applied sciences

MDPI

Article

Multi-Output Sequential Deep Learning Model for Athlete Force Prediction on a Treadmill Using 3D Markers

Milton Osiel Candela-Leal [1], Erick Adrián Gutiérrez-Flores [1], Gerardo Presbítero-Espinosa [1], Akshay Sujatha-Ravindran [2], Ricardo Ambrocio Ramírez-Mendoza [1], Jorge de Jesús Lozoya-Santos [1] and Mauricio Adolfo Ramírez-Moreno [1,*]

[1] Mechatronics Department, School of Engineering and Sciences, Tecnologico de Monterrey, Av. Eugenio Garza Sada 2501 Sur, Tecnológico, Monterrey 64849, NL, Mexico; a01197730@tec.mx (M.O.C.-L.); a00828091@tec.mx (E.A.G.-F.); presbitg@tec.mx (G.P.-E.); ricardo.ramirez@tec.mx (R.A.R.-M.); jorge.lozoya@tec.mx (J.d.J.L.-S.)

[2] Independent Researcher, California City, CA 94022, USA; akshay.s.ravindran@gmail.com

[*] Correspondence: mauricio.ramirezm@tec.mx

Abstract: Reliable and innovative methods for estimating forces are critical aspects of biomechanical sports research. Using them, athletes can improve their performance and technique and reduce the possibility of fractures and other injuries. For this purpose, throughout this project, we proceeded to research the use of video in biomechanics. To refine this method, we propose an RNN trained on a biomechanical dataset of regular runners that measures both kinematics and kinetics. The model will allow analyzing, extracting, and drawing conclusions about continuous variable predictions through the body. It marks different anatomical and reflective points (96 in total, 32 per dimension) that will allow the prediction of forces (N) in three dimensions (F_x, F_y, F_z), measured on a treadmill with a force plate at different velocities (2.5 m/s, 3.5 m/s, 4.5 m/s). In order to obtain the best model, a grid search of different parameters that combined various types of layers (Simple, GRU, LSTM), loss functions (MAE, MSE, MSLE), and sampling techniques (down-sampling, up-sampling) helped obtain the best performing model (LSTM, MSE, down-sampling) achieved an average coefficient of determination of 0.68, although when excluding F_z it reached 0.92.

Keywords: biomechanics; artificial intelligence; deep learning; force prediction; treadmill; dataset; recurrent neural network; LSTM

check for updates

Citation: Candela-Leal, M.O.; Gutiérrez-Flores, E.A.; Presbítero-Espinosa, G.; Sujatha-Ravindran, A.; Ramírez-Mendoza, R.A.; Lozoya-Santos, J.d.J.; Ramírez-Moreno, M.A. Multi-Output Sequential Deep Learning Model for Athlete Force Prediction on a Treadmill Using 3D Markers. *Appl. Sci.* **2022**, *12*, 5424. https://doi.org/10.3390/app12115424

Academic Editors: Rosa Senatore, Antonio Parziale and Nicole Dalia Cilia

Received: 1 April 2022
Accepted: 18 May 2022
Published: 27 May 2022

1. Introduction

The development of reliable methods to estimate generated forces during physical tasks is critical in the field of biomechanical sports research [1]. The study and analysis of forces exerted by athletes when conducting their activities can improve performance and technique, and reduce the risks of injuries and fractures [2]. Wrongly executed self-performed movements in sports might result in unexpected injuries and/or fractures, for which the causes include bad posture [3], inadequate technique [4], generation of dangerous biomechanical forces in joints [5], and impact [6], among others.

To prevent undesired injuries, the refinement of methods to estimate biomechanical forces in a reliable manner is needed. Traditional setups of force analyses in biomechanics include the use of force platforms to obtain three-dimensional force measurements when jumping [7], walking, running [8]; flexible force sensors (FFS) for gait detection and sport training [9]; inertial measurement units (IMUs) [10]; and strain sensors [11].

Although these sensors provide insight into the generated forces to some extent, it is limiting due to the confined space (platform) and the limitation of movement caused by the attachment of sensors. A less limiting approach is the use of video tools to estimate the generation of forces in physical tasks; however, its progress in biomechanical research has been relatively slow when compared to its use in the fields of robotics [12,13] and automated navigation [10]. According to a recent review on video-based biomechanics tools for injury assessment in sports, there is a gap in the development of real-time applications in this field [14]. The aim and scope of this study was to demonstrate the results of a proposed framework using a database of physical and biomechanical markers of joints during treadmill exercises.

The use of video for biomechanics research is not new; there have been various studies analyzing biomechanical parameters from videos of athletes. Two sub-fields exist when discussing video-based biomechanical tools: marker and markerless analysis. Marker-based-tools rely on placing markers onto the athlete's joints to track them in the video (and thenceforth process relevant information); markerless research is not constrained by this and instead uses artificial vision strategies to identify the joints in the video [10,15].

Although more complex (computationally speaking), markerless analysis is better suited for real-life scenarios (e.g., daily activities outside the laboratory). Although some studies reported biomechanical sports research using both methods, the vast majority focused on obtaining joint angles (to assess and correct technique [16], or risk of injuries [4]) and the position/speed of relevant markers [17] and did not analyze the generated forces in a detailed manner.

A recurrent neural network (RNN) is a type of artificial neural network (ANN) which is composed of a series of neurons that learn from data and adapt to it by changing their weights, hence being dynamic in the feed-forward process. More specifically, RNNs have the peculiarity that their structure allows them to learn temporal sequences [18]. This, in turn, makes them the gold standard for analyzing biomechanics, kinetics, and kinematics in a three-dimensional motion environment. Recent neuroscience findings have suggested that motor cortices can be modeled as a dynamical system. This perspective is opposite to the traditional view that neural activities in motor cortices represent the kinetic parameters of movements [19].

The adoption of neural networks allows the computation of joint contact force and muscle force via musculoskeletal modeling techniques, which offer valuable metrics for describing movement quality, thereby having numerous clinical applications and research applications. Critical applications include exercise-induced fatigue (considered one of the essential factors leading to musculoskeletal disorders), detection of fall-risk-linked balance impairment, and the ability to measure movement behavior through "skeletonized" data and 3D shapes.

Image analysis in [20] is used to extract biomechanical features in swimming; this is a common practice for stroke correction, technique analysis, and as a visual aid for the athletes. Due to how different the conditions in the water are, this leads to problems such as blurred vision from the camera and bubbles from cavitation effects. Image analysis techniques are fundamental to enhancing clarity and overcoming this problem, and to automating the detection of limb segments related to important metrics.

Fall prediction could use markers' positions [21] or bone maps estimations via a deep learning (DL) approach, composed of a convolutional neural network (CNN) with 97.1% accuracy [22]. In [23], an ANN predicted lower extremity joint torque based on ground reaction force (GRF) features, using the GRF and related parameters derived by the GRF during counter-movement jump (CMJ) and squat jump (SJ), calculating joint torque using inverse dynamics and an ANN.

The following cost functions were used to track the model's performance through epochs: mean absolute error (MAE) [24], mean squared error (MSE) [25], and mean squared logarithmic error (MSLE) [26].

Based on a database of 28 regular runners, physical marker positions based on 32 anatomical and reflective markers, resulting in 96 tags, were sampled at 150 Hz. A model used these markers to predict three-dimensional forces (F_x, F_y, F_z) sampled at 300 Hz. However, due to differences between sampling frequencies, for each marker datum (source), two force data were given (target), making it impossible to merge both datasets and generate a prediction using a model. In this sense, two methods solved data granularity to fit a machine learning (ML) model, thereby matching both data streams and merging them into a single dataset, divided into down-sampled (force) and up-sampled (positions).

In the sections of this report:

- We describe the methodology of the study: a description of the database and variables used (in the analysis), the architectures of the evaluated models, and their performance tests, in Section 2.
- The results from the proposed analysis are shown in Section 3.
- In-depth discussion going into aspects such as future work and limitations is in Section 4.
- Finally, this paper concludes in Section 5 with the discoveries of this work.

2. Materials and Methods

2.1. Dataset

A public dataset was the primary resource for processing and analysis in this work. This public dataset from [8] consists of a sample of 28 regular runners, which had familiarity and comfort with running on a treadmill, weekly mileage greater than 20 km, and a minimum average running pace of 1 km in 5 min during 10-km races. The dataset contains 48 anatomical and reflective markers (mm), of which 20 are anatomical, 20 technical, and 8 technical/anatomical; these were tracked using a 3D motion-capture system of 12 cameras, which had 4 Mb resolution, and Cortex 6.0 software (Raptor-4, Motion Analysis, Santa Rosa, CA, USA).

During a trial, researchers recorded only data from technical and technical/anatomical markers. In contrast, most anatomical markers were used for a standing calibration trial during the calibration phase, except for the first and fifth metatarsal: (R.MT1, L.MT1) and (R.MT5, L.MT5), respectively; thus, all fully available $20 + 8 + 4 = 32$ markers were considered as source features when using their three dimensions. These resulted in $32 \times 3 = 96$ features. Unfortunately, as shown in Section 3, it was found that some markers (2 anatomical and 1 technical/anatomical) have missing values in the dataset; these are: left 1st metatarsal (L.MT1), right 1st metatarsal (R.MT1), and left anterior superior lilac spine (L.ASIS); these 9 (3×3) features were removed from the initial source features considered; hence, data from only 87 features were used ($96 - 9 = 87$).

Physical marker protocol in Figure 1—the figure created by the authors of the dataset [8]. Table 1 is a descriptive table of each marker used, ans was also extracted from the dataset's authors [8].

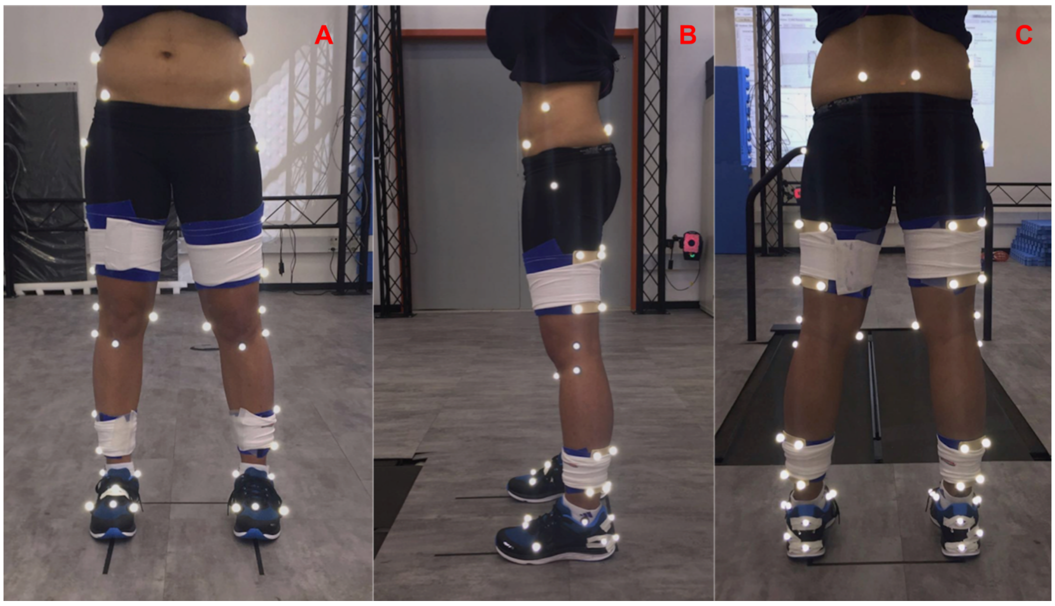

Figure 1. Markers' placement protocol during an anatomical calibration trial, including the anterior (**A**), lateral (**B**), and posterior (**C**) views, obtained from [8].

Table 1. Description of each marker's ID, label, and type to determine whether it is either anatomical (A) or technical (T), based on the anatomical position, obtained from [8].

Marker #	Label	Type	Name
1	R.ASIS	T/A	Right anterior superior lilac spine
2	L.ASIS	T/A	Left anterior superior lilac spine
3	R.PSIS	T/A	Right posterior lilac spine
4	L.PSIS	T/A	Left posterior lilac spine
5	R.Lilac.Crest	T	Right lilac crest
6	L.Lilac.Crest	T	Left lilac crest
7	R.Thigh.Top.Lateral	T	Right thigh top lateral marker
8	R.Thigh.Bottom.Lateral	T	Right thigh bottom lateral marker
9	R.Thigh.Top.Medial	T	Right thigh top medial marker
10	R.Thigh.Bottom.Medial	T	Right thigh bottom medial marker
11	R.Shank.Top.Lateral	T	Right shank top lateral marker
12	R.Shank.Bottom.Lateral	T	Right shank bottom lateral marker
13	R.Shank.Top.Medial	T	Right shank top medial marker
14	R.Shank.Bottom.Medial	T	Right shank bottom medial marker
15	R.Heel.Top	T/A	Right heel top
16	R.Heel.Bottom	T/A	Right heel bottom
17	R.Heel.Lateral	T	Right heel lateral
18	L.Thigh.Top.Lateral	T	Left thigh top lateral marker
19	L.Thigh.Bottom.Lateral	T	Left thigh bottom lateral marker
20	L.Thigh.Top.Medial	T	Left thigh top medial marker
21	L.Thigh.Bottom.Medial	T	Left thigh bottom medial marker
22	L.Shank.Top.Lateral	T	Left shank top lateral marker
23	L.Shank.Bottom.Lateral	T	Left shank bottom lateral marker
24	L.Shank.Top.Medial	T	Left shank top medial marker
25	L.Shank.Bottom.Medial	T	Left shank bottom medial marker
26	L.Heel.Top	T/A	Left heel top
27	L.Heel.Bottom	T/A	Left heel bottom
28	L.Heel.Lateral	T	Left heel lateral
29	R.GTR	A	Right greater trochanter
30	R.Knee	A	Right knee
31	R.Knee.Medial	A	Right knee medial
32	R.HF	A	Right head of fibula
33	R.TT	A	Right tibial tuberosity
34	R.Ankle	A	Right ankle
35	R.Ankle.Medial	A	Right ankle medial
36	R.MT1	A	Right 1st metatarsal
37	R.MT5	A	Right 5th metatarsal
38	R.MT2	A	Right 2nd metatarsal
39	L.GTR	A	Left greater trochanter
40	L.Knee	A	Left knee
41	L.Knee.Medial	A	Left knee medial
42	L.HF	A	Left head of fibula
43	L.TT	A	Left tibial tuberosity
44	L.Ankle	A	Left ankle
45	L.Ankle.Medial	A	Left ankle medial
46	L.MT1	A	Left 1st metatarsal
47	L.MT5	A	Left 5th metatarsal
48	L.MT2	A	Left 2nd metatarsal

By only considering technical (20), technical/anatomical (8), and 4 anatomical markers, the dataset then contained recording data from 32 markers in 3 dimensions (XYZ), which resulted in a total of 96 marker positions ($32 \times 3 = 96$), sampled at a frequency of 150 Hz and used as source features. A predictive model would use such features to predict the target features, which were forces (N) in three directions (F_x, F_y, F_z), sampled at a frequency of 300 Hz. An instrumented, dual-belt treadmill (FIT, Bertec, Columbus, OH, USA) collected these forces while the subjects performed the requested trials. Trials consisted of three phases: the subject walking at 1.2 m/s for 1 min; then the treadmill increased its speed incrementally to 2.5 m/s, and after 3 min, data were recorded for 30 *s*, which was repeated at speeds of 3.5 m/s and 4.5 m/s; after the trials, the speed was set back to 1.2 m/s for a 1-min cool-down period before stopping completely.

The sampling frequency of target features was doubled to 300 Hz, compared to the sampling frequency of source features of 150 Hz, generating more data for the target features concerning source features. Thus, for each sample in the markers (source) dataset, two examples are in the forces (target) dataset, making it impossible to merge both datasets and create a prediction using a model. Two methods solved the data granularity issue, matched both datasets, and combined them into a single dataset to fit an ML model:

- Down-sampling: Given that the forces dataset has double the samples of the markers dataset, index numbers could eliminate odd index numbers and thus cut the dataset by half. This sequential elimination of samples removed data from the doubled dataset but allowed us to join both datasets without bias in predicting a sample using the current data.
- Up-sampling: Given that the markers dataset has half the samples of the forces dataset, blank numbers were created in odd indices, thereby expanding the dataset with empty data to predict the missing values. In this sense, linear interpolation made the dataset, given a series of odd numbers and predictions for even numbers, having one valid number, one blank number, and one valid number. This method can introduce bias due to the creation of new points, and they might not follow a linear trend as assumed, although it conserves all the data.

The dataset is composed of forces and 3D marker locations of running trials for multiple treadmill velocities, 2.5, 3.5, and 4.5 m/s for each regular runner. The total number of subjects determined the data division, based on a 70:10:20 split for training, validation, and testing datasets. Considering the total number of participants, $n = 28$, and the aforementioned data-split, $n_{training} = 19$, $n_{validation} = 3$, $n_{testing} = 6$. Figure 2 represents the applied data division, where proportional data from each velocity were extracted based on randomly sampled subject ID, and the original seed was kept for reproduction purposes. The model's performance was obtained using a single data split, with fixed participants' IDs assigned to each dataset independently of treadmill velocity, thereby ensuring the three datasets had equal data proportions of each speed.

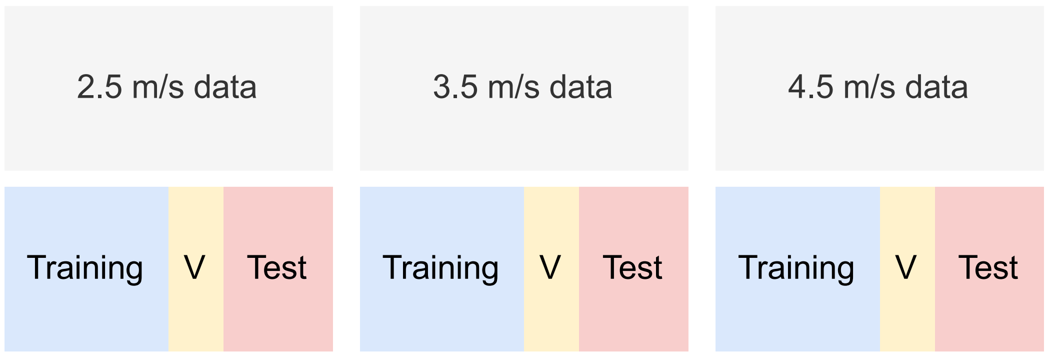

Figure 2. Data division, using a 70:10:20 split into training, validation (V) and testing (test) datasets, for each treadmill velocity, based on random sampling of subjects proportional to each dataset.

2.2. Loss Functions

In order to measure the performance of continuous variable predictions, a set of evaluation metrics were used as loss functions so that the DL-based model's performance could be evaluated, and hence its weights updated. A cost function compares a model's predictions and the actual data; in a regression model, the function computes the distances between reference and predicted values [27]. The objective is to minimize the evaluation metrics, as they measure the differences between the reference values and their predictions.

The first loss function used was the MAE, which represents the average absolute of the differences between predicted values and observed values, calculated as shown in Equation (1), where the difference between the reference value y_i and the predicted value $\hat{y}_i$ is summed over n testing samples. MAE follows the L1-norm of normalization or the Manhattan norm, which measures the distance between two points by summing the absolute differences between measurements in all dimensions [28].

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{1}$$

Another metric used was the MSE, which represents the average of squares of the differences between predicted values and observed values, calculated as shown in Equation (2), where the squared difference between the reference value y_i and the predicted value $\hat{y}_i$ is summed over n testing samples. MSE follows the L2-norm of normalization or the Euclidean norm, which measures the Euclidean distance between two points in a given vector space; this distance is significantly affected by outliers when compared to the L1-norm, which takes the absolute value of differences between two points as $\epsilon^2 > \epsilon$ [28].

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2}$$

Finally, MSLE measures the ratio between predicted and observed values; it is quite different from the other metrics because it measures the percent of difference and thus is better for extreme values [28]; it is calculated as shown in Equation (3), where the squared difference between the logarithm of the reference value y_i, and the predicted value $\hat{y}_i$ is summed over n testing samples—regularized using the log function on predictions and reference values. The $+1$ operation removes errors due to log being a 0-sensitive function. In addition, negative values are not possible in this metric (one of the reasons why all the data was scaled using MinMaxScaler).

$$MSLE = \frac{1}{n}\sum_{i=1}^{n}(\log(y_i + 1) - \log(\hat{y}_i + 1))^2 \tag{3}$$

2.3. Performance Metrics

Coefficient of determination (R^2) is a widely used regression performance metric to measure how well the predictions of a model fit the actual data [28]. The metric's domain is $R^2 \leq 1$, where $R^2 = 1$ means perfect predictions, $R^2 = 0$ is the baseline model that predicts $\bar{y}$, and $R^2 < 0$ means poor predictions; it is calculated as shown in Equation (4). It is composed of both the sum of squared estimates of error (SSE) and the sum of squares total (SST), Equations (5) and (6), respectively, and was calculated for each target feature (F_x, F_y, F_z) using the testing dataset. When the error between predictions and the reference value is minimized, $SSE \approx 0$, so $R^2 = 1$, which is concordant with the aforementioned fact, as the minimal error between samples would mean the predictive model is perfect, and hence, $R^2 = 1$.

$$R^2 = 1 - \frac{SSE}{SST} \tag{4}$$

SSE, known as the deviation between predicted and reference values, was calculated as shown in Equation (5). $\hat{y}_i$ refers to the ith prediction and y_i to the reference value, and their squared difference is summed across n testing samples.

$$SSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5}$$

SST, known as the total variation in the data, was calculated as shown in Equation (6). y_i refers to the ith sample and $\bar{y}$ to the target feature's mean, and their squared difference is summed across n testing samples.

$$SST = \sum_{i=1}^{n} (y_i - \bar{y})^2 \tag{6}$$

The Pearson correlation coefficient (r) was also used as a performance metric to measure the linear relationship between predictions and reference values [28]. The metric's domain is $-1 \leq r \leq 1$: $r = 1$ means a strong positive linear relationship, $r = -1$ means a strong negative linear relationship, and $r = 0$ means a poor linear relationship. It is calculated as shown in Equation (7). x_i is the ith sample of X series (in this case $\hat{y}$ for predictions) and y_i is the ith sample of Y series (in this case y for reference).

$$r_{xy} = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^{n} (y_i - \bar{y})^2}} \tag{7}$$

2.4. Data Pre-Processing

Finite impulse response (FIR) is a digital filter which was applied using the *lfilter* function from the *scipy* package in Python, with parameters $n = 10$, $a = 1$, and an additional parameter b calculated as shown in Equation (8). It is worth also noting that parameter b (numerator coefficient in a 1D sequence) must be an array of length n, and so should be repeated n times, whereas a (denominator coefficient in a 1D sequence) remains constant.

$$b = \begin{bmatrix} (\frac{1}{n})_1 & (\frac{1}{n})_2 & \cdots & (\frac{1}{n})_n \end{bmatrix} \tag{8}$$

The filter helped avoid noisy signals being used to train the predictive model, applied before normalization to have cleaner data on each force (F_x, F_y, F_z).

Min-max feature scaling used *MinMaxScaler* class from the *scikit-learn* package in Python to normalize data depending on both the subject and the feature (independently). Data normalization helps transform each subject's data into a shared space; moreover, the scaler transforms values into a domain $0 < x < 1$, which works best when using a DL approach to avoid exploding gradients. Feature scaling would need two equations: Firstly, a X_{std} value calculated as shown in Equation (9).

$$X_{std} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{9}$$

Secondly, max and min for each subject and feature were also calculated; thus, for each X_{std} value, we transformed it into an X_{scaled} value, which was calculated as shown in Equation (10).

$$X_{scaled} = X_{std} \cdot (max - min) + min \tag{10}$$

2.5. Deep Learning Models

2.5.1. RNN

Instead of having a feed-forward network (from input to output), an RNN also takes into account past output $h_{(t-1)}$ and thus is connected to the future input $x_{(t+1)}$. The neuron is connected to itself when not considering time t, as shown in Figure 3 on the left, which is the same structure unrolled through time on the right, where it is more apparent that output h_0 is not only the output of time 0, but is part of the operation of time 1. This process continues until frame t, and thus an ANN can process sequences via the inclusion of past output into the next operation.

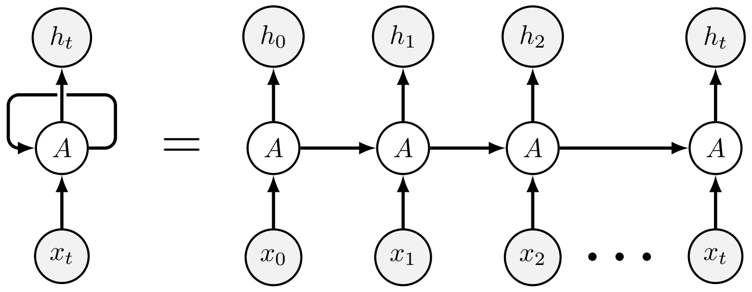

Figure 3. The basic structure of an RNN. On the left of the figure are the changes through time, and on the right of the figure is a static version. x_t represents a sample at time t, A is a neuron with a certain activation function (usually tanh) that behaves like an ANN's neurons, and h_t is not only the output at time t but also the input at time $t+1$, with its respective sample x_{t+1}.

A 1-layer RNN would be used with a fixed set of standard hyper-parameters to test the model's performance without requiring great training times: 25 epochs, 128 batch size, and 5 sequence size. Additionally, the number of neurons was adjusted via trial and error, considering the training and validation loss graph across epochs, as the chart tells if the model is under-fitting or over-fitting the training dataset. The number of neurons changed in powers of 2 according to $f(x) = 2^x$, while adjusting x and keeping it always as a power of 2, as it helps the process. It finally reached 8 neurons per layer, where the model neither over-fitted the training dataset nor under-fitted it.

The recurrent activation function used was sigmoid, as shown in Equation (11), and the final activation function was tanh. This functions are useful when explaining both LSTM cell and GRU cell in Sections 2.5.2 and 2.5.3, respectively.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{11}$$

2.5.2. LSTM Cell

Long short-term memory (LSTM) cells have a different architecture than simple cells. Their structure is in Figure 4, and the set of equations required is in Equation (12). Three gates compose the cell: forget gate, input gate, and output gate.

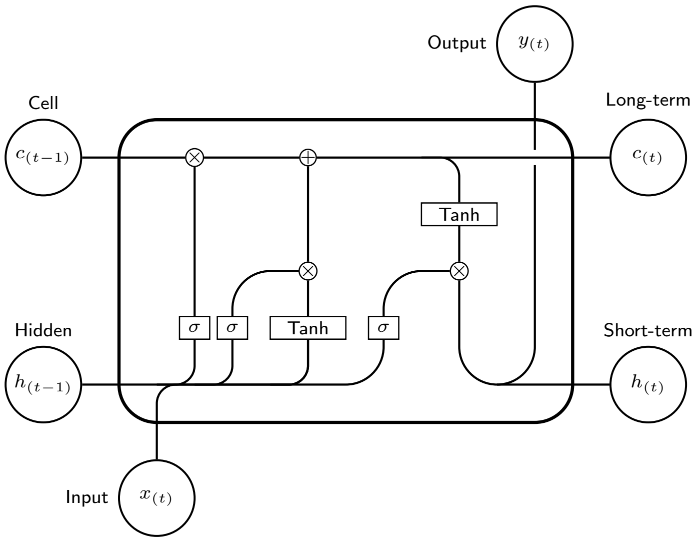

Figure 4. Default structure of an LSTM cell, based on [27]. It has a combination of $\sigma(x)$ and $\tanh(x)$ functions, to create three main gates (forget, input, output) and so generate the short-term $h_{(t)}$ and long-term $c_{(t)}$ vectors.

The processing starts on the lower part with $x_{(t)}$ and the previously generated $h_{(t-1)}$. They are used to calculate $f(t)$ (left-hand side), via the $\sigma(x)$ function in Equation (12b).

This controls the forget gate to determine which information of the long-term state should be erased, as a $\sigma(x)$ function is used, and its values range from 0 (erase all) to 1 (erase nothing). On the other hand, $x_{(t)}$ and $h_{(t-1)}$ are also used to calculate $i_{(t)}$ and $g_{(t)}$, as shown in (13a) and (13d), respectively. This consists of the input gate, where the $\sigma(x)$ function in $i_{(t)}$ controls the information of $g_{(t)}$ that should be included in the long-term state. The addition of the forget gate and input gate is calculated in $c_{(t)}$, as shown in Equation (12e), which is further used in the output gate with $o_{(t)}$, which controls how the information from the long-term state should be read and output. The element-wise multiplication of both $o_{(t)}$ and $c_{(t)}$ makes the output $y_{(t)}$, which is equal to $h_{(t)}$ (used in the next iteration as $h_{(t-1)}$), as calculated in Equation (12f).

$$i_{(t)} = \sigma(W_{xi}^T x_{(t)} + W_{hi}^T h_{(t-1)} + b_i) \tag{12a}$$
$$f_{(t)} = \sigma(W_{xf}^T x_{(t)} + W_{hf}^T h_{(t-1)} + b_f) \tag{12b}$$
$$o_{(t)} = \sigma(W_{xo}^T x_{(t)} + W_{ho}^T h_{(t-1)} + b_o) \tag{12c}$$
$$g_{(t)} = \tanh(W_{xg}^T x_{(t)} + W_{hg}^T h_{(t-1)} + b_g) \tag{12d}$$
$$c_{(t)} = f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \tag{12e}$$
$$y_{(t)} = h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)}) \tag{12f}$$

2.5.3. GRU Cell

A gated recurrent unit (GRU) cell is a simplified version of the LSTM cell; see the set of equations in Equation (13).

There are some simplifications that are done by the GRU cell with respect to the LSTM cell: The long-term state vector $c_{(t)}$ is merged into the short-term state vector $h_{(t)}$. Then, $z_{(t)}$ controls both the input gate $i_{(t)}$ and forget gate $f_{(t)}$. The output gate $o_{(t)}$ is replaced by $r_{(t)}$, which shows the information that will be shown to the main gate $g_{(t)}$.

The process starts on the left-hand with the previous state $h_{(t-1)}$ and the data $x_{(t)}$, which is used to calculate both $r_{(t)}$ and $z_{(t)}$ in Equations (13b) and (13a), respectively. The $\sigma(x)$ function is used to control the amount of past information input to the main gate $g(t)$ by $r_{(t)}$, and the proportion of information that would be forgot and learned by $z_{(t)}$ and its complement $(1 - z_{(t)})$. In this sense, $g_{(t)}$ is the output of new information learned, calculated by $r_{(t)}$ and $h_{(t-1)}$, and applying tanh in Equation (13c). The current output $h_{(t)}$ is then calculated in Equation (13d) using both the past information $h_{(t-1)}$ and the present information $g_{(t)}$, regularized by $z_{(t)}$ in order to forget and learn at a complementary rate of information, such as $(z_{(t)} + (1 - z_{(t)}) = 1)$, because $0 < \sigma(x) < 1$.

$$z_{(t)} = \sigma(W_{xz}^T x_{(t)} + W_{hz}^T h_{(t-1)} + b_z) \tag{13a}$$
$$r_{(t)} = \sigma(W_{xr}^T x_{(t)} + W_{hr}^T h_{(t-1)} + b_r) \tag{13b}$$
$$g_{(t)} = \tanh(W_{xg}^T x_{(t)} + W_{hg}^T (r_{(t)} \otimes h_{(t-1)}) + b_g) \tag{13c}$$
$$h_{(t)} = z_{(t)} \otimes h_{(t-1)} + (1 - z_{(t)}) \otimes g_{(t)} \tag{13d}$$

2.6. Model Creation

The DL models used combinations of parameters (loss function, sampling method, type of layer); they are compared in Figure 5. Firstly, raw data were extracted from the .txt files from forces and markers, using either up-sampling or down-sampling depending on the sampling rate. This was done to solve granularity and then merge both datasets into a single dataset that contained XYZ markers and forces for each velocity and subject. Then, an FIR filter and min-max scaling pre-processed data for each combination of subject and velocity. Data were further split into training, validation, and testing datasets according to the data division in Figure 2. Thereafter, data passed through a process of model creation, where the training and validation datasets trained and hence validated the model, based on $n_{epoch} = 25$ to train, compute the loss function, and update its weights. According to the loss on the final epoch, for training and validation datasets, the model could over-fit, under-fit, or ideal-fit the training dataset; only when the model had a perfect fit could it be used with the testing dataset to compute the performance. Hence, we chose the best model using these metrics.

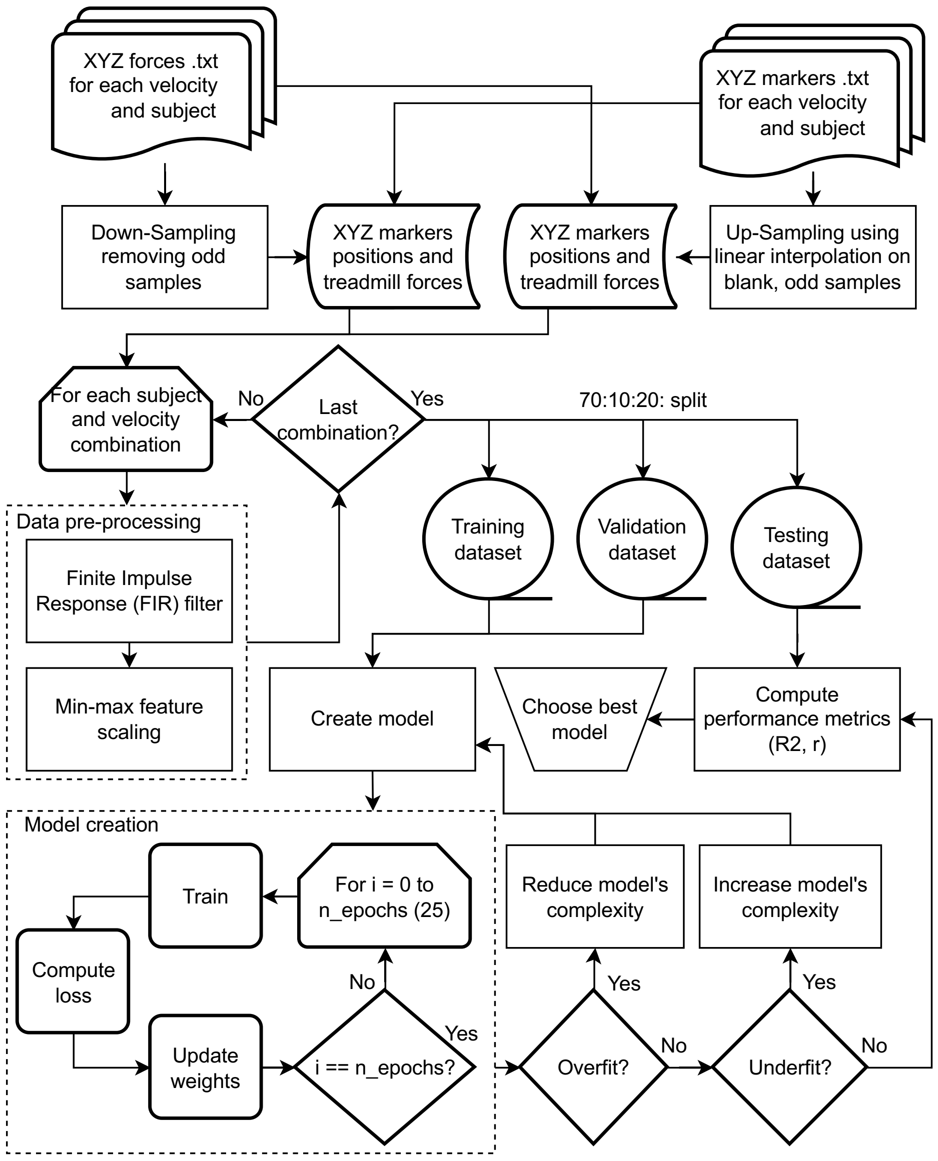
XYZ forces .txt for each velocity and subject
XYZ markers .txt for each velocity and subject
Down-Sampling removing odd samples
XYZ markers positions and treadmill forces
XYZ markers positions and treadmill forces
Up-Sampling using linear interpolation on blank, odd samples
For each subject and velocity combination
No
Last combination?
Yes
70:10:20: split
Training dataset
Validation dataset
Testing dataset
Data pre-processing
Finite Impulse Response (FIR) filter
Min-max feature scaling
Create model
Choose best model
Compute performance metrics (R2, r)
Model creation
Train
For i = 0 to n_epochs (25)
Compute loss
Update weights
i == n_epochs?
No
Yes
Reduce model's complexity
Increase model's complexity
Overfit?
Underfit?
Yes
No
Yes
No

Figure 5. Flow diagram of the DL models' creation. The process starts at the top when solving granularity and merging the raw .txt files of forces and markers. Then, data were pre-processed using FIR and min-max based on the combination of subject and velocity. Afterwards, the model was created using n_{epoch} and regularized (over-fit, under-fit), and performance metrics were used to determine the best model.

As for the evaluation of the model using training and validation datasets, the model at the ith epoch, e_i, computed training and validation loss. These computations across n_{epochs} created a line plot with two curves. These curves validated the model before testing with the testing dataset, as they first required parameters' modifications. The three types of "fits" are:

1. Under-fit: The validation curve is under the training curve, as the model is not complex enough to learn the training dataset. The parameters are not good enough to model the relationship between source and target features, so the loss function does not decrease at a stable rate. This model would not be valid, as the relationship between variables is not completely understood; the model should be more complex, adding more layers of neurons.
2. Over-fit: The training curve is under the validation curve, as the model is over-learning from the training dataset by analyzing the tiniest details that do not appear when testing the model's performance on the validation dataset. The model becomes too specific for the training dataset, making it not helpful in a real-life scenario on an unseen dataset (validation or testing dataset).
3. Ideal fit: Training and validation curves are similar; neither under-fits data nor over-fits them. The learning rate is relatively equal to the speed of generalization, so it can learn new information while still understanding the training dataset.

3. Results

Each model was structured with a similar kind of architecture, as each model has: an input layer, a 1-layer RNN type of cell (LSTM, GRU, Simple), and a dense layer for one-dimensional outputs. Each RNN model has a specific type of layer on the second layer; the loss function does not change the architecture but how weights are updated. Table 2 represents the architectures of these models, where the output shape for the first layer (input) defines the form of input data: n_{seq} represents the sequence size (5), and thus the set of samples used; n_{feat} means the number of source features used (87). While the dataset was composed of 96 features (32 data-available markers and 3 dimensions, $32 \times 3 = 96$), missing values were present in 9 features (L.MT1X, L.MT1Y, L.MT1Z, L.ASISX, L.ASISY, L.ASISZ, R.MT1X, R.MT1Y, R.MT1Z); hence, these features were dropped, and thus the number of features used was 87 ($96 - 9 = 87$). The output shapes for the second and third layers were their numbers of neurons; the only modifiable parameter was the number of neurons in the second layer, as the number of neurons in the third layer had to be three due to the three force dimensions that were being predicted (F_x, F_y, F_z).

Table 2. In each RNN architecture, for each model that has a specific type of layer, the output shape is used for each force prediction on a given dataset, where $n_{seq} = 5$ and $n_{feat} = 87$.

Type of Layer	n_{layer}	Layer	Output Shape	n_{param}	Total n_{param}
LSTM	1	Input	(n_{seq}, n_{feat})	0	3099
	2	LSTM	(8)	3072	
	3	Dense	(3)	27	
GRU	1	Input	(n_{seq}, n_{feat})	0	2355
	2	GRU	(8)	2328	
	3	Dense	(3)	27	
Simple	1	Input	(n_{seq}, n_{feat})	0	795
	2	Simple	(8)	768	
	3	Dense	(3)	27	

Even though each model employs the same architecture, each type of layer has a different number of trainable parameters; thus, the number of trainable parameters varies depending on the model's architecture. With LSTM having the highest number of n_{param} (3072) and Simple having the lowest (768), the difference comes down to the cell's architecture. As the cell's complexity increases, the number of parameters also increases. However, this does not necessarily mean that the model would take more time to train as the number of parameters increases, as the number of parameters only has to do with the modification of the cell depending on the weights; thus, it mainly represents its flexibility.

RNN training time uses epochs. A GeForce GTX 1060 6GB was used to train each DL model, although, depending on the layer type, sampling technique, and loss function selected, training time per epoch (t_{epoch}) and total elapsed training time ($t_{elapsed}$) changed. An epoch is one forward and backward propagation for the DL model; hence, numerous epochs train the DL model to understand the training data better, thereby generalizing it better for the validation and testing datasets. t_{epoch} for each model trained corresponds to the average time per epoch and determines the model that is less computationally expensive, and thus more efficient. Mean training time per epoch, shown in Equation (14), is used to calculate $t_{elapsed}$ until results, depending on the combinations of layer type, sampling method, and loss function in Table 3.

$$t_{elapsed} = t_{epoch} \cdot n_{epochs} \tag{14}$$

It is essential to see which combination has the fewest $t_{elapsed}$, as DL models could take a considerable amount of time to train. The only things that changed were the layer type, sampling method, and loss function, as other parameters and data remained constant. It is noticeable in Table 3 that layer type is a determinant factor for the t_{epoch}, as the Simple layer took 76, 66, and 73 s with the up-sampling method, which contrasts greatly with LSTM's times (31, 30, and 33 s), and GRU's times (33, 30, and 32 s), so, at first glance, it seems that the Simple layer is less efficient than LSTM and GRU layers.

Table 3. Mean training time per epoch (t_{epoch}), for each combination of layer type, sampling method, and loss function. Total training time elapsed ($t_{elapsed}$) was also calculated as shown in Equation (14), considering $n_{epochs} = 25$ and using a GeForce GTX 1060 6GB with NVIDIA Compute Unified Device Architecture (CUDA) ®Deep Neural Network library (cuDNN). Best times are in bold, which depend on average $t_{epoch} = 13$ when using (GRU, down-sampling, MSLE).

Layer	**Sampling**	**Loss Function** $L(\hat{y}, y)$	t_{epoch} **(s)**	$t_{elapsed}$ **(s)**	$t_{elapsed}$ **(min)**
LSTM	Up	MAE	31	775	12.92
		MSE	30	750	12.5
		MSLE	33	825	13.75
	Down	MAE	19	425	7.08
		MSE	19	475	7.92
		MSLE	15	375	6.25
GRU	Up	MAE	33	825	13.75
		MSE	30	750	12.5
		MSLE	32	800	13.34
	Down	MAE	16	400	6.67
		MSE	17	425	7.08
		MSLE	**13**	**325**	**5.42**
Simple	Up	MAE	76	1900	31.67
		MSE	66	1650	27.5
		MSLE	73	1825	30.42
	Down	MAE	32	800	13.34
		MSE	40	1000	16.67
		MSLE	32	800	13.34

It is worth noting that the more data, the greater the t_{epoch} would be as more data are processed; this is represented in each combination, as down-sampling times mostly are half of the up-sampling times because the up-sampling, in this case, was conserving the 9000 samples, whereas down-sampling was dropping even samples to reduce the number to 4500 samples. This phenomenon is represented in Simple's down-sampling t_{epoch} (32, 40, 32) when compared with up-sampling t_{epoch} (76, 66, 73). This can also be seen in the GRU layer down-sampling t_{epoch} (16, 17, 33) when compared with its up-sampling

(33, 30, 32). On the other hand, the loss function seems not to have a significant role in changing the t_{epoch}, although in some cases of down-sampling using MSLE, t_{epoch} seems to be less in LSTM and GRU layers ($\approx$4 s) when comparing (19 s, 19 s) with 15 s in the case of LSTM, and (16 s, 17 s) with 13 s in the case of GRU.

Moving on to prediction performance, two metrics were used: the Pearson correlation coefficient (r), shown in Equation (7), and the coefficient of determination (R^2), shown in Equation (4). The models used the following set of parameters: sampling method (up-sampling, down-sampling); type of layer (GRU, LSTM, Simple); loss function (MAE, MSE, MSLE); in addition, performance metrics obtained based on training, validation, and testing datasets—for the force on every axis (F_x, F_y, F_z), an additional average performance metric was calculated using all forces (Avg).

Based on the trained RNN's performance, the coefficient of determination (R^2) was used as a performance metric to measure a model's performance (higher is better). Results are in Table 4, divided by the type of performance metric (R^2, $|r|$) and sampling technique applied to the data (up-sampling, down-sampling). In ML, testing the dataset's performance is usually considered the most valid result because it reassembles a real-world scenario where the model would not have access to the new data beforehand. Therefore, the given table shows only testing's dataset performance.

Table 4. Coefficient of determination (R^2) and the absolute value of Pearson correlation coefficient ($|r|$) on the testing dataset, while varying RNN's loss function $L(\hat{y}, y)$, sampling technique, and type of layer, for each force dimension (F_x, F_y, F_z), and the average coefficient overall dimensions (Avg). Highest performance coefficients are in bold, which indicates the model with the best combination of parameters: up-sampling, MSE as the loss function, and LSTM as the type of layer.

Metric	**Layer**	$L(\hat{y}, y)$	**Up-Sampling**				**Down-Sampling**			
			F_x	F_y	F_z	**Avg**	F_x	F_y	F_z	**Avg**
R^2	GRU	MAE	0.76	0.96	0.1	0.6	0.64	0.96	0.09	0.56
		MSE	0.77	0.96	0.17	0.63	0.75	0.95	0.06	0.58
		MSLE	0.78	0.9	0.15	0.61	0.73	0.75	0.18	0.55
	LSTM	MAE	0.81	0.96	0.11	0.63	0.79	0.96	0.15	0.63
		MSE	**0.87**	**0.96**	**0.22**	**0.68**	0.76	0.96	0.2	0.64
		MSLE	0.82	0.94	0.2	0.65	0.71	0.91	0.18	0.6
	Simple	MAE	0.69	0.96	0.06	0.57	0.57	0.93	−0.19	0.43
		MSE	0.65	0.93	0.01	0.53	0.56	0.91	−0.04	0.48
		MSLE	0.65	0.6	0.19	0.48	0.72	0.8	0.01	0.51
$\|r\|$	GRU	MAE	0.89	0.98	0.41	0.76	0.83	0.98	0.4	0.74
		MSE	0.89	0.98	0.48	0.79	0.87	0.97	0.37	0.74
		MSLE	0.9	0.98	0.5	0.79	0.86	0.97	0.49	0.77
	LSTM	MAE	0.92	0.98	0.51	0.8	0.9	0.98	0.45	0.78
		MSE	**0.94**	**0.98**	**0.54**	**0.82**	0.89	0.98	0.51	0.79
		MSLE	0.92	0.98	0.51	0.8	0.87	0.98	0.49	0.78
	Simple	MAE	0.87	0.98	0.43	0.76	0.76	0.96	0.45	0.72
		MSE	0.82	0.97	0.42	0.73	0.77	0.97	0.32	0.69
		MSLE	0.86	0.96	0.54	0.78	0.87	0.96	0.39	0.74

This average performance for each metric is in Figure 6. A single plot could plot both metrics, as their domains are the same when considering the absolute value of r, as $-1 \leq r \leq 1$ and $R^2 \leq 1$, but $0 \leq |r| \leq 1$, thereby not showing if the Pearson correlation is positive or negative, but only measuring how strongly correlated are both sets of data. There is a vertical division on the plot: by the sampling method, then by the type of dataset, and finally by the loss function used. Moreover, the color represents the type of layer, and the type of dot the metric ($|r|$, R^2). If both metrics are at maximum, the model would achieve the best performance, as higher is better for both metrics.

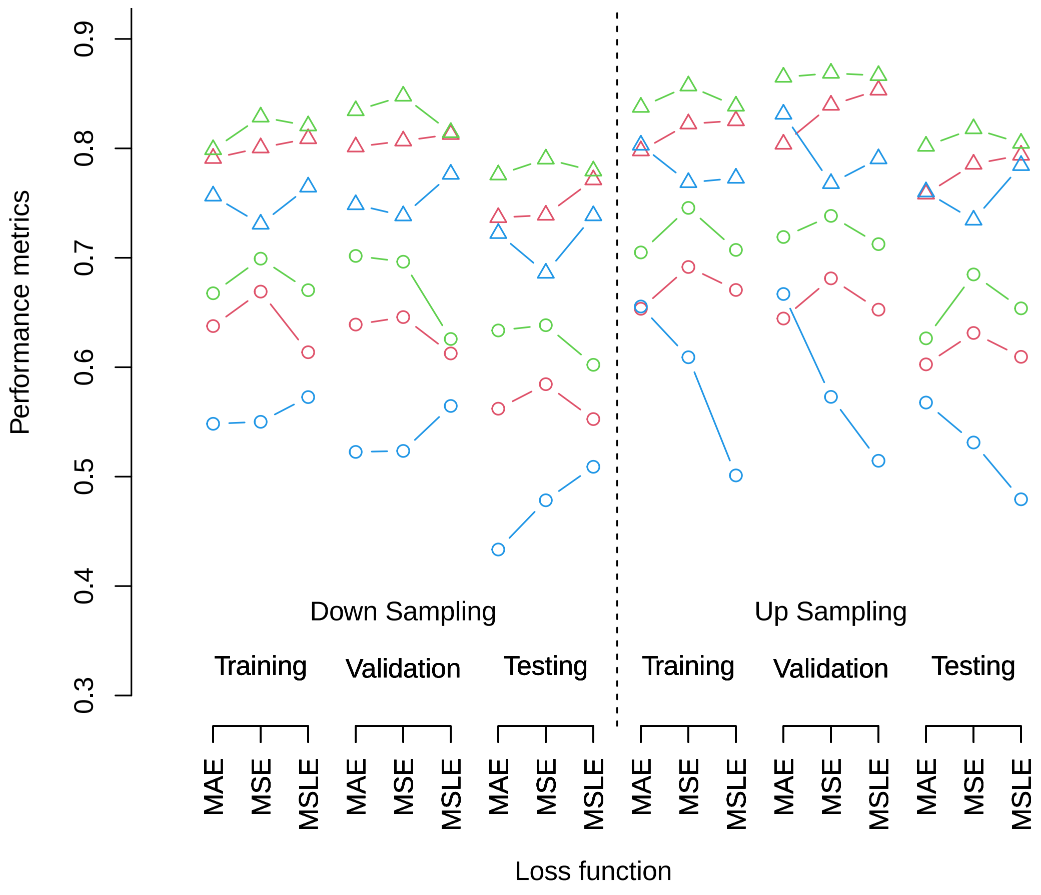

Figure 6. Models' performances considering Pearsons' $|r|$ and R^2 performance metrics. Models were created by changing a set of parameters: sampling method used for data (Up, Down), loss function (MAE, MSE, MSLE), and type of layer (GRU, LSTM, Simple). They were evaluated on several datasets (training, validation, testing) and performance was averaged for the three force dimensions (F_x, F_y, F_z) into a single metric (Avg). Each type of marker represents a performance metric (circle, triangle), and color represents the type of layer used (red, green, blue). The x-axis shows the combinations of loss functions, datasets, and sampling techniques; thus, the combination of marker and color reflects the performance of a given model in the y-axis. In this case, the combination with the highest R^2 on the testing dataset was (up-sampling, MSE, LSTM), which is the same combination that had the best $|r|$, as shown in Table 4.

As shown in Figure 6, the best $|r|$ performance was achieved using up-sampling (right side) on the validation dataset, via LSTM, which also had the best R^2's performance and MSE. However, performance should be evaluated on the data unknown to the model, the testing dataset, as that reassembles a real-life situation. When considering only the testing dataset, the LSTM layer delivered the best performance, using the up-sampling method and MSE loss function, by both $|r|$ and R^2 performance metrics. Even though the performance was better using the up-sampling method, models trained using up-sampled data took double the time used for training using down-sampled data. Considering the time factor, differences in metrics on the testing dataset are not extensive, considering the best performance delivered by LSTM and MSE.

Overall, the type of layer seems to be a determinant factor in model performance: LSTM showed the best performance in both metrics, with all combinations of loss functions and sampling methods; followed by GRU with medium performance in most cases, with a few exceptions in which the Simple layer performed better, such as on the validation dataset using MAE as the loss function; and finally, the Simple layer, which not only took longer for training, but also performed worst when compared to the other types of layers. Moving on to loss functions, MSE performed best, and both MAE and MSLE decreased the model's performance depending on the dataset, type of layer, and sampling method, as MSLE with Simple on the testing dataset performed best when using down-sampled data, whereas the same combination performed worst using up-sampled data.

Now moving on to the training-validation graphs concerning the number of epochs, only for the model with the best performance without considering training time (up-sampling, LSTM, MSE), did the training process generate four graphs (loss function value, and the other loss functions). Even though the parameters were changed only due to the loss function, keeping track of different metrics also helps to diagnose better how well the model is adapting to the training dataset, and the generalization process that it is following on the validation dataset, as it would be helpful later on the testing dataset.

For the best performing model, the MAE function graph of 4, 8, and 16 neurons is in Figure 7. Although the loss function is MSE, the MAE plot shows more clearly how the model was over-fitting when using 4 and 16 neurons, as the validation curve tends to be higher than the training curve; that it is to say that when using $n_{epoch} = \infty$, the trend of $MAE_{validation} > MAE_{training}$ would most likely still be present. On the other hand, when using eight neurons, it seems to converge on similar MAE values for both validation and training datasets $MAE_{validation} \approx MAE_{training}$; this would then represent an ideal fit rather than over-fit of the training dataset, and thus validates the use of eight neurons, based on $f(x) = 2^x$ to draw possible neuron values.

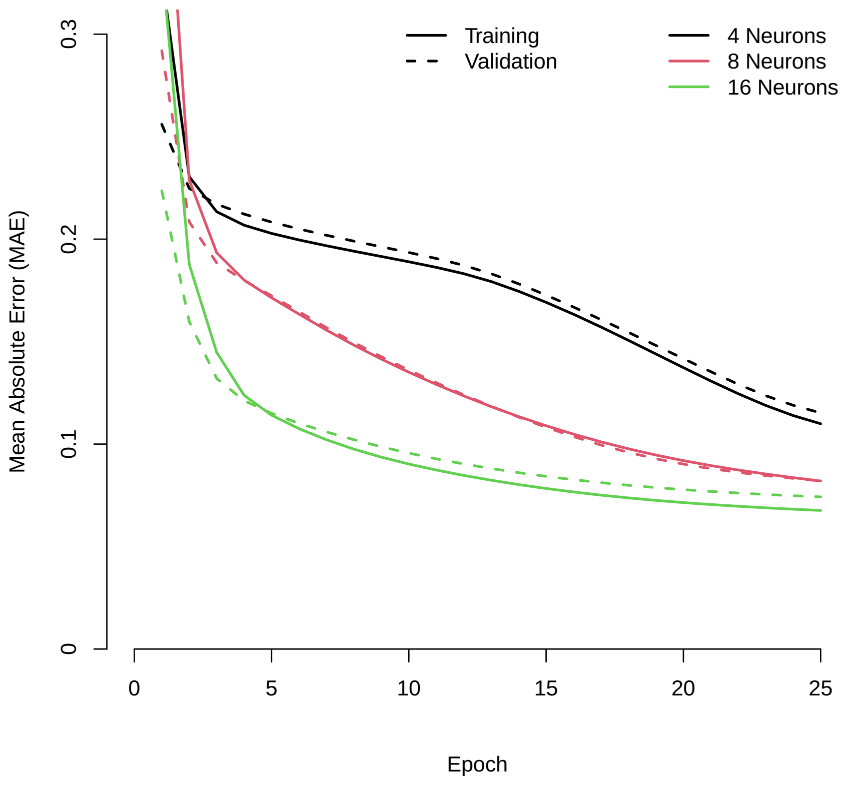

Figure 7. Training and validation MAE curves when using 4 (black), 8 (red), and 16 (green) neurons on the second layer (LSTM, GRU, Simple). These curves correspond to the best performing model, which used down-sampled data, LSTM as the type of layer, and MSE as the loss function. In the cases of using 4 and 16 neurons, the validation curve (dotted line) tends to be above the training curve (straight line), leading to over-fit, whereas the use of 8 neurons shows an ideal fit, as both training and validation curves are similar when $n_{epoch} > 5$.

Based on the previously mentioned descriptions, Figure 7 would be ideal because both training and validation loss converged to a similar loss when considering 25 epochs, even though validation loss was lower than training loss when $n_{epochs} < 5$. The under-fit behavior was corrected in the following epochs because the neural network learned from the training dataset, lowering the training loss while still lowering the validation loss at a similar rate to not over-fit the dataset. Most of the models created showed a similar trend, which validates the models, as they fitted the training dataset ideally and thus are capable of predicting.

To visually evaluate predictions done by the best performing model (up-sampling, LSTM, MSE), a line plot for each force dimension (F_x, F_y, F_z) contrasting the measured and predicted values is shown in Figure 8. It comprises three plots with scaled force values (N) on each vertical axis, and the horizontal axis represents time in seconds (s). In this case, a plot of 500 s was enough to show a pattern and how well the model was performing at predicting each force.

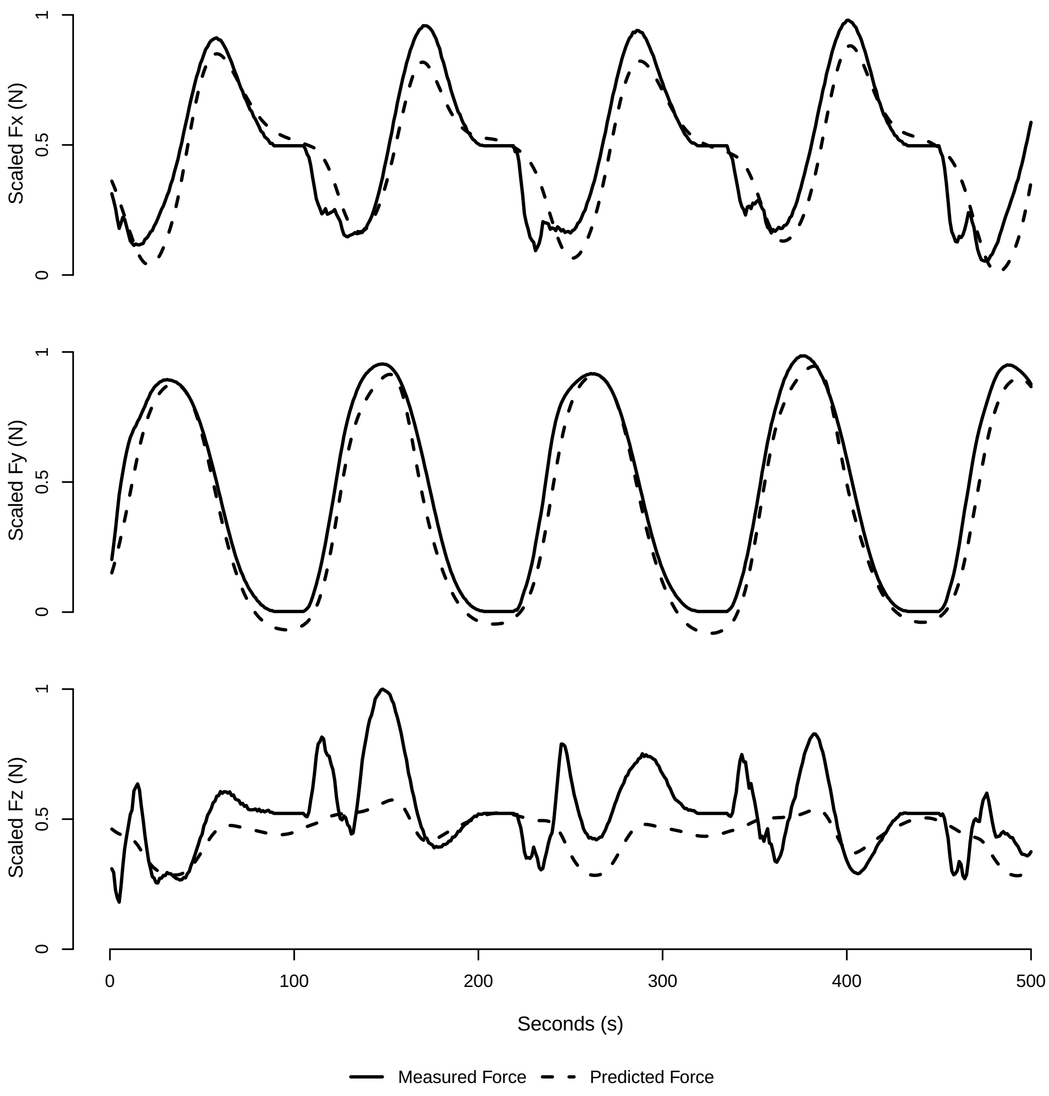

Figure 8. Scaled forces (N), reference (straight line) and predicted (dotted line) values for each dimension: F_x (**top**), F_y (**middle**), F_z (**bottom**). The best performing model (up-sampling, MSE, LSTM) according to Figure 6 and Table 4 was used to predict the values using the testing dataset. F_x and F_y seem to have clear cyclical and predictable trends, with which the model performed better when compared to F_z, which appears to have some randomness involved.

Based on predictions and measured values in Figure 8, force predictions are close to measured values on F_x and F_y. This might be because measured values behave cyclically, and thus an RNN with a specific sequence size can detect the trend and adapt their parameters more quickly than when no movement is present. Moreover, predicted values in F_y reached values < 0; this is an impossible value for scaled forces, as the min-max scaler changes value's domain to be between 0 and 1, this would only create faulty, impossible predictions, and so an improvement in forecasts would be to limit the predicted value to be between 0 and 1, improving performance metrics when comparing with the reference values. This could be achieved by using a $\sigma(x)$ function as the activation function of the final dense layer, given the proposed architecture in Table 2, as its range $0 \leq \sigma(x) \leq 1$, thereby guaranteeing that the output is between 0 and 1, which is perfect with a min-max scaler's domain.

4. Discussion

It is interesting noting the inverse relationship between n_{params} and t_{epoch}, as the number of trainable parameters within the layer only makes the model more flexible, not slower to train. In fact, when relating trainable time, Table 3, and trainable parameters, Table 2, the Simple cell had the highest t_{epoch} but the least n_{params}, while LSTM had the lowest t_{epoch} and the highest n_{params}. Another reason that could have been behind the short training times of LSTM and GRU is the graphical processing unit (GPU) optimization for those layers when using default cell parameters ($\sigma(x)$ as recurrent activation function and $\tanh(x)$ as activation function).

LSTM and GRU not only had the shortest training times in Table 3, but also performed best according to the performance metrics in Table 4. This suggests that an increase in complexity, and hence an increase in the number of parameters, would let the cell adapt better to the training data, and thus deliver better performance with less training time (when considering LSTM and Simple). The relations between the complexity of cell between the used cells, according to trainable parameters in Table 4, and the overall structure of each cell explained in Section 2.5, are the following: GRU is more complex than Simple, as it uses Equation (13) to control the flow of information with $z_{(t)}$ and $r_{(t)}$, and also requires more trainable parameters than only changing weights of A, as seen in Figure 3. LSTM is more complex than GRU, as it consists of three gates (input, forget and output) controlled by Equation (12), which separates the long-term memory $c_{(t)}$ from the short-term memory $h_{(t)}$. This would then increase the number of trainable parameters and lets it better fit the training data to not only determine which information to remember, but also how long the information should be remembered.

Moving on to force prediction, F_x and F_y had higher performance metrics than F_z in Table 4, and some even predictions had an $R^2 < 0$, which means bad predictions; this was the case when predicting F_z using down-sampled data, Simple layer type, and MAE/MSE as loss functions, with R^2 of -0.19 and -0.04, respectively. Although terrible performance was not present for F_x and F_y, the previously mentioned combination had positive R^2 values of 0.57 and 0.56 for F_x and 0.93 and 0.91 for F_y. The significant differences between these forces may be due to the cyclical pattern they display, as shown in Figure 8. F_y and F_x shared a more cyclical pattern than F_z, even though all forces were filtered using an FIR filter. Another reason may have been the orientations of the XYZ axes in the treadmill used in [8]. In this study, the x-axis is parallel to the walking direction of the participants, the y-axis follows the opposite direction of gravity force, and the z-axis is orthogonal to them. Under this axis definition, the lowest quantity of movement and force would be expected in the z-axis during a walking exercise. The average of forces performance (Avg) is greatly affected by the low performance when predicting F_z, when considering R^2 of the best performing model (LSTM, up-sampling, MSE), it had Avg $= 0.68$, although when removing F_z and only taking the average performance of F_x and F_y, $R^2 = 0.92$, which suggests that great predictions could further improve.

Many changes could be done to improve model's predictions, including: (1) Increasing the number of subjects in the dataset; hence, more data could be used to train the model, and so it could generalize more. (2) Using the subjects' meta-data dataset, as [8] included a dataset that provides for more information regarding the persons, such as weight and height, which could be used to adjust the force prediction for each subject, as the force not only depends on acceleration, but also on mass, which is not considered on the current model. This could be used when using an architecture of two neural networks, the first consisting of an RNN that makes predictions for each force (F_x, F_y, F_z), and the second an

ANN that takes as input both the predicted forces and the subject's meta-data to adjust the predictions. (3) Increasing the number of epochs, as in this work, $n_{epoch} = 25$ was used to find the best combination of parameters to avoid spending longer than 30 min on a model's training; however, increasing the number of epochs would most likely increase the model's performance as it would learn better on the training data; although, if not evaluated correctly, this could lead to over-fitting or under-fitting by the model, but the training/validation curves in Figure 7 seem to show an ideal fit trend as the number of epochs increases. (4) Trying other ML algorithms [29] or DL architectures [30]. CNN [31], which is usually used to analyze images, could also be used in time-series analysis [32].

The current work presented a quick grid search that used a one-layer RNN with $n_{epoch} = 25$ to determine which model performed best, according to a grid search that iterated over the combination of a given set of parameters (loss function, type of layer, sampling technique). The model's predictions could be improved when considering more complex DL architectures, such as the 4-layered DL models used in [32], which combined CNN, RNN, and ANN layers to achieve a 99% accuracy fall-risk prediction using a force plate. However, the model had $362,243$ parameters, as opposed to our 1-layer RNN with LSTM using 3099 parameters; it also used force, moment, and center of pressure (CoP) measurements, whereas our model only used markers positions without further processing. On the other hand, other studies used 1-layered ANNs biomechanical features, such as [23], which used GRF, vertical displacement, velocity of the center of mass, and jump power to predict joint torque. The correlation coefficient was 0.967, making it efficient but not very practical, as data have to be processed and converted to biomechanical features. In contrast, [21] used a similar approach concerning the current work on using raw XYZ positions and body weight to predict joint angles and accelerations, reaching an $R^2 > 0.95$.

The presented work can be used to predict in real-time forces generated at specific joints with given video; and can be applied to applications in ergonomy [33], posture correction, safety in the workplace [34], training, sports performance, and injury prevention [14], among others. The proposed framework can be implemented easily with open-source programming tools (e.g., Python); however, knowledge of deep learning and biomechanics is needed. Therefore, in order to provide insight into athletes' performance, it would be ideal to use the support of a biomechanical analyst to act as a mediator for a sports–biomechanics translation [35]. Furthermore, to allow a more user-friendly approach, the method could be embedded into a mobile application so that athletes and coaches can use it in a more direct manner, similarly concept of commercial applications such as Kinovea [36] and Dartfish [37].

5. Conclusions

In this work, the use of RNN structures in the context of sports biomechanics was successful, as the comparisons between the original and the predicted forces were satisfactory, as observed in Figure 8. While investigating the RNN training performance using different architectures, the t_{epoch} and $t_{elapsed}$ parameters allowed us to confirm that simple layering (in this application) was less efficient than GRU and LSTM. In our case, the more complex architectures were the ones that generated the best results.

Rather than proposing a unique solution, in this work, many parameters were explored to obtain the best model. For example, performance was measured using metrics such as the Pearson correlation coefficient (r) and coefficient of determination (R^2). Additionally, combinations of modeling parameters, such as sampling methods (up-sampling, down-sampling), type of layer (GRU, LSTM, Simple), and loss function (MAE, MSE, MSLE), were added to the evaluation of the models to find the best possible model.

Considering all the work presented in this manuscript, it is clear that the proposed framework was successful at correctly predicting the forces generated on a force platform, using positions from markers. This is a significant result in sports biomechanics because it means that using similar methods, applications can be built to predict real-time forces generated in athletes' joints, and coaches could use such predictions for injury prevention, technique assessment, and personalized coaching, among others.