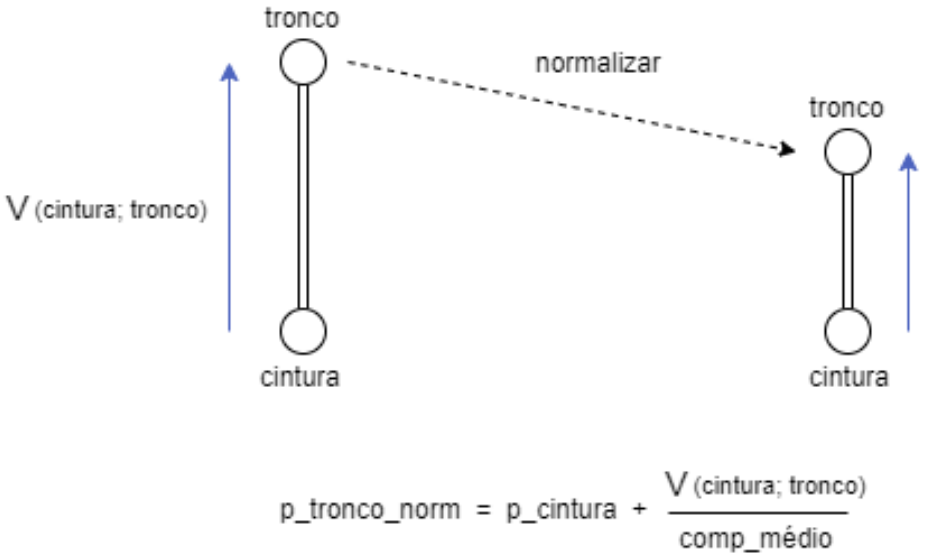


Figura 5.6: Exemplo da normalização da altura da junta "tronco"

A junta "cintura" é a raiz da hierarquia, como tal mantém a sua posição original. As juntas devem ser percorridas em hierarquia, iniciando na raiz, para que

as juntas acima tenham já os valores actualizados. Da mesma forma que o processo anterior, deve-se guardar o comprimento médio dos segmentos, de cada movimento, para voltar aos valores originais.

Por último, as rotações são ajustadas para que os vectores de direcção entre a junta pai e a junta filho fiquem alinhados com o respectivo eixo. Assim, a mesma configuração de rotações representa a mesma pose quer seja no *dataset* da CMU ou nos dados devolvidos da Kinect. O primeiro caso da figura 5.7 ilustra um exemplo onde os eixos de rotação podem não estar alinhados correctamente. Admite-se que o vector de direcção da junta "ombro" deve estar alinhado com o eixo X, sendo que na pose inicial "T" a rotação é $(0, 0, 0)$. Podem haver casos que durante a captura do movimento tenham sido definidos outros vectores de direcção, como apresenta o 2º esquema da figura, nesse caso apesar da pose ser visualmente a mesma o ângulo não será igual ao anterior e na perspectiva da rede neuronal será uma nova pose. Deve ser aplicado um deslocamento na rotação tal que o eixo X coincida com o vector de direcção, como demonstra o 3º esquema e assim a rotação retoma o valor correto.

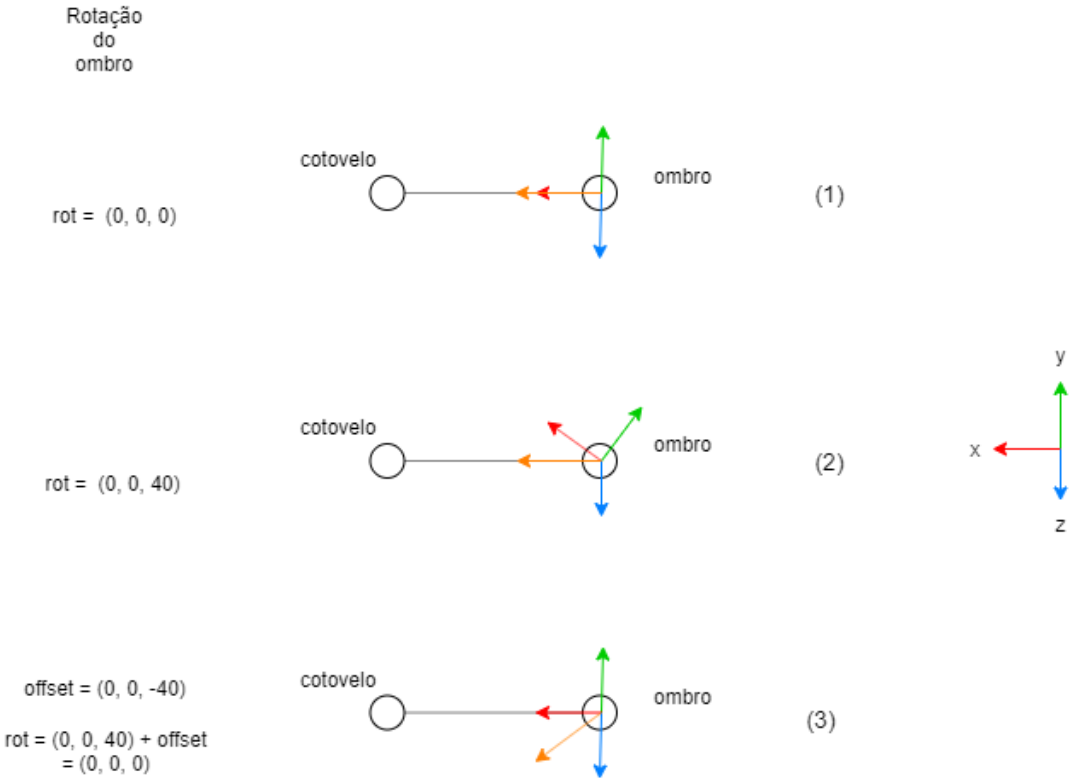


Figura 5.7: Correcção dos eixos de rotação

O eixo que deve-se alinhar com os vectores de direcção depende do tipo de junta. Esses eixos foram definidos da seguinte forma:

- As juntas do braço direito alinham o eixo X;
- As juntas do braço esquerdo alinham o eixo X negativo;
- As juntas da cintura até à cabeça alinham o eixo Y;
- As juntas das duas pernas alinham o eixo Y negativo.

O vector de direcção é dado pela posição local da junta filho, no caso da figura 5.7 o vector é a posição local de "*elbow*". Para determinar o ângulo de rotação, a posição é projectada nos respectivos planos, por exemplo se o objectivo for alinhar o eixo X então projecta-se o ponto no plano XY para rodar em Z e no plano XZ para rodar em Y.

5.5 Rede Neuronal Profunda

O bloco da rede neuronal profunda é utilizado para eliminar os ruído das poses capturadas pela Kinect, para tal são fornecidas as poses originais (*ground truth*) e as ruidosas durante a fase de treino como ilustra a figura 5.8. A rede irá modificar os seus parâmetros ao longo das iterações de maneira a reconstruir as poses originais a partir das poses com ruído.

Nos problemas de redução de ruído as redes do tipo **Autoencoders** são geralmente as mais usadas porque tem características que permitem reconstruir os dados de entrada sem a maior parte do ruído. No entanto, com o surgimento da **ResNet** [29] (*Residual Network*) começou-se a usar cada vez mais esta arquitectura, incluindo na redução de ruído. Tal como em [30] é utilizada uma rede neuronal do tipo ResNet.

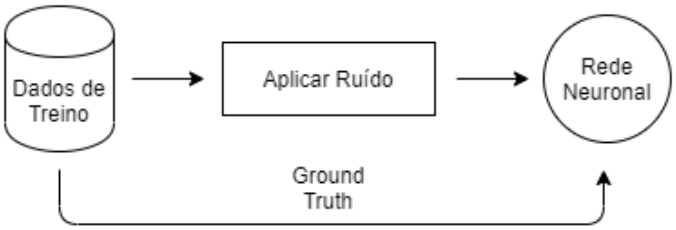


Figura 5.8: Dados de treino para a rede neuronal

5.5.1 Função do Ruído

O sensor Kinect capta a informação da pose com dois tipos de ruído:

- **Oclusões**: As juntas afastam-se muito da posição correta porque estão ocluídas, por um objecto ou parto do corpo humano, no momento da captura.
- **Deslocamentos**: Pequenos desvios da juntas da posição correta, devido a sensibilidade do sensor e o ambiente em que se encontra.

De forma a simular esse ruído nos dados de treino da rede implementou-se o algoritmo 1, baseado no trabalho publicado em [30].

Algoritmo 1 Aplicar ruído nos dados de treino

function CORRUPTPOSE(pose, probCorrupt, probO, probS, maxShift)
  $pC = bernoulli(probCorrupt)$
  $pO = uniform(0, probO) * pC$
  $pS = uniform(0, probS) * pC$
  for feature in pose **do**
    $shift = bernoulli(pS)$
    $occlusion = bernoulli(pO)$
    $magnitude = normal(-maxShift,\ maxShift)$
    $feature = (feature + shift * magnitude) * occlusion$
  end for
end function

Este algoritmo está dividido nas três fases seguintes:

1. **Gerar as probabilidades de haver ruído numa pose**: para uma dada pose geram-se as probabilidades uniformes de acontecer uma oclusão (pO) e de acontecer um deslocamento (pS). Uma vez que é importante haver poses sem ruído, ou seja dados verdadeiros, para que a rede saiba distinguir poses ruidosas das poses corretas, gerou-se também a probabilidade uniforme da pose ser corrompida (pC).
2. **Gerar as probabilidades de haver ruído em cada característica da pose**: para cada característica da pose tiram-se as probabilidades de Bernoulli (espaço amostral discreto de 0 a 1), onde a probabilidade de acontecer uma

oclusão/deslocamento é igual à probabilidade uniforme gerada anteriormente. Se as probabilidades pO, pS ou pC tiverem o valor 0 nenhuma característica dessa posa irá ser alterada, caso contrário cada uma pode ou não ser modificada dependendo da probabilidade de Bernoulli. Deste modo consegue-se uma maior variação no número de juntas ruidosas para as diferentes poses.

3. **Gerar a magnitude dos deslocamentos**: gera-se a magnitude com distribuição gaussiana e aplica-se à respectiva característica. Embora não esteja descrito no algoritmo 1, o valor máximo do deslocamento da rotação é diferente do deslocamento da posição.

As juntas ocluídas tomam a posição (0,0,0) e a rotação (0,0,0,1) enquanto os deslocamentos afastam-se ligeiramente da junta original. As figuras 5.9 e 5.10 ilustram um exemplo dos dois tipos de ruído, onde as juntas corrompidas são apresentadas a vermelho e o *ground truth* a verde. Pode-se ver que na figura da esquerda a junta vermelha que deveria sobrepor a verde foi movida para a origem (0,0,0) e na figura da direita as juntas a vermelho sofrem alguns desvios.

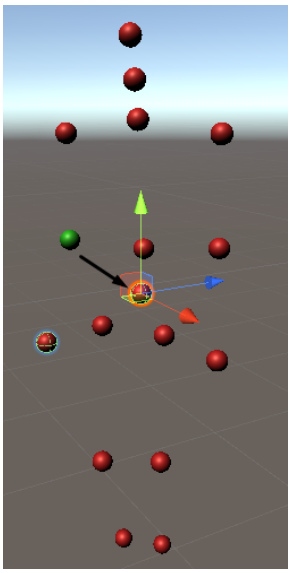


Figura 5.9: Pose com as oclusões geradas

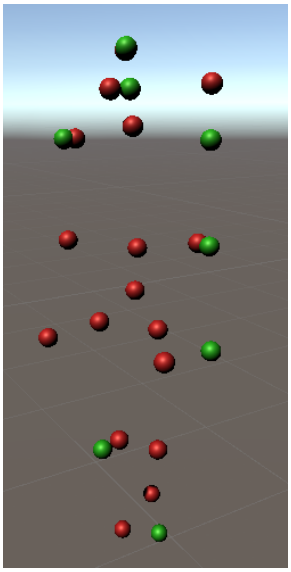


Figura 5.10: Pose com os deslocamentos gerados

Quando uma junta está ocluída é necessário eliminar a rotação da junta "pai" isto porque a Kinect, embora consiga determinar a posição do "pai", não consegue determinar a sua rotação sem saber a posição da junta "filho".

Quando uma junta está ocluída é necessário eliminar a rotação da junta "pai" isto porque a rede tira partido do vector de direcção para determinar a posição da junta ocluída. No caso da Kinect, quando existe uma oclusão, é impossível determinar exactamente a rotação do "pai" sem saber a posição da junta "filho", ficando com um valor incorrecto. Desta forma a rede irá devolver uma posição também incorrecta por isso é necessário, tanto nas poses da Kinect como nas da CMU, alterar a rotação da junta "pai" igual a $[0,0,0,1]$ para que a rede aprenda a corrigir as oclusões sem conhecimento do vector de direcção.

5.6 Filtro Temporal

O filtro Savitzky–Golay [17] suaviza os dados com o propósito de melhorar a precisão sem distorcer o sinal, através da convolução com uma função polinomial de grau n. Dado um conjunto de pontos (x_j, y_j) onde $j = [1, ..., n]$, x é uma variável independente e y é o valor observado, o resultado do filtro é um conjunto de convoluções entre o sinal e os coeficientes C_i, que dependem do comprimento da janela m e do polinómio definido.

$$Y_j = \sum_{i=\frac{1-m}{2}}^{\frac{m-1}{2}} C_i * y_{j+1} \quad , \quad \frac{m-1}{2} \leqslant j \leqslant n - \frac{m-1}{2} \tag{5.8}$$

6

Implementação

Neste capítulo serão apresentadas as etapas da implementação do sistema, dividido em quatro componentes apresentadas no diagrama da figura 6.1.

Inicia-se com o processamento do *dataset* da CMU, descrevendo todo o processo e os ficheiros criados. De seguida, explica-se como é feita a captura das poses com a Kinect através do Unity. É detalhada a sequência do pré-processamento desses dados bem como o código envolvido. Quanto à rede neuronal inicia-se por descrever as directorias e os dados produzidos, seguido da arquitectura do código a implementar. O processo de *tuning* está dividido em secções por cada parâmetro que foi estudado, nas quais são apresentados os resultados para diferente valores e respectiva interpretação. Durante o treino será utilizado o método de validação cruzada (*cross-validation*) para medir com maior precisão a performance do modelo. O conjunto de treino será divido em 6 *k-fold*, aproximadamente 10% dos dados são utilizados na validação. Por último, é explicada a utilização do filtro temporal e o impacto que tem na sequência de poses devolvida pela rede neuronal.

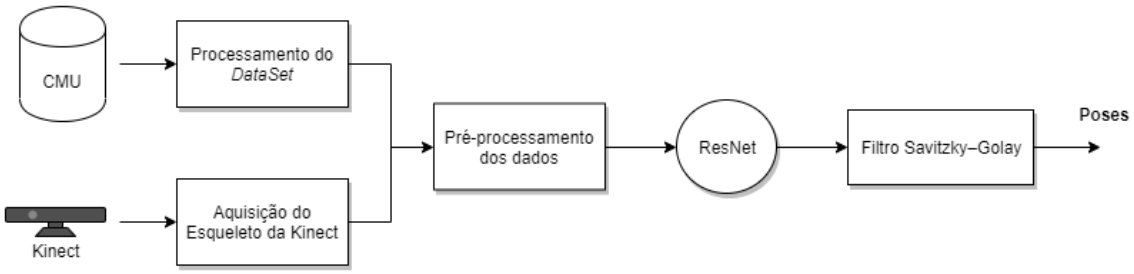


Figura 6.1: Componentes do sistema

6.1 Processamento do *dataset*

Implementou-se uma função em Python para obter automaticamente os ficheiros AMC dos últimos 54 sujeitos (aconselhado pela própria CMU) e os respectivos ASF. A figura 6.2 apresenta um diagrama deste processo juntamente com a nomenclatura utilizada, onde "X" indica o número do sujeito e "*trial*" o número da captura. De seguida, para cada ficheiro AMC gera-se o ficheiro CSV com informação do esqueleto adaptado para a Kinect e o formato indicado no capítulo 5 que será utilizado no treino da rede. Os dados são divididos em treino e teste, de forma a que o treino contenha 90% das poses, ficando com 612 movimentos na directoria "*train*" e 76 na directoria "*test*".

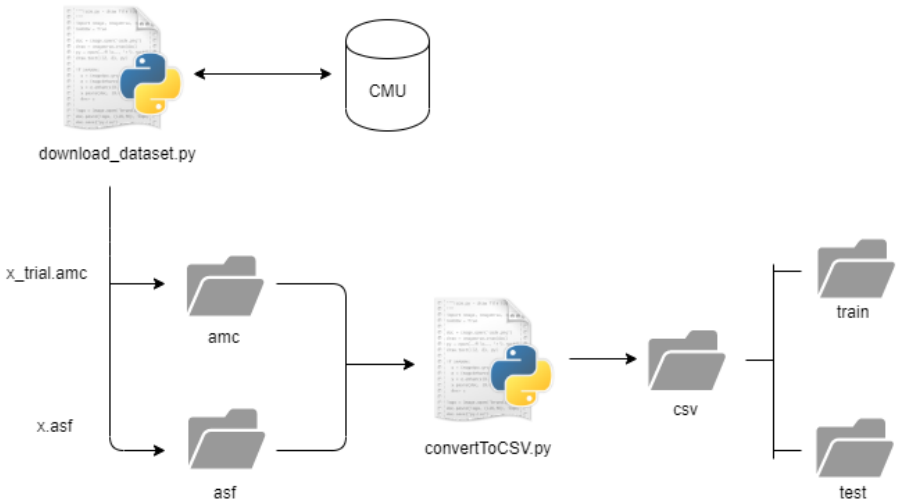


Figura 6.2: Obtenção automática do dados

Este módulo agrupa um conjunto de ficheiros Python de forma coesa, como demonstra o diagrama da figura 6.3. O ficheiro "parse_data.py" implementa os

algoritmos para a leitura dos dados AMC/ASF descritos no capítulo 4 e devolve um dicionário onde a chave é o nome do segmento e o valor é o objecto ”Bone” correspondente ou um *array* contendo os valores da animação ($[px, py, pz, rx, ry, rz]$).

O ficheiro ”skeleton.py” implementa as duas classes que representam os dados e duas funções auxiliares para obter a pose inicial ”T” e converter o segmento para junta. A classe ”Bone” representa um segmento que actualiza os seus atributos, conforme descrito nos capítulos 4 e 5, através da função ”setMotion” que recebe o *array* da animação referido anteriormente. A classe ”Joint” tem como única função representar os dados da junta.

O ficheiro ”kinect_joints.py” implementa uma função auxiliar para definir os nomes apropriados das juntas. Por último, o ficheiro ”convertToCSV.py” faz uso das funcionalidade implementadas com o objectivo de converter cada ficheiro AMC e o respectivo ASF para o formato CSV. A figura 6.4 apresenta o diagrama de sequência deste processo. Inicia-se com a leitura do ASF e AMC, resultando na variável ”skeleton” e ”animation”. Para cada *frame* da animação aplica-se o movimento ao esqueleto como explicado no capítulo 4. De seguida convertem-se os segmentos em juntas, modificam-se os nomes e adicionam-se as juntas ao ”DataFrame” da biblioteca *Pandas* de análise de dados para Python [13]. Esta biblioteca permite analisar e manipular os dados de forma mais flexível e rápida bem como ler e escrever ficheiros CSV. No fim do ciclo, converte-se o DataFrame para um ficheiro CSV com o mesmo nome do AMC, onde cada linha contém o nome, o número da *frame*, a posição e a rotação de uma junta no sistema de coordenadas globais.

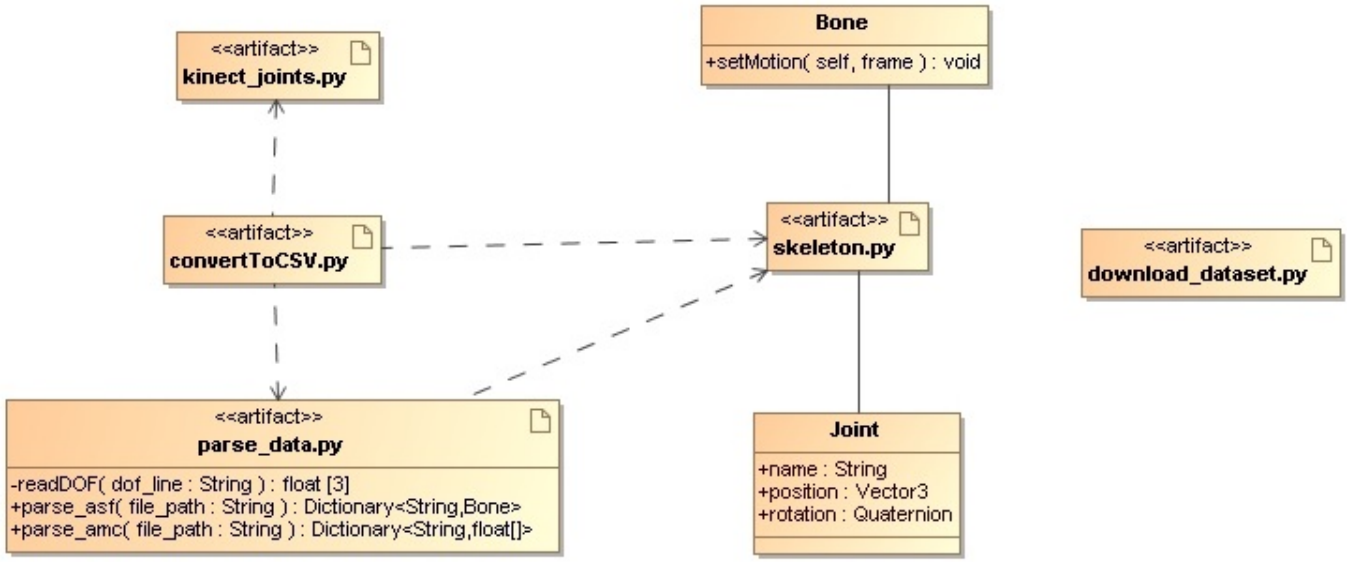


Figura 6.3: Estrutura dos artefactos criados para a interpretação dos dados da CMU

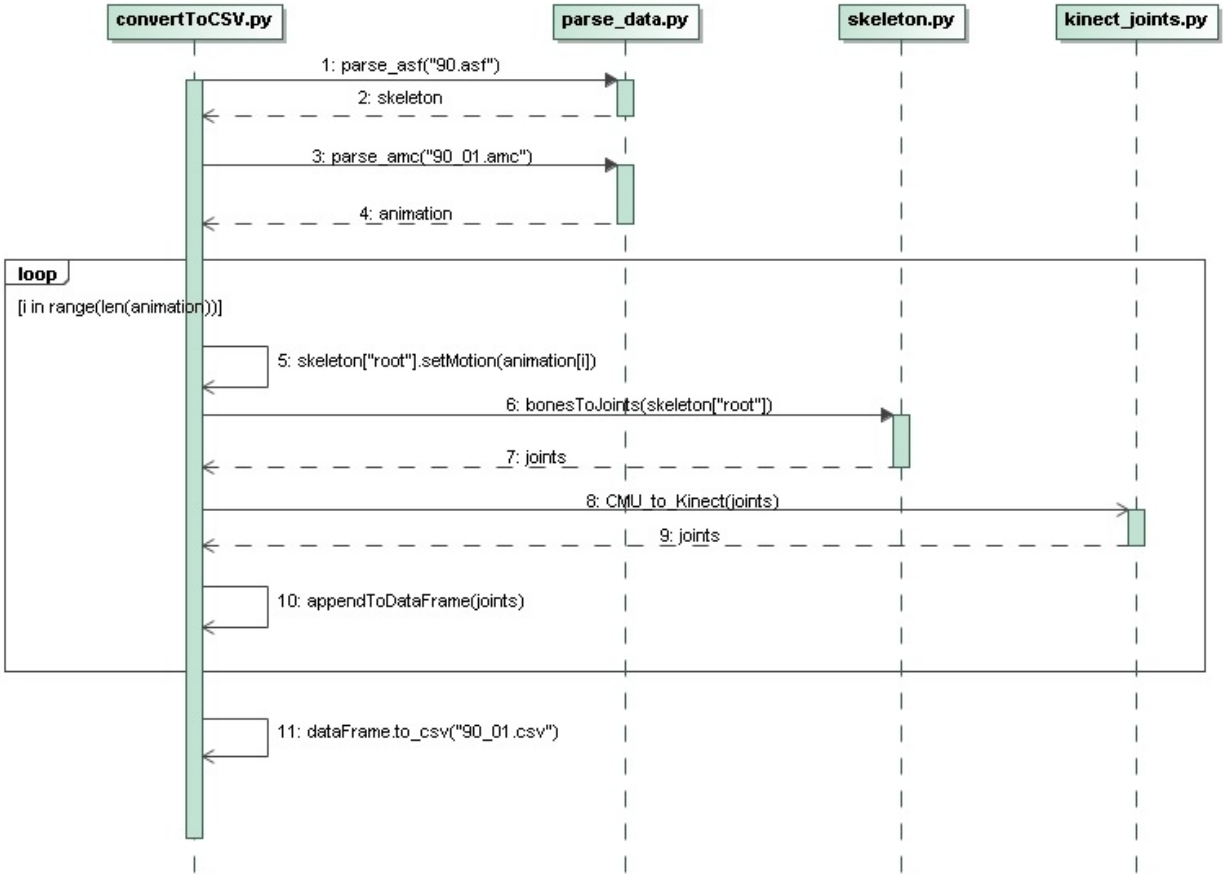


Figura 6.4: Diagrama de sequência da interpretação dos dados da CMU

6.2 Aquisição do esqueleto da Kinect

A captura do movimento do corpo humano foi efectuada através da API Nuitrack disponível para o Unity. Depois de seguir o processo de instalação descrito no site oficial [12], implementou-se a classe ”KinectRecorder” que obtém o esqueleto capturado pela Kinect a cada iteração e quando a aplicação termina escreve a informação das juntas num ficheiro CSV com o mesmo formato referido anteriormente.

A API Nuitrack tem a sua própria classe que representa o esqueleto (”Skeleton”) que por sua vez contém um conjunto de objectos que representam as juntas (”Joint”) possíveis de obter através do seu tipo ”JointType”. Para cada junta do esqueleto obtém-se a sua posição (”Vector3”), a sua rotação (”Quaternion”) e o parâmetro de confiança (”float”). o ”Vector3” da posição, o ”Quaternion” da rotação e o parâmetro *confidence*. Se este último for igual a 0, a junta encontra-se

ocluída e por isso é adicionada a *flag* "Occlusion" que tem como propósito facilitar o pré-processamento descrito mais à frente.

Para visualizar os movimentos numa personagem virtual, foi implementado um conjunto de novas classes, apresentadas na figura 6.5, incluindo para representar o esqueleto e as juntas de uma forma mais simples, sem estar dependente da API Nuitrack. A personagem que se pretende animar deve conter o *script* "ResnetTests" que actualiza o valor das juntas com base num dado ficheiro CSV interpretada pela classe "CSVReader" no início da aplicação. Essa interpretação resulta numa lista de "KinectSkeleton" que será percorrida durante as iterações da aplicação, de forma a que em cada *frame* a personagem tenha a pose dada pelo novo esqueleto.

As variáveis "updateRotation" e "updatePosition" da classe "ResnetTests" servem o propósito dos dois tipos de mapeamento. O mapeamento directo utiliza a posição e rotação das juntas capturadas. Este mapeamento é o ideal caso o esqueleto a animar tenha as mesmas proporções do utilizador, caso contrário haverá deformações no modelo. O mapeamento indirecto utiliza apenas as rotações das juntas capturadas sendo possível animar qualquer tipo de esqueleto independentemente das proporções, no entanto não causa uma experiência tão imersiva como o mapeamento directo. É preciso ter em conta que para o mapeamento indirecto o esqueleto deve estar na pose inicial "T", caso contrário as rotações aplicadas não vão transformar o esqueleto na pose desejada.

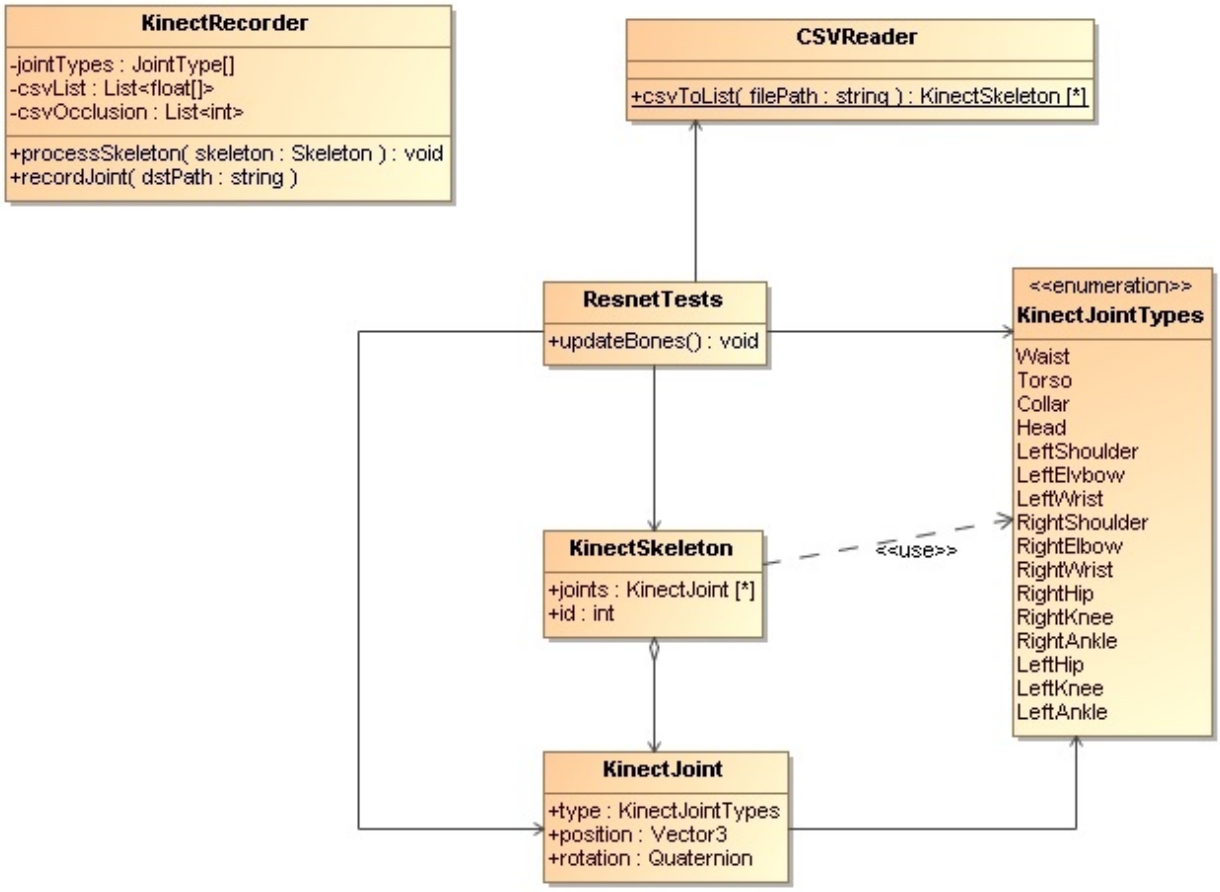


Figura 6.5: Diagrama de classes para a captura do movimento com a Kinect

6.3 Pré-processamento dos dados

O pré-processamento descrito no capítulo 5 foi implementado nos quatro ficheiros Python identificadas na figura 6.6. O ”DatasetLoader.py” implementa a classe ”DatasetLoader” que disponibiliza funções para carregar os dados de treino e teste. O ”Normalizer.py” implementa as funções de normalização, nomeadamente a transformação das coordenadas globais em locais, a normalização da altura pelo comprimento médio dos segmentos e a transformação dos vectores de direcção. O ”ErrorGenerator.py” implementa o algoritmo de corrupção das poses, disponibiliza a função ”corruptPosesUniform()” para corromper um dado conjunto de poses de acordo com os parâmetros fornecidos.

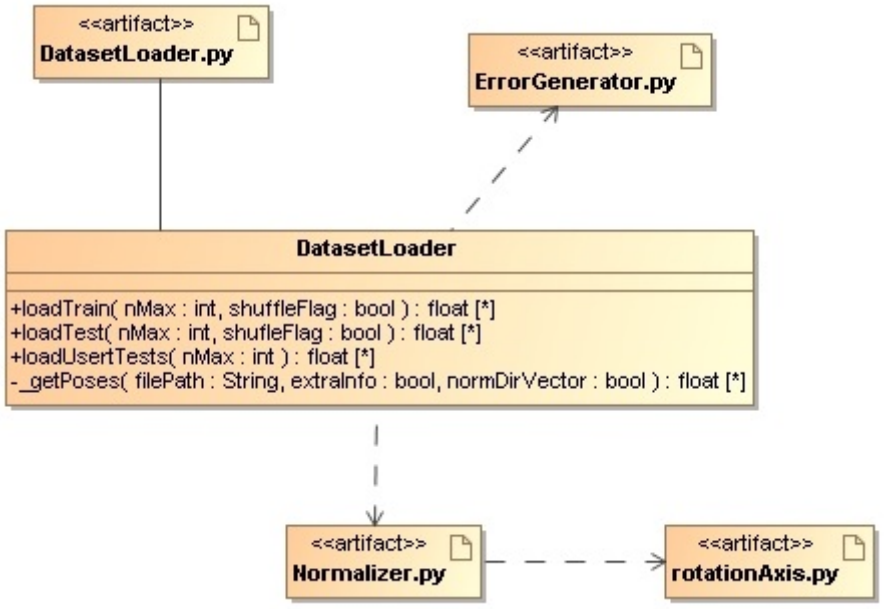


Figura 6.6: Diagrama dos artefactos criados para o pré-processamento dos dados

A função "loadTrain()" da classe "DatasetLoader" agrega numa lista as poses contidas em cada ficheiro CSV da directoria dos dados de treino. Cada CSV é passado à função "_getPoses()" que o processa de acordo com o diagrama da figura 6.7. Inicia por normalizar os vectores de direcção alterando a rotação de cada junta, como referido no capítulo 5. De seguida, para cada pose, determina a matriz de transformação local da cintura (junta *waist*) e converte todas as juntas dessa pose para esse sistema de coordenadas. O comprimento médio dos segmentos é calculado na pose inicial, sendo reutilizado nas restantes poses do movimento (ficheiro CSV). Normaliza-se a altura do esqueleto com o comprimento calculado e adiciona-se à lista das poses processadas. Terminada a iteração das poses e caso estas pertençam à Kinect, aplica-se as modificações às poses ocluídas, a partir da função "_applyOclusion()", isto porque a Kinect não devolve as juntas ocluídas com o valor pretendido para o treino da rede. Esta última função modifica a posição e rotação das juntas que tenham a *flag* "Occlusion" para os valores $[0, 0, 0, 0, 0, 0, 1]$ e rotação da junta *parent* a $[0, 0, 0, 1]$ como foi visto no capítulo 5.

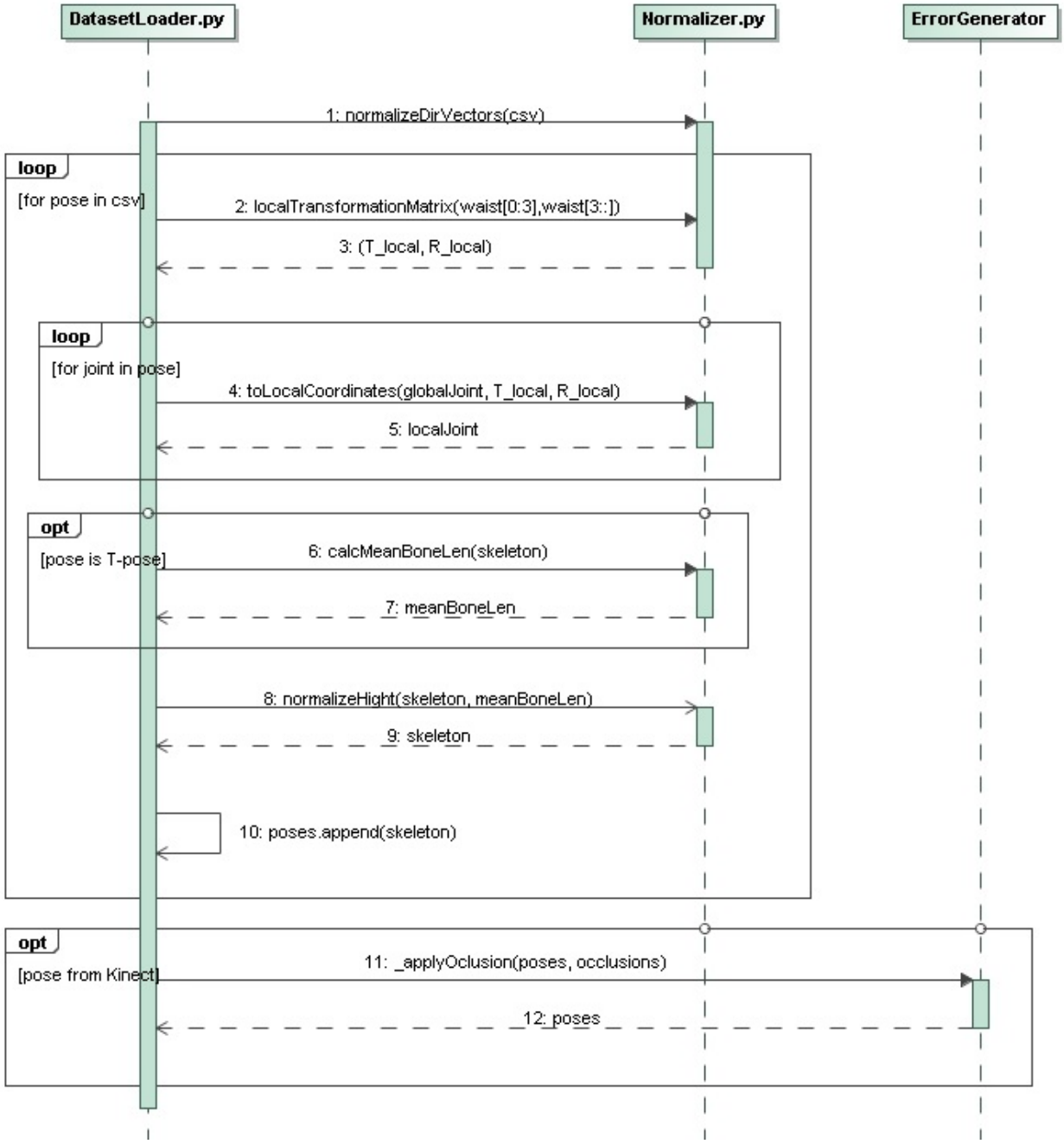


Figura 6.7: Diagrama de sequência do carregamento dos dados de treino

No final obtém-se uma lista de poses, constituídas por um conjunto de juntas que por sua vez se representam com a sua posição ($t = [tx, ty, tz]$) e rotação ($r =$

$[rx, ry, rz, rw]$) representada por quaterniões. Decompondo a pose tem-se:

$$\begin{aligned} junta &= [tx, ty, tz, rx, ry, rz, rw] \\ pose &= [junta_1, junta_2, ..., junta_n] \\ pose &= [tx_1, ty_1, tz_1, rx_1, ry_1, rz_1, rw_1, ..., tx_n, ty_n, tz_n, rx_n, ry_n, rz_n, rw_n] \end{aligned} \tag{6.1}$$

O facto da junta *waist* ser a origem faz com que esta tenha sempre o mesmo valor $[0, 0, 0, 0, 0, 0, 1]$, por isso pode ser removida, ficando no total com 16 juntas. Assim a pose é definida por um vector de $16 * 7 = 112$ características.

Os dados para o treino da rede (poses CMU) são pré-processados uma única vez e guardados no formato binário da biblioteca "Numpy", como apresenta a figura 6.8. Depois de carregar todos os dados de treino a partir da classe "DatasetLoader", baralha a ordem das poses para que os dados de treino e de validação estejam uniformente distribuídos em termos de tipo de movimento (por exemplo, correr ou saltar), ou seja, para que os dois conjuntos de dados não contenham apenas um único tipo de movimento pois isso irá enviesar os resultados. É gerado o ficheiro "motionData.npy", contendo as poses originais, e três ficheiros corrompidos com os diferentes tipos de ruído, mantendo a ordem das poses depois da função *shuffle*. Com isto evita-se processar os dados múltiplas vezes e garantem-se as mesmas condições para todos os modelos.

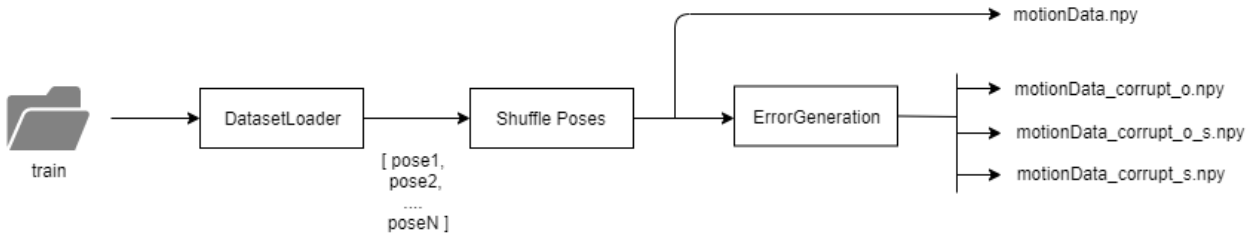


Figura 6.8: Processamento dos dados de entrada

6.4 Rede Neuronal

Nesta fase são gerado uma grande quantidade de dados que devem ser armazenados, como os modelos da rede com diferentes configurações, dados de entrada processados com diferentes tipos de ruídos (ver figura 6.8) e os resultados da rede. Para isso definiu-se as directorias da figura 6.9. Cada modelo gerado

na fase de treino é guardado numa nova pasta com o seu nome, contendo os ficheiros necessários para o reconstruir, nomeadamente os parâmetros (número de *layers*, número de neurónios, função de activação, etc.), os pesos resultantes do treino, a pose média e o desvio padrão caso seja necessário normalizar os dados de entrada. São armazenados todos os resultados para os diferentes valores dos parâmetros, incluindo o erro por época (curva de aprendizagem) e os hiperparâmetros utilizados.

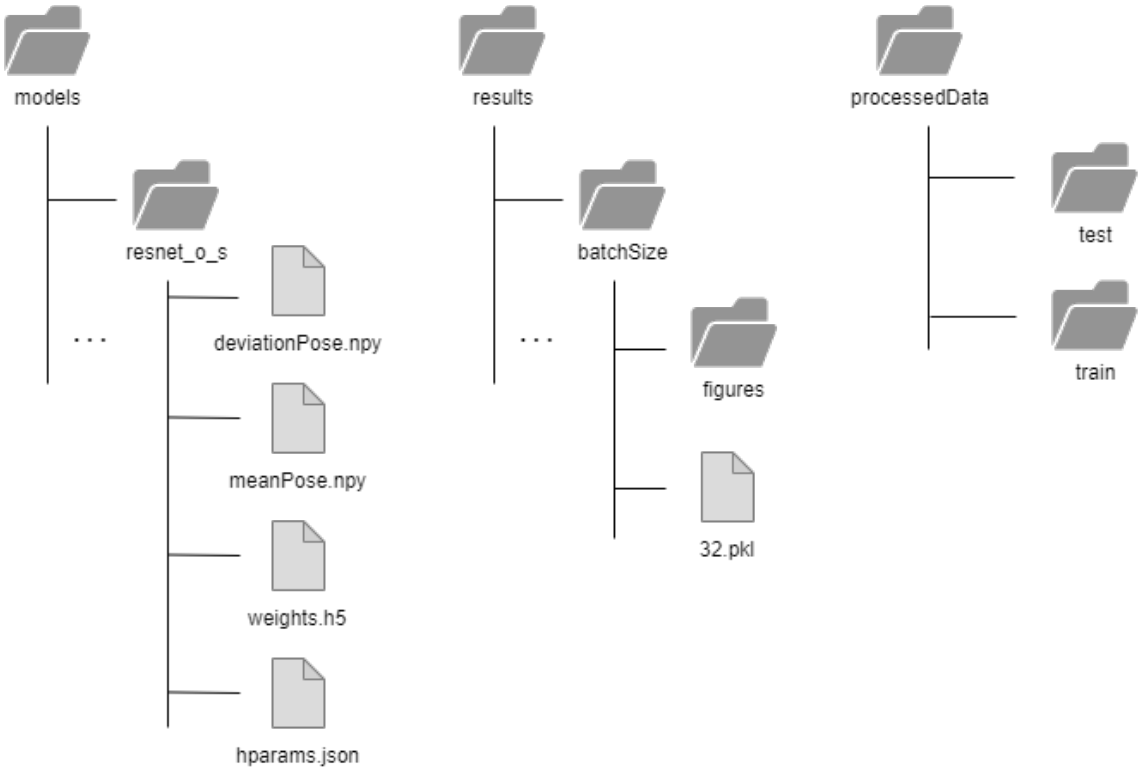


Figura 6.9: Directorias para armazenar os dados produzidos pela rede

Para esta fase foram criado três ficheiros Python, apresentados na figura 6.10. O "NetModels.py" implementa as classes para a construção da rede neuronal com o objectivo de abstrair todo esse processo durante a fase de *tuning*. No "main_tests.py" são realizados os testes para diferentes hiper-parâmetros e dados de entrada. Os modelos e resultados são armazenados nas respectivas directorias acima mencionadas. Este ficheiro tem acesso às funcionalidades de desnormalização (ficheiro "Normalizer.py") para transformar as poses devolvidas pela rede nas condições originais e assim poderem ser observadas no Unity. O "readResults.py" faz a leitura automática dos resultados, apresentando os gráficos e tabelas apropriadas.

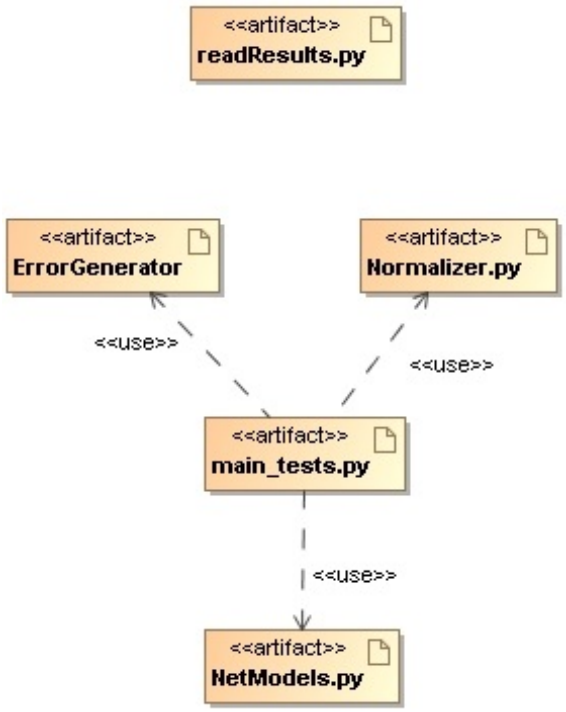


Figura 6.10: Ficheiros Python utilizados na implementação da rede neuronal

A rede neuronal foi implementada através da API TensorFlow (versão 1.14.0), integrando a API Keras que disponibiliza uma interface de mais alto nível para o desenvolvimento de redes neuronais. A API Keras não oferece um modelo do tipo ResNet, por isso foi criada uma estrutura de classes própria (figura 6.11), que implementa a arquitectura da ResNet baseada no modelo sequencial e ao mesmo tempo facilita a gestão dos modelos (guardar e carregar modelos anteriores). Para suportar diferentes arquitecturas da rede criou-se uma classe base, designada por "CustomModel", que implementa as funcionalidades gerais do modelo. A classe "ResNetModel" estende essas funcionalidade e utiliza as *layers* "ResidualBlock" no modelo sequencial. A classe "SimpleModel" segue a mesma abordagem mas com *Dense Layers* disponibilizadas pela API. As configurações do modelo são indicadas na classe "HyperParameters" de forma a agrupar todos os parâmetros num único objecto. Esta estrutura possibilita uma grande abstracção na fase de *tuning* da rede, consegue carregar modelos anteriores a partir das configurações e dos pesos guardados, mesmo que seja utilizada a normalização "z-score" porque esta é realizada no próprio modelo, do tipo "CustomModel".

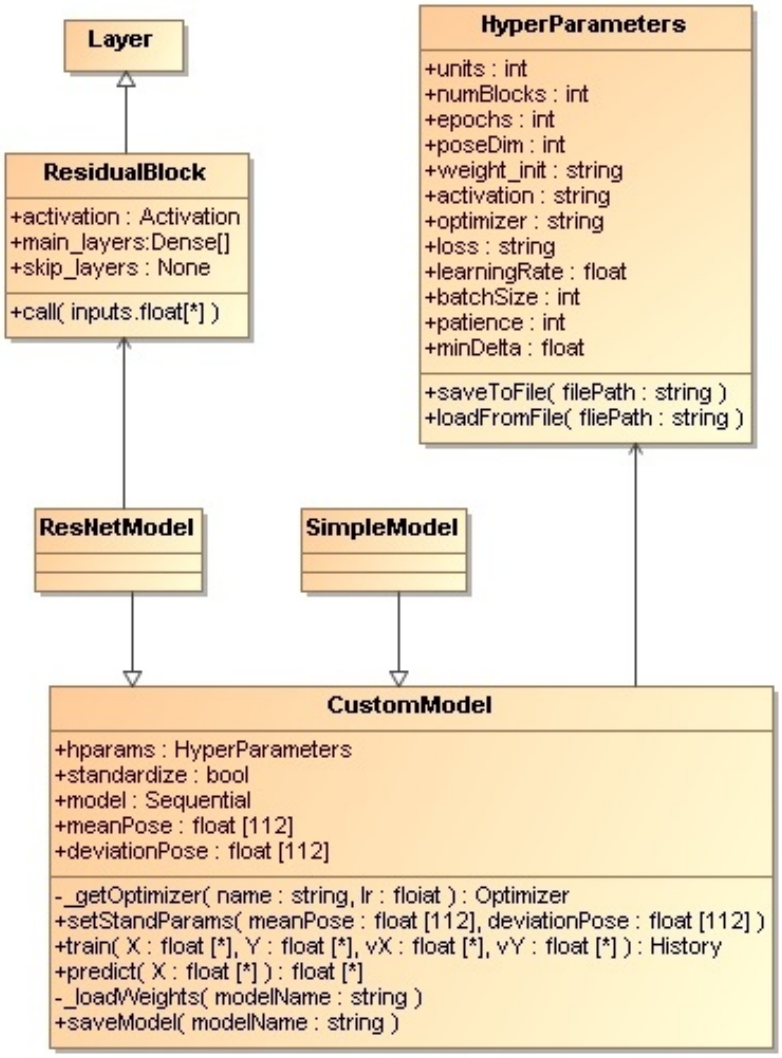


Figura 6.11: Classes para a construção da rede neuronal

Definir os hiper-parâmetros ideais pode ser uma tarefa complexa. Testar todas as suas combinações possíveis é algo impraticável pois são imensas e requer grande poder computacional. Um método conhecido, designado por "*Randomized Search*", consiste em testar um conjunto de combinações aleatórias e funciona razoavelmente bem para problemas simples, no entanto, podem não ter sido testadas as combinações mais adequadas. Optou-se por um método menos dispendioso, testando um hiper-parâmetros de cada vez e só fazer combinações com aqueles que tenham uma relação, como foi apresentado no capítulo 3 (por exemplo, entre a inicialização dos pesos e a função de activação). Irá ser utilizado a metodologia "*zoom*" que segue o principio que se uma região de parâmetros apresentar-se promissora, deve ser explorada, aumentando a discretização da sua variação.

A primeira etapa para a construção de uma rede neuronal é definir o seu tipo de

arquitectura. Inicialmente testou-se 3 modelos diferentes, um com 2 blocos residuais e 50 neurónios, outro com 5 blocos e 500 neurónios e por último 10 blocos com 1000 neurónios, de forma a encontrar o modelo inicial que se adeque melhor ao problema. Foi também alterado o número de dados do conjunto de treino para estas redes. Os hiperparâmetros usados nesta fase são os mais comuns para os problemas de regressão sugeridos no capítulo 3, ou seja, treinou-se cada modelo durante 15 épocas, com uma distribuição aleatória uniforme dos pesos, a função de activação "relu", o optimizar "nadam", a função de erro "Mean Absolute Error" (MAE) e o tamanho do *batch* de 32 amostras.

O gráfico da função de perda, figura 6.12, em relação ao número de amostras indica que são necessários pelo menos 20000 amostras para que o erro estabilize. Os seguintes testes serão realizados com um conjunto de 150000 amostras em que 15% é usado na validação e sendo que modelo de 5 blocos residuais, cada um com uma *dense layer* de 500 neurónios obteve o melhor resultado, será utilizado esse modelo como ponto de partida.

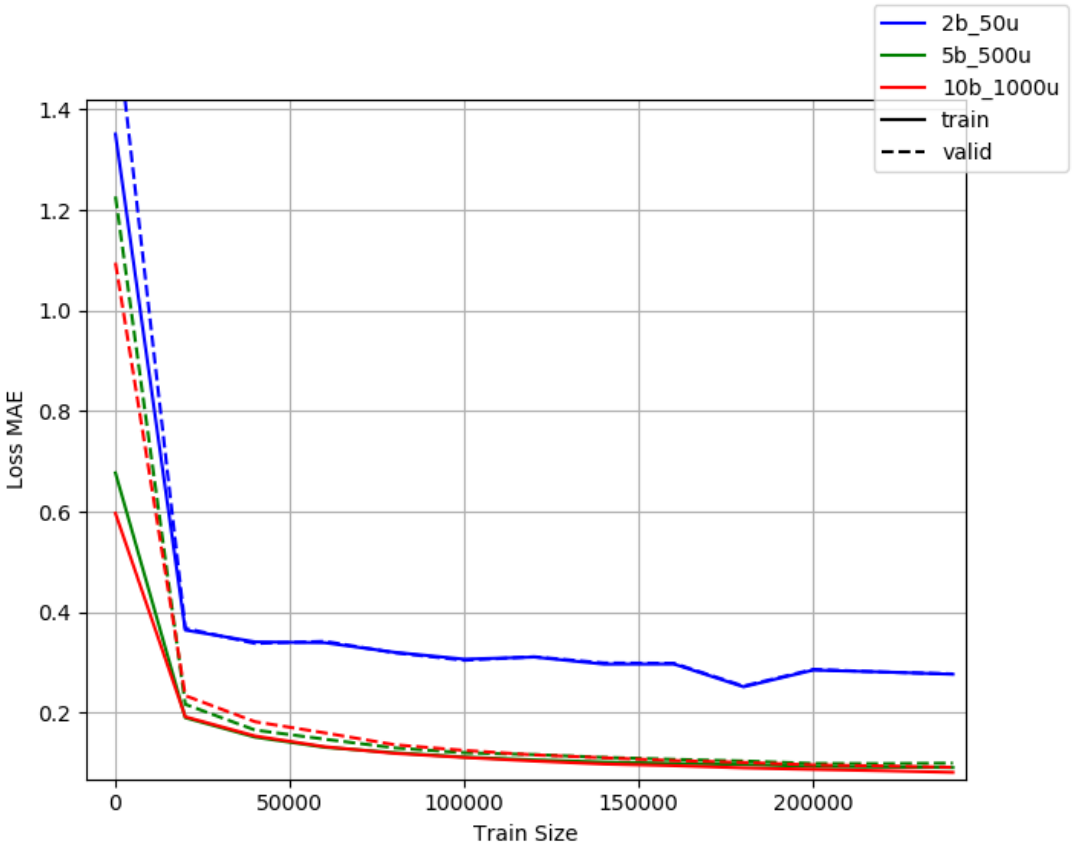


Figura 6.12: Curvas de aprendizagem para diferentes tamanhos do conjunto de treino

De seguida observaram-se os resultados do modelo escolhido com e sem normalização "**z-score**" (ver equação 6.2). Sendo que os valores numéricos dos dados normalizados estão num intervalo diferente dos dados sem a normalização, a função de perda fica também com diferentes intervalos tornando-se impossível comparar os dois modelos. Para resolver este problema é necessário normalizar os dados para o intervalo [0,1]. Uma outra hipótese seria usar a função de perda MAPE (ver equação 6.3), que calcula o erro em percentagem, mas devido a sua fórmula não é possível porque existe divisão por 0 nos dados não normalizados.

$$[H]X_{train} = \frac{X_{train} - meanPose}{deviationPose} \tag{6.2}$$

$$MAPE = 100 * \frac{abs(y_{true} - y_{pred})}{y_{true}} \tag{6.3}$$

Quanto à desnormalização, em vez de modificar as funções de erro implementadas pela API Keras para que tenham em conta o processo de desnormalização, optou-se por uma forma mais fácil normalizando também o *ground truth* (ver algoritmo 2) e assim a medição do erro é feita com ambos os dados projetados. Existem casos em que a própria rede aprende a desnormalizar, mas para este caso a descida do erro não é significativa ao contrário do processo implementado como demonstra o gráfico da figura 6.13. A linha azul apresenta o modelo sem a normalização, a vermelha verde o modelo com os dados de treino normalizados e a linha verde com os dados de treino e o *ground truth* normalizados.

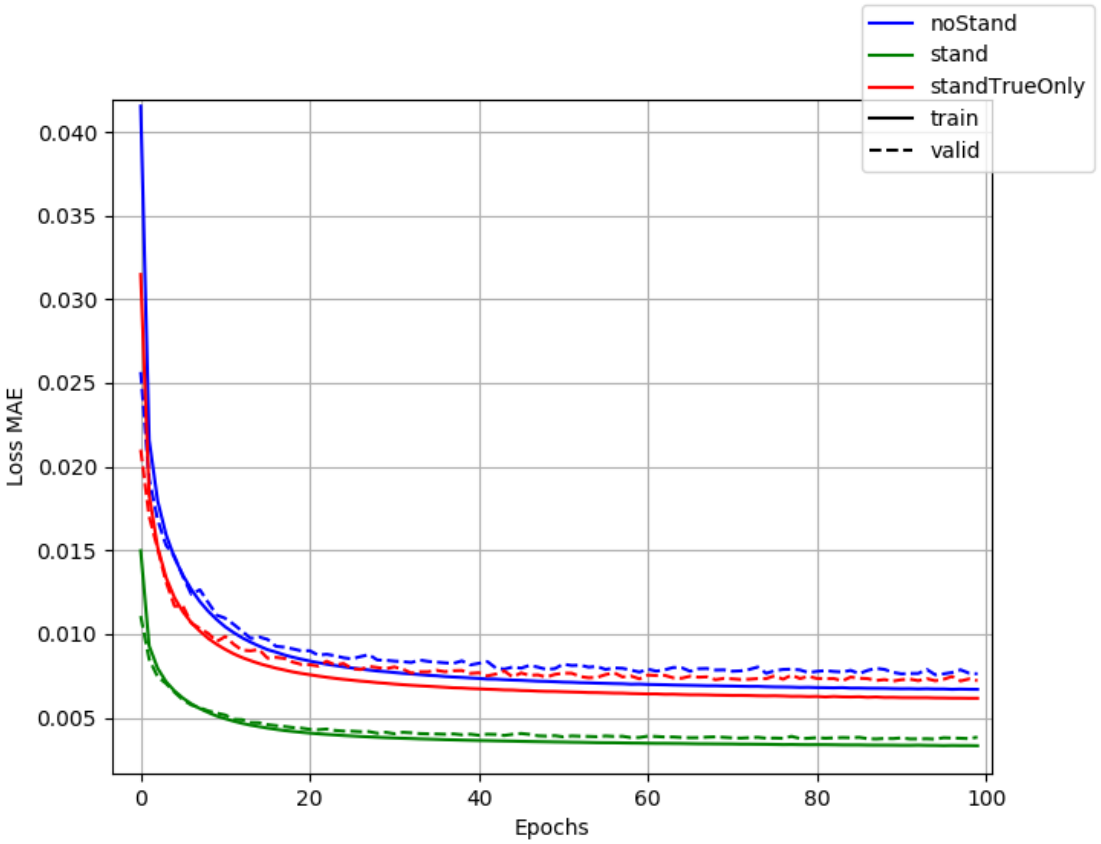


Figura 6.13: Curvas de aprendizagem para modelos com/sem normalização (*z-score*)

Algoritmo 2 Processo de normalização/desnormalização

1: $X = trainData$
2: $\hat{X} = corrupt(X)$
3: $X = (X - X_\mu)/X_\theta$
4: $\hat{X} = (\hat{X} - X_\mu)/X_\theta$
5: $Y = ResNet(\hat{X}, X)$
6: $Y = (Y * X_\theta) + X_\mu$

Repetiu-se o treino do modelo com os dados normalizados (treino e *ground truth*) durante 500 épocas, de forma a visualizar o comportamento do modelo. Como se pode ver na figura 6.14 não existe *overfitting*, tanto o treino como a validação decrescem ao longo das iterações.

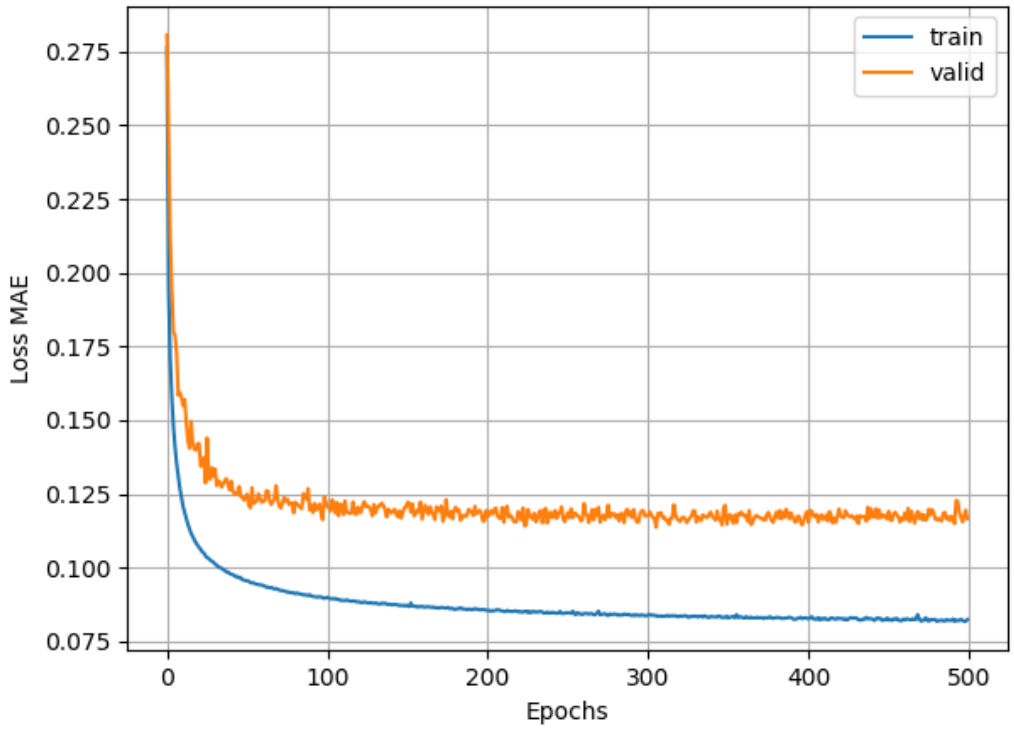


Figura 6.14: Curva de aprendizagem do modelo normalizado

Será usado o método **EarlyStopping** que interrompe o treino quando o valor da função de erro é inferior a um valor predefinido e durante um número de épocas (parâmetro *"patience"*) também predefinido.

A fim de determinar o parâmetro "*patience*" gerou-se a tabela 6.1 que apresenta o número de épocas entre cada descida da função de erro para o conjunto de validação e a respectiva diferença. Para conseguir o menor erro este método deveria definir o "*patience*" maior ou igual a 82, mas este é um número elevado, o que tornará o processo de *tunning* muito lento, sobretudo durante a *cross-validation*. Optou-se por definir o parâmetro "*patience*" a 20 que oferece uma boa relação entre o tempo de processamento e os mínimos locais obtidos.

Épocas	Decréscimo do erro
1	0.000000
1	0.035310
1	0.017780
...	
5	0.000920
2	0.000390
1	0.000950
1	0.000190
1	0.000760
1	0.000290
1	0.000850
3	0.001850
1	0.000790
19	0.000080
8	0.001620
18	0.002330
75	0.000560
38	0.000300
13	0.001310
82	0.000420

Tabela 6.1: Número de épocas até o erro voltar a diminuir e a respectiva diferença

6.4.1 Função de erro

Nesta fase pretende-se perceber qual das funções de erro, nomeadamente "**Mean Absolute Error**" e "**Mean Squared Error**" (MSE) se adequa melhor ao problema. Dada a natureza das duas expressões, a função MAE é linear em relação ao erro,

mas a MSE é quadrática, ou seja, pesa mais erros nas juntas com maior valor o que penaliza as oclusões face a pequenos deslocamentos.

A partir de dois modelos, um treinado com MAE e outro com MSE, compararam-se os resultados obtidos de um mesmo conjunto de teste, mas com diferentes tipos de ruído de forma a perceber como se comportam perante as diferentes situações. O gráfico da figura 6.15 apresenta os resultados com as funções de MAE (do lado esquerdo) e MSE (do lado direito) obtidos dos dois modelos treinados com os 2 tipos de ruído (oclusões e deslocamentos) em conjunto, para os 3 testes diferentes (oclusões e deslocamentos, só oclusões, só deslocamentos) onde o erro entre as poses corrompidas e as originais é apresentada a vermelho, o erro do modelo com MAE a azul e o erro do modelo MSE a azul escuro.

É importante frisar que as funções MAE e MSE são utilizadas em duas situações diferentes. Como função de perda para treinar a rede e como função de avaliação entre os resultados obtidos à saída da rede e o *ground truth*.

Tendo em conta os gráficos, a rede consegue diminuir o ruído e suavizar as altas frequências, à exceção do teste só com deslocamentos que pelo contrário aumenta independentemente do modelo ter sido treinado com MAE ou MSE. Isto pode ocorrer devido ao número de iterações e ao conjunto de dados de treino, como irá ser analisado mais à frente.

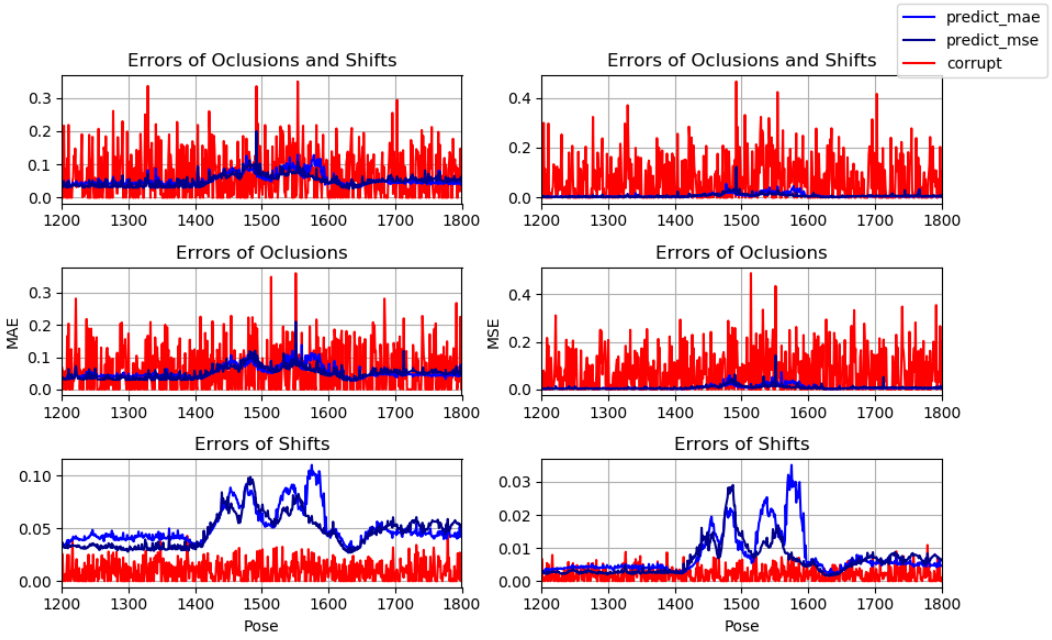


Figura 6.15: Erro MAE (esquerda) e MSE (direita) entre o *ground truth* e as poses devolvidas pelas redes treinadas com diferentes funções de perda

A tabela 6.2 apresenta a soma dos erros para todas as poses onde se observa que nas duas primeiras hipóteses de ruído, o erro MAE consegue melhores resultados independentemente se esse erro é medido com MAE ou MSE. Conclui-se que a medição MAE é mais adequada para o problema e por isso será utilizada nos teste seguintes.

	Teste com Oclusões e Deslocamentos		Teste com Oclusões		Teste com Deslocamentos	
	MAE	MSE	MAE	MSE	MAE	MSE
Modelo treinado com MAE	85.043	10.585	82.904	10.262	79.474	8.990
Modelo treinado com MSE	88.070	11.140	85.457	10.603	79.192	8.547
Dados corrompidos	143.286	154.031	130.544	154.577	20.348	4.254

Tabela 6.2: Soma total dos erros MAE e MSE

6.4.2 Arquitetura da rede

A arquitectura da ResNet é constituída por blocos residuais que por sua vez contém uma ou mais *layers*. Seguiu-se o exemplo do artigo publicado por Daniel et

al. [30] na definição dos blocos residuais, ficando cada um com uma *Dense Layer* e uma *skip connection* sem *layers* adicionais como ilustra a figura 6.16.

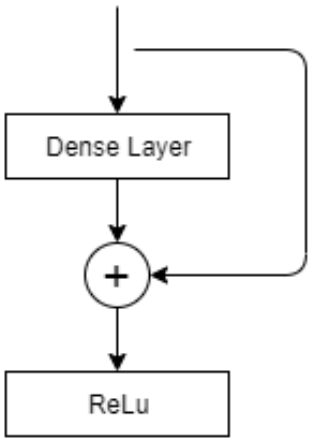


Figura 6.16: Bloco residual implementado

A figura 6.17 demonstra a arquitectura da rede neuronal implementada no mesmo artigo, onde "m" representa o número de marcadores e "j" o número de juntas. Neste projecto será implementada uma arquitectura semelhante, excluindo os marcadores e alterando o número de blocos residuais os neurónios de cada camada (*Dense Layer*).

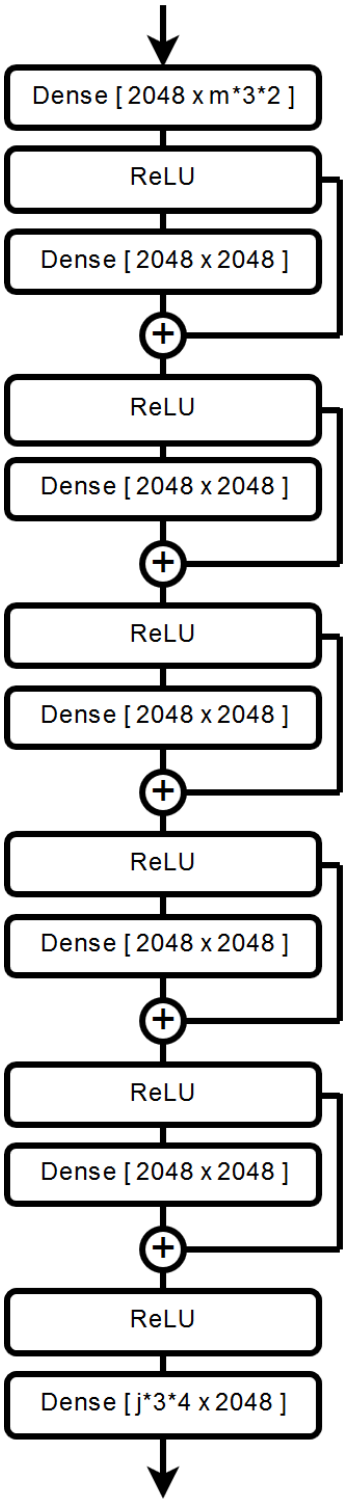


Figura 6.17: Rede neuronal implementada em Daniel et al. [30]

As *layers* captam os diferentes níveis da estrutura dos dados e por isso inicia-se pela procura do seu número ideal, para tal treinaram-se 5 modelos com diferentes quantidades de blocos residuais, mas nas mesmas condições anteriores (500

neurónios em cada *layer* e 200 épocas com *EarlyStopping*). O resultado da *cross-validation* apresentado na figura 6.18 mostra que a rede com um único bloco residual é o mais adequado ao problema.

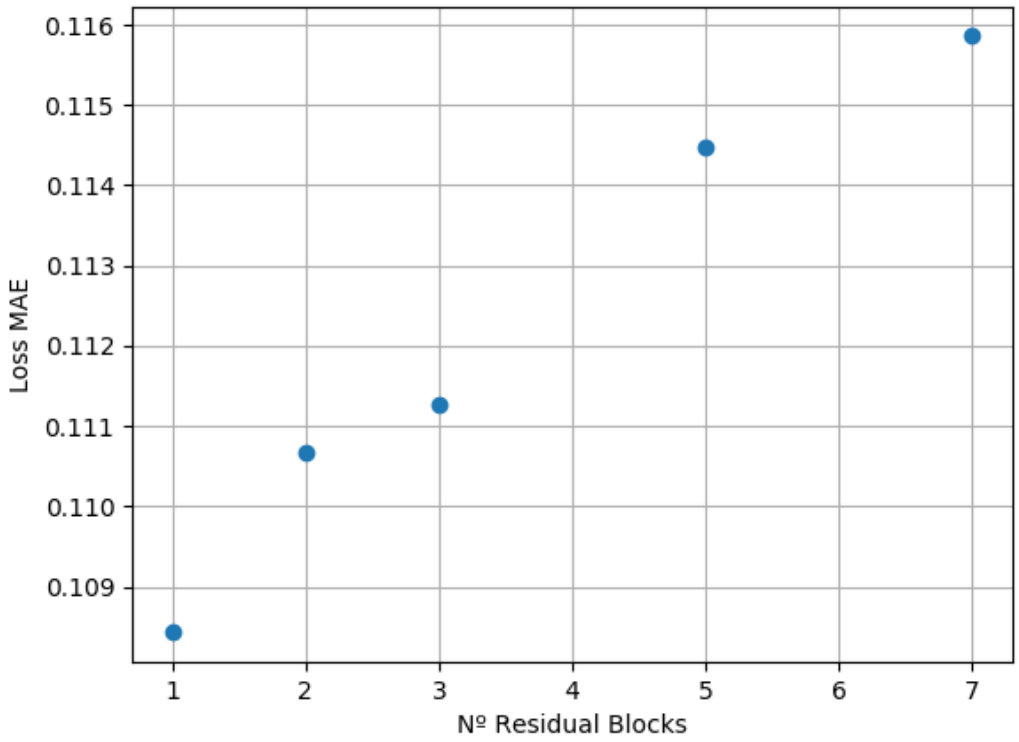


Figura 6.18: *Cross-Validation* para diferentes números de blocos residuais

O número de neurónios foi também alterado e obteve-se os resultados da figura. 6.19, onde se pode observar que a rede alcança o menor erro com 1000 neurónios. Embora o aumento dos neurónios possa contribuir para o problema de sobre-aprendizagem isso não se verifica neste caso, como demonstra a figura 6.20, devido à introdução de ruído nos dados de treino que funciona como um regularizador (por exemplo *dropout*).

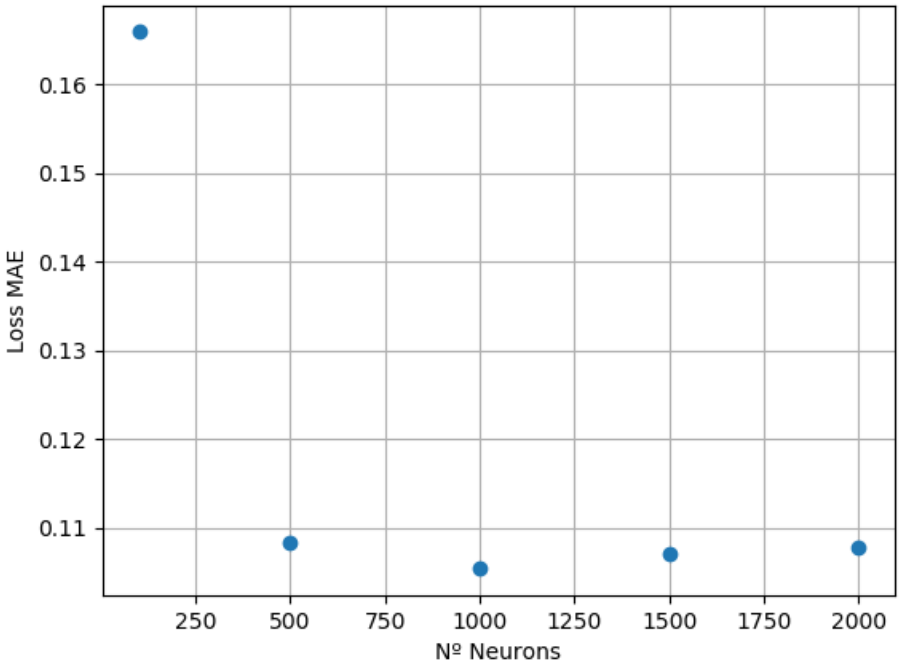


Figura 6.19: *Cross-Validation* para diferentes números de neurónios

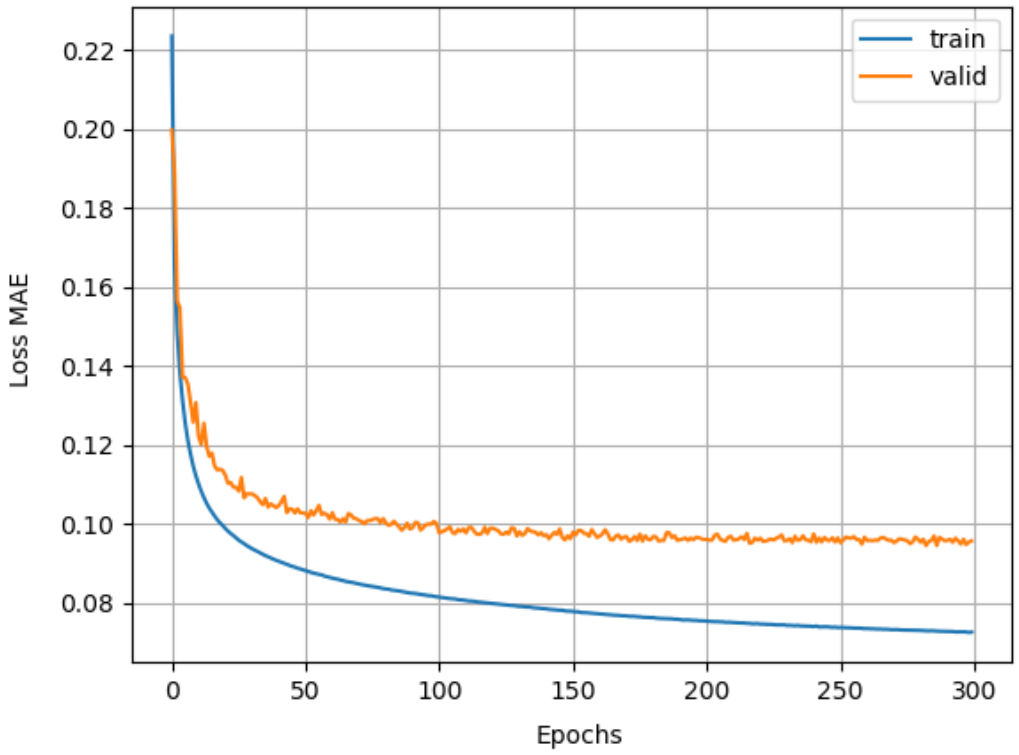


Figura 6.20: Curva de aprendizagem do modelo com 1 bloco residual de 1000 neurónios

A arquitectura da ResNet permite treinar redes com um grande número de *layers* sem o problema do desaparecimento do gradiente (ver capítulo 3). Embora este modelo tenha só um bloco residual com uma *layer*, consegue um erro menor em comparação com o modelo sequencial simples (sem *skip connection*), com uma diferença de 0.00037 que embora não seja significativa optou-se por continuar com esta arquitetura.

No final desta secção obteve-se uma ResNet de um bloco residual com uma *dense layer* de 1000 neurónios, apresentada na figura 6.21.

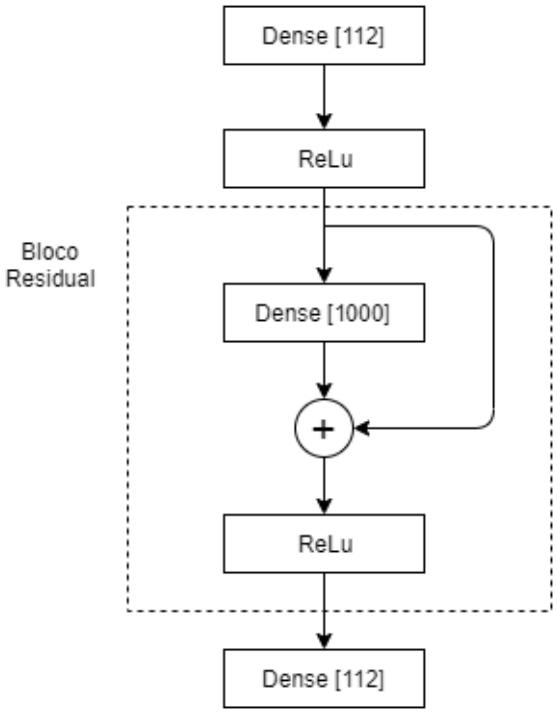


Figura 6.21: Arquitectura da rede que obteve melhores resultados para os números de camadas e neurónios

6.4.3 Inicialização dos pesos e função de activação

Nos problemas de regressão são tipicamente usadas as funções de activação **ReLu** e as suas variantes, juntamente com as funções de inicialização **He** e **LeCun** como indicado na tabela 6.3 [27].

Função de inicialização	Função de activação
He	ReLu, ELU
LeCun	SELU

Tabela 6.3: Parâmetros típicos nos problemas de regressão [27]

Como foi referido no capítulo 3, a função SELU juntamente com a LeCun permite a auto normalização da rede desde que esta só contenha *Dense layers*, por isso os testes efectuados com essa função são modelos sequências simples, sem *skip connections*.

Testaram-se as combinações, tabela 6.4, entre estas duas funções e verificou-se que a combinação (**Random uniforme**, **ReLu**) consegue o menor erro.

Função de inicialização	Função de ativação	Erro (MAE)
Random uniforme	ReLu	0.1080
Random normal	ReLu	0.1102
He normal	ReLu	0.1141
He uniforme	ReLu	0.1151
LeCun normal	SELU	0.1161
He normal	ELU	0.1163
He uniforme	ELU	0.1167

Tabela 6.4: *Cross-Validation* das combinações entre funções de inicialização e activação (ordem crescente do erro)

6.4.4 Algoritmos de Optimização

Os algoritmos de optimização escolhidos para o processo de *tuning* foram as 3 últimos referidos no capitulo 3, mais concretamente **Adam**, **Nadam** e **Adamax**. Devido a problemas de implementação do Adamax no Tensorflow (retorna valores "NaN" [3]), substituiu-se esse algoritmo pelo **Adagrad**, referido no capítulo 3 como um algoritmo mais básico e assim verifica-se a diferença entre este e os algoritmos mais avançados. A tabela 6.5 apresenta os algoritmos testados, por ordem crescente do erro.

Optimizador	Erro (MAE)
Adam	0.1008
Nadam	0.1080
Adagrad	0.1984

Tabela 6.5: *Cross-Validation* das diferentes funções de optimização (ordem crescente do erro)

O algoritmo Adam atinge o menor erro seguido do Nadam com uma pequena diferença ao contrário do Adagrad que sobe consideravelmente. Desta forma o algoritmo de optimização a usar será o Adam.

6.4.5 Tamanho do *batch*

O tamanho do *batch* pode causar um impacto significativo no desempenho e no tempo de treino da rede. Um valor pequeno acelera cada iteração, enquanto um

valor maior dá uma maior precisão no cálculo dos gradientes. No geral são utilizadas potências de 2 neste parâmetro de forma a alinhar com a memoria do CPU/GPU que também se encontram organizadas em potências de 2. Deste forma tira-se partido das optimizações do hardware.

Testaram-se os valores 32, 256, 512, 1024 e conclui-se que 512 é o valor óptimo para o tamanho do *batch*, como indicado na tabela 6.6.

Tamanho do *batch*	MAE
512	0.097
1024	0.0981
256	0.1080
32	0.1211

Tabela 6.6: *Cross-Validation* para diferentes tamanhos do *batch* (ordem crescente do erro)

6.5 Filtro Savitzky–Golay

A biblioteca "SciPy" [18] (biblioteca Python para matemática, ciências e engenharia) já implementa este filtro numa função que recebe o tamanho da janela e o grau do polinómio. Também oferece uma função que retorna os coeficientes dos dois parâmetros anteriores. Sabendo que a escolha óptima dos parâmetros é um compromisso entre a redução de ruído e a distorção do sinal, testou-se o polinómio de 2º grau para diferentes tamanhos da janela, como demonstra a figura 6.22. Quando o tamanho da janela é igual a 29, atinge-se o erro mínimo de 42.91.

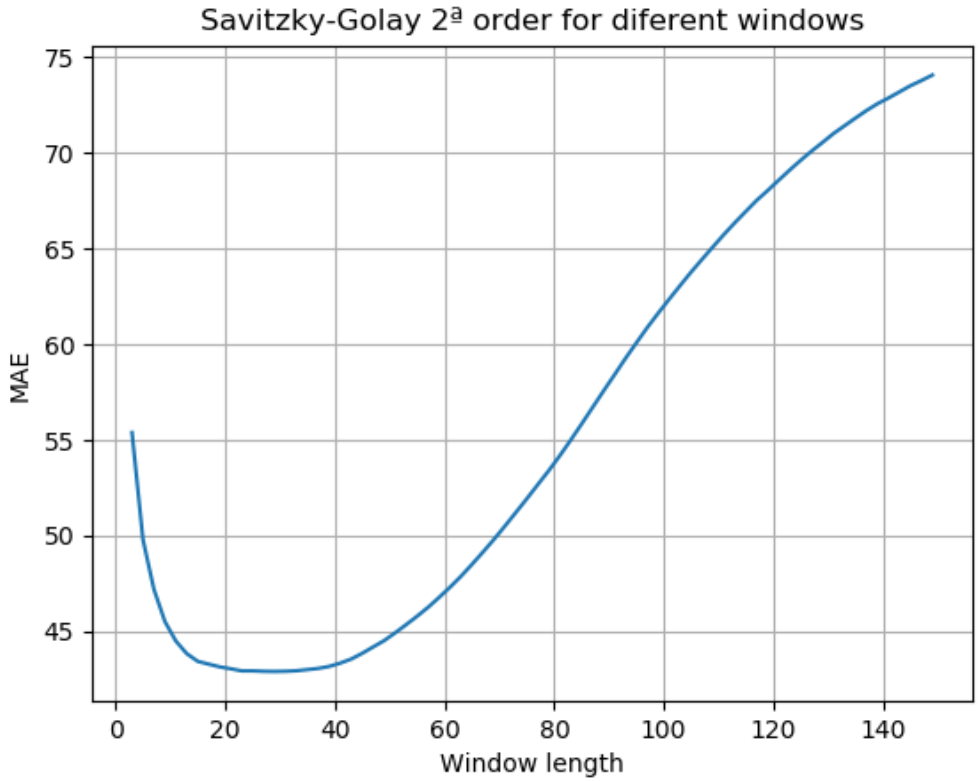


Figura 6.22: Desempenho do filtro Savitzky-Golay

A figura 6.22 mostra, em relação à coordenada "X" da junta "Head", os dados originais (*ground truth*, a verde) e os dados devolvidos pela rede sem filtro (vermelho) e com a aplicação do filtro (azul). Pela análise da figura é possível verificar que o filtro elimina grande parte do ruído que a rede não conseguiu corrigir.

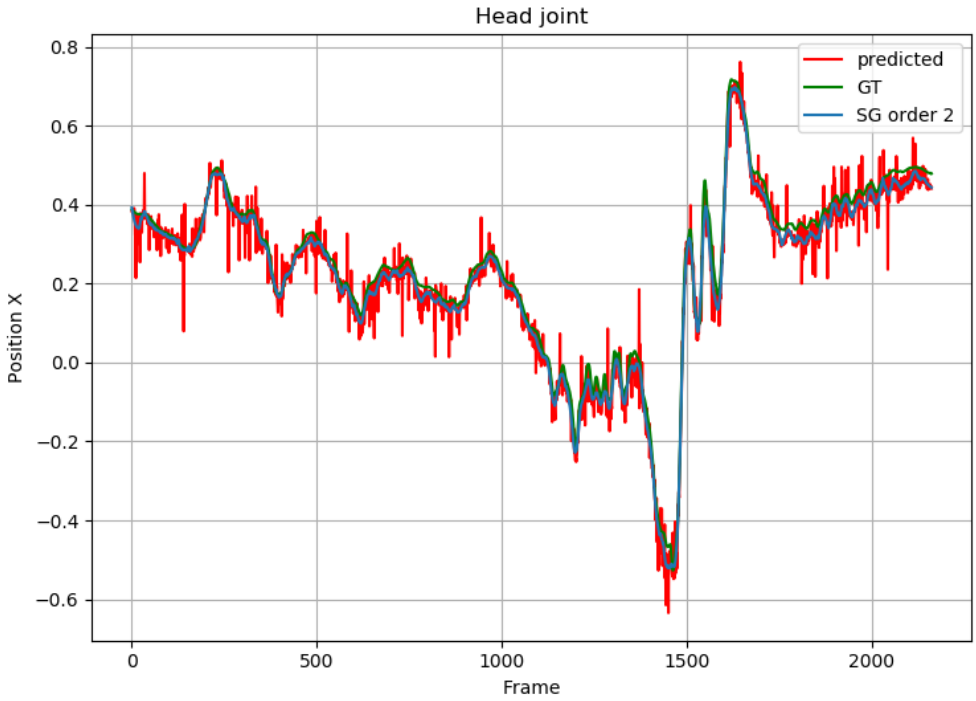


Figura 6.23: Posição "X" da junta "*Head*" antes e depois de passar pelo filtro

Embora os dados filtrados se aproximarem mais do *ground truth*, a informação começa a distorcer perante grandes variações, como demonstra o gráfico da figura 6.24. Para resolver esse problema experimentou-se, para a mesma janela, dois polinómios diferentes, um de grau 4 e outro de grau 6 (tabela 6.7). O polinómio de grau 4 consegue recuperar parte da informação perdida e ainda diminuir o ruído face aos polinómios de grau 6 e 2.

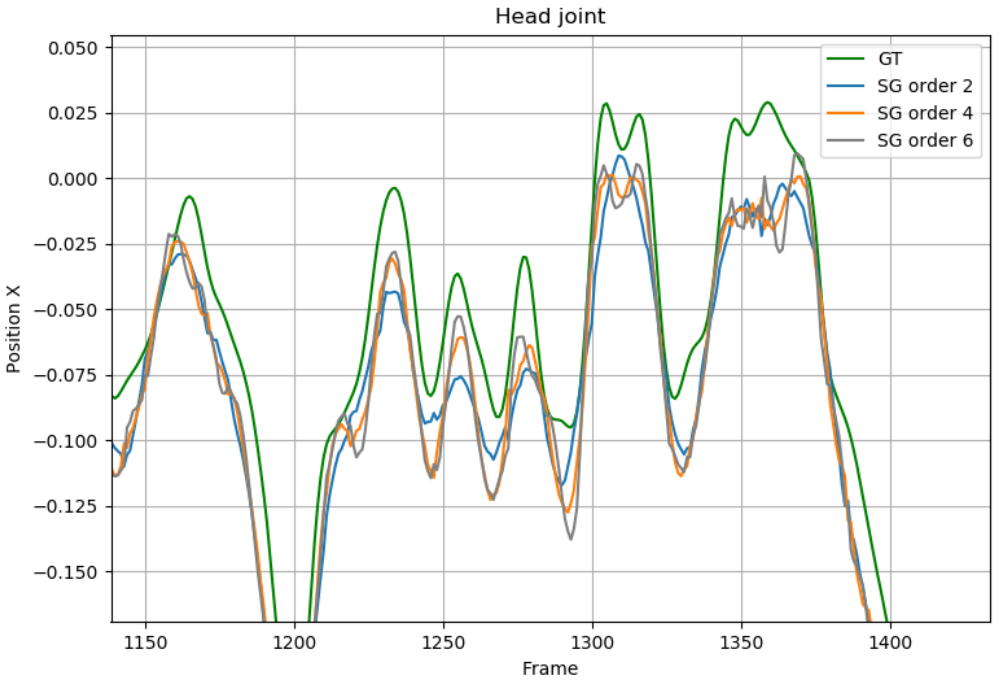


Figura 6.24: Teste do filtro com polinómios de ordem 2, 4 e 6

Polinómio	**MAE**
2	46.26
4	42.98
6	43.06

Tabela 6.7: Erro total dos filtros

Assim, o filtro Savitzky–Golay a aplicar aos dados devolvidos pela rede neuronal tem como parâmetros um polinómio de grau 4 e uma janela de 29 pontos.

7

Avaliação

Neste capítulo são apresentados e discutidos os resultados obtidos pelos testes realizados para validar o modelo proposto. Estes testes estão divididos em duas fases. A primeira fase testa os dados da Kinect e da CMU através de métricas como o MAE. A segunda fase testa os dados da Kinect através de uma uma avaliação perceptual

7.1 Introdução

Numa primeira fase, o modelo é avaliado com um conjunto de poses da CMU, diferente das poses utilizadas no treino da rede neuronal profunda, para avaliar o seu comportamento perante a mesma estrutura do esqueleto que foi treinado. Esta avaliação é feita utilizando a métrica MAE para medir o erro entre as poses originais e as devolvidas pela rede. Nesta fase também são realizados testes para verificar qual dos tipos de ruído introduzidos no treino é mais adequado e as diferenças entre o número de características usadas, isto é, as diferenças entre o modelo treinado com as posições e rotações em simultâneo e individualmente.

Numa segunda fase, o modelo é avaliado com um conjunto de poses provenientes da Kinect, para verificar se o modelo é capaz de corrigir as poses do esqueleto original da Kinect, sabendo que foi treinado com uma estrutura semelhante. Como não se conhece o *ground truth* das poses capturadas pela Kinect, optou-se por fazer uma avaliação perceptual com 12 utilizadores. Nesta avaliação foi pedido aos

utilizadores para escolherem uma de três animações do mesmo movimento. A primeira animação é obtida sem aplicação do modelo proposto e as outras com a aplicação de duas variantes do modelo proposto para correcção dos erros.

O modelo utilizado nos testes foi treinado com 885823 amostras de treino, durante 800 épocas. Quanto aos dados de teste, existem 88582 amostras, diferentes das utilizadas no treino.

7.2 Avaliação com os dados da CMU

Nesta fase são testadas 3 situações de ruído:

1. Ruído provocado por oclusões e deslocamentos;

2. Ruído provocado por oclusões;

3. Ruído provocado por deslocamentos.

O objectivo é avaliar se é favorável treinar dois modelos separadamente, um com oclusões e outro com deslocamentos. A figura 7.1 apresenta os erros das poses devolvidas pelo modelo (a azul) treinado com oclusões e deslocamentos, bem como o erro introduzido nas poses de teste (a vermelho). No primeiro gráfico da figura adicionaram-se os dois tipos de ruído às poses de teste, no segundo gráfico apenas se adicionaram oclusões e no terceiro só deslocamentos. As figuras 7.2 e 7.3 apresentam o mesmo tipo de informação, mas para os modelos treinados unicamente com oclusões e deslocamentos respectivamente.

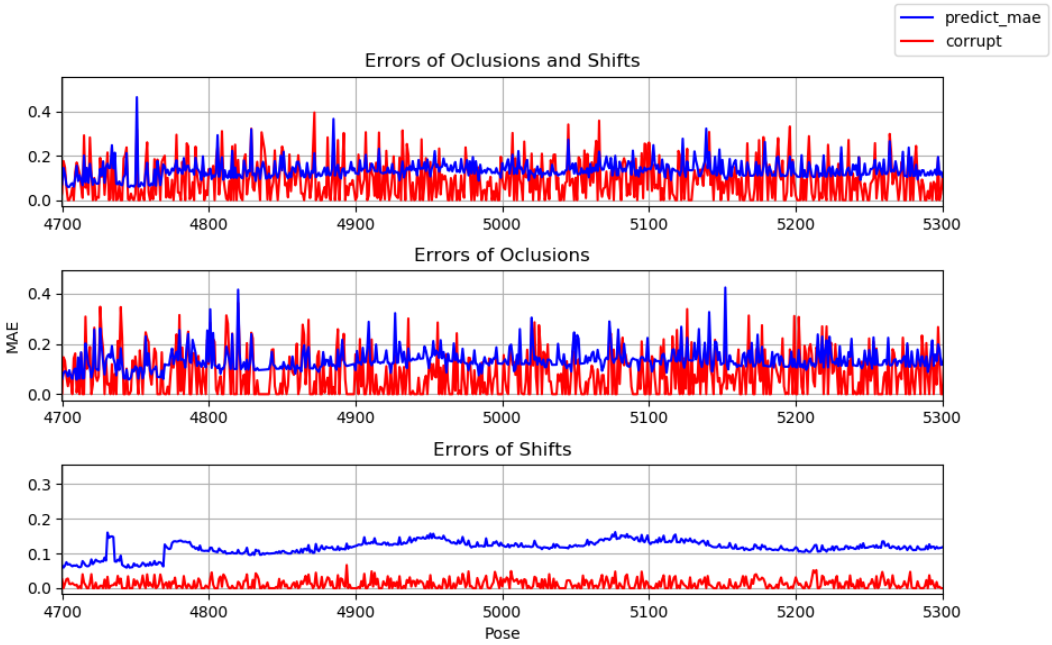


Figura 7.1: Testes com modelo treinado com oclusões e deslocamentos

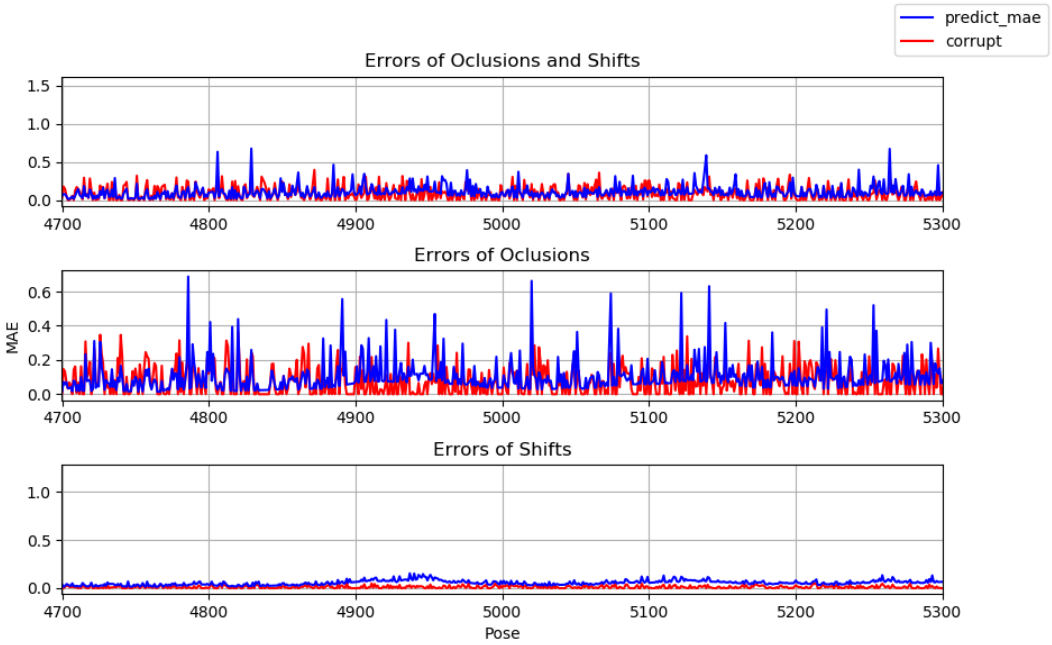


Figura 7.2: Testes com modelo treinado com oclusões

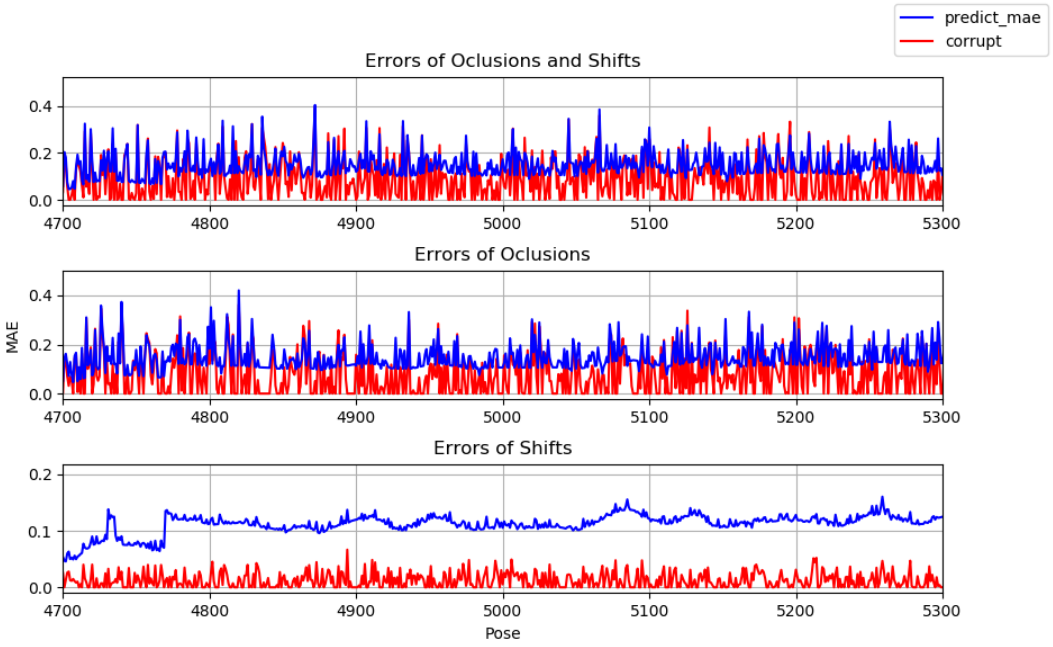


Figura 7.3: Testes com modelo treinado com deslocamentos

Através dos gráficos conclui-se que o ruído é reduzido significativamente à excepção dos deslocamentos, que embora tente suavizar as altas frequências, o erro total aumenta (tabela 7.1). Tal acontece porque na maior parte das poses a função de corrupção desloca as juntas para outra posição possível e a rede tende a aprender a deslocar sempre as juntas independentemente da posição. Assim, haverá mais deslocamentos e como estes não tem uma distância tão grande como as oclusões, o erro acaba por ser maior. Mas isto não significa que as poses não sejam visualmente mais corretas, a figura 7.4 ilustra a correcção de uma pose com deslocamentos que se aproxima mais do *ground truth* do que a pose corrupta. O modelo treinado exclusivamente com oclusões apresenta o erro mais baixo independentemente do tipo de ruído aplicado nos dados de teste. Na figura 7.5 pode-se observar que este modelo consegue corrigir a pose ocluída, tornando-a semelhante à verdadeira.

		Teste CMU		
		Oclusões e Deslocamentos	Oclusões	Deslocamentos
Treino	Oclusões e Deslocamentos	0.0461	0.042	0.0334
	Oclusões	0.0383	0.0239	0.0225
	Deslocamentos	0.0766	0.0734	0.0303
	Corrupt	0.0724	0.0618	0.0127

Tabela 7.1: Erro MAE para diferentes tipos de ruído no treino e teste

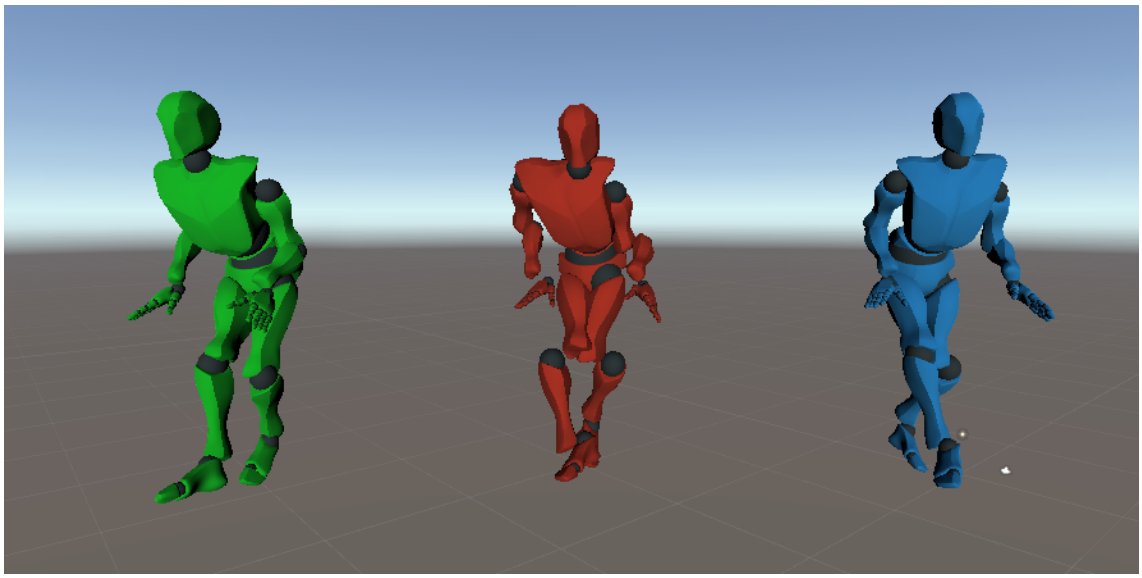


Figura 7.4: Correcção dos deslocamentos. Verde: pose verdadeira; Vermelho: pose corrupta; Azul: pose predita.

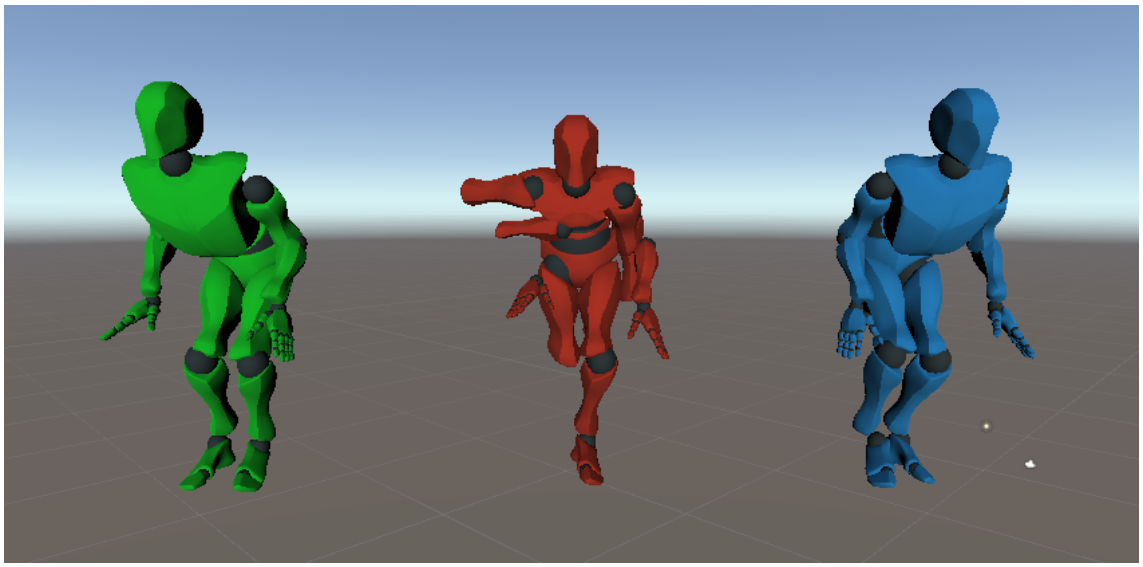


Figura 7.5: Correcção das oclusões. Verde: pose verdadeira; Vermelho: pose corrupta; Azul: pose predita.

Nas figuras 7.6, 7.7 e 7.8 são apresentadas as curvas de aprendizagem para verificar que os modelo comportam-se como esperado sem entrar em sobre-aprendizagem. Tanto o treino como a validação decrescem a longo das épocas.

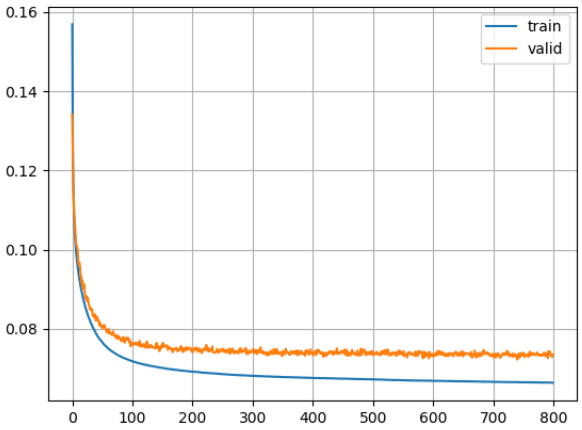


Figura 7.6: Curva de aprendizagem - Modelo O/D

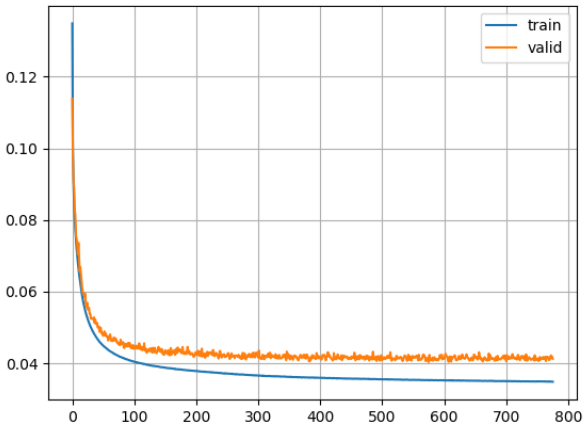


Figura 7.7: Curva de aprendizagem - Modelo O

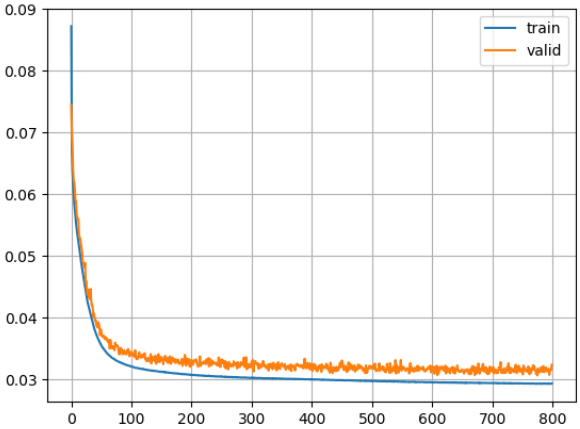


Figura 7.8: Curva de aprendizagem - Modelo D

7.3 Avaliação com dados da Kinect

Para os dados da Kinect, a rede não consegue corrigir as poses com a mesma precisão dos testes da CMU. Apesar das transformações efectuadas para aproximar as estruturas dos dois esqueletos, as posições das juntas não são exactamente as mesmas o que irá influenciar o resultado da rede. Testou-se um conjunto de poses sem ruído capturadas pela Kinect e introduziu-se oclusões e deslocamentos com a função de corrupção implementada. A tabela 7.2 mostra que em qualquer uma das condições de treino, o erro da rede nunca é menor que o erro introduzido.

		Teste Kinect		
		Oclusões e Deslocamentos	Oclusões	Deslocamentos
Treino	Oclusões e Deslocamentos	0.093	0.0894	0.083
	Oclusões	0.1096	0.0874	0.0928
	Deslocamentos	0.1012	0.0989	0.0647
	Corrupt	0.0621	0.0522	0.0122

Tabela 7.2: Erro MAE para os dados de teste da Kinect

Neste caso, o modelo dos deslocamentos é o que causa menos ruído mas isso não significa que as poses estejam visualmente mais próximas do *ground truth*. De forma a perceber as alterações efectuadas pela rede, observaram-se as poses no Unity (ver figura 7.9). Dos 3 modelos treinados, o que aproxima-se mais do *ground truth* é o modelo treinado com ambos os ruídos. Este consegue corrigir a posição da junta juntamente com a sua rotação, ao contrário do modelo treinado com oclusões, que embora pareça acertar as posições falha nas rotações. Apesar do modelo dos deslocamentos não ser apropriado para corrigir ruído, este não causa tantas alterações nas juntas quando a pose fornecida não contém ruído como demonstra a figura 7.10. Por essa razão e pelo facto dos outros dois modelos não serem precisos, leva a que o erro total deste modelo seja o menor.

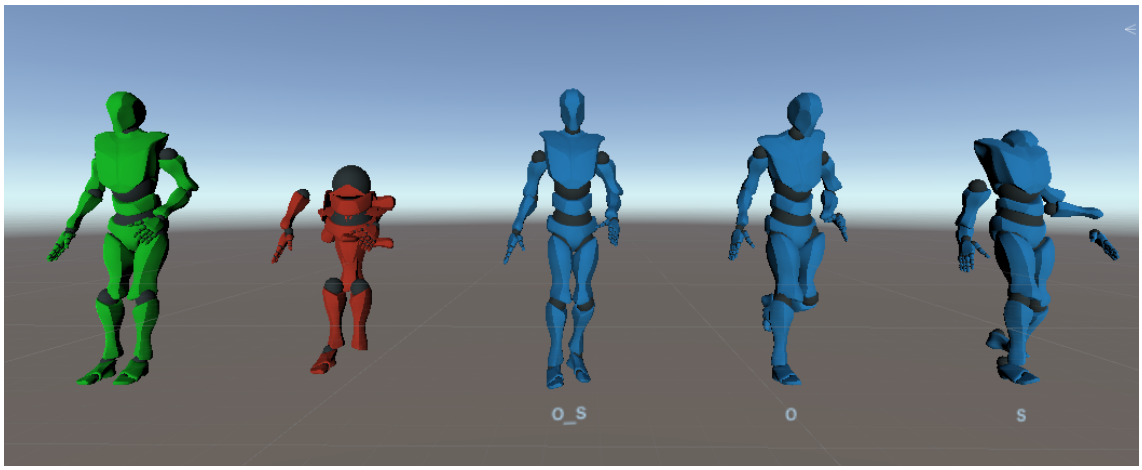


Figura 7.9: Resultados da rede para os dados de teste da Kinect, corrompidos com a função de corrupção implementada. Verde: pose verdadeira; Vermelho: pose corrupta; Azul: pose predita pela rede treinada com os diferentes ruídos.

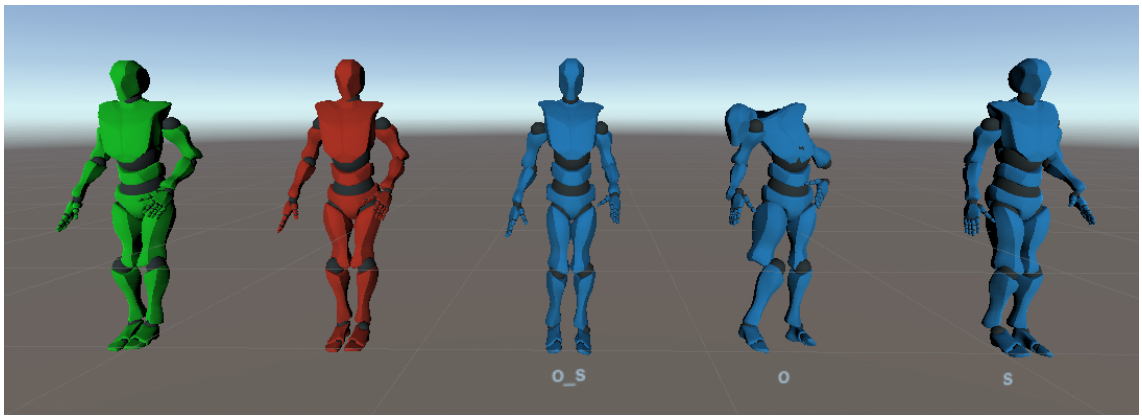


Figura 7.10: Resultados da rede para uma pose da Kinect sem erro adicional. Verde: pose verdadeira; Vermelho: pose corrupta; Azul: pose predita pela rede treinada com os diferentes ruídos.

De seguida testou-se para um movimento real da Kinect, onde podem haver oclusões e deslocamentos. Não existindo informação do *ground truth*, as poses foram observadas no Unity onde se concluiu que o resultado da predição piora relativamente as poses corrompidas através da função implementada. Na figura 7.11 nota-se que o modelo dos dois ruídos em simultâneo origina uma pose mais próxima da realidade.

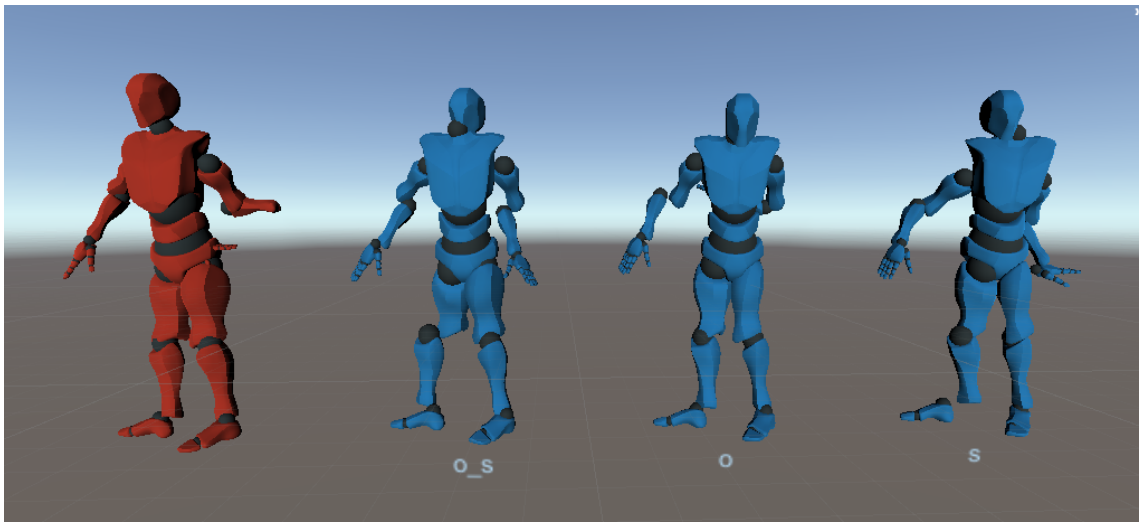


Figura 7.11: Resultados da rede para os dados ruidosos da Kinect. Vermelho: pose corrupta; Azul: pose predita pela rede treinada com os diferentes ruídos.

A rede treinada apenas com as posições das juntas devolve também só as posições. Tal não é desejável porque sem as rotações não se sabe a direcção dos segmentos, resultando no esqueleto da figura 7.12.

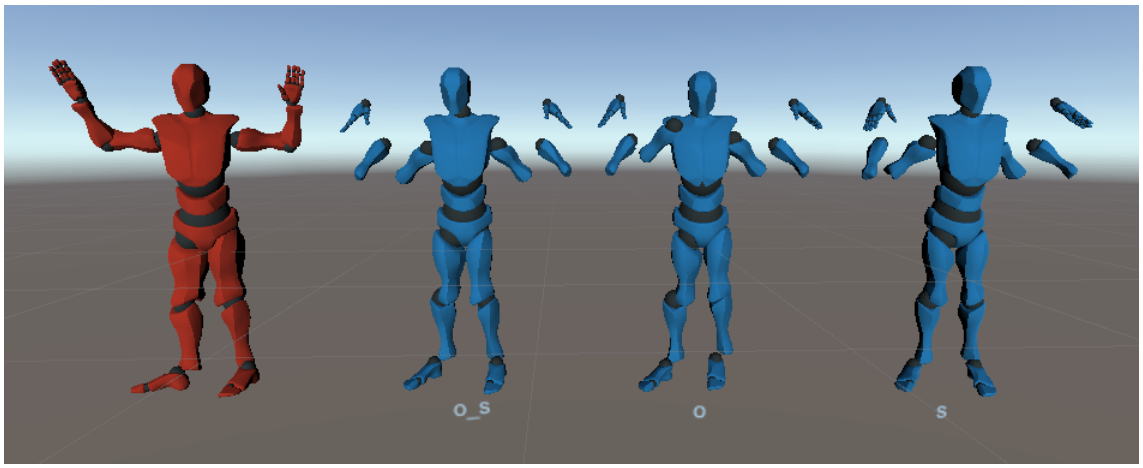


Figura 7.12: Resultados da rede, só com informação das posições das juntas, para os dados ruidosos da Kinect. Vermelho: pose corrupta; Azul: pose predita pela rede treinada com os diferentes ruídos.

Ao contrário do modelo acima, é possível treinar a rede só com as rotações desde que se saiba a posição e rotação da pose inicial T. A partir dessa, serão aplicadas as transformações das rotações de forma hierárquica. Na figura 7.14 nenhum dos

modelos conseguiu prever com precisão a rotação do braço ocluído, no entanto apresentam poses mais naturais que o modelo treinado com posições e rotações em simultâneo.

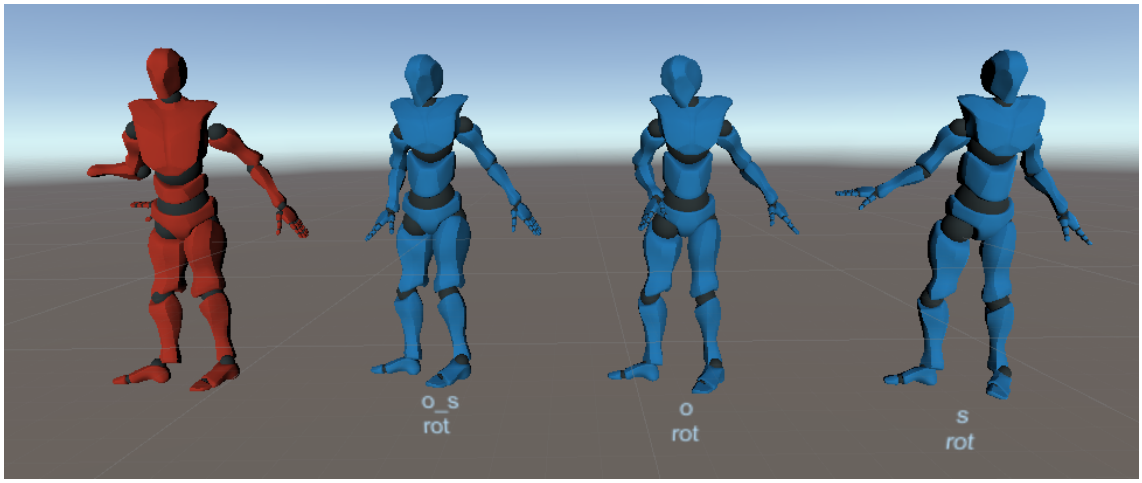


Figura 7.13: Resultados da rede, só com informação das rotações das juntas, para os dados ruidosos da Kinect. Vermelho: pose corrupta; Azul: pose predita pela rede treinada com os diferentes ruídos.

A utilização do filtro temporal nos dados devolvidos pela rede ajuda a diminuir o ruído, principalmente o *jittered* causado pela rede. Contudo, nas poses da Kinect, o erro continua acima do erro introduzido.

		Teste CMU - Filtro Temporal		
		Oclusões e Deslocamentos	Oclusões	Deslocamentos
Treino	Oclusões e Deslocamentos	0.0397	0.0382	0.0314
	Oclusões	0.0289	0.0218	0.0172
	Deslocamentos	0.061	0.0599	0.0283
	Corrupt	0.0724	0.0618	0.0127

Tabela 7.3: Erro MAE para diferentes tipos de ruído no treino e teste com passagem no filtro temporal

A diferença dos resultados entre as poses da CMU transformadas e as poses da Kinect pode ser devido ao facto dessa transformação não ser apropriada. Por isso adicionou-se algumas poses adquiridas com a Kinect, sem a presença de oclusões, aos dados de treino, para que a rede aprenda a estrutura real do esqueleto da Kinect. A tabela 7.5 mostra, que com este procedimento, é possível diminuir 6.3% do erro total em relação ao modelo anterior treinado só com as rotações. Quanto

		Teste Kinect - Filtro Temporal		
		Oclusões e Deslocamentos	Oclusões	Deslocamentos
Treino	Oclusões e Deslocamentos	0.0869	0.086	0.0826
	Oclusões	0.0923	0.0814	0.0855
	Deslocamentos	0.0868	0.086	0.0655
	Corrupt	0.0621	0.0522	0.0122

Tabela 7.4: Erro MAE para os dados de teste da Kinect com passagem no filtro temporal

ao modelo treinado simultaneamente com a posição e rotação, existe um ligeiro aumento no erro total. No entanto, a figura 7.14 apresenta o contrário, onde o modelo treinado com alguns dados da Kinect devolve uma pose mais próxima da realidade e com desvios mais pequenos, sobretudo na posição da cabeça e do joelho direito. Assim pode-se concluir que adicionar poses da Kinect aos dados de treino é benéfico à rede mesmo que o erro total calculado não o represente.

	Pos+Rot	**Rot**
CMU	0.093	0.0598
CMU+Kinect	0.0932	0.056
Corrupt	0.0621	0.0321

Tabela 7.5: Erro MAE dos modelos treinados com/sem os dados da Kinect e para diferentes dimensões

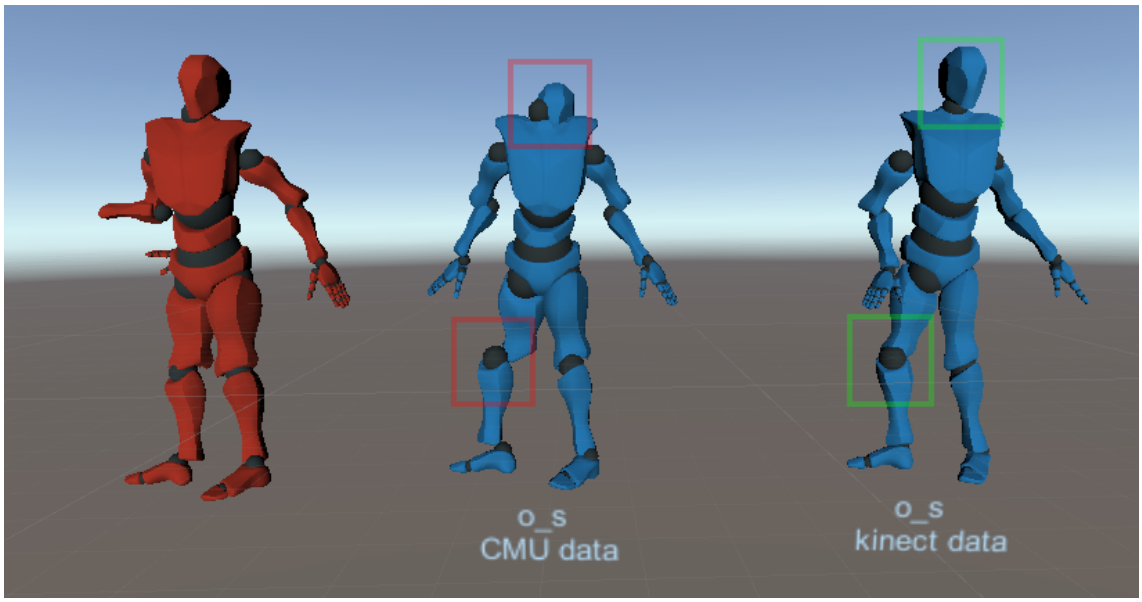


Figura 7.14: Resultados da rede para os dados ruidosos da Kinect. Vermelho: pose corrupta; Azul: pose predita pela rede treinada com/sem os dados da Kinect.

7.4 Testes do utilizador

Os movimentos captados com a Kinect não têm *ground truth* tornando-se difícil de perceber se a rede consegue corrigir esse movimentos. Para isso, realizou-se um questionário com 29 movimentos distintos (anexo A), captados com este sensor, em que para cada um é apresentado um vídeo com 3 sequências ordenadas aleatoriamente. Uma das sequências é obtida directamente do sensor (sem aplicar a rede neuronal profunda) e duas provenientes dos dois modelos que apresentaram visualmente poses mais naturais, nomeadamente o modelo treinado com rotações e o modelo treinado com rotações e posições. Ambos treinados com oclusões e deslocamentos. No treino destes dois modelos, foram introduzidas algumas poses da Kinect que, tal como foi provado anteriormente, melhora o resultado juntamente com passagem do filtro temporal.

Para cada movimento é pedido ao participante para seleccionar a sequência que representa o movimento mais natural.

Os testes foram realizados a um conjunto de 12 participantes com conhecimento na área da informática e 4 deles com experiência na área da animação e modelação 3D. Foi enviado um *link* do questionário a cada um dos participantes através

da aplicação Google Forms [7], para que preenchessem individualmente.

O resultado apresentado na tabela 7.6 prova que a rede tem um efeito positivo na correcção das poses da Kinect. Cerca de 45% das respostas apontam para a sequência da "Resnet" (modelo treinado com posição e rotação) e cerca de 35% para a "ResnetRot" (modelo treinado só com a rotação) como as sequências mais naturais. A verificação da segunda sequência mais natural prova que os dois modelos conseguem geralmente melhorar as poses provenientes da Kinect.

	1º mais natural	**2º mais natural**
Kinect	67	107
Resnet	157	116
ResnetRot	124	125

Tabela 7.6: Resultado geral do questionário

Com o intuito de analisar o comportamento dos modelos perante diferentes níveis de oclusão, desenhou-se o gráfico da figura 7.15 onde apresenta o número de votações de cada sequência ("Kinect", "Resnet", "ResnetRot") num conjunto de movimentos classificado, em uma escala de 1 a 5, de acordo com o número de oclusões existentes. O número de votações em cada classe foi normalizado de 0 a 100 devido ao diferente número de movimentos entre elas. Pode-se observar que em movimentos com poucas oclusões ambas as redes tem a mesma votação, superando consideravelmente as poses da Kinect. Nos níveis de oclusão intermédios, 2 e 3, a "Resnet" consegue os melhores resultados seguido do "ResnetRot". Este último tem um grande impacto nas poses com maior número de oclusões, mas surpreendentemente tem um desempenho abaixo da sua média no nível máximo, sendo que neste caso, nenhuma das redes consegue melhorar as poses da Kinect.

No geral, o modelo treinado com as posições e rotações parece ser o mais adequado, o que era esperado visto que contém mais características para o treino, apesar de as sequências apresentarem ainda poses não naturais.

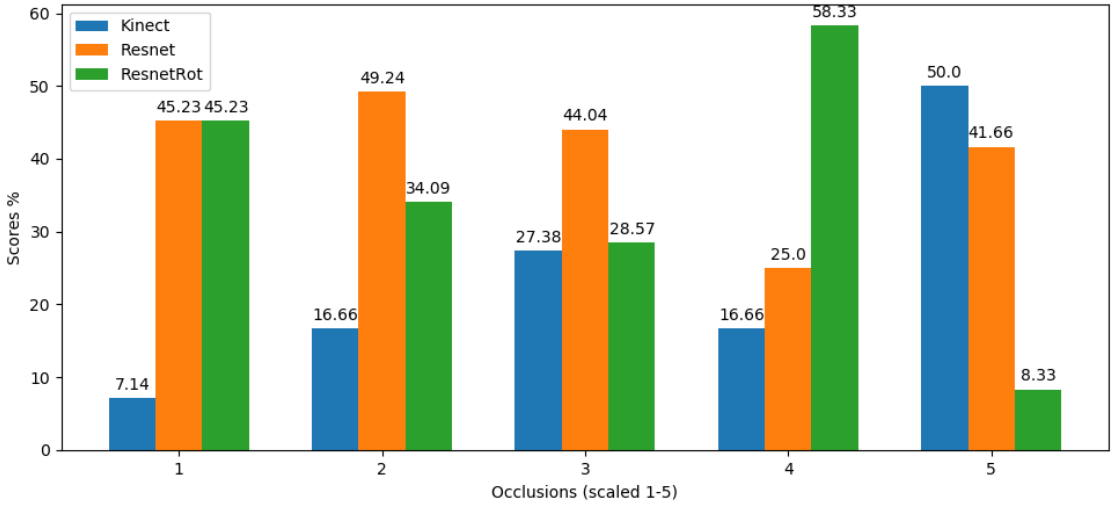


Figura 7.15: Resultados do questionário em relação ao nível de oclusão das animações. Oclusão a 1 agrupa todos os movimentos com poucas oclusões. Oclusão a 5 agrupa todos os movimentos com bastantes oclusões.

8

Conclusões e trabalho futuro

A utilização das redes neuronais profundas para resolver problemas de *denoising*, como o problema descrito, apresenta resultados promissores à custo de um grande conjunto de dados de treino.

O sistema baseado numa rede neuronal profunda implementado consegue diminuir consideravelmente o ruído das poses com a mesma estrutura que foi treinada, ou seja, as poses da CMU adaptadas. No entanto, os testes com as poses do sensor Kinect mostram que a rede não se adapta tão bem a esqueletos com outra arquitectura, mesmo com o adaptação realizada. O facto das juntas nos dados de treino não estarem perfeitamente alinhadas com as juntas do esqueleto da Kinect influencia negativamente a predição. Este problema pode ser ligeiramente atenuado ao adicionar poses correctas provenientes do sensor Kinect, isto para que a rede tenha algum conhecimento da estrutura desse esqueleto.

Os valores numéricos dos resultados obtidos nem sempre permitem determinar o melhor modelo. Como explicado no capítulo 7, o modelo treinado com os deslocamentos apresenta um erro total acima do esperado, mas consegue visualmente corrigir as poses da CMU corrompidas. Torna-se difícil determinar qual dos tipos de ruído é mais adequado. De acordo com as tabelas dos erros calculados para os testes da CMU, o modelo treinado exclusivamente com oclusões consegue os melhores resultados nas três situações.

Para os dados de teste da Kinect, cada modelo serve o seu propósito, ou seja, corrige melhor o erro para o qual foi treinado. Apesar da soma total do erro

para estes dados seja sempre maior do que o esperado, os testes do utilizador apresentados no capítulo 7 mostram que a predição feita pela rede juntamente com a aplicação o filtro temporal resulta em movimentos mais naturais, mesmo que apresentem poses incorrectas.

Para trabalho futuro, pretende-se seguir outro processo de adaptação do esqueleto que consiste em inverter essa transformação, ou seja, converter o esqueleto da Kinect para o da CMU adicionando as juntas que faltam em vez de as eliminar. Outra hipótese seria construir um *dataset* com o esqueleto verdadeira do sensor Kinect, a partir de três câmaras para evitar oclusões nas juntas. Por último pretende-se testar o sistema implementado numa aplicação de realidade virtual/aumentada com o intuito de analisar o desempenho em tempo real e qual o impacto na experiência do utilizador.

Referências

[1] Rmsprop. URL https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[2] Acclaim asf/amc. URL https://research.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/ASF-AMC.html.

[3] Problemas do adamax no tensorflow. URL https://github.com/tensorflow/tensorflow/issues/26256.

[4] Autodesk maya. URL https://www.autodesk.com/products/maya/overview.

[5] Blender. URL https://www.blender.org/.

[6] Cmu mocap dataset. URL http://mocap.cs.cmu.edu/.

[7] Google forms. URL https://www.google.com/forms/about/.

[8] Hardware do sensor microsoft kinect. URL https://www.microsoftpressstore.com/articles/article.aspx?p=2201646.

[9] Leap motion. URL https://www.ultraleap.com/.

[10] Microsoft kinect. URL https://developer.microsoft.com/pt-pt/windows/kinect/.

[11] Sistemas de motion capture. URL https://en.wikipedia.org/wiki/Motion_capture.

[12] Nuitrack unity sdk. URL http://download.3divi.com/Nuitrack/doc/UnityBasic_page.html.

[13] Pandas library. URL https://pandas.pydata.org/.

[14] Intel realsense. URL https://www.intelrealsense.com/skeletal-tracking/.

[15] code-ai. URL https://code-ai.mk/neural-network-with-c-from-scratch/.

[16] Rokoko. URL https://www.rokoko.com/.

[17] Savitzky-golay filter. URL https://en.wikipedia.org/wiki/Savitzky%E2%80%93Golay_filter.

[18] Scipy. URL https://www.scipy.org/.

[19] Shadow. URL https://www.motionshadow.com/.

[20] Xsens. URL https://www.xsens.com/.

[21] Andreas Aristidou and Joan Lasenby. Real-time marker prediction and cor estimation in optical motion capture. The Visual Computer, 29:7–26, 01 2013. doi: 10.1007/s00371-011-0671-y.

[22] Andreas Aristidou, Daniel Cohen-Or, Jessica Hodgns, and Ariel Shamir. Self-similarity analysis for motion capture cleaning. Computer Graphics Forum, 37, 05 2018. doi: 10.1111/cgf.13362.

[23] Jinxiang Chai and Jessica K. Hodgins. Performance animation from low-dimensional control signals. ACM Trans. Graph., 24(3):686–696, July 2005. ISSN 0730-0301. doi: 10.1145/1073204.1073248. URL https://doi.org/10.1145/1073204.1073248.

[24] Jimmy Ba Diederik P. Kingma. Adam: A method for stochastic optimization. URL https://arxiv.org/abs/1412.6980.

[25] Sepp Hochreiter Djork-Arné Clevert, Thomas Unterthiner. Fast and accurate deep network learning by exponential linear units (elus). URL https://arxiv.org/abs/1511.07289.

[26] Timothy Dozat. Incorporating nesterov momentum into adam. URL http://cs229.stanford.edu/proj2015/054_report.pdf.

[27] Aurélien Géron. Hands-On Machine Learning with Scikit-Learn, Keras TensorFlow. O'REILLY, 2019.

[28] Andreas Mayr Sepp Hochreiter Günter Klambauer, Thomas Unterthiner. Self-normalizing neural networks. URL https://arxiv.org/abs/1706.02515.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. pages 770–778, 06 2016. doi: 10.1109/CVPR.2016.90.

[30] Daniel Holden. Robust solving of optical motion capture data by denoising. ACM Transactions on Graphics, 37:1–12, 07 2018. doi: 10.1145/3197517.3201302.

[31] Yoram Singer John Duchi, Elad Hazan. Adaptive subgradient methods for online learning and stochastic optimization. URL http://jmlr.org/papers/v12/duchi11a.

[32] Pedro Nogueira. Motion capture fundamentals a critical and comparative analysis on real-world applications. 2012.

[33] Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. URL https://www.researchgate.net/publication/243648538_Some_methods_of_speeding_up_the_convergence_of_iteratior

[34] Jun Saito, Daniel Holden, and Taku Komura. Learning motion manifolds with convolutional autoencoders. 11 2015. doi: 10.1145/2820903.2820918.

[35] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network, 2018. URL http://arxiv.org/abs/1808.03314. cite arxiv:1808.03314Comment: 39 pages, 10 figures, 66 references.

[36] Chunwei Tian, Lunke Fei, Wenxian Zheng, Yong xu, Wangmeng Zuo, and Chia-Wen Lin. Deep learning on image denoising: An overview, 12 2019.

Questionário

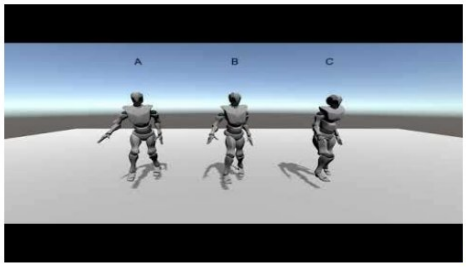
Kinect Mocap

This questionnaire aims to evaluate the result of a pose correction system from the Kinect sensor.
Three sequences (A, B and C) will be presented, for each of 29 motion captures, which the participant must select, subjectively, the first and second most natural sequence.

We appreciate your time to help us with this evaluation.

Note: if you need to watch the video more than once, use the video player controls to navigate the video.
*Obrigatório

Movement - "Arrow"



http://youtube.com/watch?v=dbJ6c3N7AjA

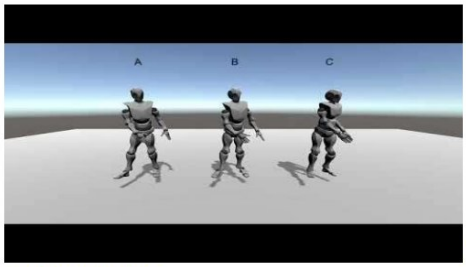
1. Movement - "Arrow" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.1: Questionário (1)

Movement - "Carry"



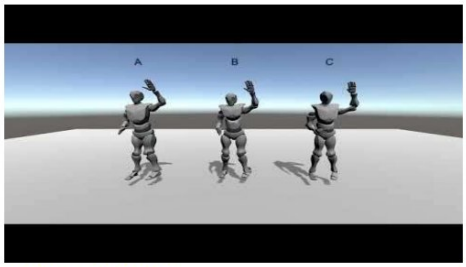
http://youtube.com/watch?v=2Sqdi6lKR7E

2. Movement - "Carry" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Clap"



http://youtube.com/watch?v=Ze1md0ab1b0

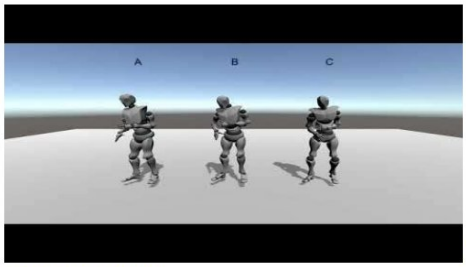
3. Movement - "Clap" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.2: Questionário (2)

Movement - "Clean"



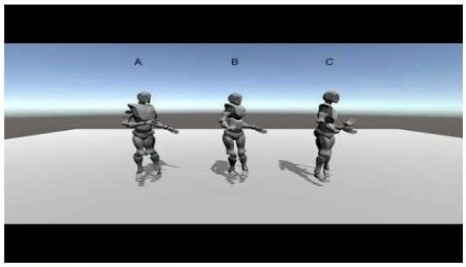
http://youtube.com/watch?v=Ubu97J0tBeI

4. Movement - "Clean" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Clean2"



http://youtube.com/watch?v=UdmriOtS1WY

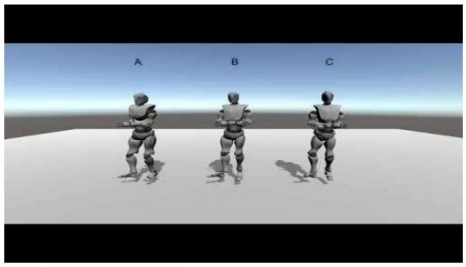
5. Movement - "Clean2" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.3: Questionário (3)

Movement - "Cross Arms"



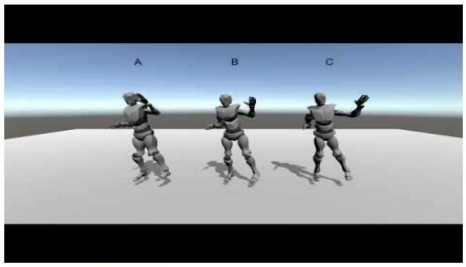
http://youtube.com/watch?v=DcAzm9utFZw

6. Movement - "Cross Arms" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Dance"



http://youtube.com/watch?v=FiVUPikg5Nk

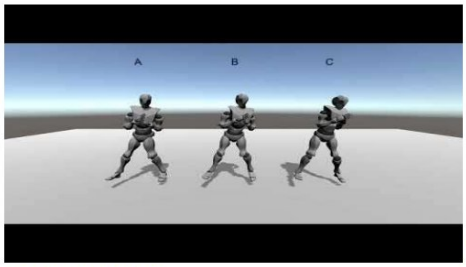
7. Movement - "Dance" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.4: Questionário (4)

Movement - "Dodge"



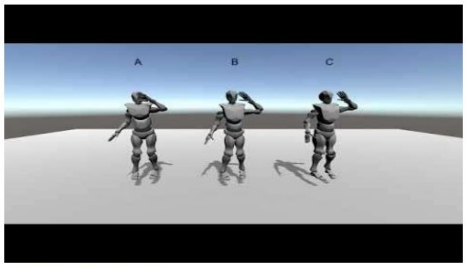
http://youtube.com/watch?v=7CjKTd8fVLA

8. Movement - "Dodge" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Drink"



http://youtube.com/watch?v=S3pFYr4EZP0

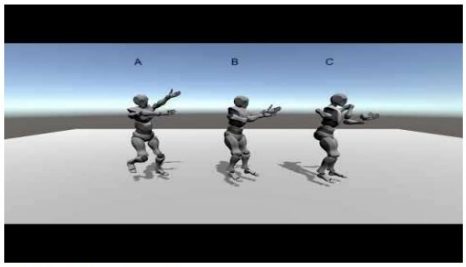
9. Movement - "Dodge" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.5: Questionário (5)

Movement - "Gun"



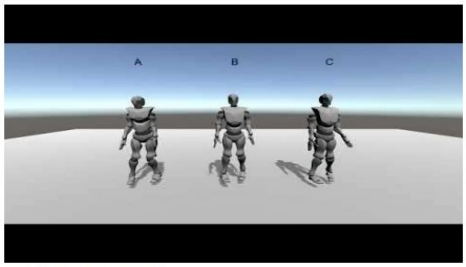
http://youtube.com/watch?v=Ga3sBJy7NBg

10. Movement - "Gun" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Gym"



http://youtube.com/watch?v=YPXSc5LEyDI

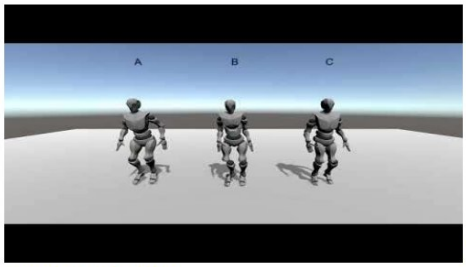
11. Movement - "Gym" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.6: Questionário (6)

Movement - "Jump"



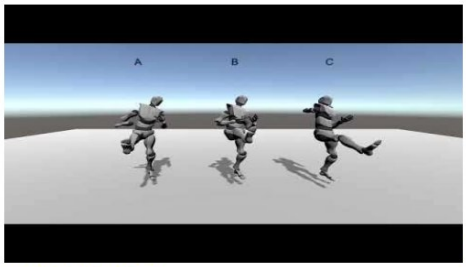
http://youtube.com/watch?v=kcdM55GOi9E

12. Movement - "Jump" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Kicks"



http://youtube.com/watch?v=Zei1I47GTV8

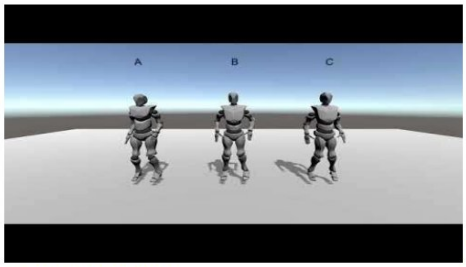
13. Movement - "Kicks" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.7: Questionário (7)

Movement - "Laugh"



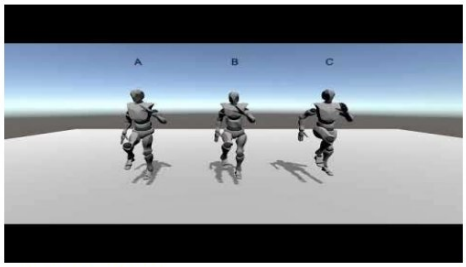
http://youtube.com/watch?v=sQvu7KLC5q0

14. Movement - "Laugh" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "March"



http://youtube.com/watch?v=BAkcjegQdvE

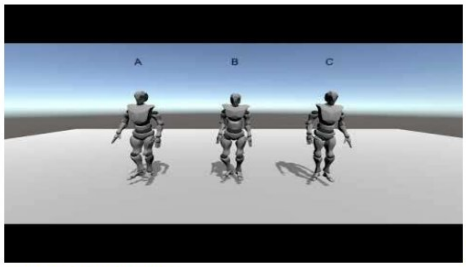
15. Movement - "March" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.8: Questionário (8)

Movement - "Pick and Throw"



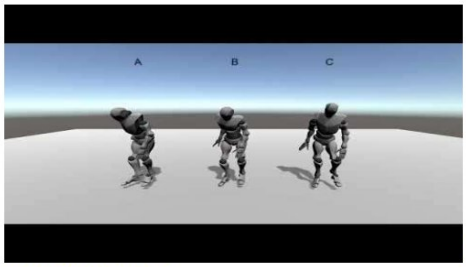
http://youtube.com/watch?v=DXqZpf50GBw

16. Movement - "Pick and Throw" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Pick Up"



http://youtube.com/watch?v=ZMF59rhJQgo

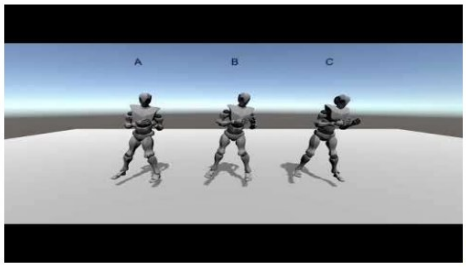
17. Movement - "Pick Up" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.9: Questionário (9)

Movement - "Punch"



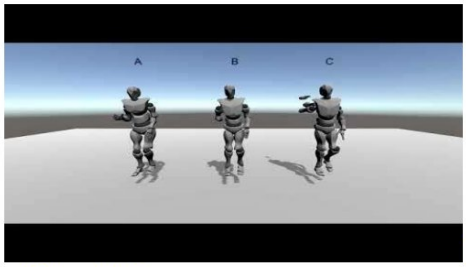
http://youtube.com/watch?v=IezsNCxvq1I

18. Movement - "Punch" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Run"



http://youtube.com/watch?v=8AkqnWgQcow

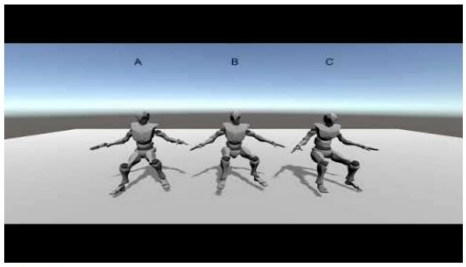
19. Movement - "Run" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.10: Questionário (10)

Movement - "Sit"



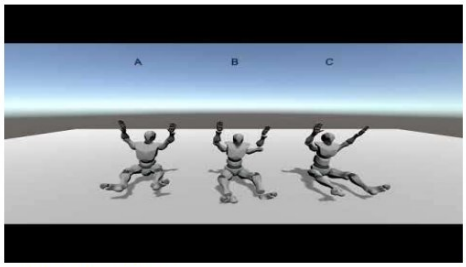
http://youtube.com/watch?v=QsRZo8-vfZU

20. Movement - "Sit" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Sit2"



http://youtube.com/watch?v=D5mQzyUzGJc

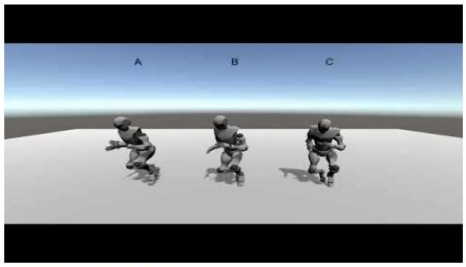
21. Movement - "Sit2" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.11: Questionário (11)

Movement - "Sneaking"



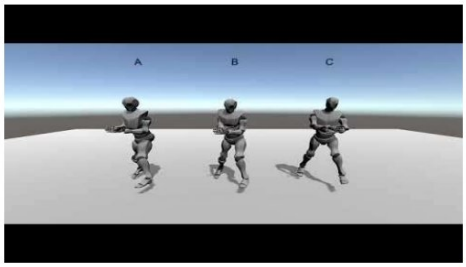
http://youtube.com/watch?v=1YWQ4EDzXOw

22. Movement - "Sneaking" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Sword"



http://youtube.com/watch?v=4j2YIqDfhAc

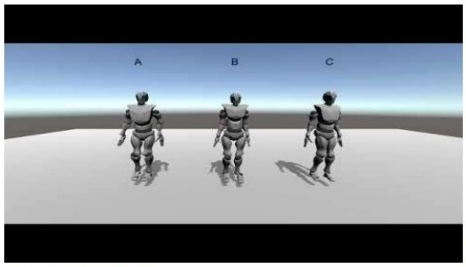
23. Movement - "Sword" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.12: Questionário (12)

Movement - "Talk"



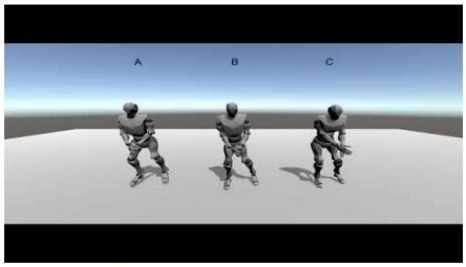
http://youtube.com/watch?v=Idisn3zx7UQ

24. Movement - "Talk" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Throw"



http://youtube.com/watch?v=IKRqPPpWGwU

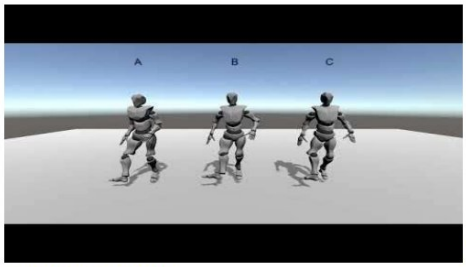
25. Movement - "Throw" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.13: Questionário (13)

Movement - "Throw2"



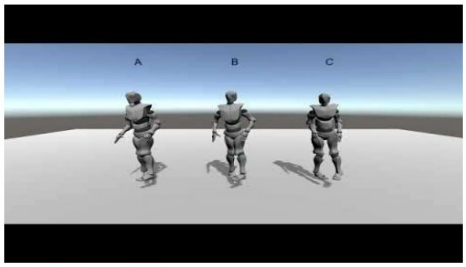
http://youtube.com/watch?v=uSCy8e5ASNY

26. Movement - "Throw2" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Wait"



http://youtube.com/watch?v=ic6VYgScsys

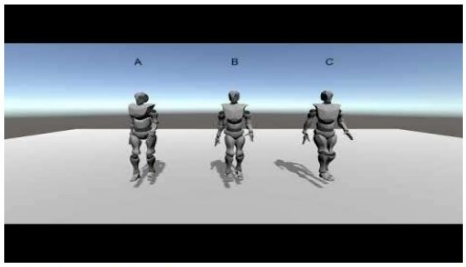
27. Movement - "Wait" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.14: Questionário (14)

Movement - "Walk"



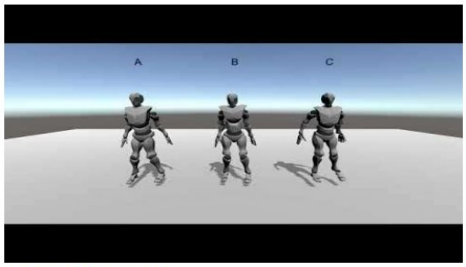
http://youtube.com/watch?v=lZ0SQnOY4sw

28. Movement - "Walk" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Movement - "Wear"



http://youtube.com/watch?v=Xqv5YxydmVc

29. Movement - "Wear" *

Marcar apenas uma oval por linha.

	A	B	C
Which is the most natural?	◯	◯	◯
Which is the second most natural?	◯	◯	◯

Figura A.15: Questionário (15)

B

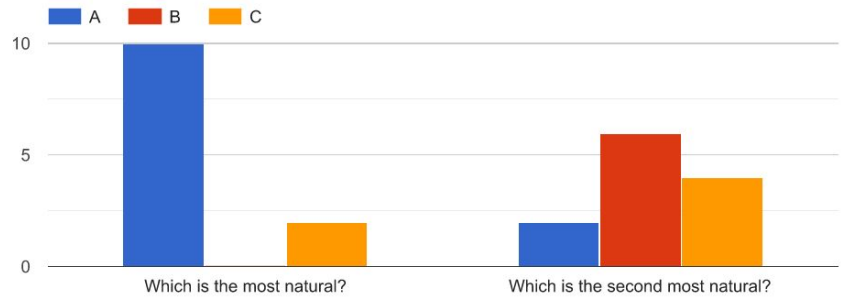
Respostas ao Questionário

Kinect Mocap

12 respostas

Publicar estatísticas

Movement - "Arrow"



Movement - "Carry"

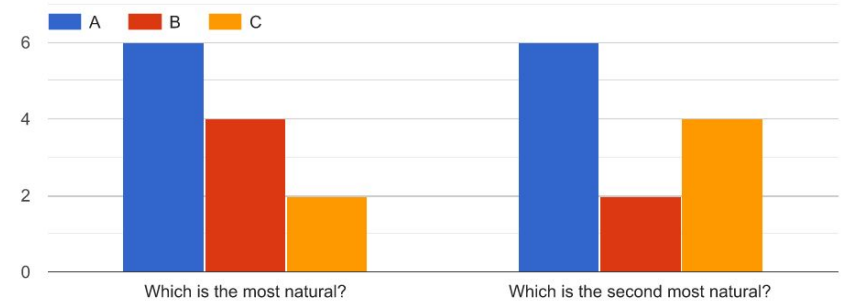


Figura B.1: Resposta (1)

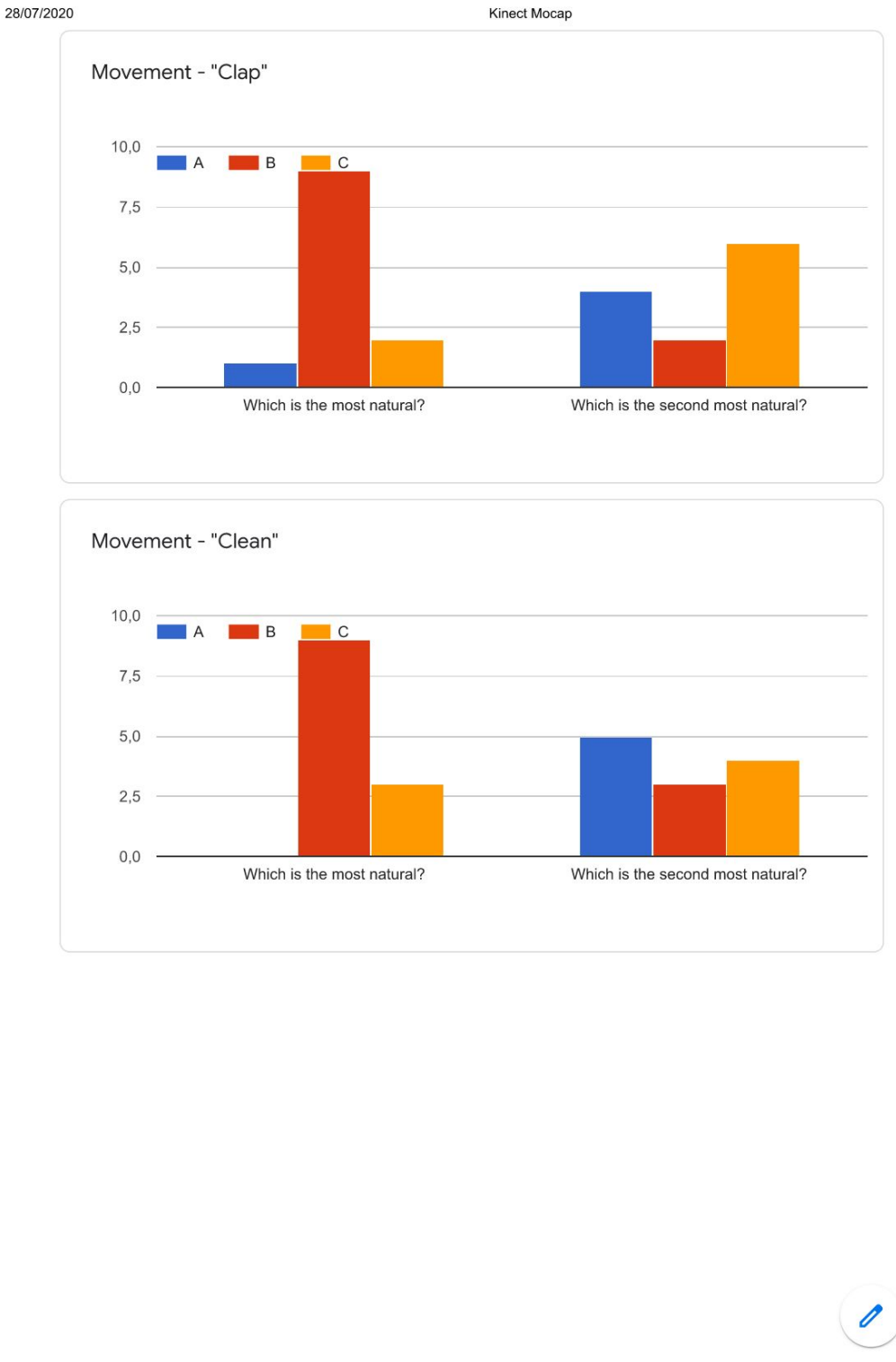


Figura B.2: Resposta (2)

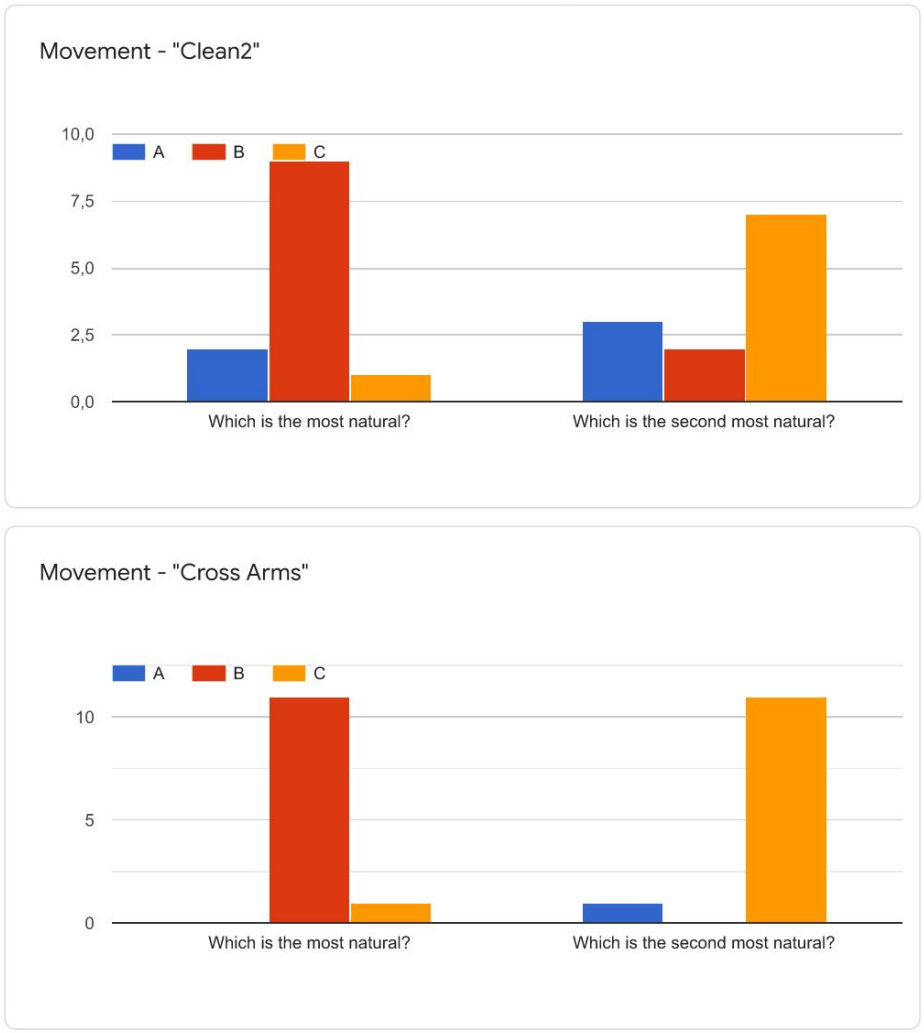


Figura B.3: Resposta (3)

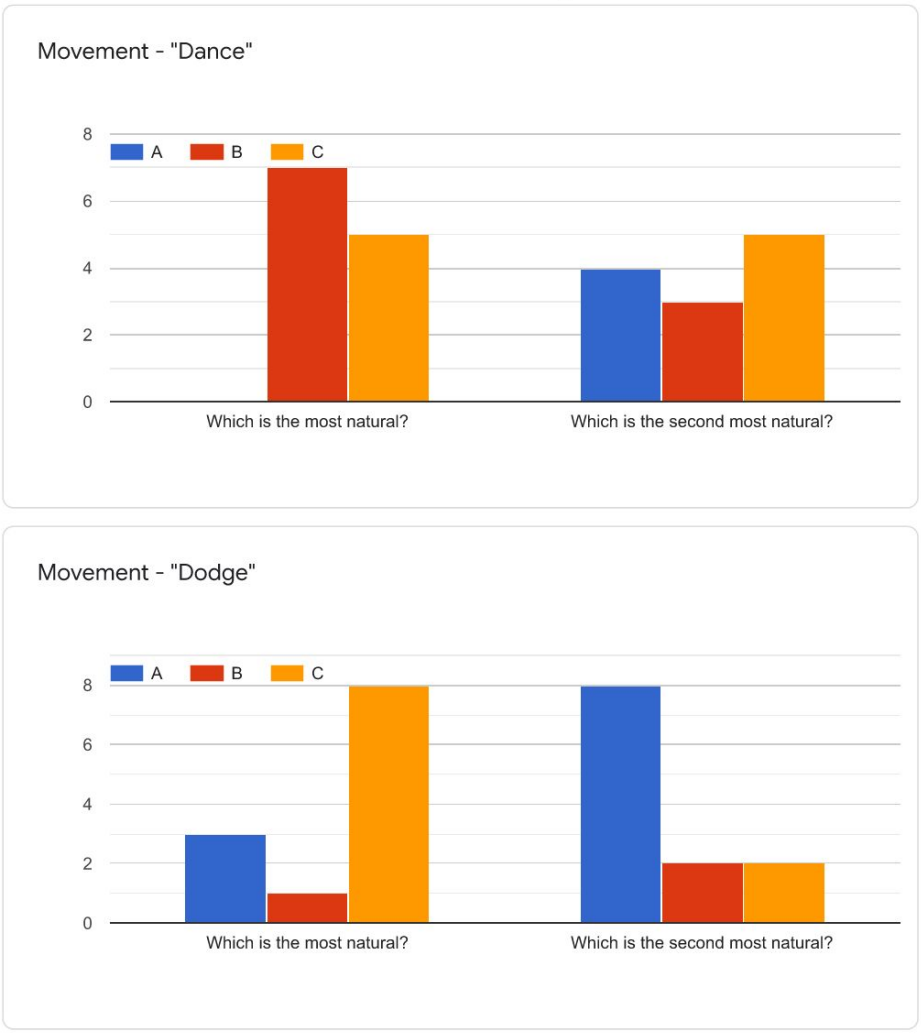


Figura B.4: Resposta (4)

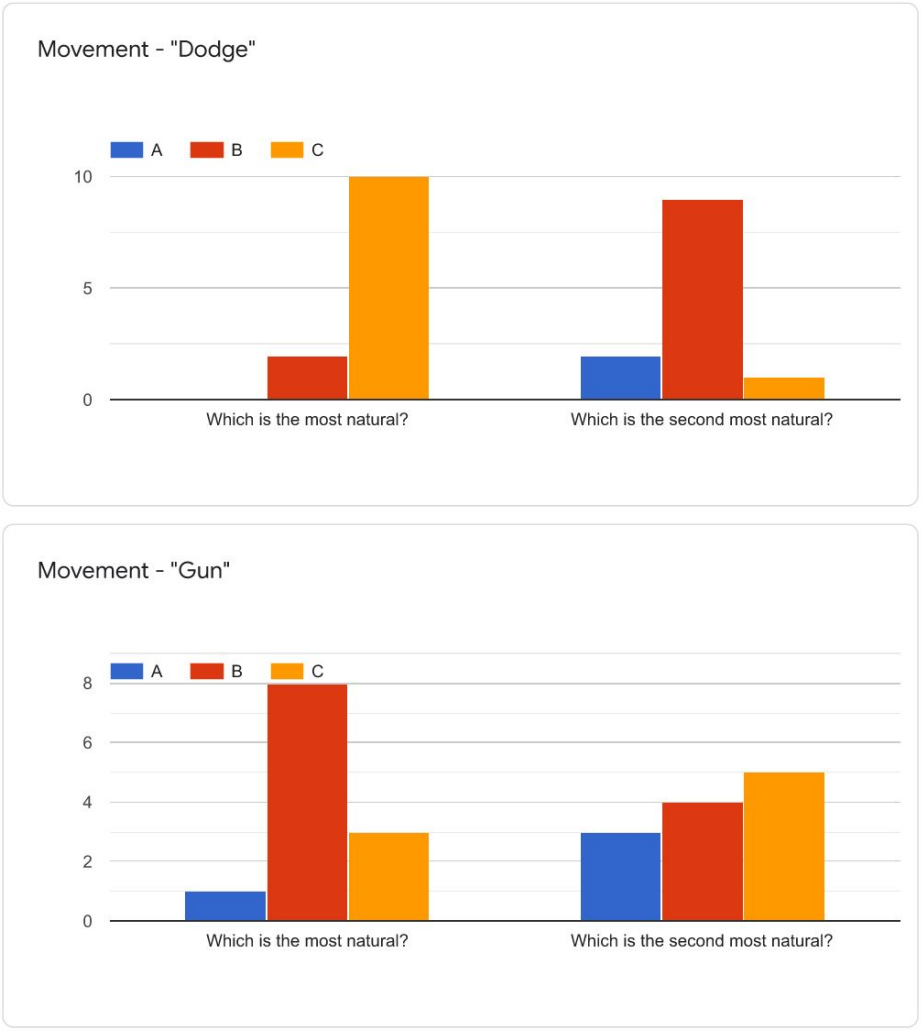


Figura B.5: Resposta (5)

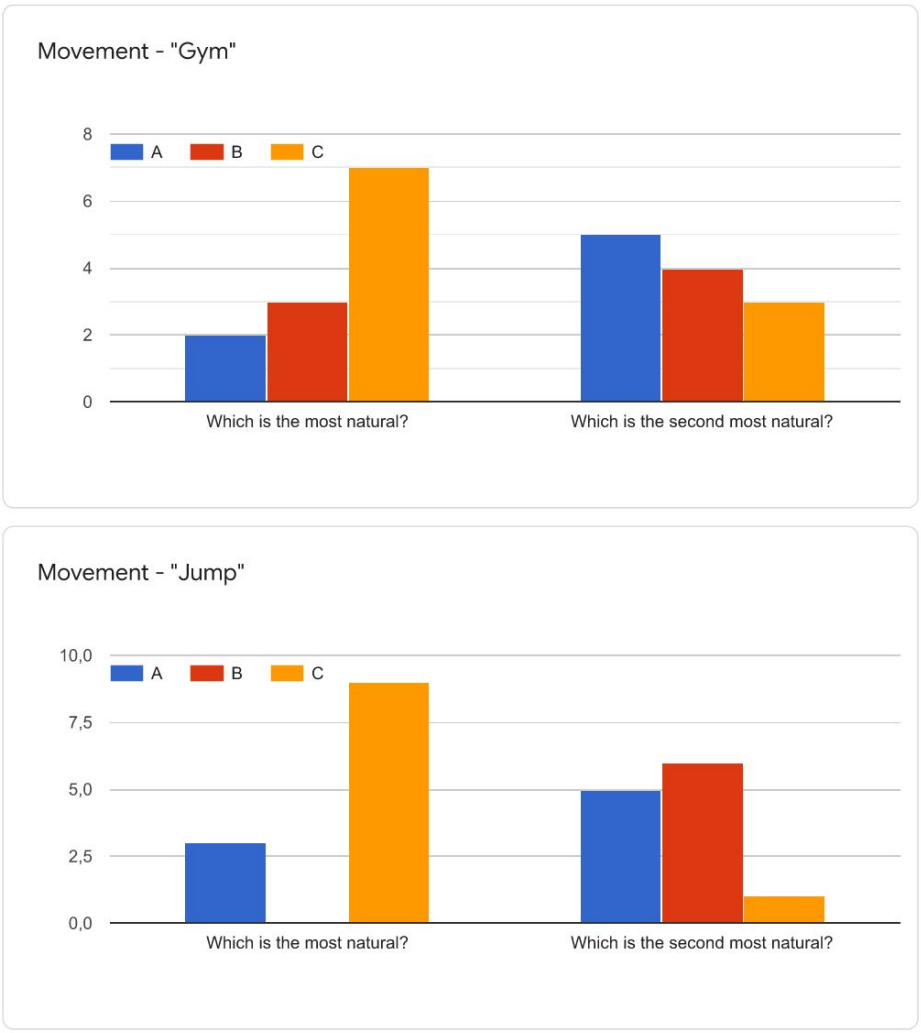
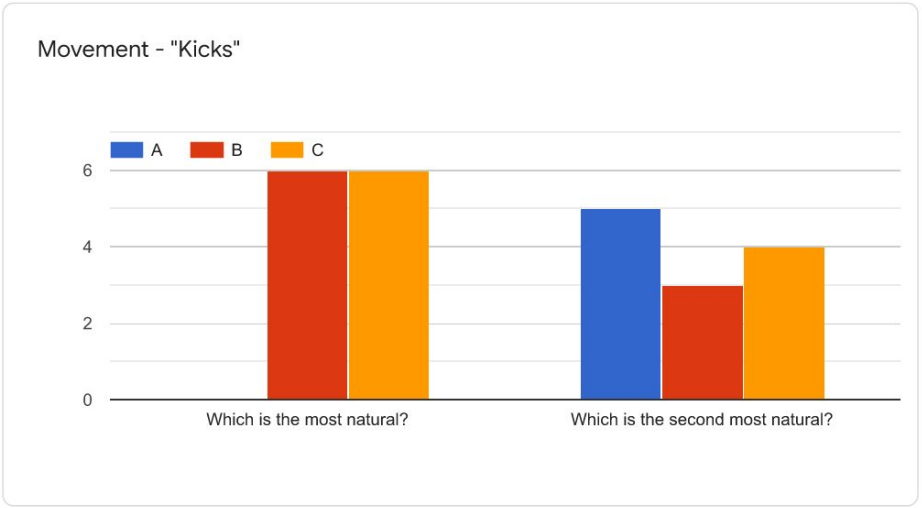


Figura B.6: Resposta (6)



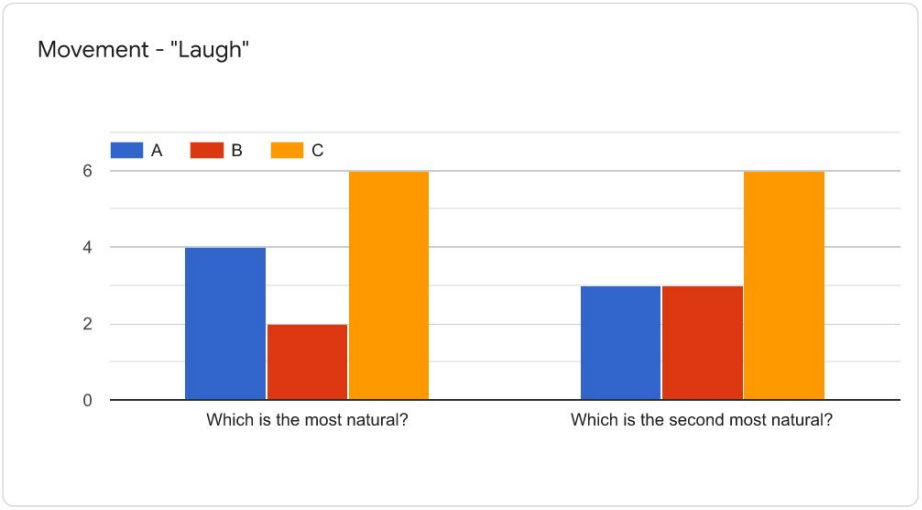


Figura B.7: Resposta (7)

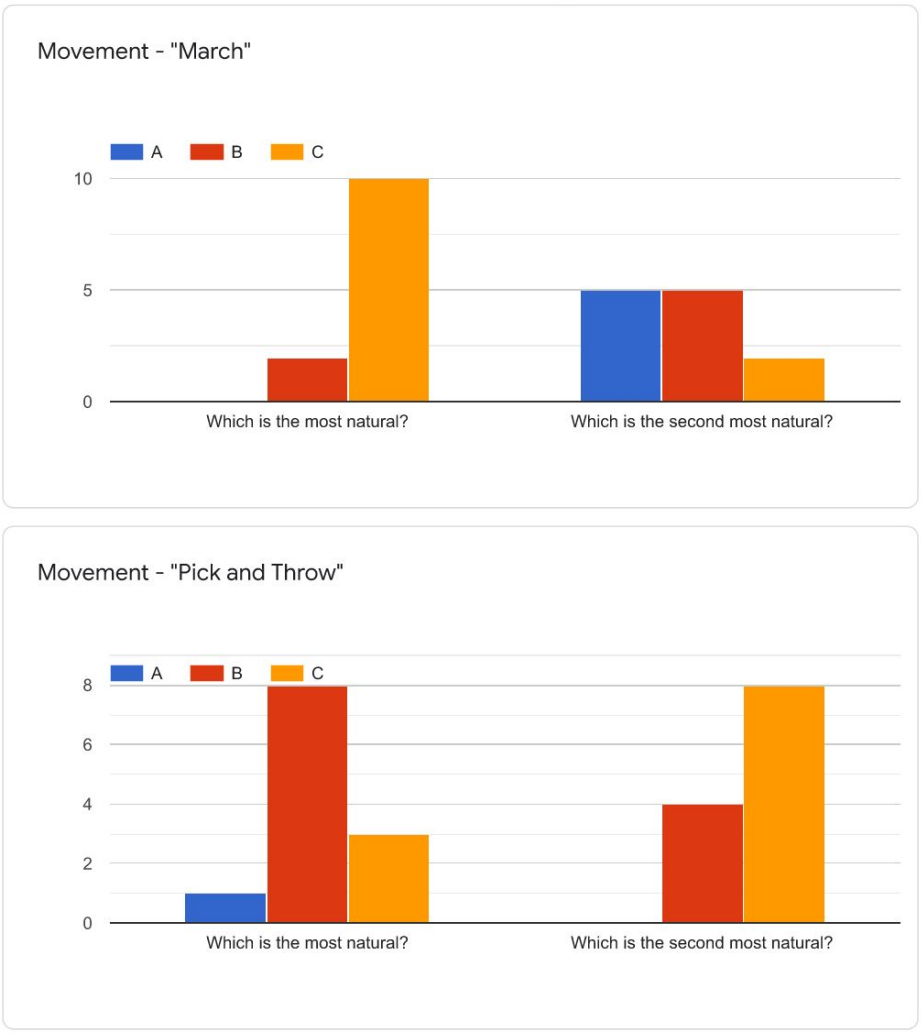


Figura B.8: Resposta (8)

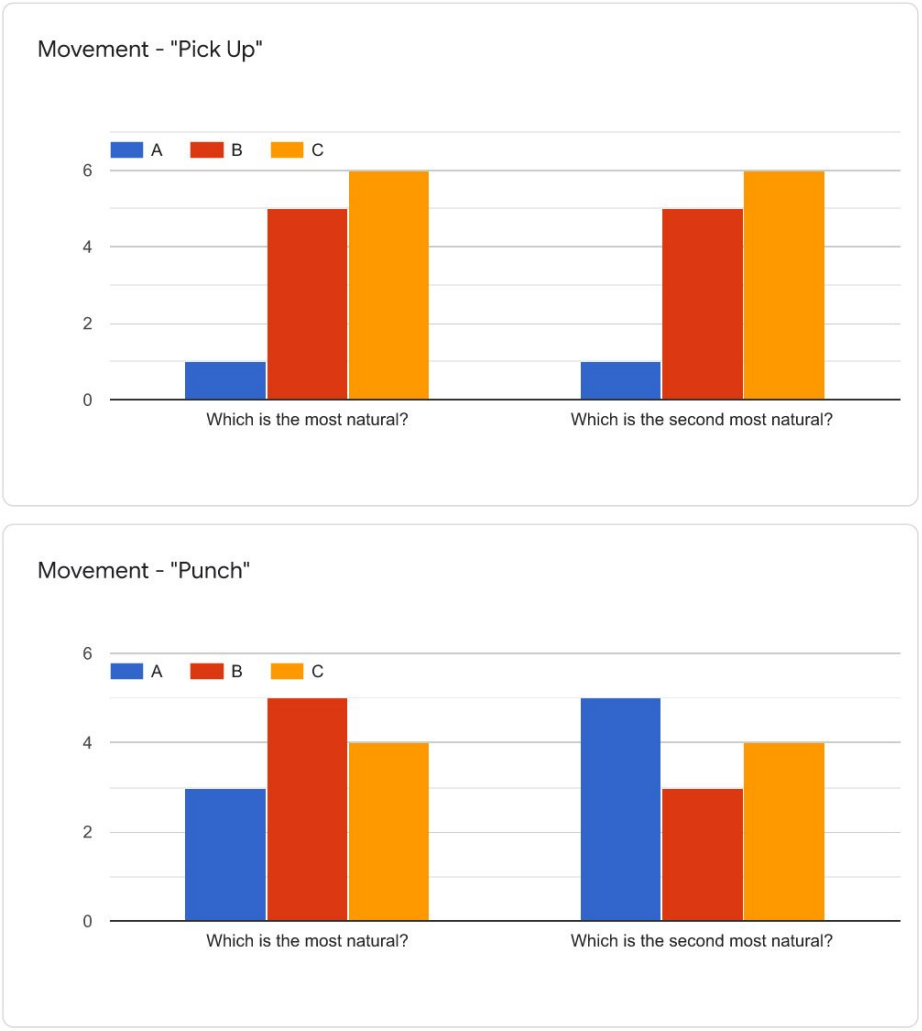


Figura B.9: Resposta (9)

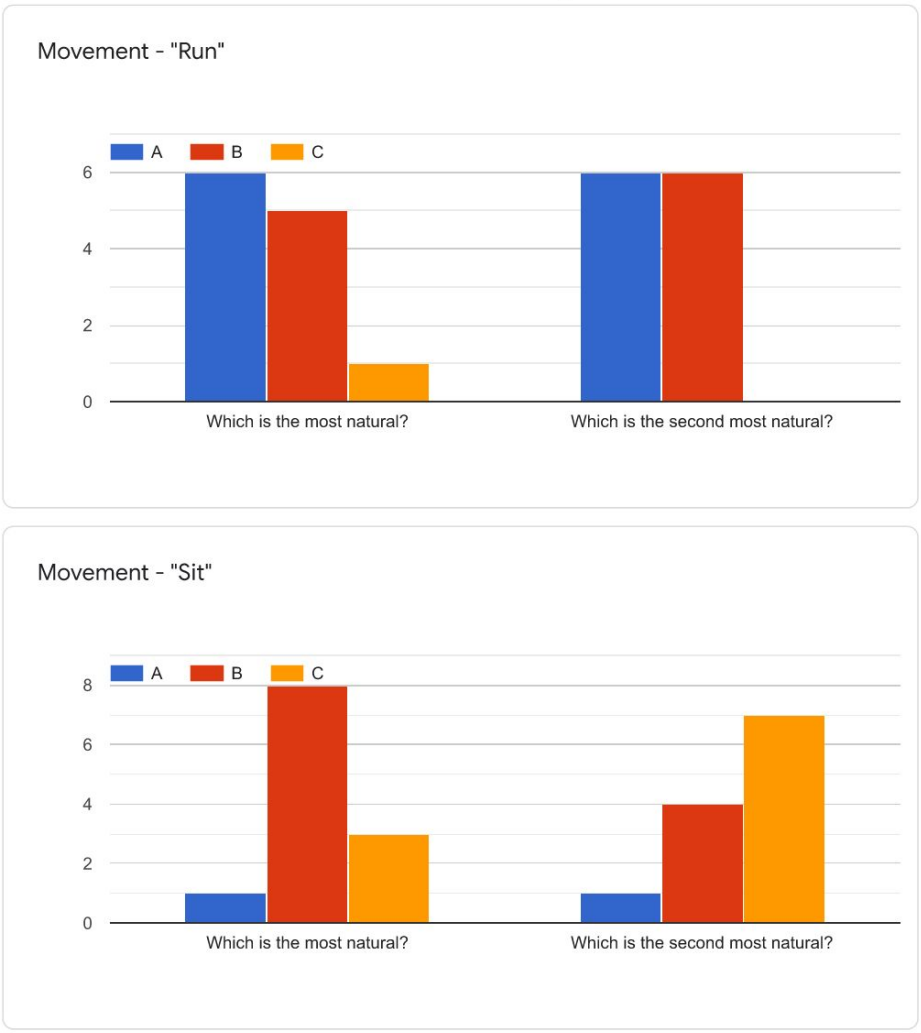


Figura B.10: Resposta (10)

Figura B.11: Resposta (11)

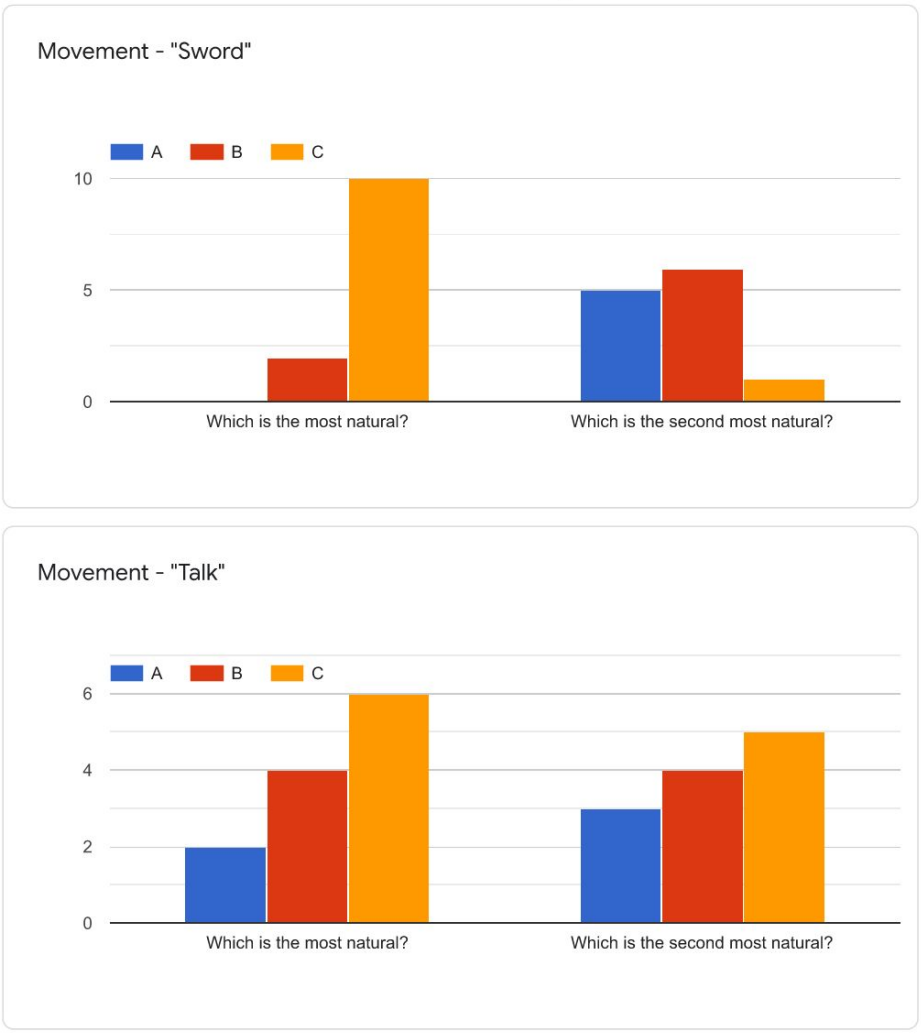


Figura B.12: Resposta (12)

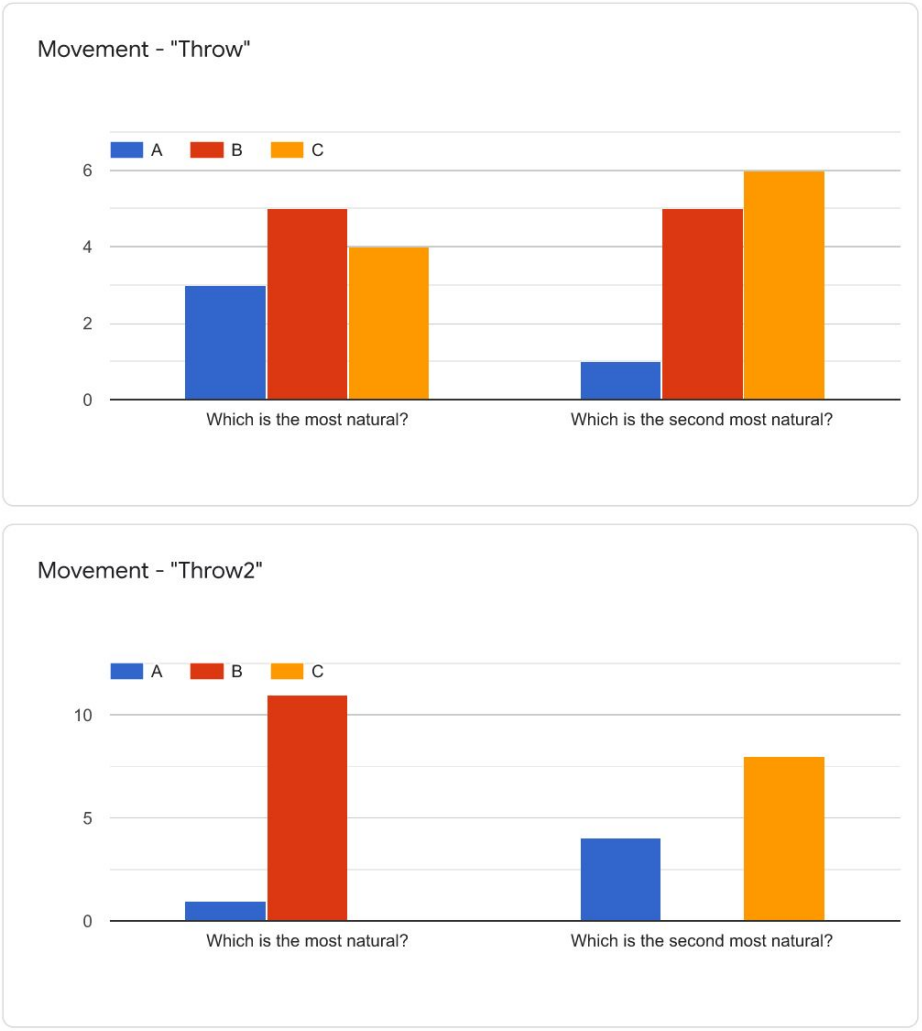


Figura B.13: Resposta (13)

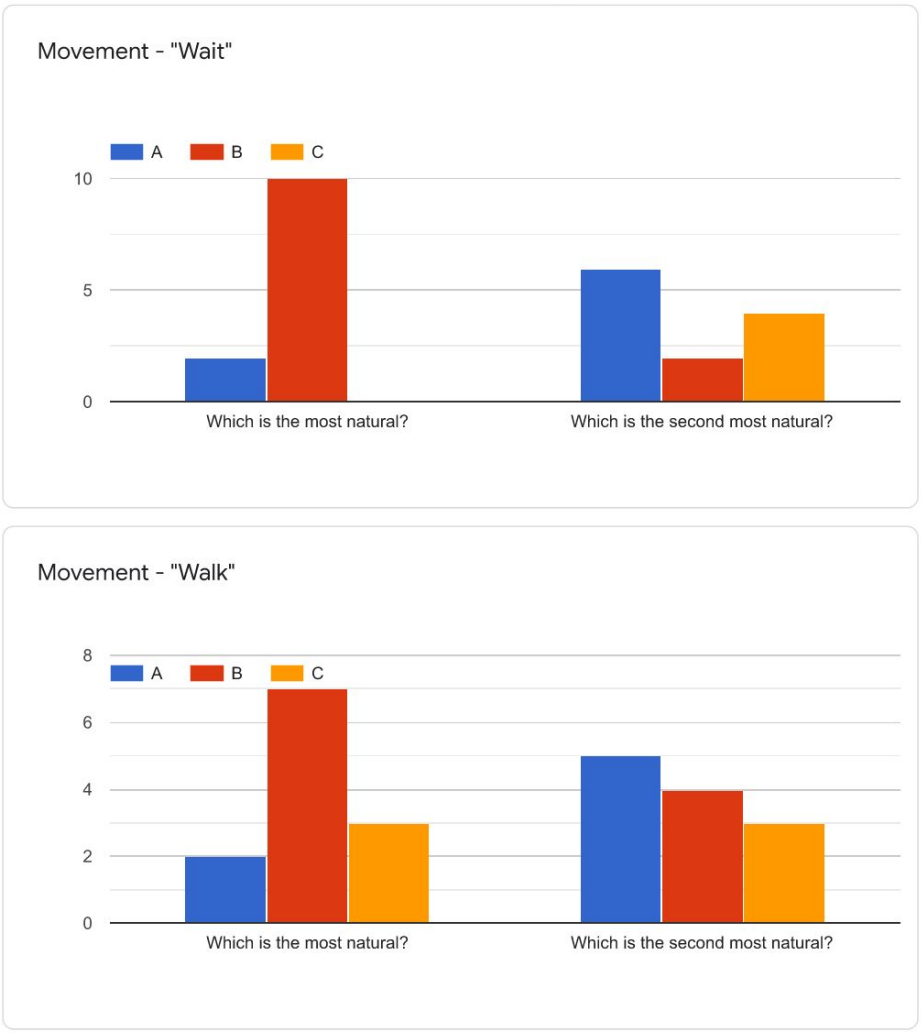


Figura B.14: Resposta (14)

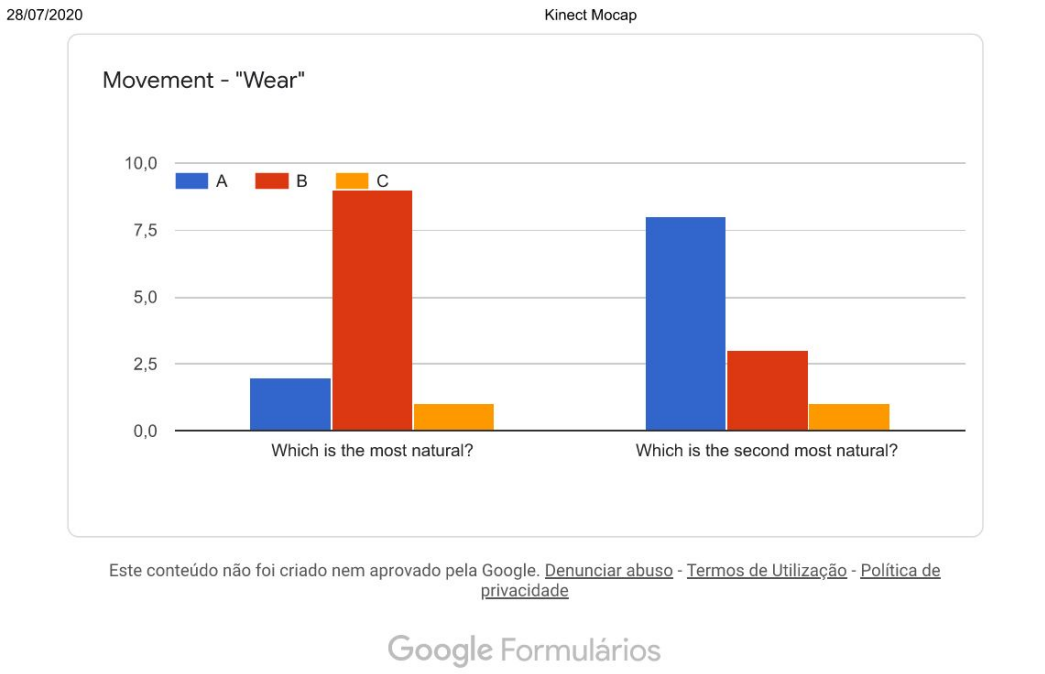


Figura B.15: Resposta (15)