Final Master Thesis

MASTER'S DEGREE IN AUTOMATIC CONTROL AND ROBOTICS

Deep Learning: Creating bridges between DMPs in Auto Encoders and Recurrent Neural Networks

MEMORY

Author: Marc Arbones

Advisor: Dr. Cecilio Angulo

Date: September, 2017

Escola Tècnica Superior

d'Enginyeria Industrial de Barcelona

Abstract

The complexity in modeling human movement increases as the dimensionality of these movement grows. Since searching more precision and flexibility involves more variables in the model. Dynamic Movement Primitives (DMP) have shown the ability to generate joint movements with high complexity. However, the problem remains in the interaction between several joints since DMP alone is not able to deal with it. To solve this problem a new model called ***autoencoded dynamic movement primitive (AE- DMP)*** is introduced in the work *"Efficient movement representation by embedding Dynamic Movement Primitives in Deep Autoencoders*"[2]. The proposed approach uses autoencoder in order to find a representation of the movement in a latent space. Consequently, the DMP model is able to reconstruct the complete movement. In this Master Thesis we will study the implementation of this model and study its performance. All the features stated in the original paper are checked, as multiple movements, sparsity and reconstruction of missing or corrupted data.

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS AND SYMBOLS

During the writing of this thesis I tried to be as much coherent as possible with the notation and symbol usage.

LIST OF ABBREVIATIONS

AE	Autoencoder
ANN	Artificial Neural Network
DMP	Dynamic Movement Primitive
DNN	Deep Neural Network
NN	Neural Network
MSE	Mean Square Error
PCA	Principal Component Analysis
SD	Standard Deviation

LIST OF SYMBOLS

d	Number of joints in the joint space
N	Number of frames in the demonstration
x	Input space, dimensions $[d \times 1]$
$\tilde{x}$	Corrupted input space
d'	Latent space dimension
y	Latent space of the AE
yd	Latent space obtained from the demonstration
$\tilde{y}$	Latent space generated with the DMP
$h_\theta(x)$	AE encoder part function
$g_{\theta'}(y)$	AE decoder part function

θ, θ'	AE parameters
W, W'	Weights in the AE
b, b'	Bias in the AE
w	Weights of the basis functions in the DMP
W_d	Transformation matrix to obtain frame by frame first derivates
W_{dd}	Transformation matrix to obtain frame by frame second derivatives
$\dot{y}$	Latent space first derivative
$\ddot{y}$	Latent space second derivative
g	Goal of the DMP
α_y, β_y	DMP parameters
α_s	Canonical system parameter
τ	Time constant in DMP
s	DMP canonical time
$f(s)$	Forcing term function in a DMP
f_t	Target forcing term in a DMP obtained from demostration
$\Psi(s)$	Gaussian basis function
z	Output space of the AE
$L(a, b)$	Loss function
λ, μ	Regularization parameters
$exp(a)$	Refers to expression e^a
A_1, B_1, C_1, D_1	Parameters in the generation of $\tilde{y}$
A, B, C, D	Matrix implementation of A_1, B_1, C_1, D_1

CHAPTER 1

OBJECTIVES

The general motivation for this master thesis is to explore links between deep autoencoders and forms of information representation as dynamical systems, more precisely dynamic movement primitives. The project is starting from the work in [2], where movement is represented by embedding dynamic movement primitives into deep autoencoders. This approach is a dynamical solution to the increasing problem of human movement replication.
This starting motivation leads me to set up the objectives of my master thesis:

- implement and re-visit the approach introduced in [2];
- analyze how to improve its performance; and
- consider future implementations in real applications, as exoskeletons and general mobile robots.

In parallel, practical skills are improved during the master thesis development:

- study and analysis of autoencoders;
- study and analysis of dynamical movement primitives;
- general literature review on these subjects;
- study and analysis of machine learning methods;
- TensorFlow and Python skills are needed; and
- human movements are studied for extrapolation to robotic applications.

Based on [2], the first steps in the master thesis development were the implementation and reproduction of their results. During this time, most of the practical skills were acquired. Next, several structures in the literature were analyzed and considered. Associated algorithms and procedures to these structures were mathematically introduced and then programmed in Python, using the machine learning library TensorFlow, in order to test their performance in a simulation.

For all the proposed structures, the different characteristics attributed to the method were discussed: joints reconstruction, goal changing, movement interpolation. All of them were observed from different views, looking for its strengths and weaknesses.

Experimentation was developed having as the first aim the results precision and fidelity to the original data. After checking the right performance, a more generalized solution is searched in the results with the denoised autoencoder and the movement interpolation. Hence, we are paving the way for future projects that have as objective a real implementation.

CHAPTER 2

CONCEPTS

In this section, a brief explanation of the main concepts that will be used in the development of this master thesis is introduced. The basics about Neural Networks and Dynamical Movement Primitives **verb**.

2.1 DEEP NEURAL NETWORKS

Artificial Neural Networks (ANNs) have been a hot topic since their first formulation during the 40s. After their introduction a lot of uses have been found in machine learning and artificial intelligence. But the progress of ANN was slowly due to the small computational power available in that time. Another important drawback was the inaccessibility to large sets of data to train and obtain a real and useful implementation.

During the last years we started to gather data about all different actions, behaviours, weather patterns, medical records, communication systems, data about our travels, and information about what we do at work, in order to create a huge library that machines can use to learn. Currently during these years the computational power increased exponentially and allowed us to start implementing neural networks in a lot of different applications without overwhelming the devices. Due to this leap in computing power, a huge variety of different types of neural structures appeared, which were not possible to compute before. One of these types are the Deep Neural Networks (DNNs). DNNs are a specific types in which instead of adding nodes in one or two layers, more layers are added to the structure, as shown in the Figure 2.1. Adding more layers forces the NN to learn the important features of the training data, leading to better results in some topics, allowing the construction of more combinations of the data.

DNNs have a drawback: when humans try to understand the meaning of the computation in the layers, it becomes a difficult and complex task and sometimes unintelligible. For this reason, some people are hesitant of accepting the success of DNNs. The inability of tweaking this parameters with some knowledge of their implication in the process creates that dissatisfaction. In image processing, sometimes patterns appear, programmers are able to observe nodes that learn to search vertical lines, other that search for specific shapes, etc. But a full understanding of them is beyond human comprehension.

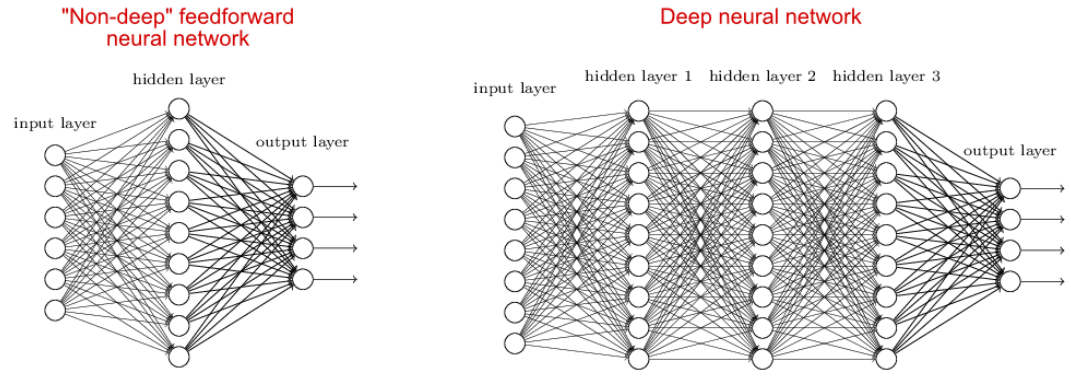


Figure 2.1: Difference between Neural Network architectures: Deep NN (right) and standard feedforward NN(left).

2.2 AUTO-ENCODERS

Auto-Encoders (AEs) are a special construction of DNN and its objective is the reduction and uncorrelation of a set of observations. AEs are the expansion of Principal Component Analysis (PCA), where PCA only composes linear combinations. The AE adds the possibility of adding non-linear combinations using the activation functions in the NN. This structure creates a latent space, with less or equal dimensions to the original space and where data can be processed more easily with the chosen machine learning method.

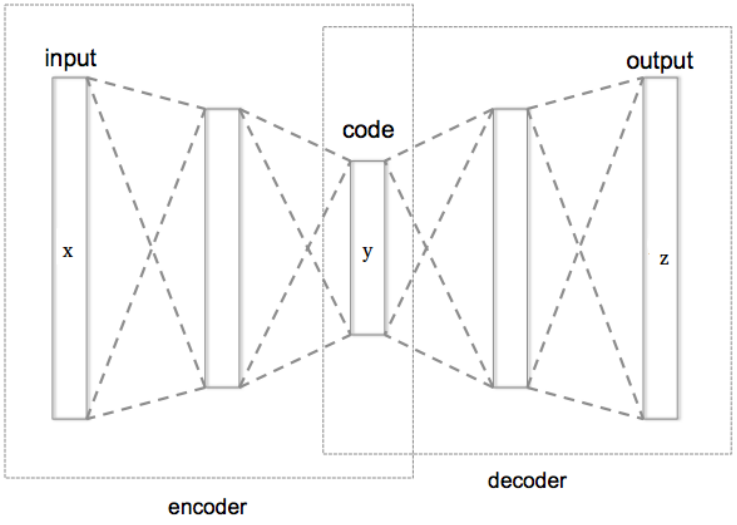


Figure 2.2: Basic Auto-Encoder structure and the different parts. Source: neuralnetworksanddeeplearning.com

The AE consists of an encoder part and a decoder part as shown in Figure 2.2. The encoder part codifies the data into a lower space, obtaining a called latent space.

In our setup for this project, the encoder takes the d joints angles $\mathbf{x} \in [-1,1]^d$ of the moving body as inputs to the network. The encoder maps these inputs to a reduced latent space, passing through the different hidden layers. Every hidden layer computes the mapping $\mathbf{y} = h_\theta(\mathbf{x}) = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$, where $\theta = \{\mathbf{W}, \mathbf{b}\}$ are the NN parameters.

Matrix **W** corresponds to the connection weights of the different layers, and **b** is the bias vector, these are the basic parameters existent in almost all the NN structures. The activation function proposed in the paper is a hyperbolic tangent, but in this thesis other options will be explored.

This mapping leads to $\mathbf{y} \in [-1,1]^{d'}$, a latent representation obtained made of d' dimensions.

From this latent space the feature representation is reconstructed back to the joint space $\mathbf{z} \in [-1,1]^{d}$ through the decoder network. The decoder has the same mapping but in the reverse way, $\mathbf{z} = g_{\theta'}(\mathbf{y}) = \tanh(\mathbf{W}'\mathbf{x} + \mathbf{b}')$ where $\theta' = \{\mathbf{W}', \mathbf{b}'\}$, $\mathbf{b}'$ is the bias of the decoding layers, and the weight matrix is $\mathbf{W}'$. It exists a tied weights relation, which is $\mathbf{W}' = \mathbf{W}^T$, the implementation of this change will be discussed during this thesis. To seek the parameters θ and θ', the optimization problem becomes:

$$\theta^\star, \theta'^\star = \arg\min_{\theta,\theta'} \frac{1}{n} \sum_{i=1}^{n} L\left(\mathbf{x}^{(i)}, \mathbf{z}^{(i)}\right) = \arg\min_{\theta,\theta'} \frac{1}{n} \sum_{i=1}^{n} L\left(\mathbf{x}^{(i)}, g_{\theta'}\left(h_\theta\left(\mathbf{x}^{(i)}\right)\right)\right), \tag{2.1}$$

where n is the number of instances in a training set. L is a loss function, the squared error $L(\mathbf{x},\mathbf{z}) = \| \mathbf{x} - \mathbf{z} \|^2$; also other types of error approximation will be discussed.

2.3 Denoising

Denoising is a process in which the inputs are corrupted during the training steps. After training the model with this process, the model is able to robustly reconstruct the set of observations from a corrupted one [7]. The denoising process consist in, starting from an initial instance **x**, generating a corrupted vector $\tilde{\mathbf{x}}$ as the new input. For every input frame, each input joint is picked with a probability p, and its value is set to 0, while the rest are unchanged.

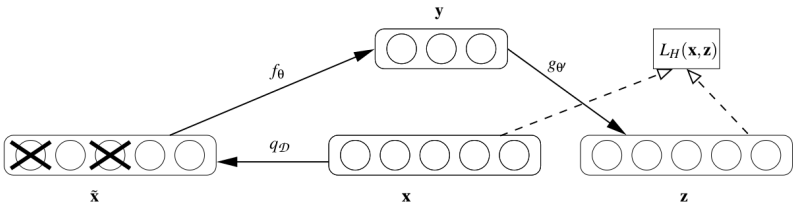


Figure 2.3: Schematized process of a Denoised Auto-Encoder. Source: eric-yuan.me

The optimization problem is modified to adapt to the new changes:

$$\theta^\star, \theta'^\star = arg\min_{\theta,\theta'} \frac{1}{n} \sum_{i=1}^{n} L\left(\mathbf{x}^{(i)}, g_{\theta'}\left(h_\theta\left(\tilde{\mathbf{x}}^{(i)}\right)\right)\right). \tag{2.2}$$

The layers are connected, except the removed neurons, but as the inputs are 0 there is no need to make any modification to the NN. During the testing process, all the joints are presented without corruption and the weights of the first input layer to the first encoder layer are scaled by multiplying $1-p$. The weights and bias in the other layers remain unchanged.

2.4 Dynamical Movement Primitives

Finding an appropriate dynamical system model for a given behavioral phenomenon is a nontrivial task due to the parameter sensitivity of nonlinear deferential equations and their lack of analytic predictability. The use of Dynamical Movement Primitives (DMPs) is presented as a solution to this problem, generating a multidimensional system to capture an observed behavior in an attractor landscape. All the work in this topic is obtained from [1].

The simplest system is a point attractor using a second-order system:

$$\tau \ddot{y} = \alpha_y(\beta_y(g-y) - \dot{y}) + f, \tag{2.3}$$

where τ is a time constant and α_y and β_y are positive constants. in order to obtain a critically damped system is recommended to use $\beta_y = \alpha_y/4$ in order to monotonically converge towards the goal g. The approach in this thesis is to use the latent space in the AE as the dynamical system y in the DMP.

The forcing term f in (2.3) is chosen as a linear combination of basis functions Ψ_i:

$$f(t) = \frac{\sum_{i=1}^{N} \Psi_i(t) w_i}{\sum_{i=1}^{N} \Psi_i(t)}. \tag{2.4}$$

To decouple this from the dynamics of the data, a replacement for the time is introduced by means of the following first-order linear dynamics in s:

$$\tau \dot{s} = -\alpha_s s, \tag{2.5}$$

where α_s is a constant. Starting from some arbitrarily chosen initial state s_0, such as $s_0 = 1$, the state s converges monotonically to zero. This equation is named the *canonical system* because it models the generic behavior of the model equations:

$$f(s) = \frac{\sum_{i=1}^{N} \Psi_i(s) w_i}{\sum_{i=1}^{N} \Psi_i(s)}. \tag{2.6}$$

Basis function $\Psi_i(s)$ can be formulated as:

$$\Psi_i(s) = \exp\left(-h_i(s-c_i)^2\right), \tag{2.7}$$

where h_i and c_i are constants that determine, respectively, the width and center of the basis functions. For more information about the distribution of the basis functions, look for in Section 3.2.2 in this same thesis.

The DMP is trained using the objective force, also known as target force, is the force that the model needs to achieve the demonstration movement. The target force is obtained from the demonstration values, **yd** and its respective computed first and second derivative, all of these obtained from the latent space. This target is calculated in the following way:

$$\mathbf{f_{target}} = \tau^2 \ddot{\mathbf{yd}} - \alpha_y(\beta_y(g - \mathbf{yd}) - \tau \dot{\mathbf{yd}}). \tag{2.8}$$

As it has been previously mention the weighted summation of basis functions have to achieve this $\mathbf{f_{target}}$. Hence, the performance of the model is measured taking into account the difference between this two values. The

corresponding solution is derived as:

$$w^\star = arg\min_w \sum_{i=1}^{m} L\left(\mathbf{f_t}^{(i)}, \mathbf{f}^{(i)}\right), \tag{2.9}$$

with $\{(\mathbf{f_t}^{(1)}, \mathbf{f}^{(1)}), ..., (\mathbf{f_t}^{(m)}, \mathbf{f}^{(m)})\}$ of length m, number of time frames.

2.5 Auto-Encoder Dynamical Movement Primitive

The objective of this model is to embed the DMP into a denoising AE. The set of parameters w can be trained using a weighted linear regression or any other machine learning methods. However, in order to fit DMP into auto-encoder, the learning is made using the optimization problem:

$$\theta^\star, \theta'^\star, w^\star = \arg\min_{w,\theta,\theta'} \frac{1}{n} \sum_{i=1}^{n} L\left(\mathbf{x}^{(i)}, \tilde{\mathbf{z}}^{(i)}\right) + \lambda \cdot L\left(\mathbf{f_t}^{(i)}, \mathbf{f}^{(i)}\right), \tag{2.10}$$

where λ is a regularization parameter. And the model is trained with backpropagation like a normal NN. The second term $L\left(\mathbf{f_t^{(i)}}, \mathbf{f}^{(i)}\right)$ allows the DMP output to follow the demonstration in the latent space. During this training the weights must be updated for a whole demonstration.

2.6 Spare AE-DMP

In order to codify different movements a new term is added in the optimization formula:

$$L(\mathbf{y}) = \mu \sum_{i=1}^{m} \left\| \mathbf{y}^{(i)} \right\|_1, \tag{2.11}$$

where the parameter μ is used to regularize the term. This sparsity is able to deactivate various hidden neurons, leading to a possibility of codifying new movements in this "unsued" neurons.
The optimization problem is modified to:

$$\theta^\star, \theta'^\star, w^\star = \arg\min_{w,\theta,\theta'} \frac{1}{n} \sum_{i=1}^{n} L\left(\mathbf{x}^{(i)}, \tilde{\mathbf{z}}^{(i)}\right) + \lambda \cdot L\left(\mathbf{f_t}^{(i)}, \mathbf{f}^{(i)}\right) + \mu \sum_{i=1}^{m} \left\| \mathbf{y}^{(i)} \right\|_1. \tag{2.12}$$

When this codification of different movements is accomplished the model is able to interpolate between different movements or even create new movements like a slow jogging as a combination of running and walking.

CHAPTER 3

METHODOLOGY

In this chapter some topics will be discussed and compared in order to determine the most efficient in terms of performance through the available options. Firstly the structure of the NN and the complications in the construction procedures. The training methods and regularization. And finally the mathematical implementation of the model in [2].

3.1 AUTO-ENCODERS

3.1.1 NUMBER OF LAYERS AND RATE OF NEURONS PER LAYER

It is well know that there is no standard method to find the optimal values for the number of layers and the rate of neurons per layer. However, exist some rules applied in general at NN: the relation between the input and the output layers, the complexity of the problem and the size of the training data. If the problem is a simply linear classification theoretically there is no need in using hidden neurons. But if a more complex nonlinear problem must be solved; additional hidden layer must be added.
Defining the number of nodes is also a complex task; small number of nodes leads to error in the approximation but having a lot of them will create a overfitting in the result, Figure 3.1.

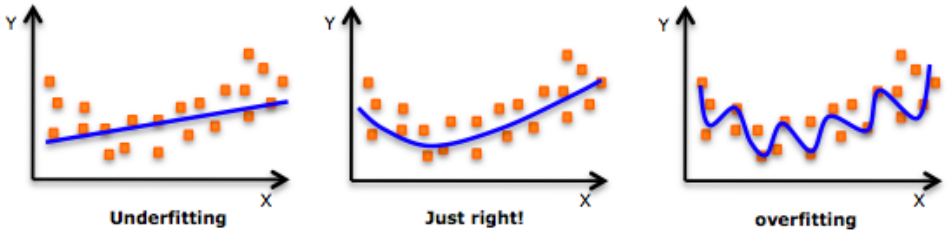


Figure 3.1: Example of the possible problems when defining the number of nodes. Source: pingax.com

However for our case we are using an autoencoder, a special case of NN where our objective is to “compress” the information into features and then reconstruct them. This compressed information is mean to be more descriptive of the whole pattern in the original data.
First aboard the problem with the number of layers. Our objective, as previous said, is to “compress” the joints

space into a smaller features space, through a nonlinear transformation. It is assumed the need of at least one hidden layer in the encoder and one in the decoder. The introduction of at least one nonlinearity to the computation is what differentiates this method from the PCA. From there, continue expanding until the one that gives better performance is found.

But the problem lies in when to stop adding layers. A common problem in DNN is that the gradient tends to get smaller as we move backward through the hidden layers. The phenomenon is known as the *vanishing gradient problem*. That means that neurons in the first layers learn much slower than neurons in last layers. As this error is decreased exponentially during the propagation. With this problem also exist the counterpart, the *exploding gradient problem*, where the gradients change drastically due to singularities in the gradient surface. Exist different methods that solves this problems but during this thesis we are studying the implementation of a AE-DMP not the solution of this problems.

The results obtained from an experiment with different structures comparing their performance are presented in Figure 3.2. The same experiment is repeated twice per structure for robustness in the results.

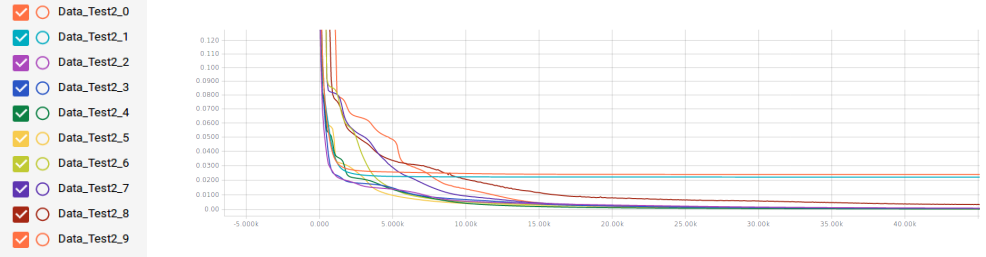


Figure 3.2: Different structures of DNN perfmance comapared.

For the experiments, the same parameters have been used for all of the structures: activation function, training time, batch size, inputs set at 50 and the latent space a dimension of 5. Results in Data_Test2_0 and Data_Test2_1 are obtained from a DNN structure with basic connection with only one layer, 50 to 5. As observed, the Auto-Encoder is not able to reconstruct the data as the other structures did, these converge to a larger error. The results in Data_Test2_2 and Data_Test2_3 are obtained from a DNN structure in the form [50,36,5]. They have the faster training of all the test. The error in these examples drop significantly at the start but they do not reach the same minimum than the next structure. Data_Test2_4 and Data_Test2_5 are obtained from a DNN in the form [50,36,20,5], which is the structure proposed in the paper used as starting point of this Master Thesis [2]. It is the one that best works in terms of final accuracy. Looking at the results plotted in Data_Test2_8 and Data_Test2_9, obtained from a DNN scturctured as [50,42,36,20,12,5], overfitting appears, the phenomenon previously mentioned. Learning period using that structure is slowed and even it converges to larger error than the previous one.

This experiment manifest all the problems early mentioned, a simple structure will not be able to reconstruct the data. And a too complex structure will impact in the learning speed. The point lies into find the adequate structure.

In the experimenting part of this project two structures that gave the better results in the preliminary experiments are used. The 3 layers[36,20,5] and 4 layers [60,42,25,5].

3.1.2 ACTIVATION FUNCTION

Each neuron in an NN represents a mathematical operation. First of all, all the inputs are added or multiplied, the sum is the most common operation but the multiplication is also used in probabilistic networks. Next, the result is introduced into a activation function and the output is passed to the next neurons. The activation function defines the output of the neuron. There is a wide range of functions to be used. Usually a non-linear function is used to add this particularity to the network, but in some cases special functions may also be used to obtain some special characteristics into our network.

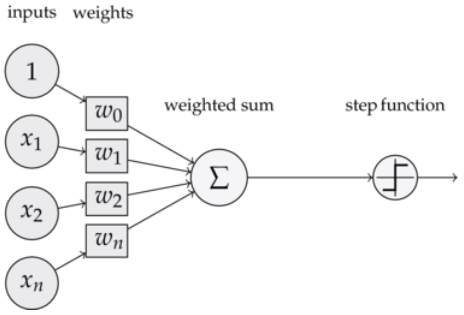


Figure 3.3: Example of a neuron. Source:blog.dbrgn.ch

In the statement of the problem it has been declared that the inputs and the outputs are in the space of $\mathbf{x}, \mathbf{z} \in [-1,1]^d$. For this reason only functions that are able to produce outputs between these values are chosen in order to do not lose information.

With the previous statement there are 3 common activation functions that fulfill the requirement,

- Tanh
- Softsign
- Sinusioudal

Tanh:

$$f(x) = \frac{2}{1+\exp^{-2x}} - 1 \tag{3.1}$$

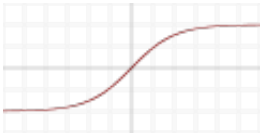


Figure 3.4: Tanh activation function

This is the proposed function in the source paper [2]. Also is a common choice in Deep Neural Networks. Produce a sigmoid-like response but in the range $[-1,1]$.

Softsign:

$$f(x) = \frac{x}{1+|x|} \tag{3.2}$$

Gives almost the same distribution as the tanh function but is less computationally stressful.

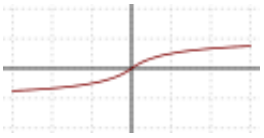


Figure 3.5: Softsign activation function

Sinusoidal:

$$f(x) = sin(x) \tag{3.3}$$

This option is not common, due that the sinus is not a increasing function and don't fulfills the universal approximation theorem. But in certain systems can be useful using them, with precaution [6].

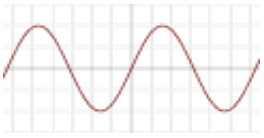


Figure 3.6: Sin activation function

For the purpose of following the original paper [2] the tanh function is chosen. But all these functions could work well for our setup. That the softsign is only smooth C^1 is not a problem because the optimization usually only use the first derivative. Other options like step or binary are not good because the continuous nature of our data.

3.1.3 Denoising process

If our interest is the extraction of the main features of a large set of data, one option is to force it. This is done trying to reconstruct the data from a corrupted version of it. With this procedure we are forcing the autoencoder to obtain the major information possible of the underlying structure.
Three types of noise are commonly used [8]:

- Additive isotropic Gaussian noise: $\tilde{X}^n \sim N(X^x, \sigma^2 I)$
- Masking noise: Some elements of the input vector are randomly changed for 0
- Salt-pepper noise: Some elements of the input vector are randomly changed to the maximum or the minimum values allowed.

In the project the Masking noise is the one implemented.

3.1.4 Tied weights

In Auto-Encoders, there are 2 parts clearly defined, the encoder and the decoder. Both of them have a set of weights which are the connections of the layers. If the AE have a large number of neurons the space of search growth having more dimensions to find the gradient, slowing down the learning process. One proposed solution commonly used is to tie mathematically the weights from both sides. This is made as $\mathbf{W}_2 = \mathbf{W}_1^T$. If the system is working with linear functions the solution of the latent space converge to a PCA solution. The learning time is reduced because now the system have the half of variables. Some discussion exist about the optimality and the

huge restriction that adds this method to the system, a good discussion about this topic and AE general related can be found here [5], but this is not the topic of this thesis. Our model is not huge and the training time is not a critical restriction, so the option of keeping the weights independent is chosen.

3.1.5 Loss function

In order to do the training the optimizer have to follow the gradients. These gradients come from the minimization of the error. In this project is used the sum of a weighted *loss functions*. To calculate the distance of the output from the input, and the same for the DMP part. This function is chosen to be an l_2-norm.

$$L(\mathbf{x},\mathbf{z}) = \| \mathbf{x}-\mathbf{z} \|^2 \tag{3.4}$$

Other options exist in the l-norm family, from l_0 to l_∞, all of them have some special characteristics [4]. But for this project only the l_2-norm is tested following the steps in [2].

3.1.6 Training method

TensorFlow software provides some classes to compute the gradients from a loss function and apply the calculated gradients to the variables. Also exist the possibility of creating your own optimizer.
The next ones are a few of the options given, the ones considered for this project:

- Gradient Descent. Is the simplest implementation of a optimizer, is the search of the minimum point of a multivariable function following this equation:

$$a^{n+1} = a^n - \gamma \nabla F(a^n), \tag{3.5}$$

  where γ can be calculated by different methods.

- Adagrad. Implements a method that dynamically incorporates knowledge of the geometry of the data observed in earlier iterations to perform a more informative gradient learning[1].

- Adadelta (Adaptive learning rate method). This method, presented in Algorithm 1[2], dynamically adapts over time using only first order information and has minimal computational overhead beyond vanilla stochastic gradient descent.

Algorithm 1 Computing ADADELTA update at time t

Require: Decay rate ρ, Constat ϵ
Require: Initial parameter x_1
1: Initialize accumulation variables $E[g^2]_0 = 0, E[\Delta x^2]_0 = 0$
2: **for** $t = 1$ **do** %% Loop over # of updates
3: Accumulate Gradient: $E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2$
4: Compute Update: $\Delta x_t = -\frac{RMS[\Delta x]_{t-1}}{RMS[g]_t} g_t$
5: Accumulate Updates: $E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1-\rho)\Delta x_t^2$
6: Apply Update: $x_{t+1} = x_t + \Delta x_t$

[1] For a wider explanation check this publication [3].
[2] A deeper formulation of the algorithm can be found in [9].

In the implementation of the model a Adadelta optimization is used. Gives a better performance than the basic gradient descent without over-complicating the computation. Due that the objective of the project is not to test different leaning methods, or their success, only this one will be implemented.

3.2 DMP

3.2.1 Periodic/Limit cycle canonical system

The canonical time can be computed in different ways. The most common is in the form of a first-order dynamical system, as previous introduced:

$$\tau \dot{s} = -\alpha_s s, \tag{3.6}$$

but in some specific cases a cyclic canonical system can be used. This modified system provides a periodicity to the basis functions:

$$\tau \dot{\phi} = 1, \tag{3.7}$$

where $\phi \in [0, 2\pi]$ is the phase angle of the oscillator in polar coordinates and the amplitude is r. Then the basis should be substituted by von Mises basis functions:

$$f(\phi, r) = \frac{\sum_{i=1}^{N} \Psi_i(t) w_i}{\sum_{i=1}^{N} \Psi_i(t)} r, \tag{3.8}$$

$$\Psi_i = \exp\left(h_i \left(cos(\phi - c_i) - 1\right)\right). \tag{3.9}$$

The amplitude and the period of the oscillations can be modulated by modifying the value of the variables r and τ. Commonly τ is chosen to be closest to the signal period.
The human walking is a cyclic movement, but each part of the body works at different periods, furthermore in the latent space the period can be whatever the NN adapt the combination of all the joints. For these reasons adapting the structure to a cyclic canonical will require a lot of preprocessing working, against the objective of simplifying the implementation and reproduction of the human movement.

3.2.2 Basis function distribution

A common problem using DMPs rise due that the system time is passed by a exponential system. If the centers of the basis functions are grouped as **mean** = $[0, .., t\_final]$, uniformly distributed during all the demonstration time. The basis actually are activated as represented in the Figure 3.7.
The basis function are packed during the firsts moments of the execution. In order to spread out the functions during all the movement, the center of our basis functions must be shifted. If we define the new centers as:

$$\mathbf{c} = S_0 \cdot \exp\left(-\frac{\alpha_s \cdot \mathbf{mean}}{\tau}\right). \tag{3.10}$$

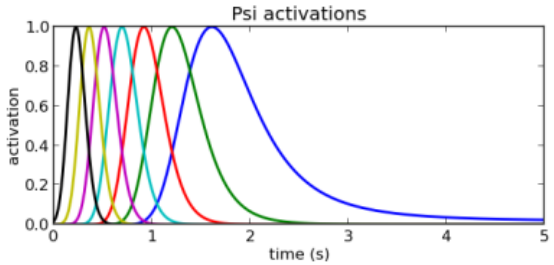


Figure 3.7: Basis function distribution using the uniformly distributed means in the time axis. Source: study-wolf.wordpress.com

Other modification in the variance can be added to obtain better results:

$$h_i = \frac{\#BF}{c_i}. \tag{3.11}$$

With this new centers and variance introduced in the same equation previous presented [2.7]. Now we obtain an evenly distribution in the basis functions during the new canonical time.

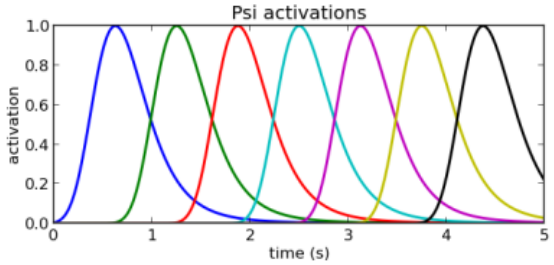


Figure 3.8: Basis function distribution spaced. Source: studywolf.wordpress.com

3.3 AE-DMP

After all the previous knowledge is time to introduce the main topic of this thesis, the AE-DMP which is the mix of both. Uses the AE to find a latent space where the DMP can be trained easily. In our case we have worked in the human body movement, simplified to 50 joints, low variance joints are discarded. Traditionally in order to apply the DMP to a whole structure with multiple joints, a DMP function per joint or dimension of the state space is required to reproduce the movement. But if this space is reduced to a latent space with 5 dimensions, only 5 DMP are needed in order to reproduce the movement. Also the embedding the DMP into a Denoising AE allows the model to reconstruct corrupted data of missing joints or even complete sections.

3.3.1 Structures

In order to validate the performance and the execution of this model, 3 different structures are proposed[3]:

[3] I gave the freedom to myself to name the structures with this names in order to facilitate the differentiation, there is no official nomenclature or definition for these structures.

1. Basic DMP: Indivual DMP applied to all the 50 joints.
2. AE-DMP Non-Integrated: DMP applied from the latent space.
3. AE-DMP Integrated: DMP completely integrated into the AE as new layer in the whole structure.

3.3.1.1 Basic DMP

The first structure considered is the classical DMP. It consists of applying for each joint an independent DMP with their respective forcing term, acceleration and speed. Only the canonical time is shared.

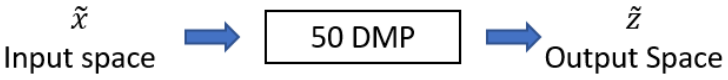


Figure 3.9: Basic DMP structure, one dmp for each joint

The set of parameters w can be trained with locally weighted regression(LWR).
The optimization problem for each kernel function Ψ_i in f corresponding w_i, is:

$$J_i = \sum_{t=1}^{P} \Psi_i(t) \cdot \left(f_{target}(t) - w_i \xi(t)\right)^2, \tag{3.12}$$

where $\xi(t) = x(t)(g - y_0)$.
This is a weighted linear regression problem with the solution:

$$w_i = \frac{\mathbf{s}^T \Gamma_i \mathbf{f_{target}}}{\mathbf{s}^T \Gamma_i \mathbf{s}}, \tag{3.13}$$

where:

$$\mathbf{s} = \begin{pmatrix} \xi(1) \\ \xi(2) \\ \dots \\ \xi(P) \end{pmatrix} \quad \Gamma_i = \begin{pmatrix} \Psi_i(1) & & & 0 \\ & \Psi_i(2) & & \\ & & \dots & \\ 0 & & & \Psi_i(P) \end{pmatrix} \quad \mathbf{f_{target}} = \begin{pmatrix} f_{target}(1) \\ f_{target}(2) \\ \dots \\ f_{target}(P) \end{pmatrix} \tag{3.14}$$

3.3.1.2 AE-DMP Non-Integrated

The second one is a naive implementation of the model AE-DMP presented in this thesis. Where the DMP only acts as an observer to this latent space and forcing the AE to adapt the codification of this space without considering the interaction from the DMP to the output. As show in the scheme the target force is obtained in the latent space, so only 5 dimensions are needed.
For the training the new loss function is the combination of both parts, where the AE is the main objective and the DMP is the constraint that regularizes the training:

$$\theta^\star, \theta'^\star, w^\star = \arg\min_{w,\theta,\theta'} \frac{1}{n} \sum_{i=1}^{n} L\left(\mathbf{x}^{(i)}, \tilde{\mathbf{z}}^{(i)}\right) + \lambda \cdot L\left(\mathbf{f_t}^{(i)}, \mathbf{f}^{(i)}\right). \tag{3.15}$$

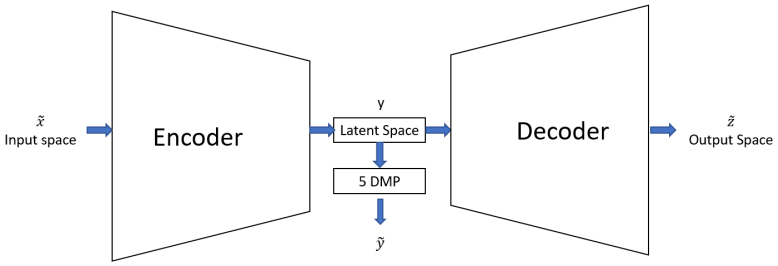


Figure 3.10: AE-DMP non-Integrated structure. The DMP is not interfering directly with the AE.

Starting off with a demonstration of a unique joint in the joint space $\mathbf{x} = [x_1, x_2, ..., x_n]$, where n is the number of frames. Introduced to the autoencoder is translated to the latent space $\mathbf{yd} = [yd_1, yd_2, ..., yd_n]$ as previous shown in the Section 2.2. To simplify the equations are referenced to only one joint or dimension to extrapolate for bigger dimensions is only needed to repeat the equations for each of the dimensions. In order to obtain the forcing term the first and second derivatives of this demonstration have to be found. The discretized derivation of a sampled data is:

$$\begin{aligned} \dot{yd}_t &= \frac{yd_t - yd_{t-1}}{ts}, \\ \ddot{yd}_t &= \frac{yd_{t-1} - 2yd_t + yd_{t+1}}{ts^2}. \end{aligned} \tag{3.16}$$

This previous equations can be translated to a matrix form. Allowing us to apply the derivatives calculation to all the members in a array at the same time:

$$\mathbf{W_d} = \begin{bmatrix} -1 & 1 & 0 & \cdots & 0 \\ -1 & 1 & 0 & \cdots & 0 \\ 0 & -1 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \quad \mathbf{W_{dd}} = \begin{bmatrix} 1 & -2 & 1 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & 1 & -2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \tag{3.17}$$

The solution of applying this matrices to the demonstration array is an array with all the step first and second derivatives:

$$\begin{aligned} \dot{\mathbf{yd}} &= (\mathbf{W_d} \cdot \mathbf{yd}) \frac{1}{ts}, \\ \ddot{\mathbf{yd}} &= (\mathbf{W_{dd}} \cdot \mathbf{yd}) \frac{1}{ts^2}, \end{aligned} \tag{3.18}$$

where $\dot{\mathbf{yd}} = [\dot{yd}_0, \dot{yd}_1, ..., \dot{yd}_n]$ and $\ddot{\mathbf{yd}} = [\ddot{yd}_0, \ddot{yd}_1, ..., \ddot{yd}_n]$ are the array of the first derivative and the second derivative, respectively, for each step. Using (2.8) we found the target force in every step of the demonstration.

In (3.17) the first and last rows are modified. We suppose that the system is already in movement before and after the demonstration. Due that these demonstrations are obtained from a real human movement. With these adjustments we can ovoid the emergence of spikes in the first and second derivatives. Leading to a smoother forcing term. During the generation of new movements this has to be taking into account, and initialize the model with a initial speed and acceleration.

Using the back-propagation algorithm in Tensorflow, the error is propagated from the DMP part to the encoder part in the AE. With this particular structure a problem arise. Meanwhile the encoder part tries to catch up with the DMP, the decoder only decodes a data that comes from the demonstration. As such gradients coming from the error in the DMP part are larger and the latent space is shaped to adapt perfectly to the DMP but the cohesion in the AE is deformed when more precision in the DMP is required. And during the generation of new movements any little disturbance or error in the DMP calculation in the latent space can lead to unknown consequences because the decoder part is only trained with data coming from the perfect demonstration and is not able to adapt to this disturbances.

3.3.1.3 AE-DMP Integrated

The last introduced structure is embedding the DMP into the AE structure as a layer in the neural network structure. This is accomplished with a few equation manipulation to obtain the DMP transition.

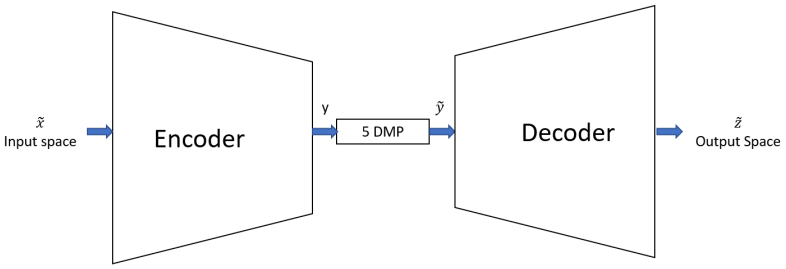


Figure 3.11: AE-DMP Integrated structure, the DMP is totally embeded into the AE

Given a demonstration in the joint space $\mathbf{x} = [x_1, x_2, ..., x_n]$ the system pass through the encoder part and obtains the $\mathbf{yd} = [yd_1, yd_2, ..., yd_n]$ in the latent space. In order to embed the DMP into that latent space we have to find transformation in the transition from y_t to y_{t+1} in a new layer in the NN allowing to use back-propagation for the training.

The first step is obtain the dynamical equations of the DMP. The second derivative equation is obtained from the basic formulation of the DMP (2.3):

$$\tau \ddot{y}_{t+1} = \alpha_y(\beta_y(y^{goal} - y_t) - \dot{y}_t) + f(s), \tag{3.19}$$

while the rest of formulas, velocity and position, are obtained integrating the acceleration:

$$\begin{aligned} \dot{y}_{t+1} &= \ddot{y}_t ts + \dot{y}_t, \\ y_{t+1} &= \dot{y}_t ts + y_t, \end{aligned} \tag{3.20}$$

where ts is the sample time, and all the y are the variables in the latent space. The $f(s)$ is the forcing term for each step.

If we substitute (3.19) into the first equation in (3.20) and then recursively the same for the second obtaining the dynamical equations with two state variables:

$$\begin{aligned}\dot{y}_{t+1} &= \frac{\alpha_y(\beta_y(y^{goal}-y_t)-\dot{y}_t)+f(s)}{\tau}ts+\dot{y}_t,\\ y_{t+1} &= \left(\frac{\alpha_y(\beta_y(y^{goal}-y_t)-\dot{y}_t)+f(s)}{\tau}ts+\dot{y}_t\right)ts+y_t.\end{aligned} \tag{3.21}$$

Two state variables are chosen, y and $\dot{y}$.

$$\begin{aligned}\dot{y}_{t+1} &= \left(\frac{\tau-\alpha ts}{\tau}\right)\dot{y}_t-\frac{\alpha\beta ts}{\tau}y_t+\frac{\alpha\beta y^{goal}ts}{\tau}+\frac{f(s)ts}{\tau},\\ y_{t+1} &= \left(\frac{\tau ts-\alpha_y ts^2}{\tau}\right)\dot{y}_t+\left(\frac{\tau-\alpha_y\beta_y ts^2}{\tau}\right)y_t+\frac{\alpha\beta y^{goal}ts^2}{\tau}+\frac{f(s)ts^2}{\tau}.\end{aligned} \tag{3.22}$$

The previous equations can be translated to the state space as:

$$\begin{bmatrix}y_{t+1}\\ \dot{y}_{t+1}\end{bmatrix} = \mathbf{A}\begin{bmatrix}y_t\\ \dot{y}_t\end{bmatrix}+\mathbf{b}. \tag{3.23}$$

From now in this section all the α_y and β_y are simply written as α,β, and the y^{goal} which is yd_N is simply written as g in order to simply the equations. Now the transitions matrices can be defined.
The state matrix is written as:

$$A = \begin{bmatrix}-ts\alpha\beta\frac{1}{\tau} & -ts\alpha\beta\frac{1}{\tau}+1\\ -\alpha\beta\frac{1}{\tau} & -\alpha\frac{1}{\tau}\end{bmatrix}ts+I, \tag{3.24}$$

and the control input as:

$$b = \begin{bmatrix}ts\\ 1\end{bmatrix}\left(\alpha\beta g+f(s)\right)ts\frac{1}{\tau}. \tag{3.25}$$

Developing the previous equations we are able to find the computation of the next step $\tilde{y}$, using the yd of the demonstration:

$$\tilde{y}_t = \left(-\frac{ts^2\alpha\beta}{\tau}+1\right)yd_{t-1}+\left(-\frac{ts^2\alpha}{\tau}+ts\right)\dot{yd}_{t-1}+\frac{ts^2\alpha\beta}{\tau}g+\frac{ts^2}{\tau}f(s). \tag{3.26}$$

The equation can be identified with constant terms that don't depend on the data or the time and variable terms:

$$\tilde{y}_t = A_1 yd_{t-1}+B_1\dot{yd}_{t-1}+C_1 g+D_1 f(s). \tag{3.27}$$

This constants only depend from the parameters defined by us or the data characteristics:

$$\begin{aligned}A_1 &= \frac{\tau-ts^2\alpha\beta}{\tau},\\ B_1 &= \frac{\tau ts-ts^2\alpha}{\tau},\\ C_1 &= \frac{ts^2\alpha\beta}{\tau},\\ D_1 &= \frac{ts^2}{\tau}.\end{aligned} \tag{3.28}$$

Now this terms, A_1, B_1, C_1, D_1, can be introduced to a matrix form, with this we are able to apply the transition to all the elements of the array:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ A_1 & 0 & \cdots & 0 & 0 \\ 0 & A_1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & A_1 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ B_1 & 0 & \cdots & 0 & 0 \\ 0 & B_1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & B_1 & 0 \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} 0 \\ C_1 \\ \vdots \\ C_1 \\ C_1 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 0 \\ D_1 \\ \vdots \\ D_1 \\ D_1 \end{bmatrix} \tag{3.29}$$

With this new transition matrices we are able to compute the $\tilde{y}$ for all the demonstration array in each of its frames:

$$\tilde{\mathbf{y}} = \mathbf{A} \cdot \mathbf{yd} + \mathbf{B} \cdot \mathbf{yd} + \mathbf{C} \cdot g + \mathbf{D} \cdot \mathbf{f}, \tag{3.30}$$

where now **yd**, $\dot{\mathbf{yd}}$, are arrays obtained from the demonstration. g is the goal position, which is constant in all the time. And the **f** is the forcing term array generated with (2.6) where the variables **w** are the ones that have to be trained.

The input and the output in this new NN layer will have a one-to-one correspondence where its functions is to generate the next step from the previous:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \rightarrow \begin{bmatrix} y_0 \\ \tilde{y}_1 \\ \tilde{y}_2 \\ \vdots \\ \tilde{y}_N \end{bmatrix}. \tag{3.31}$$

The step $\tilde{y}_0$ is the same as y_0 due that is the starting point. But the next step $\tilde{y}_1$ is generated with previous values and goes one for each frame. With this new NN layer we are to reproduce the interference of the DMP in the generation of new movement on the decoder part of the AE. If the DMP is no completely accurate the correspondence $\mathbf{y} = \tilde{\mathbf{y}}$ will not exist, not like in the AE-DMP Non-Integrated structure.But when the model is trained and the **w** properly calculated then $L(\mathbf{f_t}^{(i)}, \mathbf{f}^{(i)}) \rightarrow 0$ then $\mathbf{y} \approx \tilde{\mathbf{y}}$. With this structure the DMP is transformed to a new hidden layer between the encoder and the decoder and now can be applied back-propagation taking into account the interference of the DMP in the AE.

3.3.2 Regularization in the training step

In order to do a training, a loss function have to be defined. Decide which terms has more importance and which ones have less weight is crucial to regularize correctly all the terms.

The principal objective in the AE is to keep the cohesion between the inputs and the outputs, trying to obtain the outputs more analogous to the inputs possible. The AE-DMP have also the task to shape the latent space in the best possible way to simplify the training in the DMP part. This part is achieved adding the regularization to

the loss function in the AE:

$$\theta^\star, \theta'^\star, w^\star = \arg\min_{w,\theta,\theta'} \frac{1}{n} \sum_{i=1}^{n} L\left(\mathbf{x}^{(i)}, \tilde{\mathbf{z}}^{(i)}\right) + \lambda \cdot L\left(\mathbf{f_t}^{(i)}, \mathbf{f}^{(i)}\right). \tag{3.32}$$

The chosen function L is L_2-norm but also different function can be considered:

- SVMR
$$L(x^{(i)}, z^{(i)}) = |x^{(i)} - z^{(i)}|_\epsilon \tag{3.33}$$
- SVMC
$$L(x^{(i)}, z^{(i)}) = |1 - x^{(i)} z^{(i)}|_+ \tag{3.34}$$
- Hard margin
$$L(x^{(i)}, z^{(i)}) = \theta(1 - x^{(i)} z^{(i)}) \tag{3.35}$$
- Missclassification loss
$$L(x^{(i)}, z^{(i)}) = \theta(-x^{(i)} z^{(i)}) \tag{3.36}$$

These are the most used functions in regularization, but have a particularity in common; in some moments the error value or the derivative is 0, and no gradient can be extracted from this parts, or more specific are not C^1 smooth. Even for the $\theta(x)$ the Heaviside function being a C^{-1} function, which its derivative is the delta function. Leading that the only feasible options for our setup are L_2 and L_1 norms.

3.3.3 Latent space

The main objective of using AE is to obtain a new reduced space where DMP can be applied more easily. It is similar to SVM for classification. Having a huge and complex space reduced to a smaller and adaptable to our requirements, improving the training and the performance.

Each time this latent space will be different due that not only one solution will exist. It is chosen to be a 5 dimensional space, where this dimension do not have any meaning in the joint space or the state space. In this space the movements are codified. If it is observed some patterns can be appear (Figure 3.12). This scatter plot show which is the constitution of the different dimensions in two movements, running and walking.

As we can see the two movement have different features in each dimension and in the combination of different dimension codifies this circular movement. In the dimension number 5 this can be observed, depending of the dimension one of the movement is more predominant while the other have no significant variance. In the fourth dimension it is easy to notice that codifies the differentiation between the 2 movements and are not circular centered in the same spot so when one movement is playing here will be differentiated in where the center is allocated. The plots are obtained from independent trained models, but the results are the similar in models that are trained with both data (Figure 3.13).

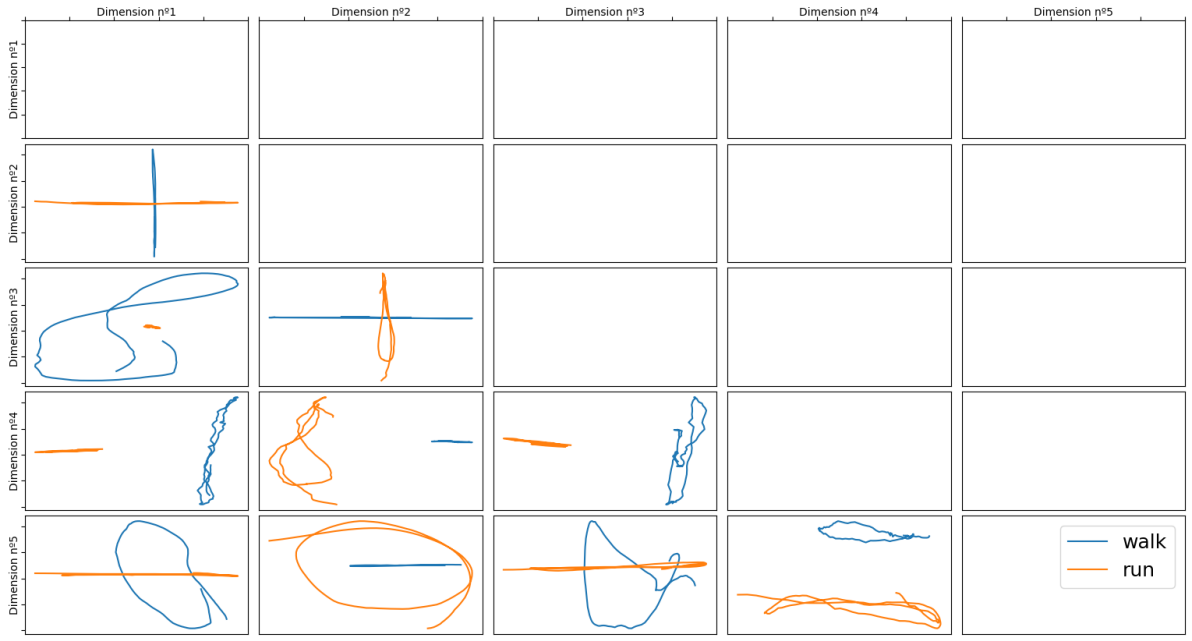


Figure 3.12: Matrix plot of all the dimensions in the latent space

3.3.4 Reconstruction of a missing joint

The implementation of the Denoising AE allows the system to reconstruct broken or corrupted data. DAE have been shown to be generative models . If the input data is corrupted the results should yield almost to the same representation that have been trained.

This is a consequence of the fully layer connected structure of the NN. The model is able to capture the dependencies and regularities characteristic of the distribution in the input data. The DAE reconstruct this blanks from the behaviour of the other variables. In the experiments this will be tested as the complete loss of information in a particular joint in the joint space. After training the model using the process explained in the Section 2.3.

3.3.5 Sparse AE-DMP

In order to use different movements the sparse AE-DMP allows to codify the movement using less dimensions so that one or more that dimensions tends to 0.

As we can see the model is forced to adapt the movement in less dimensions than the full 5-dimensional space. A few of them are reduced close to 0, minimizing its impact in the movement generation. With this characteristic if different movements are trained in the same model, the system tries to distribute the patterns for this space. The Sparse form can also used to obtain a more general model of a movement using different demonstrations for it. With this more generalized model the system is able to find a more deep representation of the dynamics in the movement. The different movements are distributed in the latent space, and comparing with the previous results of training separately Figure 3.12 the combined training of the movements in the same model gives better results Figure 3.13. The differentiation between the movements is more clear and observable in every dimension combination.

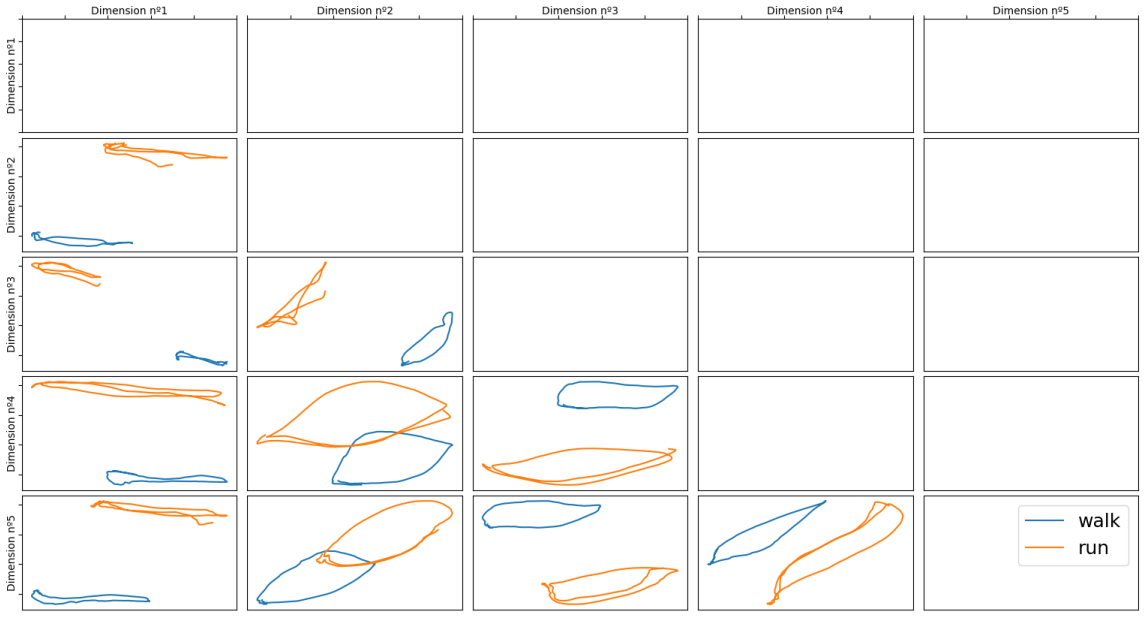


Figure 3.13: Matrix plot of all the dimensions in the latent space

3.3.6 Interpolation between movements

Starting from the Sparse AR-DMP if the model is trained with different models, then the system is able to generate new movements with the adequate tuning. Deactivating or changing some hidden neurons the movement generated will be a mix of the different demsotrations.

Also is possible to simulate the evolution from one movement to another, for example: starting running and then low the pace and continues walking. This is achieved with a combination of two DMP parameters, the weights and the goals:

$$w^* = \sum_i \left(\frac{\eta^i w^i}{\sum_i \eta^i} \right), \tag{3.37}$$

$$g^* = \sum_i \left(\frac{\eta^i g^i}{\sum_i \eta^i} \right). \tag{3.38}$$

The new w^* and g^* substitute the original variables in the model, and are tuned depending on our criterion to create the proper transition or direction. The η parameter is constrain to $\eta^i \in [0,1]$. Finally i is the number of movements learned.

Only 2 movements will be studied, walking and jogging, so only two parameters are considered $[\eta^1, \eta^2]$. To simplify the task both parameters will be complementaries, $\eta^2 = (1-\eta^1)$. If we develop the previous equation with this constraints:

$$w^* = \frac{\eta^1 w^1 + \eta^2 w^2}{\eta^1 + \eta^2}. \tag{3.39}$$

Knowing that the imposition $\eta^1 + \eta^2 = 1$ is conserved for all the interpolation :

$$w^* = \eta^1 w^1 + \eta^2 w^2. \tag{3.40}$$

A similar equation results for the goal:

$$g^* = \eta^1 g^1 + \eta^2 g^2. \tag{3.41}$$

For the η evolution is chosen to be like a logistic function. The transition is chosen in a certain point in the time and is centered in the logistic function by t_0, then the values are adapted to get a 0 close in the start and a 1 in the end of the simulation. This can also be reversed to obtain the inverse interpolation.

$$\eta(t) = \frac{1}{1 + e^{-m*(t-t_0)}} \tag{3.42}$$

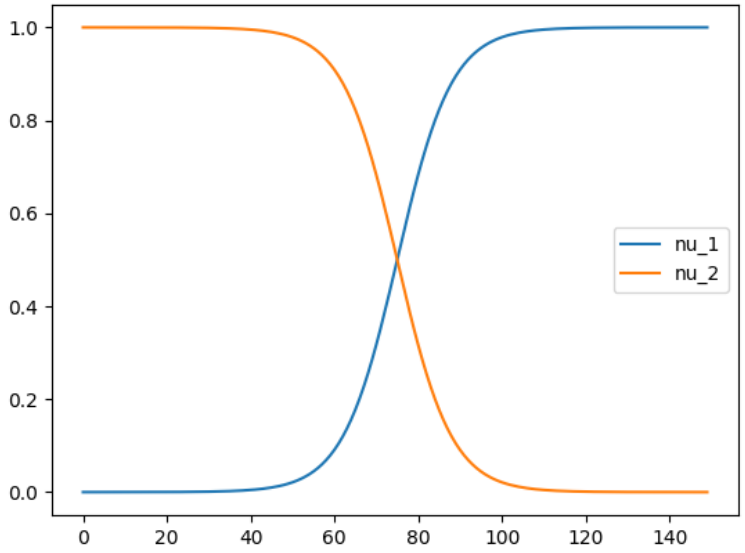


Figure 3.14: η values evolution during time

With this evolution the transition should be enough smooth and stable during the start and the end to obtain a good movement.

CHAPTER 4

PROGRAMMING

In this section a concise explanation of the programming language and library used in this thesis. Including the main points of the code used in the model.

4.1 PYTHON

Python[1] is a high-level programming language for general purpose. Is a interpreted language, its objective is to emphasize readability, with a syntax that allow to do more in fewer lines of code than other languages such as C++ or Java. Also incorporate dynamic type system and dynamic memory management.

4.2 DATA ACQUISITION

The data is extracted from CMU Graphics Lab Motion Capture Database[2], for this project the subject 35 is used to obtain the demonstration movements.
The file format chose is ".amc", it's a text format where all the movement is codified in frames, at constant framerate defined in a ".asf" file. Each one of this frame contains the angle in degrees of every joint and it's degrees of freedom. In this case the demonstration is composed by 62 features. 6 values are the root position respect to the origin, can be ignored in our purpose. Also 6 other features have a variance of $< 10^{26}$(shoulder and finger movement) and can be ignored from our input vector. This leads to a 50 joint state space.
The data is read from the ".amc" file through a personal created function "$readfromamc\_fun$" Appendix [A]. Which takes the file name as input and outputs a numpy array of dimension [50xN], where N is the number of frames.
The input array x have this form:
$[x_1, x_2, x_3; lowerback], [x_4, x_5, x_6; upperback], [x_7, x_8, x_9; thorax], [x_{10}, x_{11}, x_{12}; lowerneck],$
$[x_{13}, x_{14}, x_{15}; upperneck], [x_{16}, x_{17}, x_{18}; head], [x_{19}, x_{20}, x_{21}; rhumerous], [x_{22}; rradius], [x_{23}; rwrist],$
$[x_{24}, x_{25}; rhand], [x_{26}, x_{27}; rthumb], [x_{28}, x_{29}, x_{30}; lhumerous], [x_{31}; lradius], [x_{32}; lwrist], [x_{33}, x_{34}; lhand],$
$[x_{35}, x_{36}; lthumb], [x_{37}, x_{38}, x_{39}; rfemur], [x_{40}; rtibia], [x_{41}.x_{42}; rfoot], [x_{43}; rtoes], [x_{44}, x_{45}, x_{46}; lfemur],$

[1] Python programming language: https://www.python.org/
[2] CMU Graphics Lab Motion Capture Database: http://mocap.cs.cmu.edu/

$[x_{47}; rtibia], [x_{48}.x_{49}; lfoot], [x_{50}; ltoes]$

In the opposite way, a function that writes a ".amc" file when is needed to test the results in a simulation of a skeleton "$writetoamc\_func$" Appendix[B].

4.3 USED LIBRARIES

4.3.1 TENSORFLOW

To do all the training of the project, TensorFlow[3] is used. TensorFlow is an open source software library for numerical computation using data flow graphs. This graph construction is perfect for NN, but also is used in a wide variety of other domains as well. The ability of using GPU as a computing unit allows to reduce drastically the training time.

4.3.2 SCIPY

SciPy[4] (pronounced “Sigh Pie”) is a Python-based ecosystem of open-source software for mathematics, science, and engineering. During the execution of this thesis some of the packages of this ecosystem are used. The ones used are the following.

4.3.3 NUMPY

NumPy[5] is the fundamental package for scientific computing with Python. With Numpy we are able to work with N-dimensional arrays in a optimized way, this allows us to link the list structures to arrays and then to Tensorflow. Also that is a fully optimized library boosting the performance of math computation processes.

4.3.4 MATPLOT

Matplotlib[6] is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

4.3.5 SCIPY SIGNAL PROCESSING

The signal processing toolbox[7] currently contains some filtering functions, a limited set of filter design tools, and a few B-spline interpolation algorithms for one- and two-dimensional data. This is used to filter or resample the demonstration data.

[3] Tensorflow Machine Learning library: https://www.tensorflow.org/
[4] Scipy libraries: https://www.scipy.org/
[5] Numpy scientific computation: http://www.numpy.org/
[6] Matplotlib Python plotting library: https://matplotlib.org/
[7] Signal processing: https://docs.scipy.org/doc/scipy/reference/tutorial/signal.html

4.4 CODE STRUCTURE

All the code is separated in two parts, the model training and the model testing.

4.4.1 MODEL TRAINING

All the project is constructed around TensorFlow library, so the code structure is delimited by its rules. Firstly the a graph have to be defined.

In Tensorflow 3 main types of introducing data exist. As a "placeholder" where is the data is feed in the different executions. The "placeholder" are usually the input data for the training or parameters that can be changed o defined by the user just before the execution. Then "Variables, are the trainable parameters. Categorizing a tensor as "Variable" Tensorflow detects as trainable and when the gradient updating step is executed this parameters are changed according to the training rules. And finally "Constants", parameters that are defined in the code and do not change.

The AE model is written as sequence of fully connected layer.

Listing 4.1: Neural Network layer

```
with tf.name_scope('hidden_1') as scope:
    W_h1 = tf.Variable(tf.random_normal([inputs,h_1_neurons],stddev=0.1),name="Weights_hidden_1")#
     Neuron weights
    b_h1 = tf.Variable(tf.constant(0.1,shape = [h_1_neurons]),name = "Bias_hidden_1")#Neuron bias

    h_1 = tf.tanh(tf.matmul(x_norm,(W_h1*Denoised_W_h1))+b_h1)#Neuron operation


```

Each layer is defined by three parts, the layers weights, the layer bias, and the layer computation. Each computation in Tensoflow is build as a node in the general graph while the Tensors are the information going between this computation nodes.

After building the AE structure the optimization function have to be defined:

Listing 4.2: Optimization problem definition

```
# Loss function
with tf.name_scope('Loss_AE') as scope:
    loss_AE = tf.reduce_mean(tf.sqrt(tf.reduce_sum(tf.square(x_norm-z),1)),0)
tf.summary.scalar('Loss_AE', loss_AE)#Data login operation

#AE training function
with tf.name_scope('Opt_AE') as scope:
    train_step_AE = tf.train.AdadeltaOptimizer(0.1).minimize(loss_AE)

```

First is defined a loss function and then the optimization problem, with an objective of minimizing this loss function. We are searching the point where the difference between the inputs and the outputs are the smallest possible.

The Code: 4.2 in the line 4 the command "tf.summary.scalar('Loss_AE', loss_AE)" is used in the data logging aspect of Tensorflow. Tensorflow provides a complete data inspection with Tensorboard. This allows the user to

store the detailed data during the training, this can be any scalar, matrix, tensor, etc. After that can be visualized building a server with Tensorboard, can be private in your own computer but also provides the possibility of uploading to a web server and updating in real time.

The DMP is built in different parts. The basis functions are the essential part in the DMP and are built as this:

Listing 4.3: Kernel building

```
with tf.name_scope("Basis_funcs") as scope:
    with tf.name_scope("Kernel_fun_params") as scope:
        mean = tf.linspace(ts,t_final,num)
        mean_distributed = X0*tf.exp(-alfax*mean/tau)
        var_distributed = num/mean_distributed

    with tf.name_scope("Weights") as scope:
        w = []
        for i in range(dim):
            name = "Weights" + str(i)
            scope_name = 'Weights_kernel_' + str(i)
            with tf.name_scope(scope_name) as scope:
                w.append(tf.Variable(tf.truncated_normal([num], stddev=1),name=name))

    with tf.name_scope("Kernel") as scope:
        time = [tf.linspace(0.0,t_final,Data_in.shape[0])]
        aux = tf.zeros([num,1],dtype=tf.float32)
        s_t = tf.transpose(time + aux)

        s = X0*tf.exp(-alfax*s_t/tau)
        internm = -(var_distributed*tf.square(tf.subtract(s,mean_distributed)))
        kernel = tf.exp(internm,name="Kernel_fun")


```

With this we simulate the basis functions values in all the time axis. The time is generated with the values know, initial time, finish time, and time step.

Listing 4.4: Forcing term

```
with tf.name_scope("Forcing_term") as scope:
    f_list = []
    kernel_sum = tf.expand_dims(tf.reduce_sum(kernel,1),1)
    for i in range(dim):
        f_list.append(tf.divide(tf.matmul(kernel,(tf.expand_dims(w[i],1))),kernel_sum))

```

Combining the kernel with the weights variables the model obtains the force value for all the latent space joints in all the time axis. This forcing term will be compared with the target force obtained from the demonstration.

Listing 4.5: Force target

```
with tf.name_scope("Goal") as scope:
    goal = tf.transpose(tf.expand_dims(y[-1],axis=1))

goal = tf.stop_gradient(goal)

with tf.name_scope("Forcing_target_term") as scope:
    f_target_list = []
```

```
    y_dd_matrix = tf.matmul(W_dd_tensor,y)
    y_d_matrix = tf.matmul(W_d_tensor,y)
    for i in range(dim):
        y_dd = tf.slice(y_dd_matrix,[0,i],[-1,1])
        y_d = tf.slice(y_d_matrix,[0,i],[-1,1])
        f_target_list.append(tf.multiply(tf.square(tf.constant(tau,dtype=tf.float32)),y_dd)-tf.multiply(tf.constant(alfa,dtype=tf.float32),tf.subtract(tf.multiply(beta,tf.subtract(goal[0,i],tf.slice(y,[0,i],[-1,1]))),tf.multiply(tf.constant(tau,dtype=tf.float32),y_d))))

with tf.name_scope("Pack_tensor_lists") as scope:
    f_target = tf.transpose(tf.squeeze(tf.stack(f_target_list)))
    f = tf.transpose(tf.squeeze(tf.stack(f_list)))

```

In Code 4.5 the operation in line 4 is important for the training step. In order to avoid the optimizer to propagate the gradients through the goal value the operand "tf.stop_gradient" is added. After all this process two tensors are obtained, "f_target" and "f" which represented the objective and the generated by the current weights, respectively. And the optimization problem is built as.

Listing 4.6: Loss function AE-DMP

```
with tf.name_scope("Loss_fun_AE-DMP") as scope:
    suma_AE = tf.sqrt(tf.reduce_sum(tf.square(x_norm_target-z_decoded),1))
    loss_AE_scalar = tf.reduce_sum(suma_AE,0)
    suma_DMP = tf.sqrt(tf.reduce_sum(tf.square(f_target-f),1))
    loss_DMP_scalar = tf.reduce_sum(suma_DMP,0)
    spartsity = tf.reduce_sum(tf.abs(y))
    loss_ae_dmp = tf.reduce_sum(suma_AE+nu*suma_DMP,0)+mu*spartsity

tf.summary.scalar('Loss_AE', loss_AE_scalar)
tf.summary.scalar('Loss_DMP', loss_DMP_scalar)
tf.summary.scalar('Loss_AE_DMP', loss_ae_dmp)

with tf.name_scope("Opt_DMP") as scope:
    train_step_DMP = tf.train.AdadeltaOptimizer(0.1).minimize(loss_ae_dmp)

```

For the AE-DMP Integrated the new layer representing the DMP has to be integrated into the NN.

Listing 4.7: DMP NN layer

```
with tf.name_scope("y_computed") as scope:
    y_decode = tf.matmul(A_tensor,y)+tf.matmul(B_tensor,y_d)+tf.multiply(C_tensor,goal)+tf.multiply(f,D_tensor)

```

After building all the model and establishing the training rules. The model have to be trained. This is made inside a loop, calling each execution to the optimization operation "train_step_DMP". In this operation the optimizer computes the error, the gradients, propagates them for all the connected operations and update the variable. When the training is finished or when a snapshot is required to be saved. Calling the saver function provides the possibility of saving all the model.

Listing 4.8: Saving function

```
with tf.name_scope("Saver") as scope:
```

```
    saver = tf.train.Saver()

save_path = saver.save(sess, "tmp/model.ckpt")

```

4.4.2 MODEL TESTING

As the model can be saved it can also be loaded.

Listing 4.9: Loading function

```
with tf.name_scope("Saver") as scope:
    saver = tf.train.Saver()

saver.restore(sess, "tmp_run/model.ckpt")


```

This function will load all the variables stored, exist different ways of doing this. Exist the possibility of lading all the model from this checkpoint, only the variables and assign to a new variables or build the same model and then load the variables inside. Then the model can be test with no problem.

4.4.3 MOVEMENT INTERPOLATION

In order to achieve the interpolation and the combined training of different movements, the model previous introduced have to be tweaked a bit.

Firstly each movement will have its personal weights:

Listing 4.10: Weights model with multiple movements

```
with tf.name_scope("Weights_walk") as scope:
    w_walk = []
    for i in range(dim):
        name = "Weights_walk" + str(i)
        scope_name = 'Weights_kernel_walk_' + str(i)
        with tf.name_scope(scope_name) as scope:
            w_walk.append(tf.Variable(tf.truncated_normal([num], stddev=1),name=name))
with tf.name_scope("Weights_run") as scope:
    w_run = []
    for i in range(dim):
        name = "Weights" + str(i)
        scope_name = 'Weights_kernel_run_' + str(i)
        with tf.name_scope(scope_name) as scope:
            w_run.append(tf.Variable(tf.truncated_normal([num], stddev=1),name=name))

with tf.name_scope("Basis_funcs") as scope:
    with tf.name_scope("Kernel_fun_params") as scope:
        var = tf.ones(num, dtype=tf.float32)*0.005
        mean = tf.linspace(ts,t_final,num)
        mean_distributed = X0*tf.exp(-alfax*mean/tau)
        var_distributed = num/mean_distributed
    with tf.name_scope("Kernel") as scope:
        time = [tf.linspace(0.0,t_final,sample_num)]
        aux = tf.zeros([num,1],dtype=tf.float32)
```

```
        s_t = tf.transpose(time + aux)

        s = X0*tf.exp(-alfax*s_t/tau)
        internm = -(var_distributed*tf.square(tf.subtract(s,mean_distributed)))
        # internm = -(tf.square(tf.subtract(s,mean_distributed))/(2*tf.square(var)))
        kernel = tf.exp(internm,name="Kernel_fun")

```

The new generation of the force term is a merge of the multiple movements:

Listing 4.11: Force term with multiple movements

```
with tf.name_scope("Forcing_term") as scope:
    nu_1 = tf.placeholder(tf.float32,shape=[],name="Trainning_toggler")
    f_list = []
    kernel_sum = tf.expand_dims(tf.reduce_sum(kernel,1),1)
    w = tf.multiply(w_run,nu_1)+tf.multiply(w_walk,(1-nu_1))
    for i in range(dim):
        f_list.append(tf.divide(tf.matmul(kernel,(tf.expand_dims(w[i],1))),kernel_sum))
```

During the training both groups of data are interchanged changing the parameters in order to match the training. For each movement differnt training functions are used:

Listing 4.12: Training functions

```
kernel_weights_walk = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,"Weights_walk")
kernel_weights_run = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,"Weights_run")
AE_parameters = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,"AE")

with tf.name_scope("Opt_DMP") as scope:
    train_step_DMP_run = tf.train.AdadeltaOptimizer(0.1).minimize(loss_ae_dmp,var_list= AE_parameters+
     kernel_weights_run)
    train_step_DMP_walk = tf.train.AdadeltaOptimizer(0.1).minimize(loss_ae_dmp,var_list= AE_parameters
     +kernel_weights_walk)
sess.run(tf.global_variables_initializer())
```

Tensorflow when the function "minimize" is used take as default all the trainable variables in the model, but if the variables are specified only will train that variables. This is used to separate the training of both movements. As previous said Tensorflow provides a data logger, where you can see all the variables or tensors that you indicates, histograms, images, etc. But one of the important is the possibility of observing the graph built:

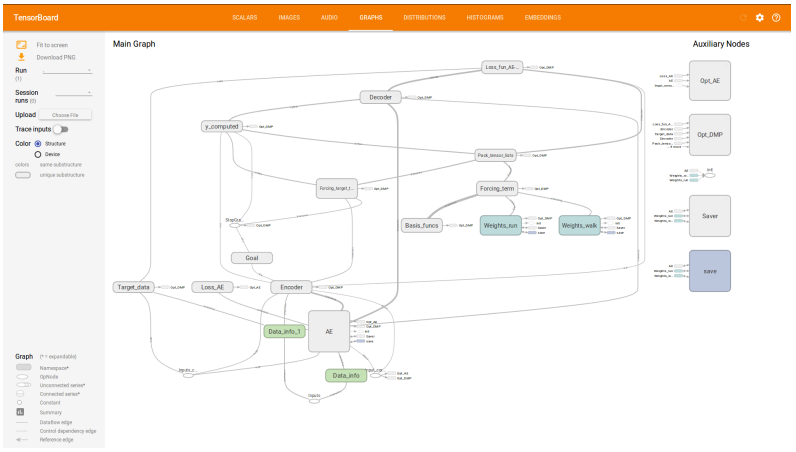


Figure 4.1: Tensorflow graph visual representation

CHAPTER 5

EXPERIMENTATION

In this section all the results obtained in the test and results of the Auto-encoder are exposed and discussed. As explained in the programming section of this project, the model is build in the library TensorFlow. Firstly the AE is tested and check separately from the DMP in order to find the structure that gives better results. And corroborate that the model works. Then the different structure proposed for the AE-DMP are tested.

5.1 AUTO ENCODER

The test of the AE is made using the demonstration of the movement. The data is separated in batches, precisely 50 for epoch, and scrambled in random order. The model is trained for a few epochs due that the objective here is not final results whereas we search only a stable snapshot to compare the different models.
During all the process only two structures will be tested, 3 layers[36,20,5] and 4 layers [60,42,25,5]. Fist with the full demonstration an then re-sampled to apply a kind of filter in the signal.
With 4 layers structure, after 4,050,000 training steps, results are obtained. With a final total loss of 0.157, Figure 5.1 depicts its behavior. Loss function decreases exponentially as expected. In fact, most of the learning models behaves like this. This also inform us that the time chosen is a good option to compare the different models.

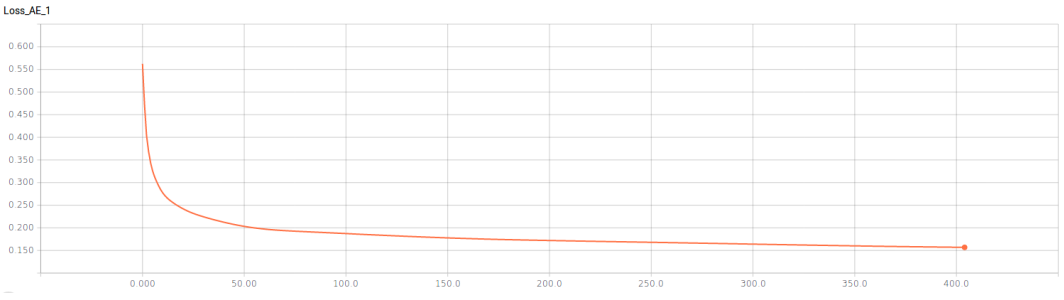


Figure 5.1: Loss evolution obtained from the 4 layer structure

Counting the average MSE ± SD for all the joints, in radians, the result is MSE: 0.133 ± 0.247. We will use the MSE plot in Figure 5.2 and the loss function to compare between the different options. The MSE plot gives information of the AE in all the joints and the precision.

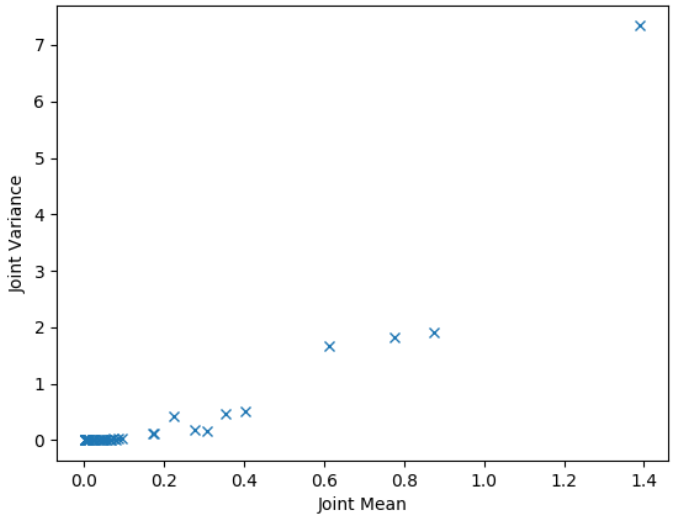
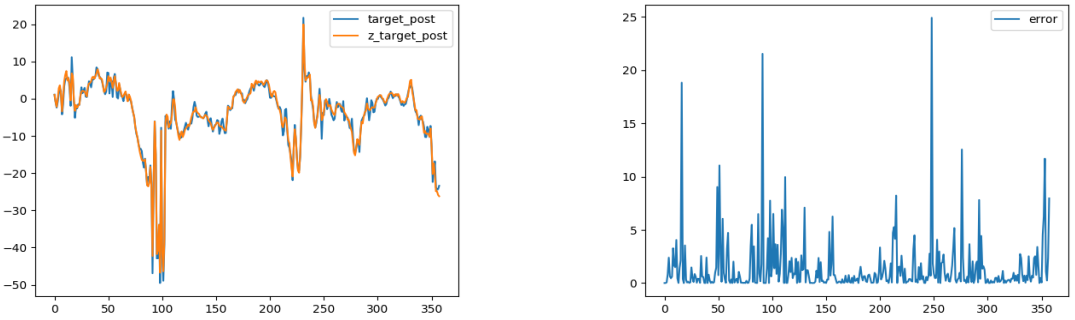


Figure 5.2: MSE plot of the joints in the 4 layer structure

As observed, a high number of joints are close to a low error but a few exceptions are far away from this objective. This is caused by the data collection method, which sometimes have error and generate unrealistic and complicated movements. For example, the joint exposed in Figure 5.3.



(a) Joint position

(b) Joint error

Figure 5.3: Example of the joint 50

Now, let us study the 3 layers structure. In this case, 4,490,000 epochs are considered and the achieved final loss is 0.131, as plotted in Figure 5.4.

The averaged MSE is 0.328 ± 0.686, clearly larger than for the 4 layers version, even with a longer training time, as it can be identified in Figure 5.5.

As the previous example, the joint number 50 leads again to problems in the training due to its complex behavior.

Now, let us observe the effects of re-sampling the signal to 100 samples instead of 358 that compose the original demonstration. The 4 layers structure, with a training of 4,910,000 epochs, leads to a final loss of 0.135 (see Figure 5.6).

The average MSE is 0.215 ± 6.389, that is a lower mean error value, but the variance increases due that less samples are used (see Figure 5.7).

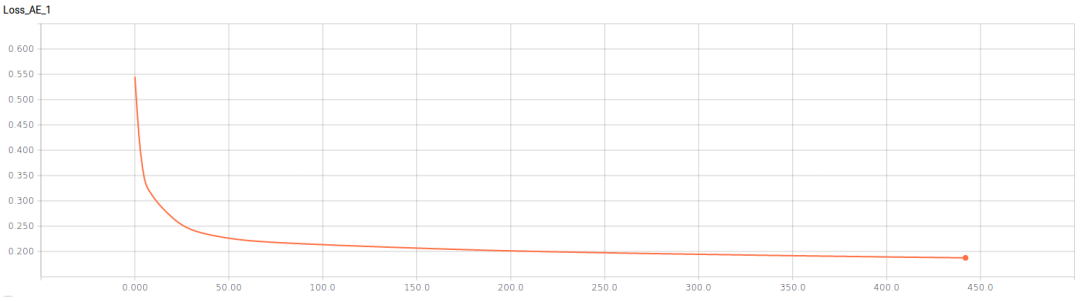


Figure 5.4: Loss evolution obtained from the 3 layer structure

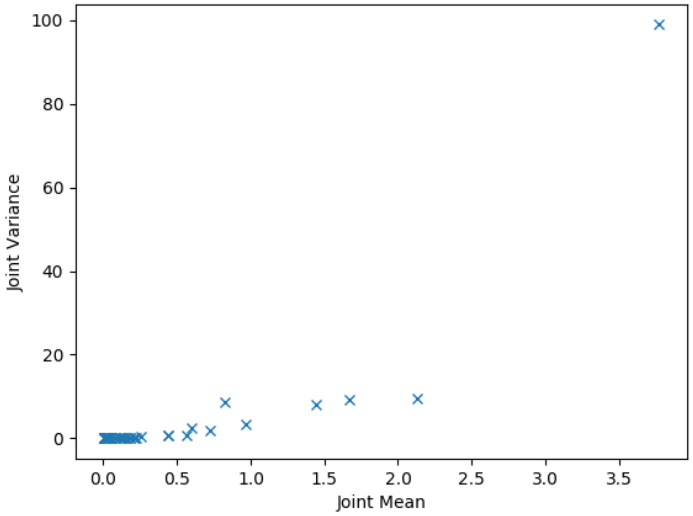


Figure 5.5: MSE plot of the joints in the 3 layer structure

The effects on resampling in the 3 layers structure, with a training of 5,000,000 and a final loss vaflue of 0.189 are depicted in Figure 5.8.

In this case, the average MSE is 0.298 ± 0.951, as shown in Figure 5.9.

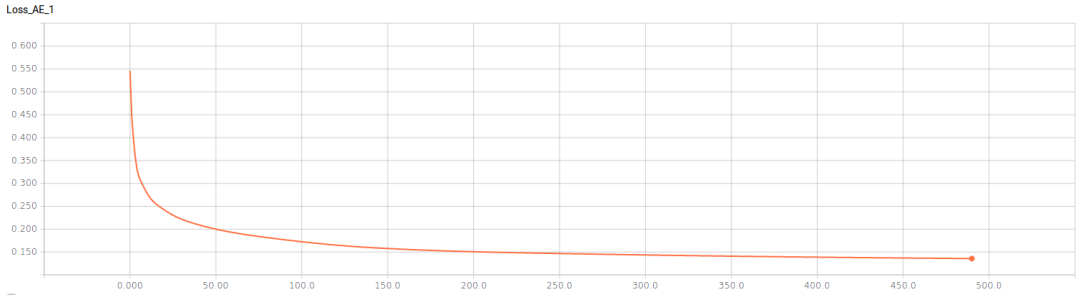


Figure 5.6: Loss evolution obtained from the 4 layer structure, data resampled to 100 samples

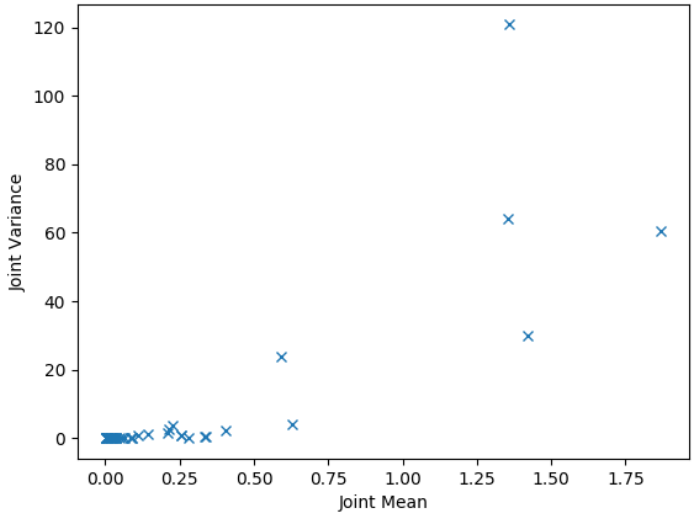


Figure 5.7: MSE plot of the joints in the 4 layer structure with 100 samples

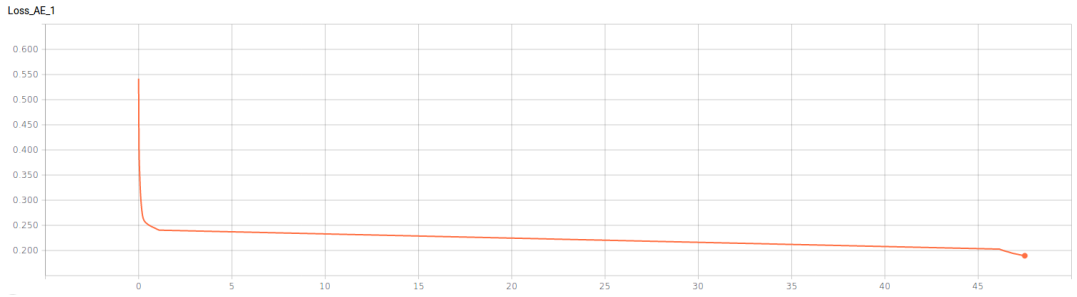


Figure 5.8: Loss evolution from the 3 layers structure, data resample to 100 samples

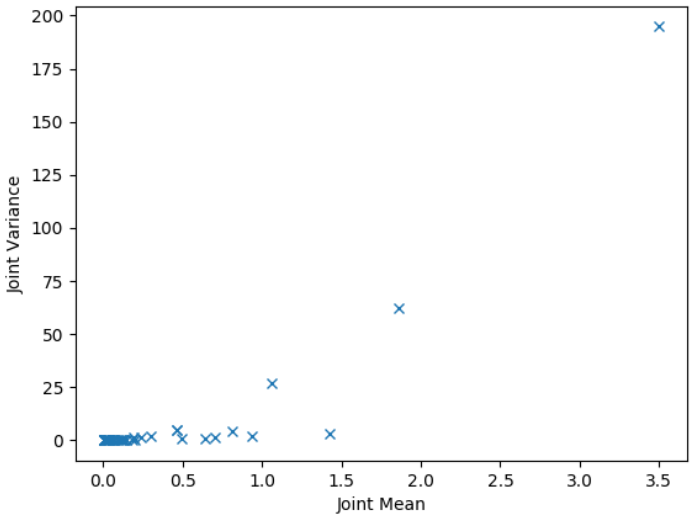


Figure 5.9: MSE plot of the joints in the 3 layers structure with 100 samples

The test for DAE is made in the same way as AE, but this time data will present some degree of corruption. Tests are conducted on 2 structures, with 3 and 4 layers, and different sampling.

Firstly, we start with a low corruption rate 0.05%. Hence, in the 50 joints and the 358 samples, for a total of 17900 instances, 895 among them will be changed to 0. At least 2 of the joints in each frame are corrupted. With this modification, DAE is forced to be able to infer the missing joints from the non-corrupted ones.

Results for the 4 layers structure, with a training of 4,550,000 epochs and a final loss value of 0.232 is presented in Figure 5.10.

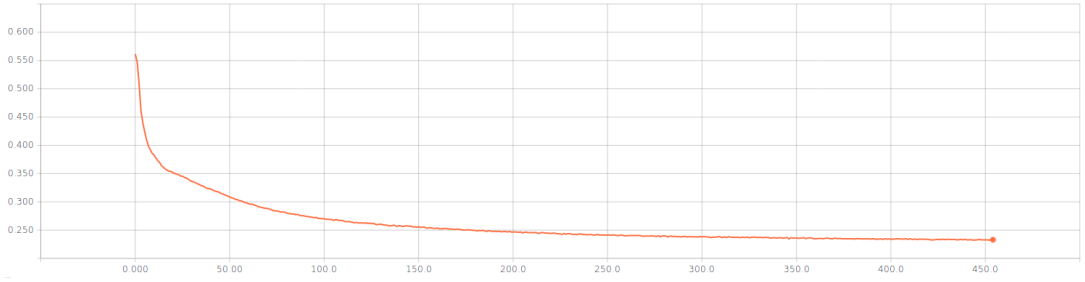


Figure 5.10: Loss evolution of the DAE 4 layers structure

The average MSE is 0.272±0.585 for this 4 layers structure with DAE (see Figure 5.11).

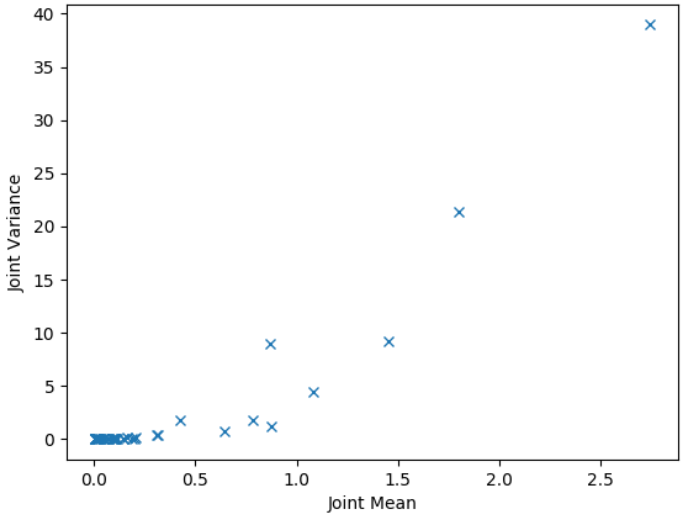


Figure 5.11: MSE plot of the joints in the 4 layers structure

The value for MSE is clearly higher than in the precedent case, but this is compensated with the DAE operation of being able to reconstruct the signal even when inputs are corrupted.

The 3 layers structure was trained for 4,450,000 epochs and the obtained final loss value was 0.219 (see Figure 5.12).

The average MSE was 0.487±1.153 (Figure 5.13).

Now, let us observe the effects of resampling the signal on the DAE. The signal now have 100 samples.

Results with the 4 layers structure, with a training of 4,630,000 epochs and a final loss of 0.164 are depicted in Figure 5.14.

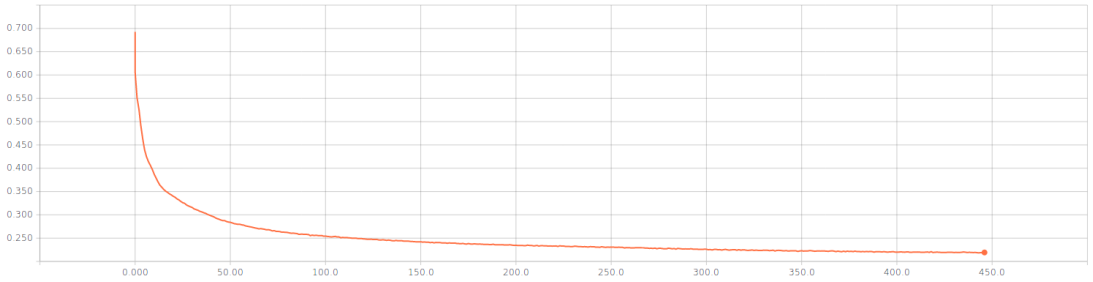


Figure 5.12: Loss evolution of the DAE 3 layers structure

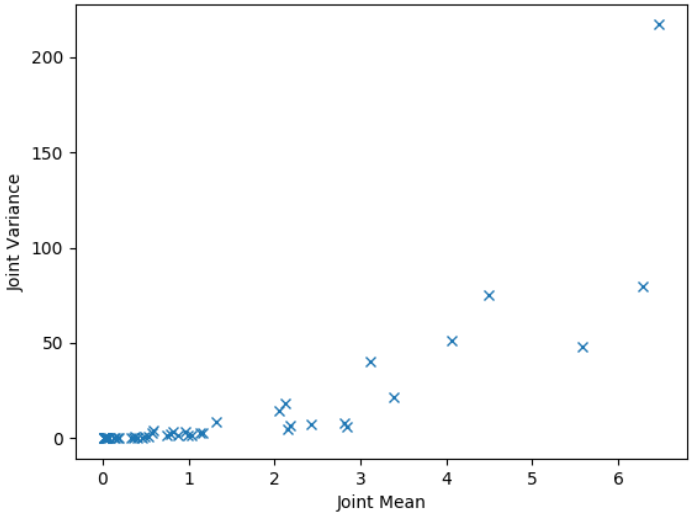


Figure 5.13: MSE plot of the joints in the 3 layer structure

In this case, the average MSE is 0.274±0.595 (see Figure 5.15). For the 3 layers structure, with a training of 4,540,000 epochs and a final loss value of 0.220, results are in Figure 5.16.

The average MSE is 0.458 ± 0.987 (see Figure 5.17).

Auto encoder conclusions:

With this test we are able to see some of the different options that exist in the implementation of the AE. Obviously, there exists an infinite space of different structures to test, however the objective of this thesis is not finding the better AE in terms of accuracy, but verify the performance of the AE-DMP approach.

The best option among the chosen structures is confirmed to be the 4 layers one. But this result leads to new

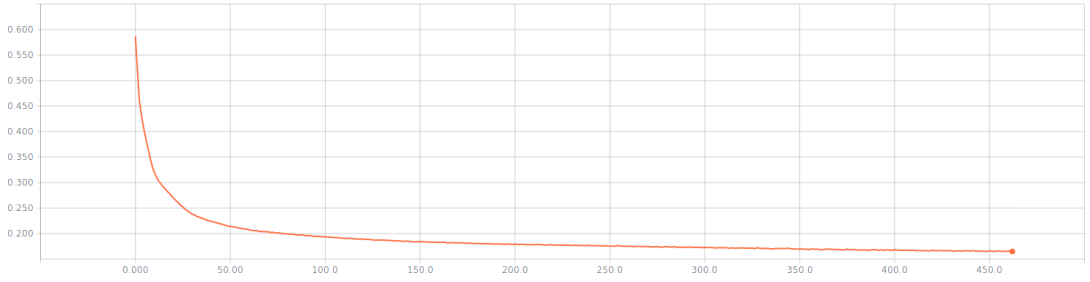


Figure 5.14: Loss evolution of the DAE 4 layers structure, data resampled to 100 samples

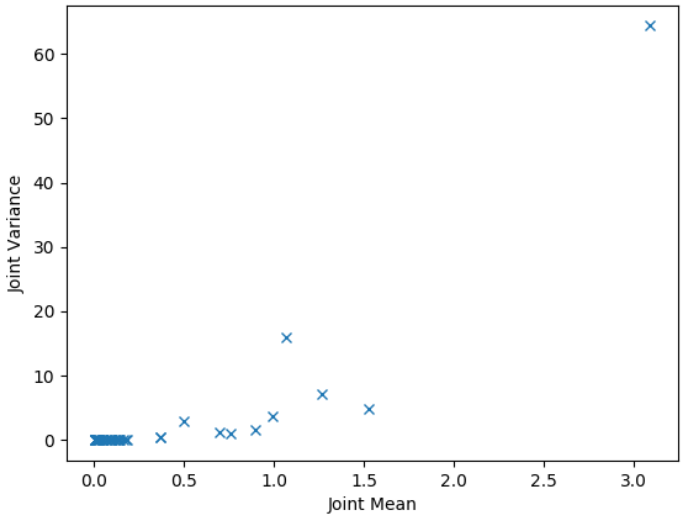


Figure 5.15: MSE plot of the joints in the 4 layers structure with 100 samples

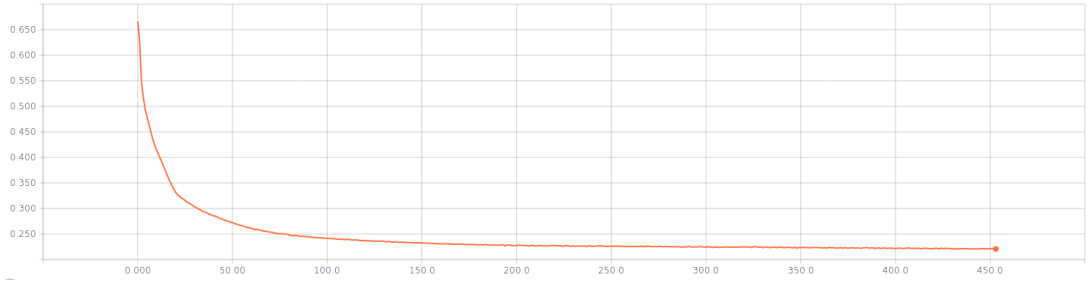


Figure 5.16: Loss evolution of the DAE 3 layers structure, data resample to 100 samples

questions; adding more layers will show better results? Which is the number of layer and nodes to chose before making the system too complex? Like in deep leaning this question don't have an exact solution.

5.2 AE-DMP

As previously mentioned, three different structures will be tested. Firstly the DMP applied to all the 50 joints in the demonstration. Then the one where the DMP "observe" the latent space; and the one where the DMP is totally integrated into the AE. In this way, we will get a control group, the basic DMP, and then different implementations in order to be able to compare and corroborate that the system works better than the basic.

5.2.1 Basic DMP

This is simply apply a DMP in each of the 50 joints. The setup is simple. For each joint the w values of the DMP are calculated. The basis functions and the canonical time is the same for all the joints.
With these model the forcing term tries to adapt to the objective force, Figure 5.20. But the complexity of the signal make impossible to find a correct match with this method.

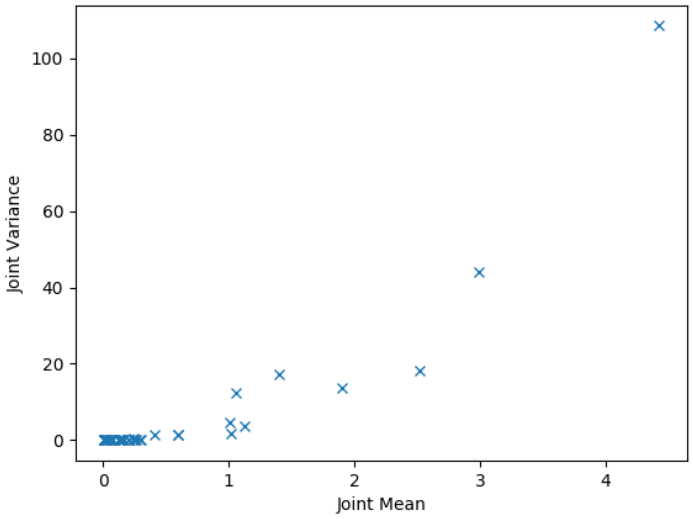


Figure 5.17: MSE plot of the joints in the 3 layers structure with 100 samples

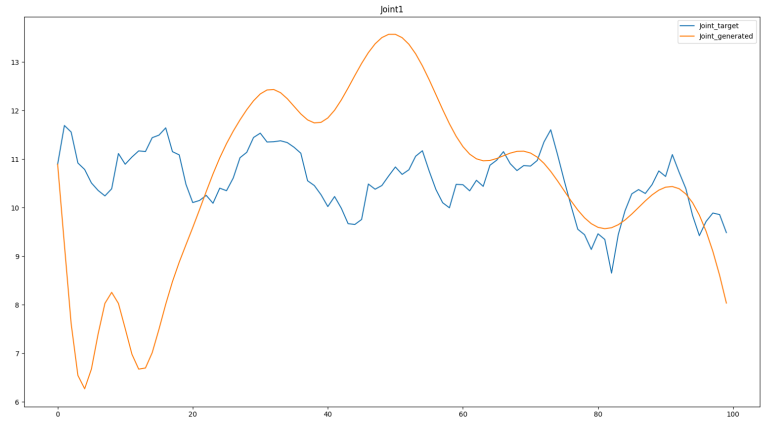


Figure 5.18: Joint 1, basic DMP

As we can observe, Figure 5.18 the DMP tries to recreate to the movement of the joint but with low success, with another joint, Figure 5.19 we can see that the results are similar. Therefore the system is not working.

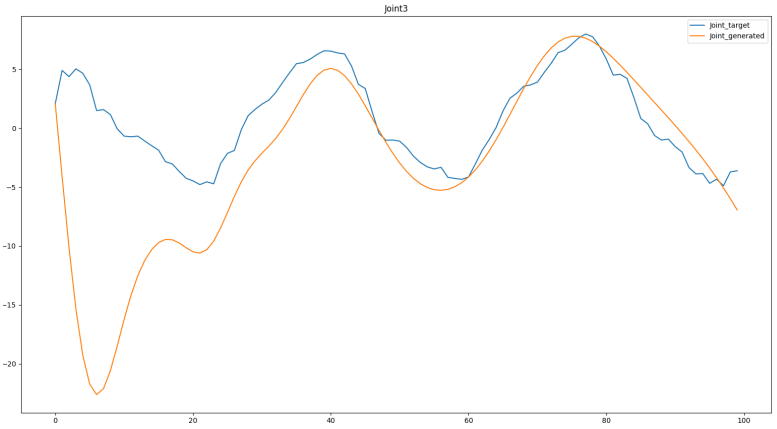


Figure 5.19: Joint 3, basic DMP

This happen as previous stated because the forcing term calculated is too complex to be approximated with a low number of basis functions, Figure 5.20.l

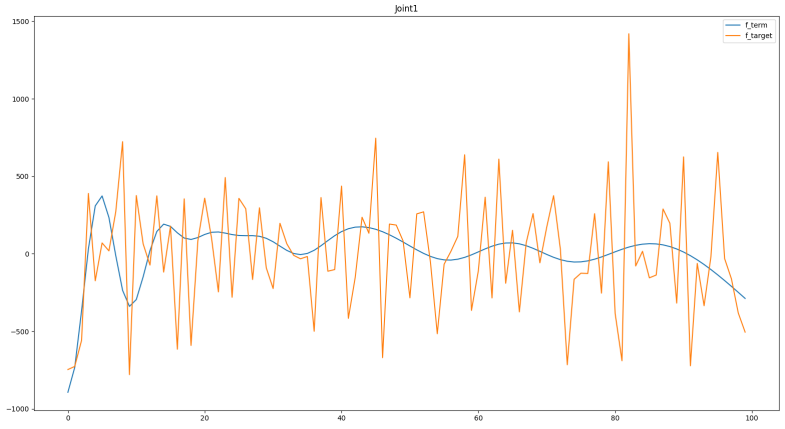


Figure 5.20: Joint 1 forcing term, basic DMP

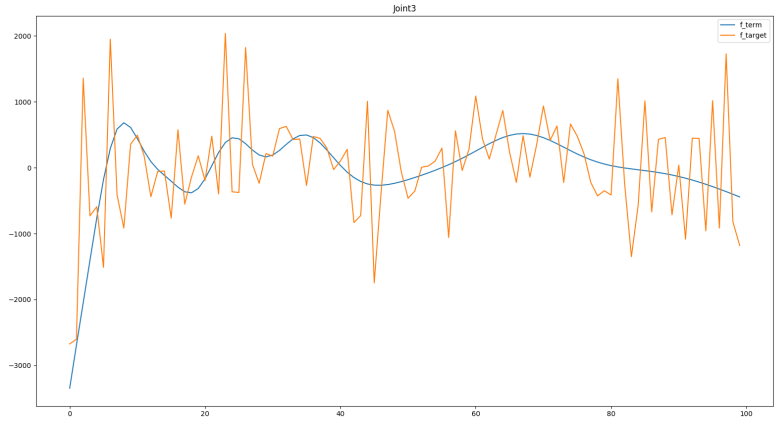


Figure 5.21: Joint 3 forcing term

To match better the function, the number of basis function should be higher than the number of samples and this makes the problem too complex.

The Figure 5.22 is the generated with 50 basis functions.

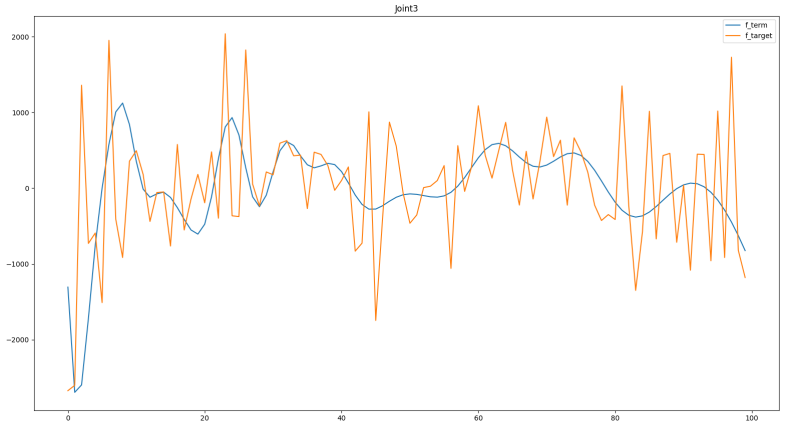


Figure 5.22: Joint 3 forcing term, basic DMP

In conclusion, in order to obtain a correct representation of the movement is required an extremely complex approximation in the DMP. The method proposed AE-DMP will be able to solve this issue.
For the next parts the control group where all the improvements will be compared, is this one:

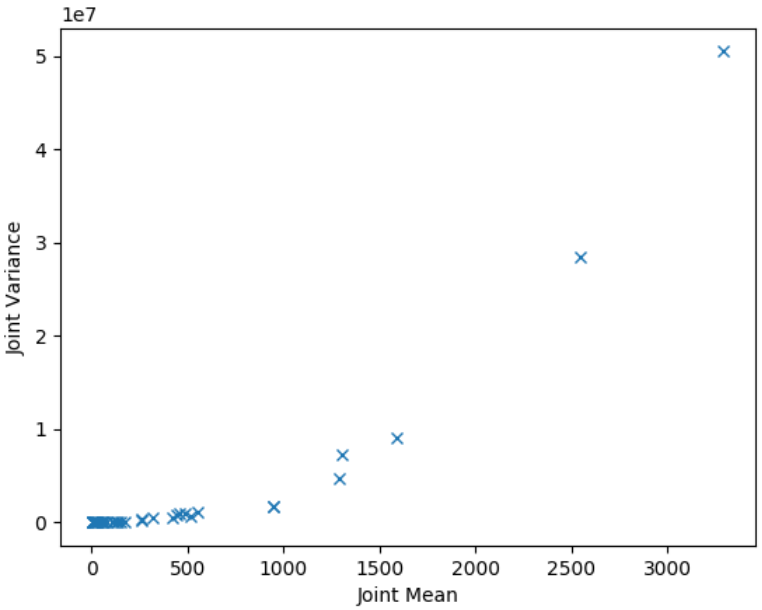


Figure 5.23: Joints MSE plot basic DMP

The MSE will be the main comparison tool due that other like loss or accuracy depends on each structure. Also is easy to compare with the previous test of the AE alone and conclude that the Basic DMP is not even close to a decent performance

5.2.2 AE-DMP NON-INTEGRATED

The next one is the DMP "observing" the latent space but not interfering directly to it. This is wide explained in the previous section 3.3.
3 layers:
The first model tested is the 3 layers structure, and here we can see clearly that with the first results the setup is working as expected. In the latent space y the f_{term} is shaped by the AE to adapt the basis functions easily. The encoder maps the inputs with the correct parameters to facilitate the task to the DMP. It is interesting to remark that the basis function initial size are a big delimiting point in the shape of the latent space. During the test is observed that if the initial vales of the weights are big the latent space will have bigger values, and if the weights are small the latent space will tend to a smoother and smaller solution.

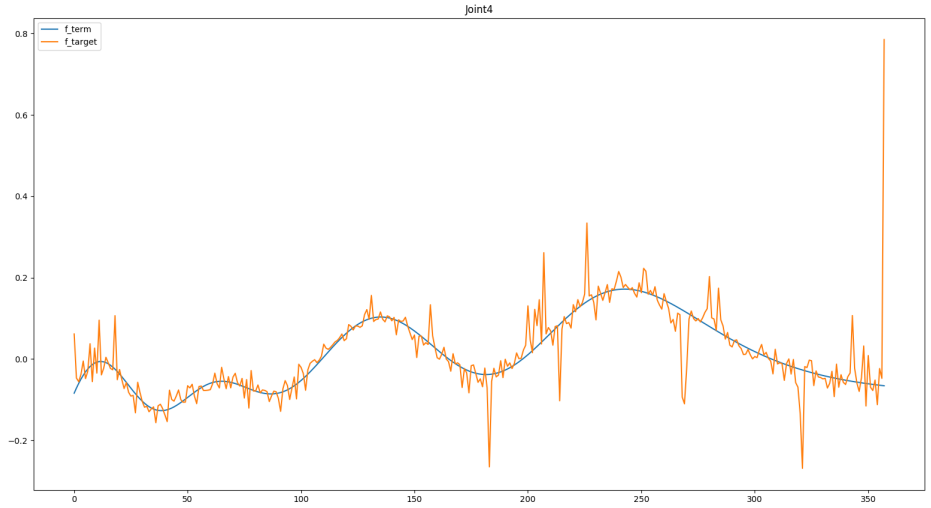


Figure 5.24: Joint 4 forcing term, AE-DMP 3 layers structure

And the effectiveness of this can be observed in the next plot which is the representation of a joint in the latent space.

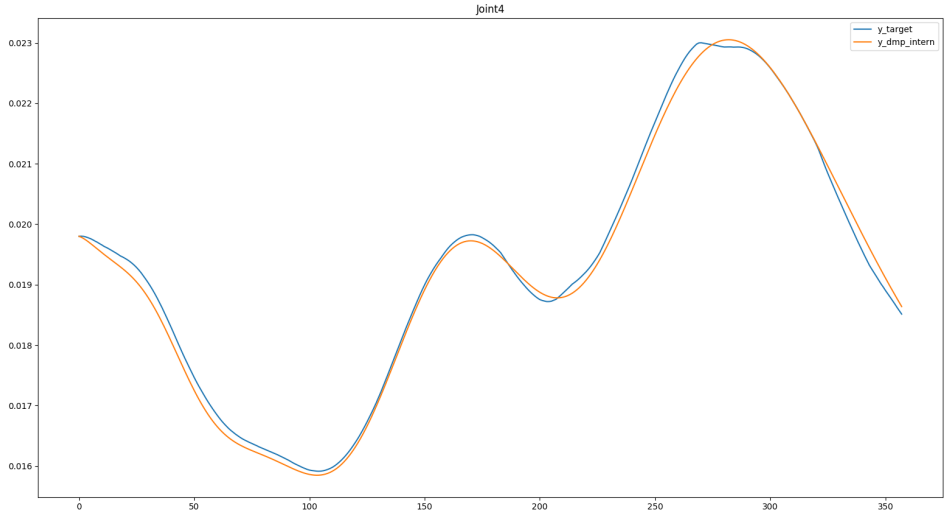


Figure 5.25: Joint 4 latent space, AE-DMP 3 layers structure

After passing thought the decoder of the AE, we take the Joint 1 as a reference for the next comparisons. Is clearly observe as the setup tries to imitate the movement with some difficulties in some parts. This is due the codification and decodification process in the AE. Using the 4 layers structure this is improved.

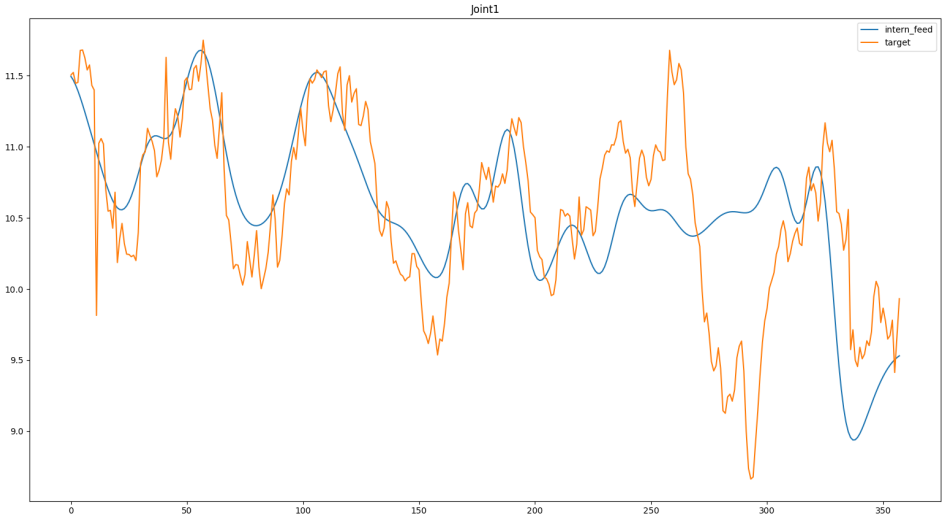


Figure 5.26: Joint 1, joint space, AE-DMP 3 layers structure

Comparing the MSE plot, Figure 5.27, it is clearly seen that a huge improvement happened compared to the Basic DMP version, Figure 5.23. Where the Mean error is reduced near to x7 times, and the variance more than x100 times. This is a lead that we are getting close to the solution.

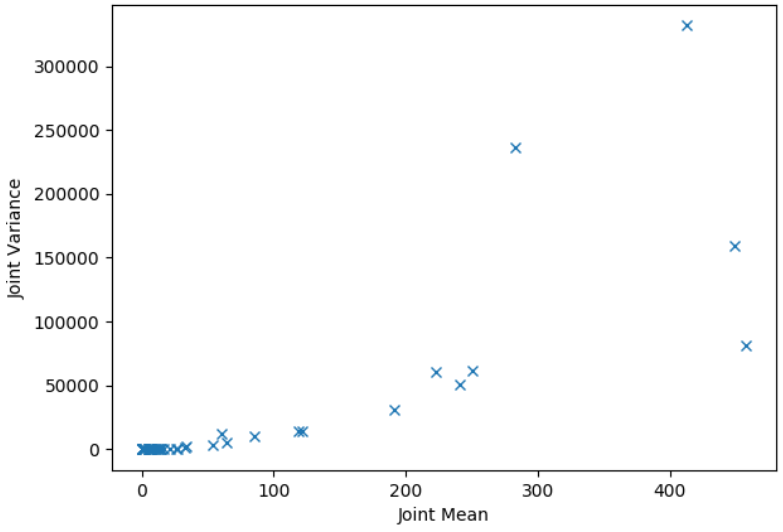


Figure 5.27: MSE plot, AE-DMP 3 layers structure

Now observe what happens with the 3 layers structure if is the signal is resampled to 100 samples and also changing the value in the regulation term, giving more importance to the DMP part.

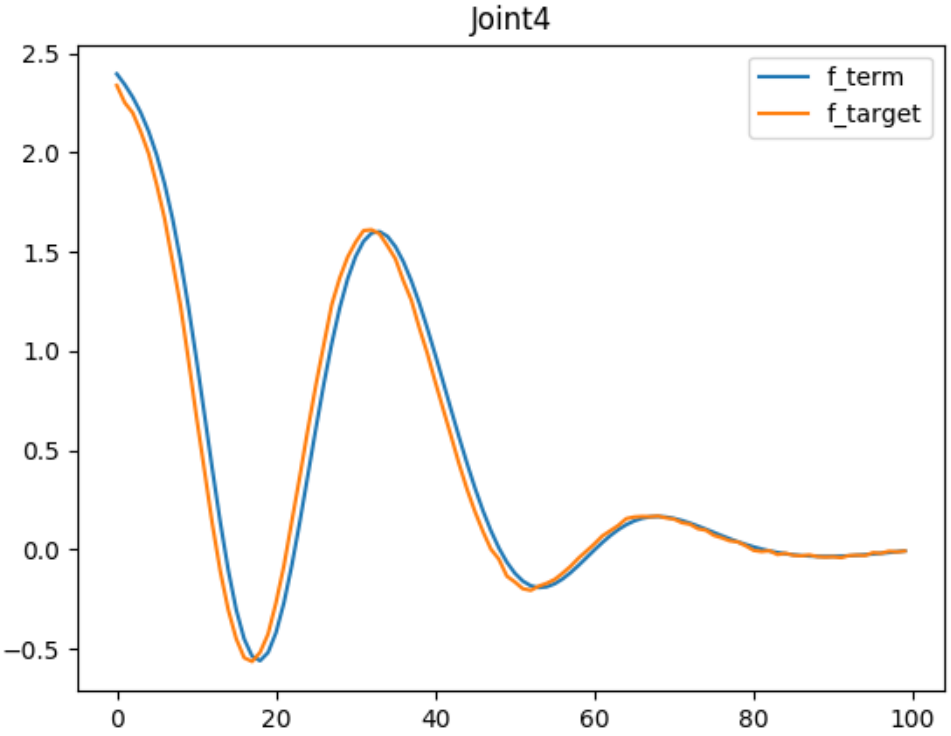


Figure 5.28: Joint 4 forcing term, AE-DMP 3 layers structure with 100 data samples

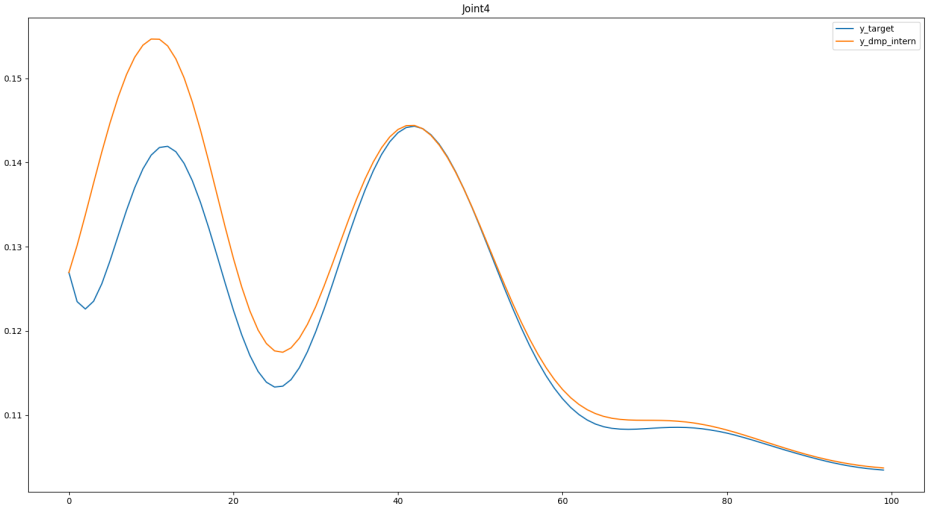


Figure 5.29: Joint 4 latent space, AE-DMP 3 layers structure with 100 data samples

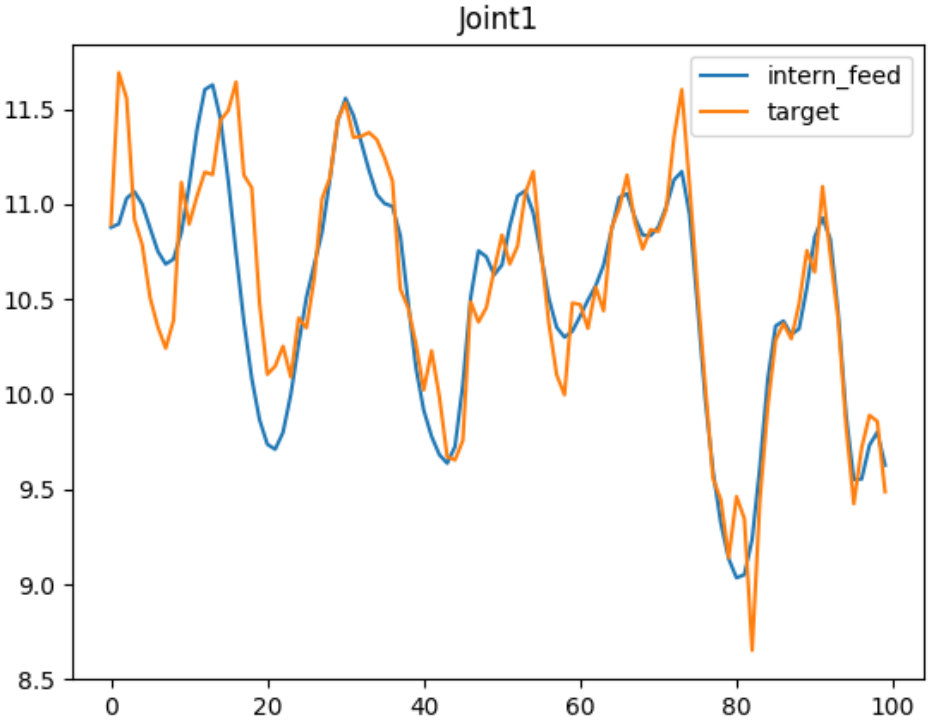


Figure 5.30: Joint 1 joint space, AE-DMP 3 layers structure with 100 data sample

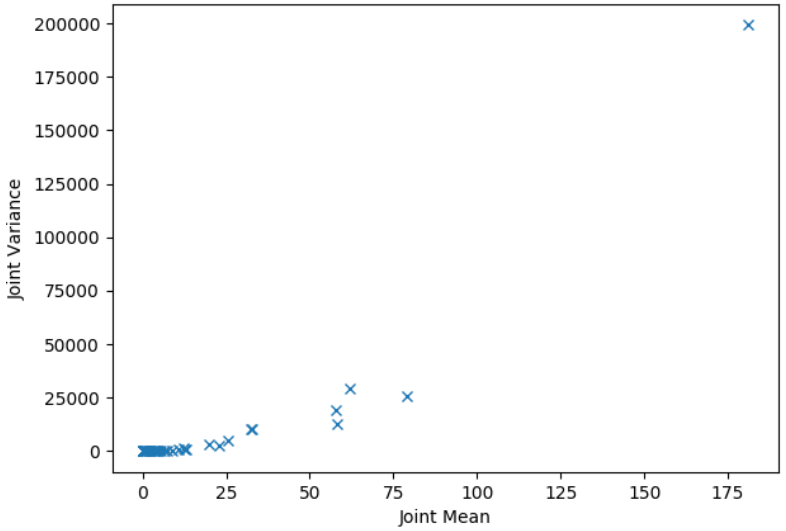


Figure 5.31: MSE plot, AE-DMP 3 layers structure with 100 data samples

4 layers:

The next structure is the 4 layers. In the forcing term and the latent space joints, not so much difference can be seen. But the improvement appears when we look to the decoded output.

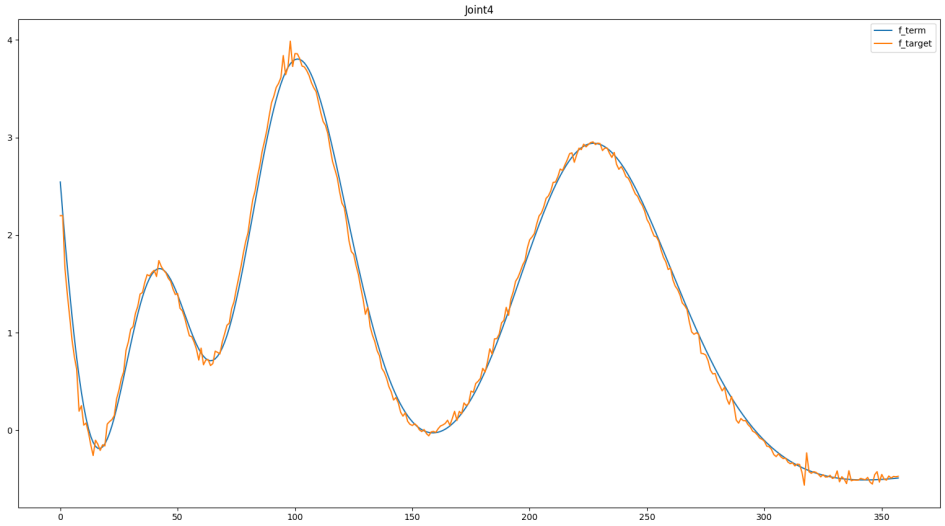


Figure 5.32: Joint 4 forcing term, AE-DMP 4 layers structure

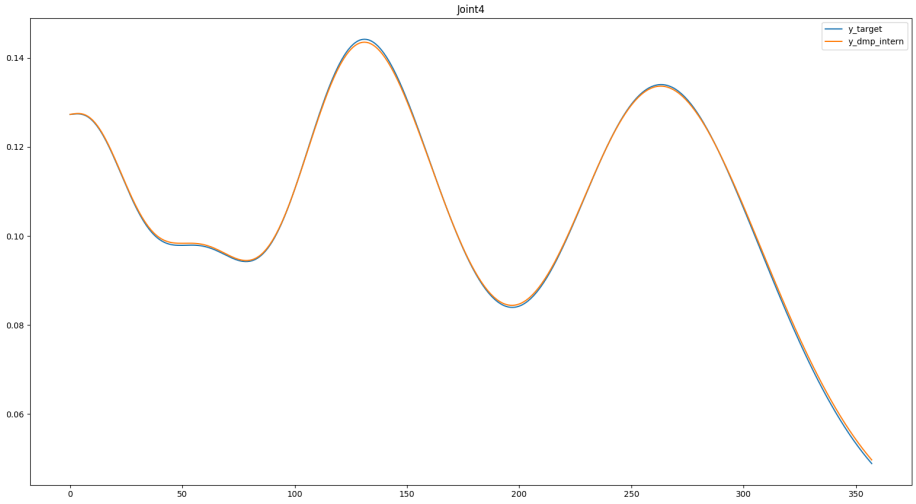


Figure 5.33: Joint 4 latent space, AE-DMP 4 layers structure

Here the system adapts with incredible precision to the target values, Figure 5.34.

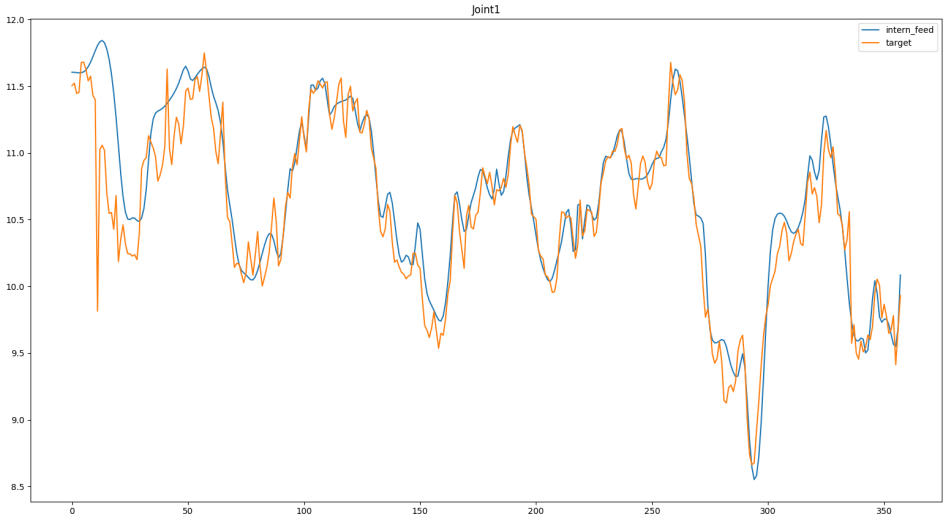


Figure 5.34: Joint 1 joint space, AE-DMP 4 layers structure

The MSE values are much better than the previous tests. The Mean is x2 times smaller and the variance x50 times. With this plot we can see clearly that the 4 layers structure with the same training time is able to adapt better. This is due to the higher possibilities in the configuration space.

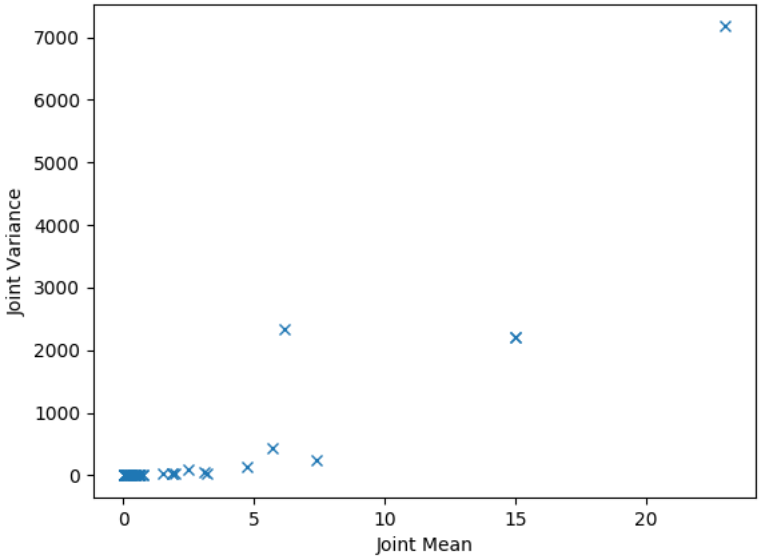


Figure 5.35: MSE plot, AE-DMP 4 layers structure

4 layers structure with corruption:

Looking to the next graphs the results are not good as the previous examples. The introduction of the DAE into the model have a big impact in the results. The forcing term is not able to adapt to the target, this can be induced due to the introduction of the corruption in the inputs.

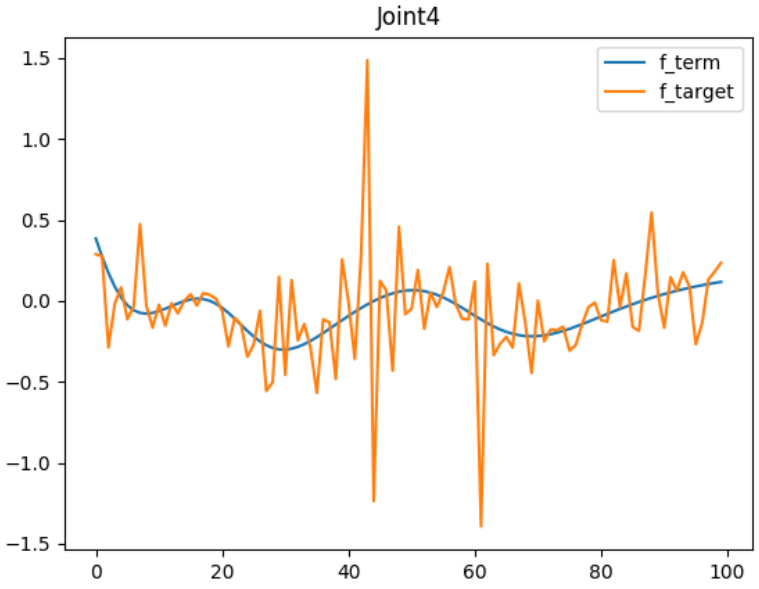


Figure 5.36: Joint 4 forcing term, DAE-DMP 4 layers structure

Is easy to see the error in the latent space, the generated movements is not precise as the previous iterations. This error will be increased after the decodification as shown in the Figure 5.38.

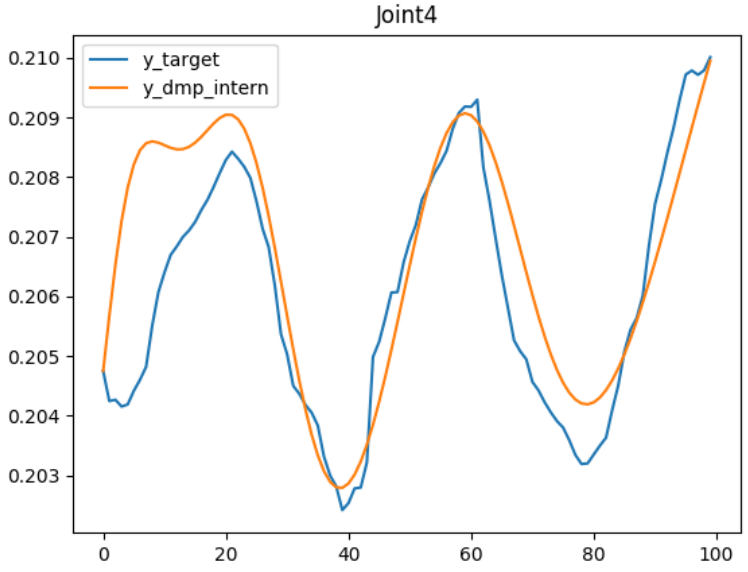


Figure 5.37: Joint 4 latent space, DAE-DMP 4 layers structure

As said the joint is completely different to the target even that the mean is relatively low the variance in the joints shows this error.

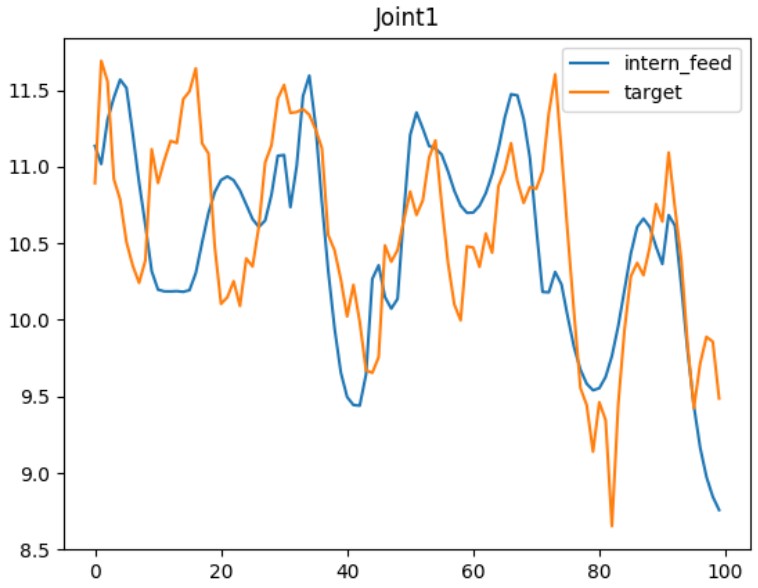


Figure 5.38: Joint 1 joint space, DAE-DMP 4 layers structure

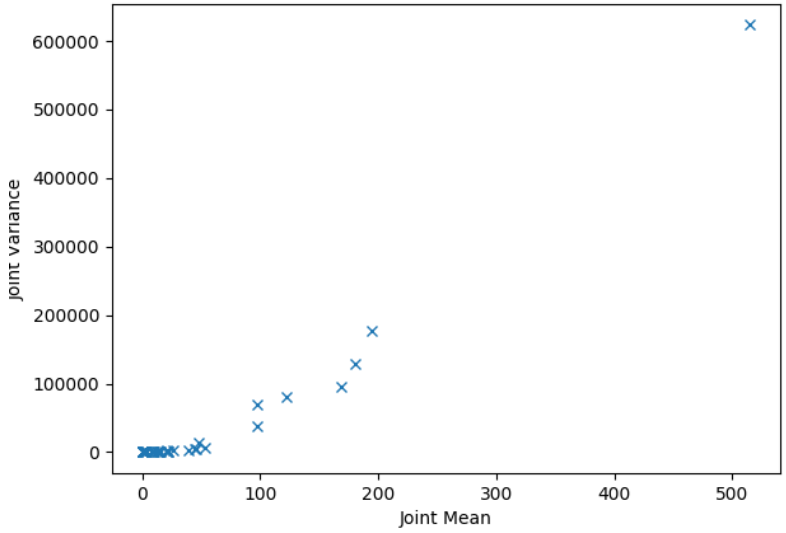


Figure 5.39: MSE plot, DAE-DMP 4 layers structure

Doing a bit of exploration, with this new plot Figure 5.40, where are represented the following signals; the demonstration($target$), the direct feed from the demonstration thought the AE($direct\_feed$) and the generated by the AEDMP($intern\_feed$). The AE is working but the latent space representation is not suitable to be trained by the DMP. The only solution is to tweak the values in the regularization, giving more weight to the DMP part. But then the cohesion for the AE is lost as the DMP takes more importance and the model starts to performs worse. A mew model is required, one which includes the dynamics of the DMP inside the AE.

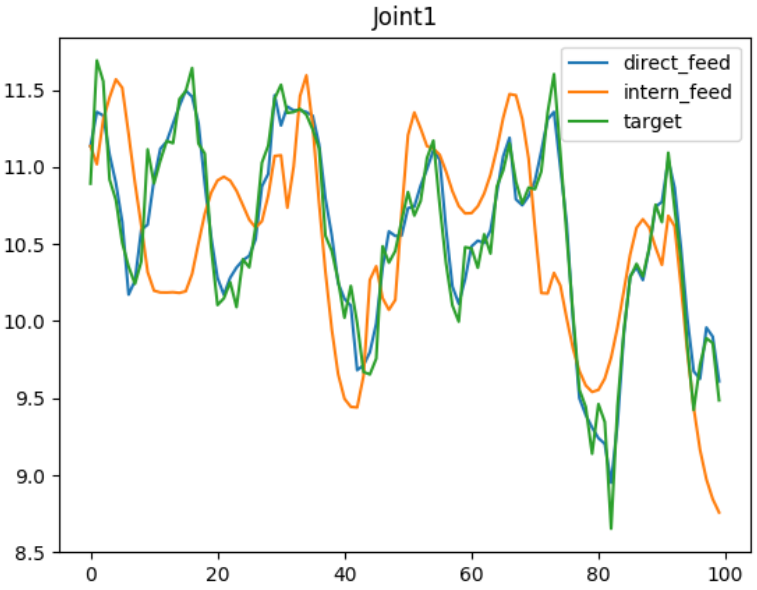


Figure 5.40: Joint 4 joint space, comparison between AE codification and AE-DMP generation

5.2.3 AE-DMP Integrated

This is the final structure where the DMP is totally embedded in the AE. During all the tests the AE structure with 4 layers is chosen. The DMP have only 40 basis functions. In this part the same model will be tested but different parameters, to adjust to the better performance, will be tweaked.

2 Main parameters can be changed in the regularization formula, λ and μ this two values controls the weight of the different parts in the loss function. The λ controls the adaptability of the forcing term. While the μ is used to add sparsity to the latent space variables, this is required if you want to use for more than movement, allowing the interpolation between movements. In the first test we can see the forcing term is almost equal to

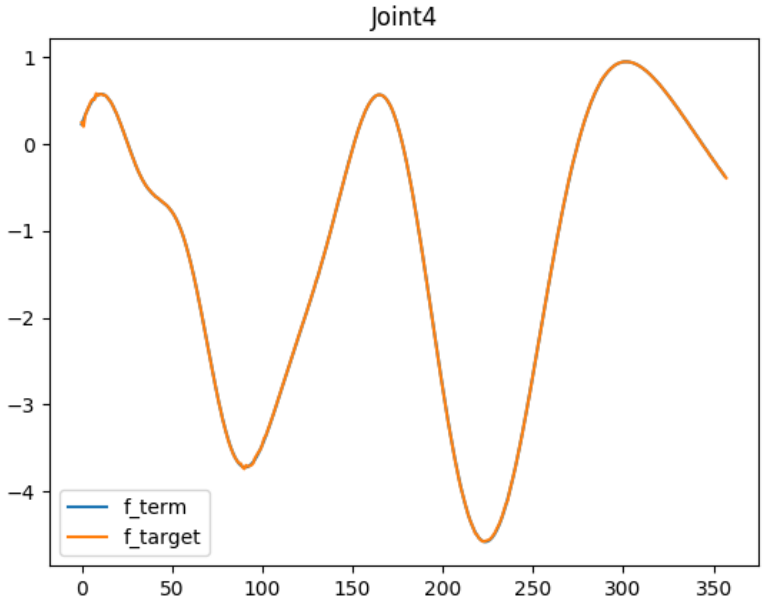


Figure 5.41: Joint 4 forcing term

the forcing terget according our predictions, then the generated latent variables will be the same as the target from the demonstration. This can be seen in the next Figure 5.24.

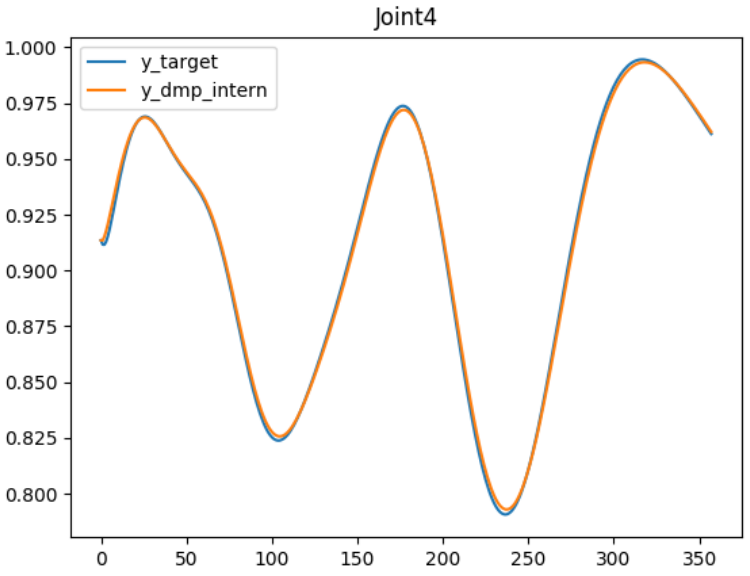


Figure 5.42: Joint 4 latent space

When passing to the joint space as we expect the system behaves almost identical to the demonstration.

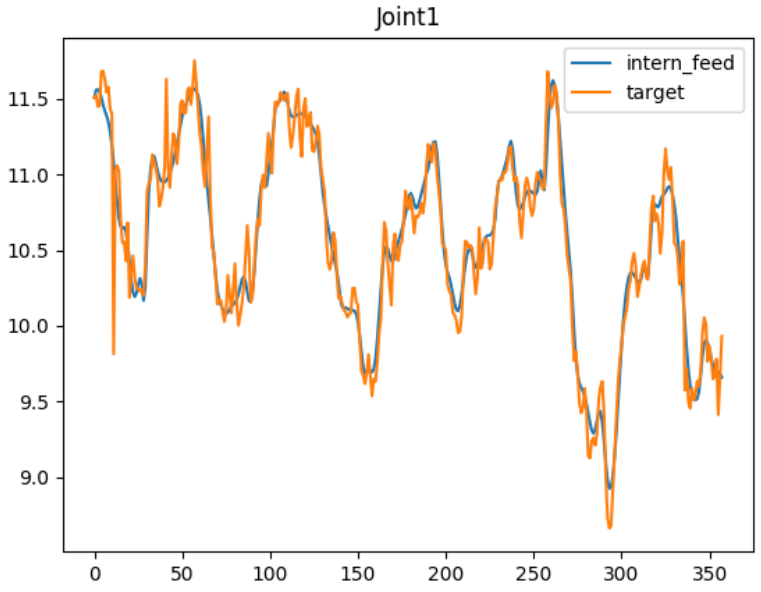


Figure 5.43: Joint 1 joint space

More than the 90% of the joints are under the MSE error of 2. And reducing the variance in a huge gap from the previous examples. This confirms our expectations about the model. Solves all the problems presented by the previous structures and is suitable to be tested with the previous exposed features.

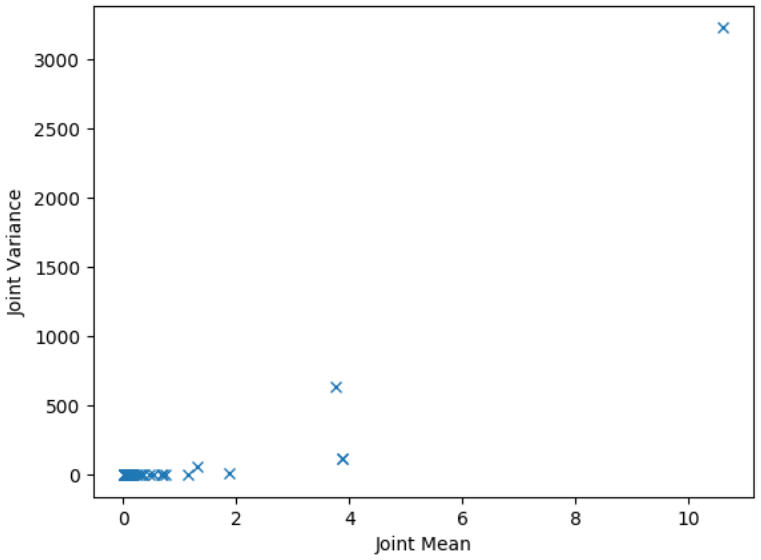


Figure 5.44: MSE plot of the AE-DMP Integrated

5.2.4 DAE-DMP

Now the inputs will be corrupted in order to introduce the ability of reconstructing missing parts.

But firstly lets observe what happen when we introduce the corruption to a previous trained AE-DMP. All the next test will taking as a base the AE-DMP Integrated structure.

Here is the example of different inputs with random corruption in each one, Figure 5.45.

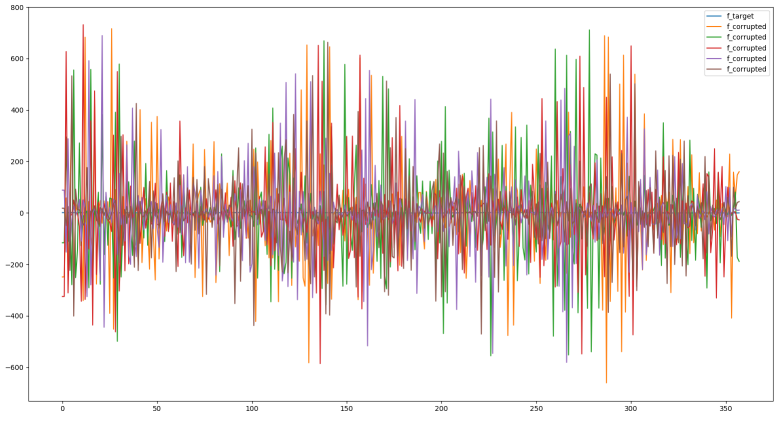


Figure 5.45: AEDMP forcing term

The target force becomes a completely mess and a stable movement is difficult to be learned. Furthermore if the DMP does not work properly then the supposition that when the error in the DMP goes to 0, then the equivalence $y = \tilde{y}$ is not fulfilled and the training is more complicated .

Taking from a starting point the trained version without corruption. The forcing term evolves in this way after a few training steps:

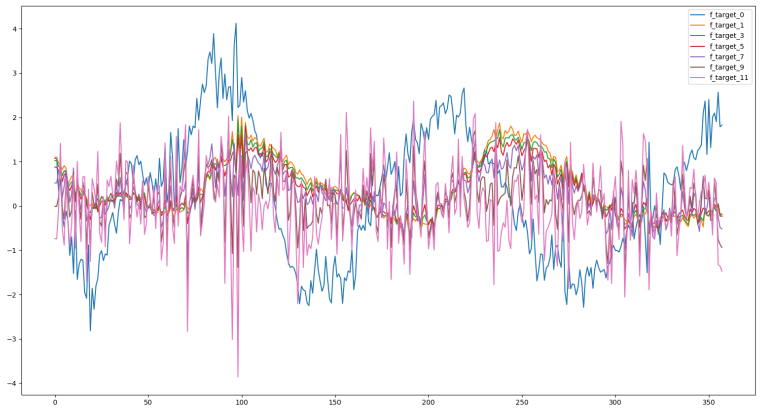


Figure 5.46: AEDMP forcing term training evolution

The forcing term collapses to near to 0, as for the latent space goes near to 0 also. I did not find any reason to explain this phenomenon but in all the test happens the same. The forcing term loses the pattern learned and becomes a mess, this requires more steps to stabilize and find another solution. After close to 10 million training steps we achieve this:

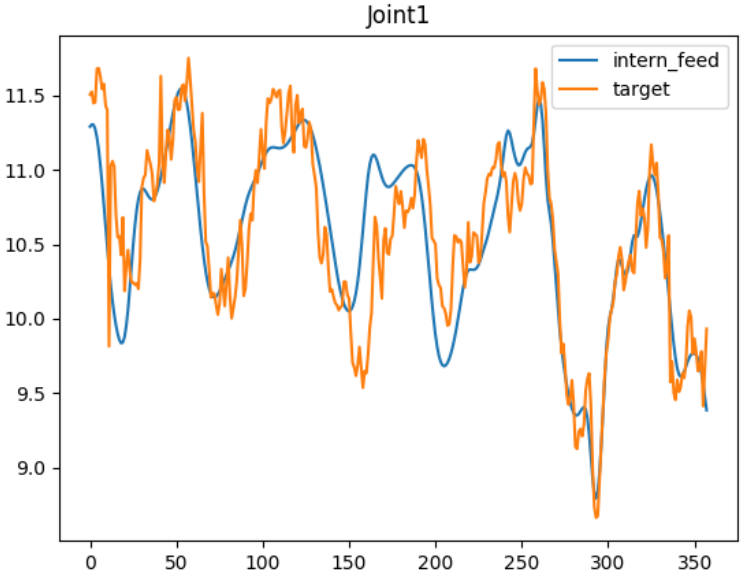


Figure 5.47: Joint 1 joint space

It's a good improvement compared in the corrupted version of the previous structures. The underling pattern in the movement is obtained. Even can be considered as a filtered version of the input data.

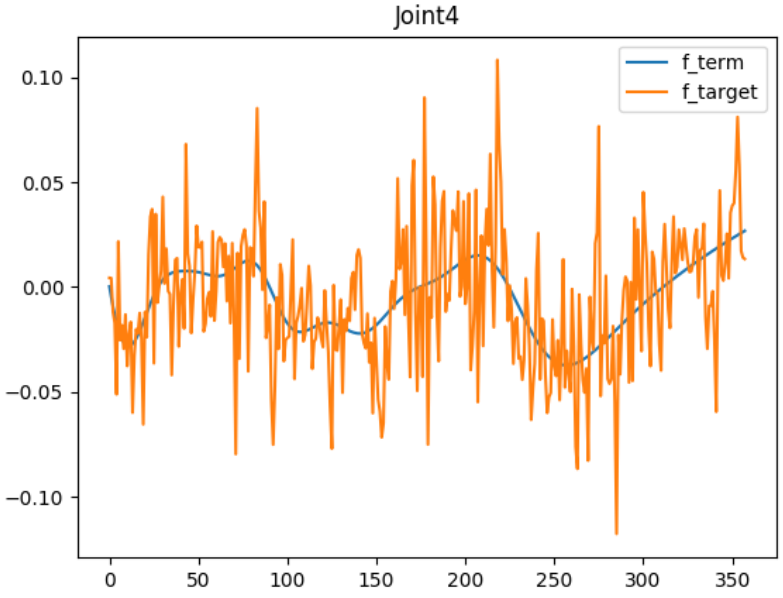


Figure 5.48: Joint 4 forcing term

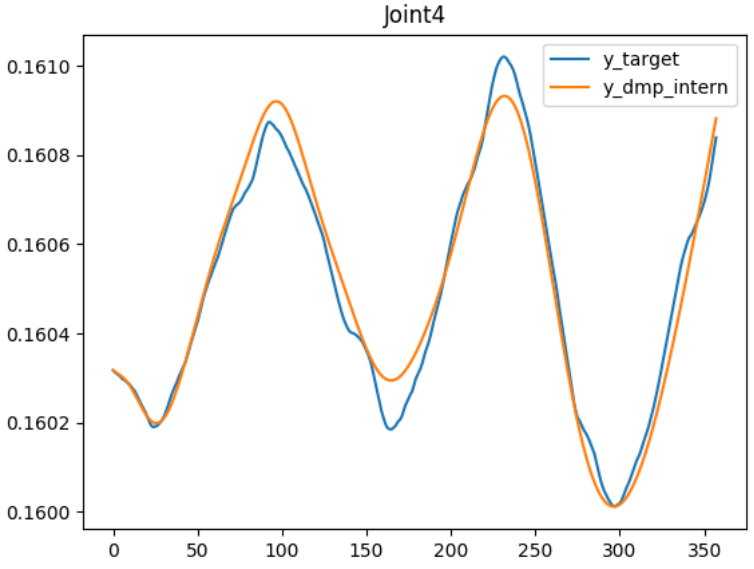
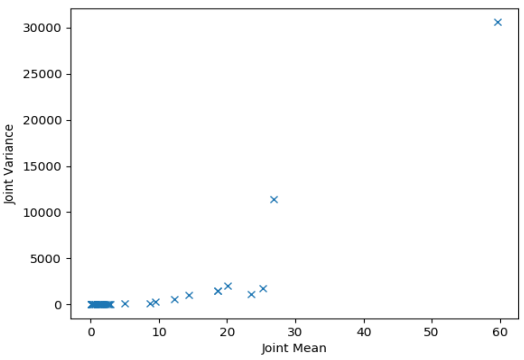
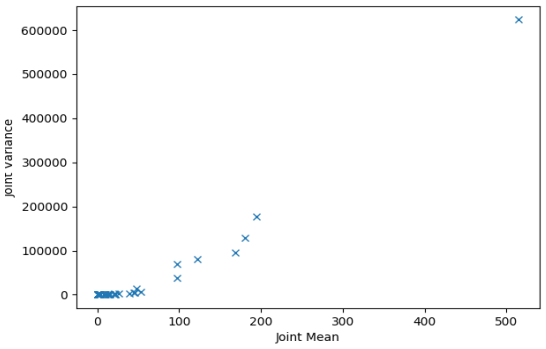


Figure 5.49: Joint 4 latent space

The forcing term and the evolution for the latent space is also a bit noisy but the performance is again much better than the previous structures.



(a) MSE plot of the DAE-DMP Integrated



(b) MSE plot of the DAE-DMP Non-Integrated

Figure 5.50: MSE plot comparison of the different corruption effects in the structures

In the previous picture we have compared the MSE from the DAE-DMP integrated 5.50a and the DEA-DMP not integrated 5.50b, as I have advanced the new implementation gives a much better performance not close to the one without corruption but in this case the structure is able to perform as expected. Giving us the option to implement the reconstruction of missing joints.

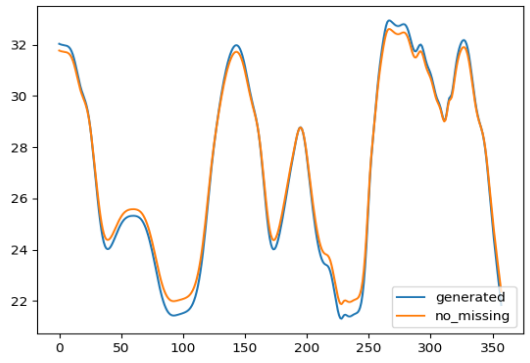
5.2.5 Missing joint reconstruction

As previous said the objective of the DAE-DMP is the possibility of recreating the movement from missing, or corrupted, joints or even full body sections.

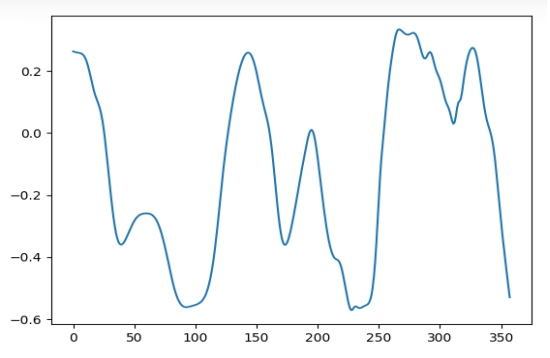
This test are made introducing a 0 in the initial state of the movement in the joints that we want to corrupt. To show the performance in the graphs are shown the joint movement when there is no joint missing, and the one generated after deleting the joint/s. Also the error between both graphs is shown to help in some case due that in some the error is not observable compared to the scale.

Elbow joint missing:

We make $x_{22} = 0$, elbow joint index, and execute the algorithm:



(a) Joint position

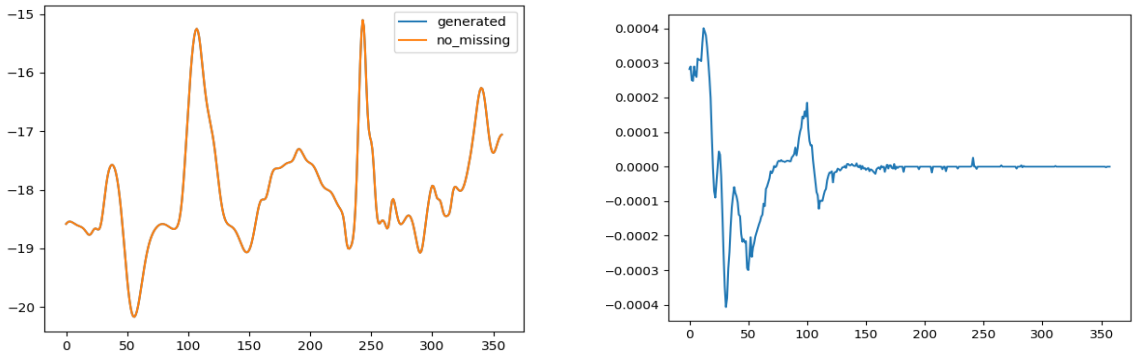


(b) Joint errror

Figure 5.51: Example of the missing joint 22; right radius

In this case the elbow joint is missing, as we can observe the algorithm fulfills its task of compensating this error.

We can observe the effect of this joint in the neighbour joints, for example the hand.



(a) Joint position

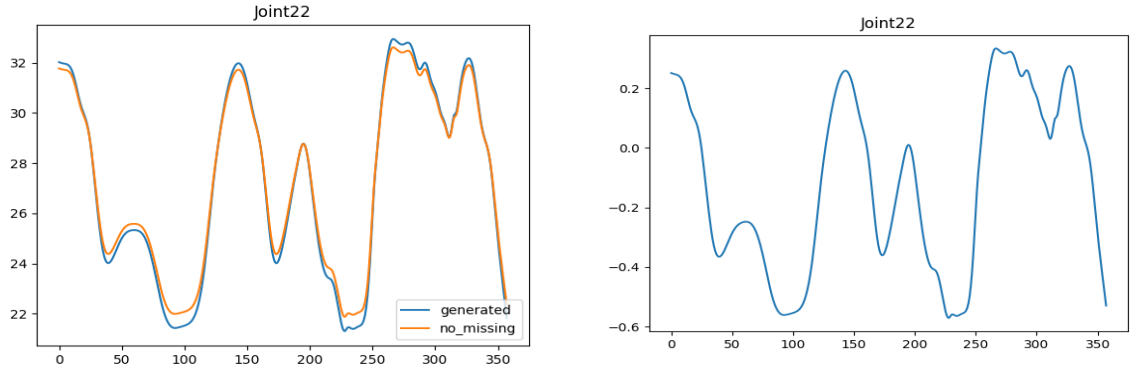
(b) Joint error

Figure 5.52: Example of the missing joint 22; right hand

The algorithm is completing the missing one without notable interferences in the subsequent joints in the arm.

Forearm missing:

In this case all the forearm joints are missing, this includes; elbow, wrist, 2 joints in the hand and 2 in the thumbs. A total of 6 joints.



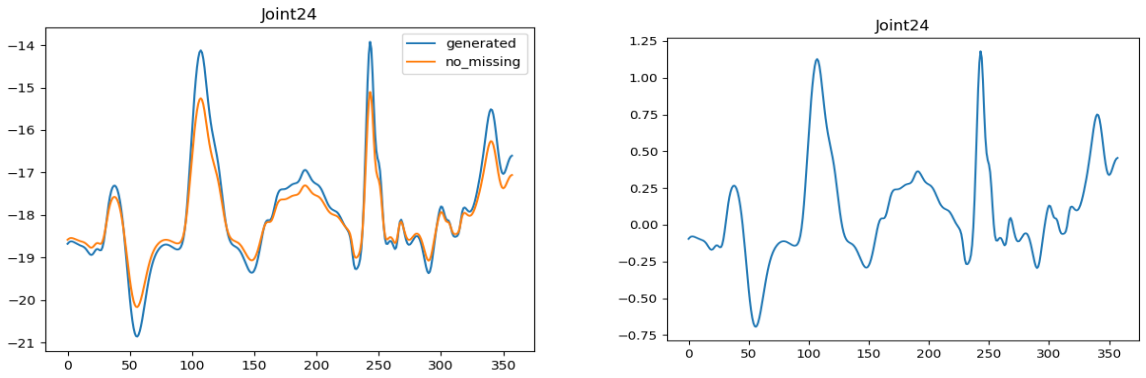
(a) Joint position

(b) Joint errror

Figure 5.53: Example of the missing section right forearm; right radius

Is observable that in the radius is almost identical to the previous.

In the hand joint which is the one that have the most error in this example,



(a) Joint position

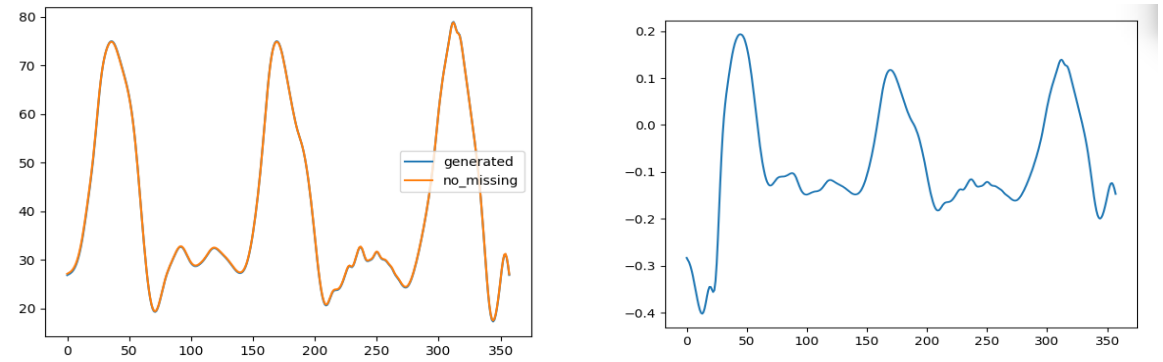
(b) Joint error

Figure 5.54: Example of the missing section right forearm; right hand

And more instructing is to compare this with the same joint when only the elbow is missing, Figure 5.52. Where is observable that this missing section is interfering in the generation. Even losing all the initial information of the arm, the model is able to reconstruct it without major information loss.

Knee joint missing:

In this case we are studying the effect in the right leg.



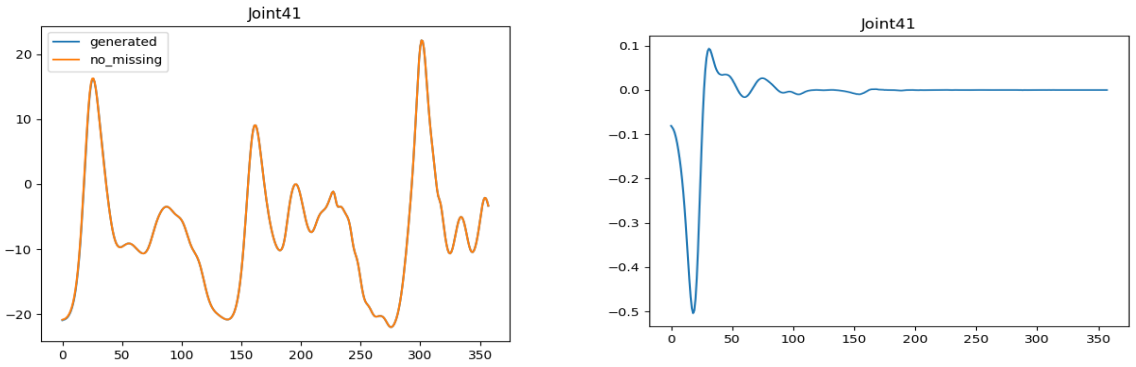
(a) Joint position

(b) Joint error

Figure 5.55: Example of the missing joint 40; right tibia

The error is not comparable to the scale of the movement representing only a 1%.

But in this case the elimination of the joint have a higher impact into the movement of the next joint, the foot. In the Figure 5.56 we can observe a more prominent interference form the missing joint but as evolves in the time this error disappears.



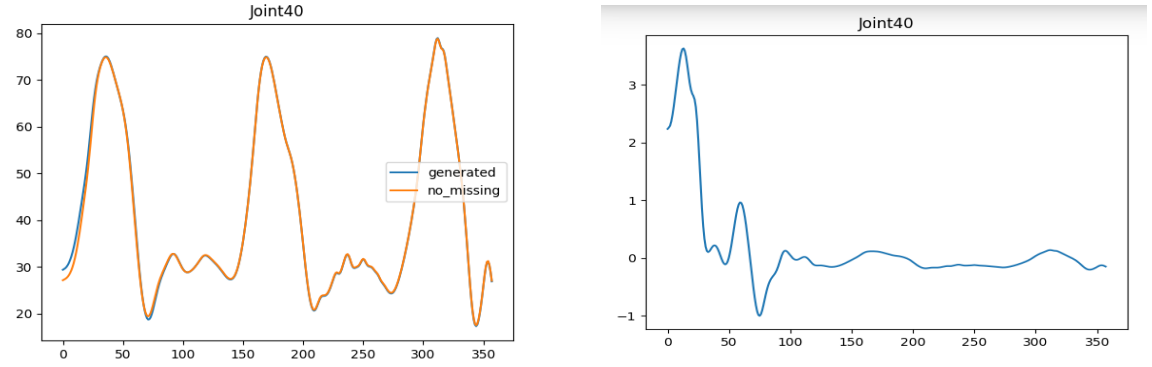
(a) Joint position

(b) Joint error

Figure 5.56: Example of the missing joint 40; right foot

Right foreleg missing:

In this case the right foreleg is missing form the knee including 2 joints in the foot and the toes joint.

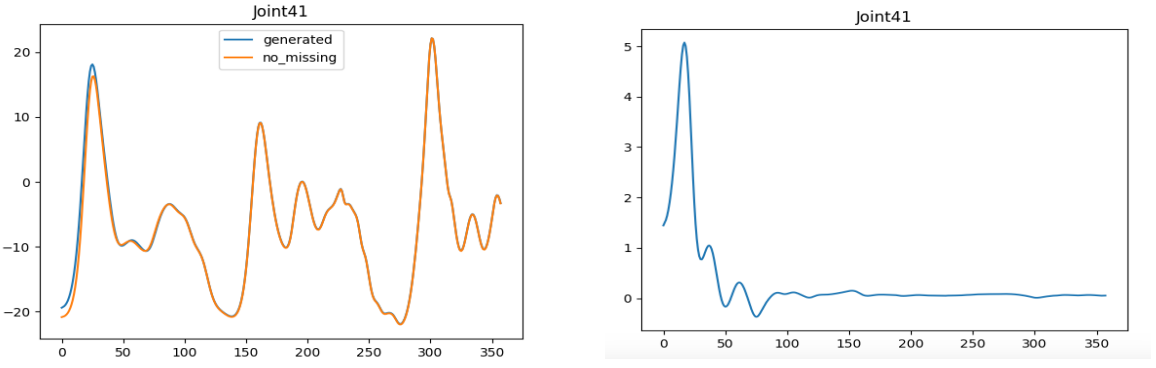


(a) Joint position

(b) Joint error

Figure 5.57: Example of the missing section right leg; right tibia

Here the error is bigger than in the arm but the difference is that the DMP is able to stabilize the movement and tends to a 0 error.



(a) Joint position

(b) Joint error

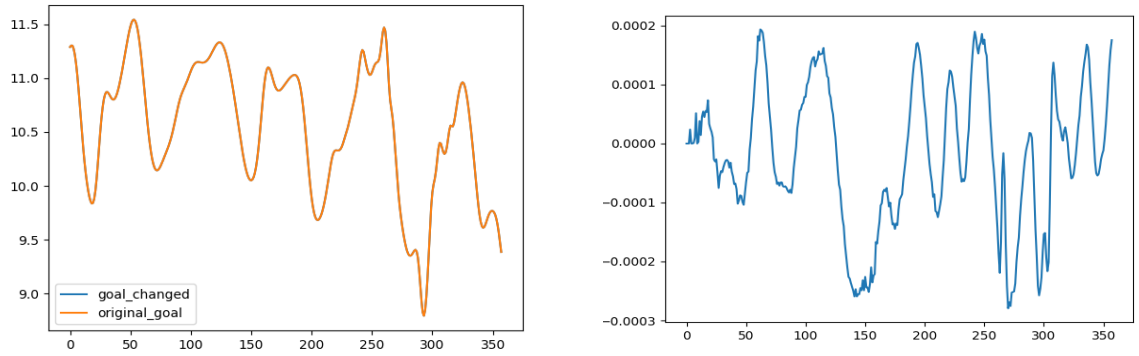
Figure 5.58: Example of the missing section tight leg; right foot

Compared to the previous version, Figure 5.56, the error is more notable, but in the same way with the evolution

of the time this error tends to 0. The results of this features are very successful, as stated the DAE-DMP provides a robust model.

5.2.6 GOAL CHANGING

With this characteristic of the DMP a problem occurs due to the particularities of our system.
The DAE can be understand as an attractor, the DAE learns the underling manifold of the data. And every point that diverges from the demonstration is attracted to this manifold. So any change in the initial point or in the goal point will be diminish and attracted to the demonstration learned. This is can see in the Figure 5.59



(a) Joint position

(b) Joint difference

Figure 5.59: Example of the joint 22 with goal changed; right radius

In this case the goal is changed more than 10 degrees but the movement generated remains almost unchanged. As we can see in the difference between the generated from the original goal and the changed goal.
With this said a possible solution is that if we can not change the goal points in the joint space, they have to be changed in the 5-dimension latent space where the DMP is allocated.

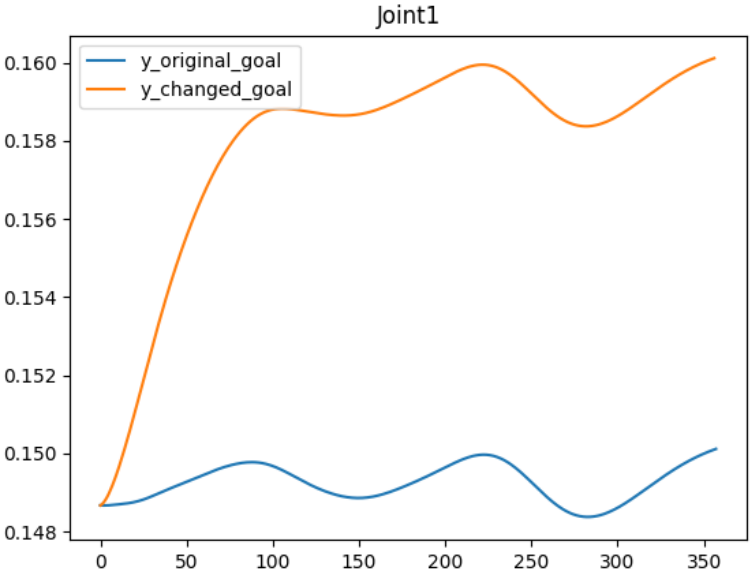
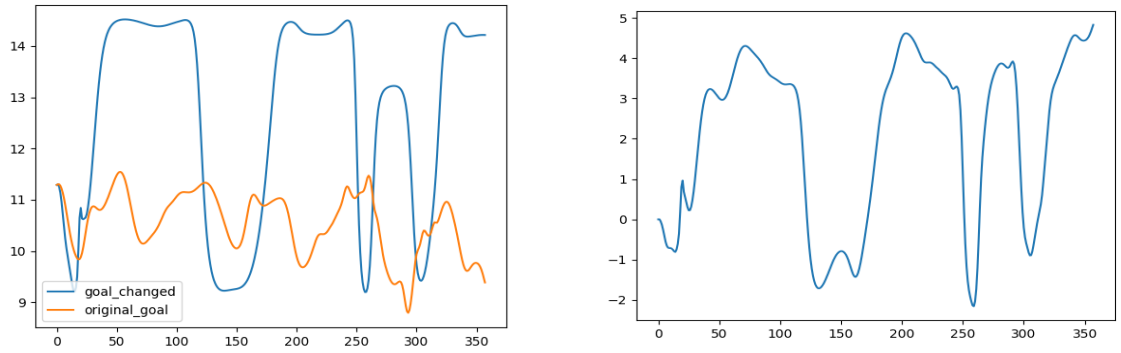


Figure 5.60: Joint 1 latent space goal changed

It behaves as expected, the DMP follows the new trajectory to the new goal, but the new problem relays in what means this new goal after the decodification into the joint space.

If we give a look to one of the joints in the joint space it easy to see in the Figure 5.61b. The movement in the



(a) Joint position

(b) Joint difference

Figure 5.61: Example of the joint 22 with goal changed; right radius

joint space is totally changed and this will be observed in all the joint where the latent space changed dimension have effect. For this reason the DMP ability to adapt to the changing goal points is inapplicable in this system.

Applied the same procedure to the AE-DMP without corruption, the results are similar. Is sensible to changes in the goal in the joint space. But when translated in the DAE as this values are not in the learned data set the results are totally unpredictable.

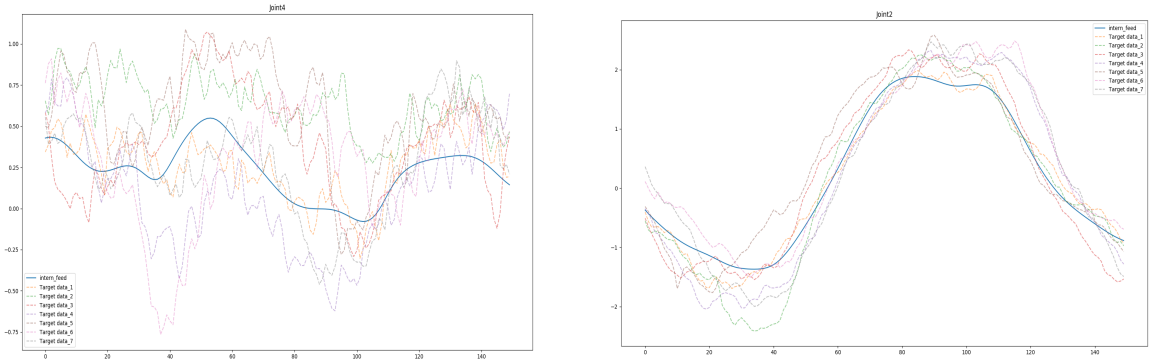
5.2.7 Movement generalization

In this section we will put on test the ability of the system to generalize a solution and generate new movements from different demonstrations learned.

First we will test the generalization of this movements. The walking model is trained with 8 different demonstrations. The data is reduced to 150 samples, for both test walking and running. In order to facilitate the learning task some demonstrations are shifted a few frames in this way the movements are more coherent during all the time. Because the objective is to find the underlying pattern of the movement.

Walking:

In the photos we can see that the model is working as intended, providing a movement more generalized. In joints where the movement follows a observable pattern, the model is able to learn this pattern and reconstruct it, Figure 5.62b. While in other joints when the movement is more erratic and no defined pattern is seen is able to stay in the more consistent path, Figure 5.62a.



(a) Joint 4, joint space walking demonstration

(b) Joint 2, joints space walking demonstration

Figure 5.62: Example of two joints, of the generalization for various demonstrations. Dashed lines are the demonstration and the continuous blue line is the generated by the Sparse AE-DMP

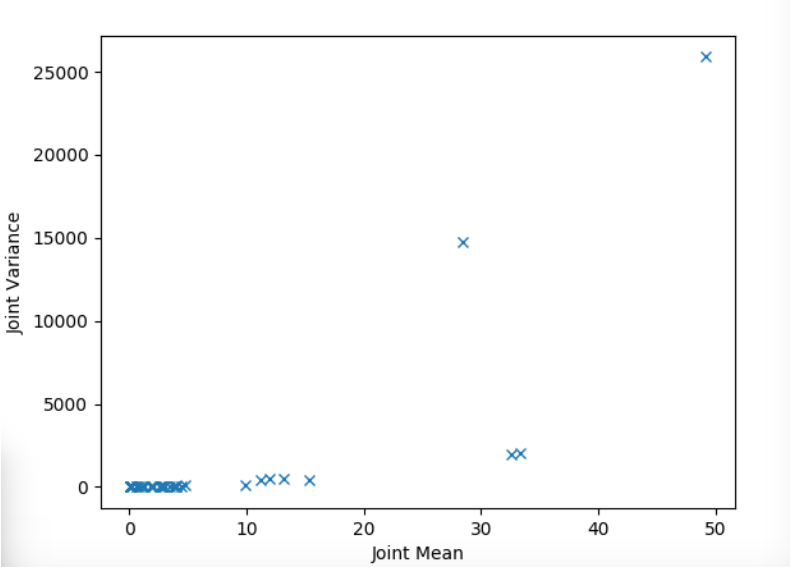
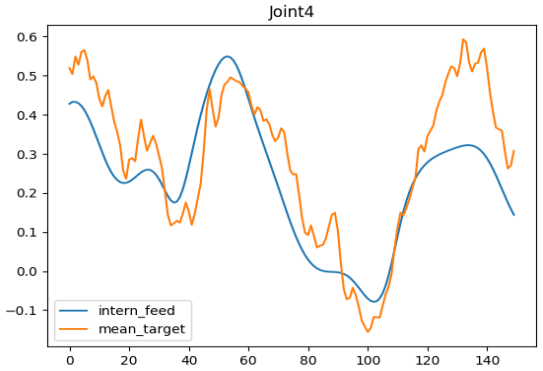


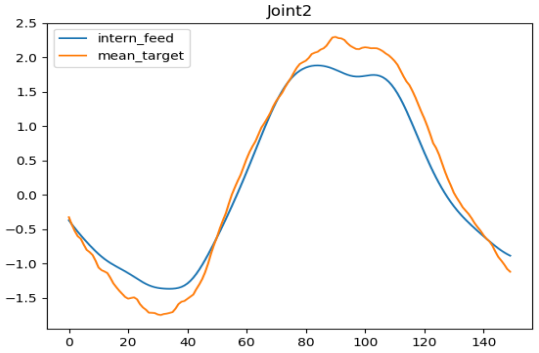
Figure 5.63: MSE plot of the walking movement generated compared with a single demonstration

This generalization will affect the precision compared to a single demonstration. As seen in the Figure 5.63 the error is increased compared to a single demonstration.

To check if the model is working the generated movement will be compare with the mean of all the demonstrations. The generated movement is more close to the pattern represented by the mean of all the demonstrations.



(a) Joint 4, joint space mean of the walking demonstration



(b) Joint 2, joints space mean of the walking demonstration

Figure 5.64: Example of two joints, are compared with the mean obtained in from all the demonstration

We can observe that the movement generated is close to the intern pattern. The MSE plot confirms this, the error is more comparable to the one obtained in the previous results.

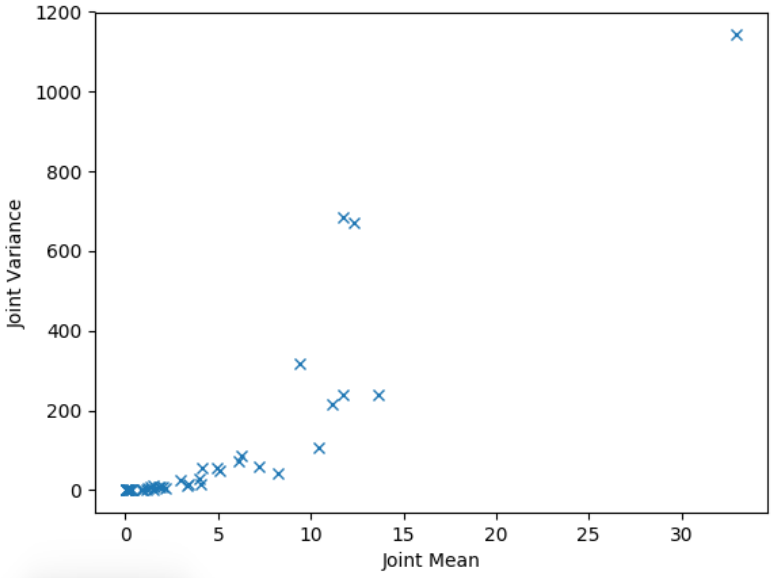
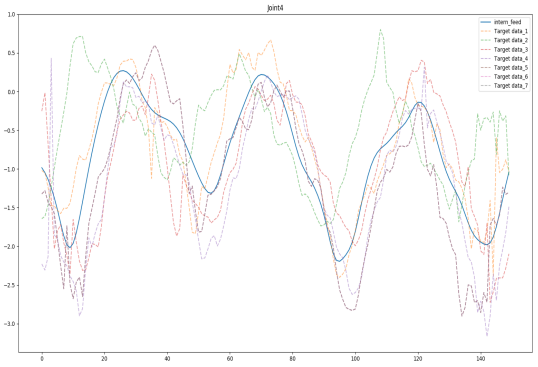


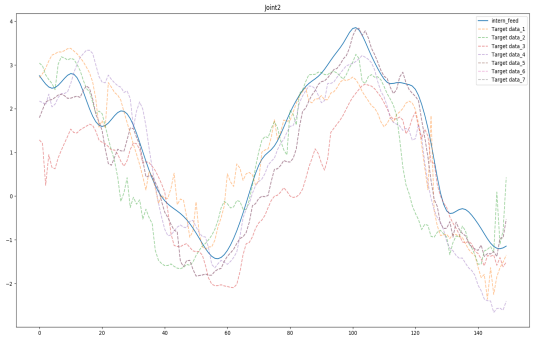
Figure 5.65: MSE plot of the running movement generated compared with the mean of all the demonstrations

Running:

The running movement have a faster pace, representing more movement cycles in the same time frame than the walking movement. This means that more information is concentrated in less space. The model have to be faster enough and have the flexibility to keep this rate.



(a) Joint 4, joint space of the running demonstration



(b) Joint 2, joints space of the running demonstration

Figure 5.66: Example of two joints, are compared with all the taught demonstrations

The model is able to generalize the movement in the same way as in the walking, Figure 5.67. Comparing with the

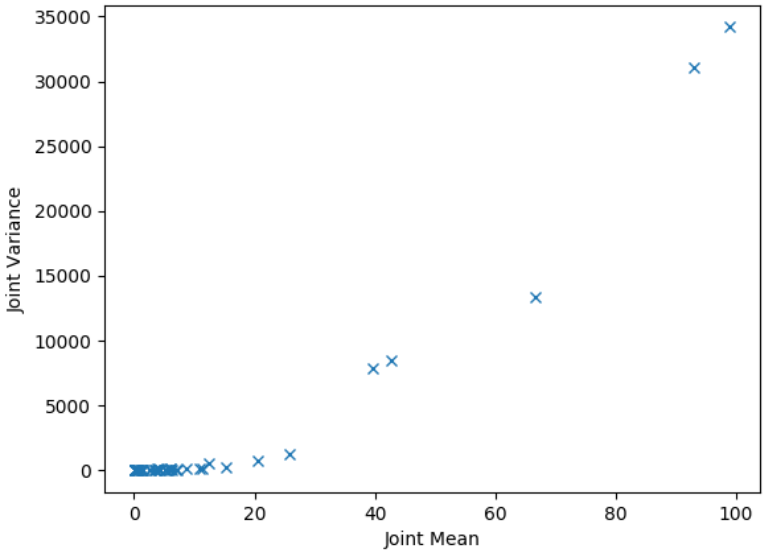
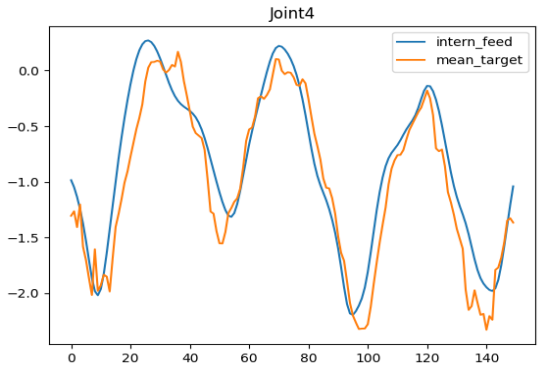
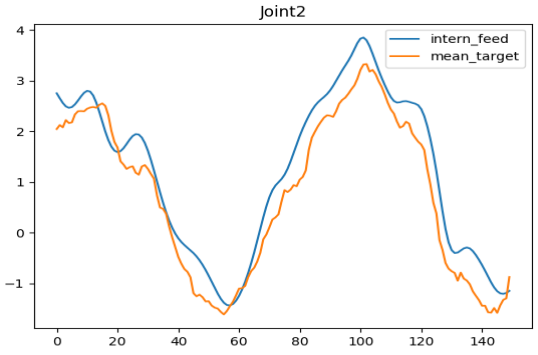


Figure 5.67: MSE plot of the walking movement generated compared with a single demonstration

mean of all the movements is easy to see the adequate performance of the model, Figure 5.68. Both movement



(a) Joint 4, joint space mean of the running demonstration



(b) Joint 2, joints space mean of the running demonstration

Figure 5.68: Example of two joints, are compared with all the taught demonstrations

separately work really well now the trick point is to merge both movements in the same model. Both movements must be trained at the same time and in the same model, training in different models results to problems in the merge.
When the model is trained with different movements at the same time, the Tensorflow model becomes a bit more complicated. Different weights for the walking and running have to be defined, and during the training this weights have to alternate during their corresponding time.

The training have to be modified, and one problem arise with the different data. During the training all the data is normalized to $[-1,1]$ to do this, the mean and the variance is calculated and used. However when different movements are trained at the same time, this normalization process should be the same for both. Otherwise when new movement is generated the decoding process at the last step, the values obtained have no relation to the body position. Due that to pass to the body real joint values the mean and variance have to be added. For new generated movements from both demonstrations the decodification process fails.

5.2.8 Movement interpolation

The model is trained with 2 movements at the same time, walking and running. The AE parameters will be the same for both, but the weights for in the DMP are changed depending of the movement trained in that step. So the model will have 3 kinds of variables, the AE parameters, the DMP weights for the walking and the DMP weights for the running. The canonical time and the basis functions are the same for both movements.

The first test is to observe only the walking codification, so only the walking weights are used.

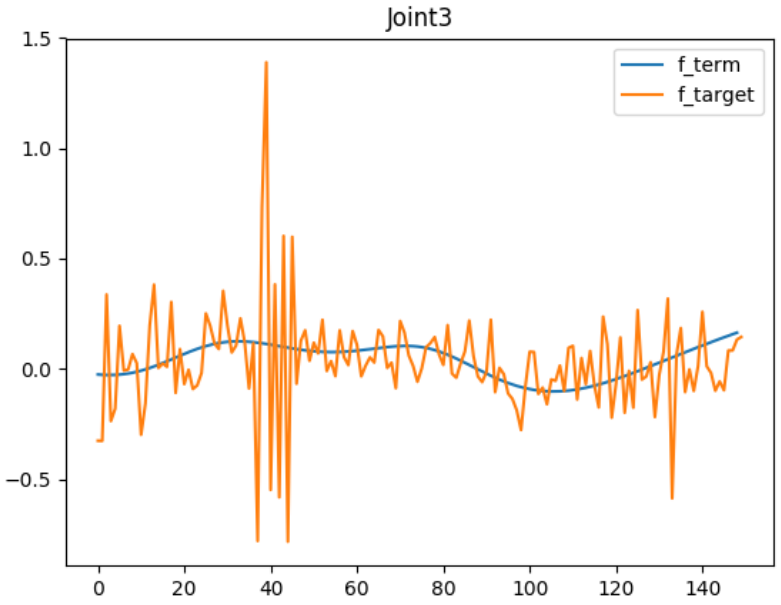


Figure 5.69: Force term of the joint 4 generated with the walking of the merged model

The model tries to follow the objective in the force term. In the Figure 5.69 the forcing term is adapting to the target, and when is used to generate a new movement, Figure 5.70, the model performs inside a relaxed

expectations. In the graph we can observe the "y_target" that represents the walking demonstration in the latent space, the "y_target_run" that is the running demonstration, and finally the "y_dmp_intern" which is the generated by the AE-DMP. The model is able to copse the movement close to the walking.

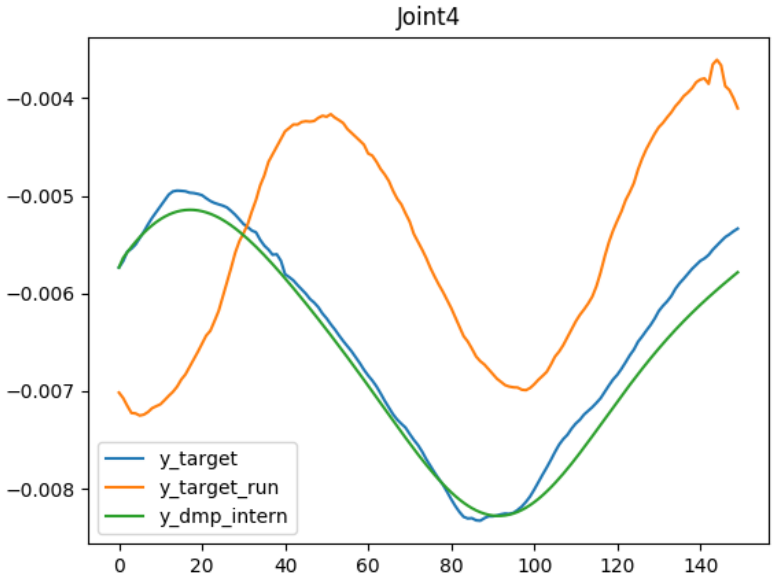


Figure 5.70: Latent space Joint 4 generated from the walking of the merged model

It is interesting to observe the behaviour if we interchange the initial state, position, speed and acceleration for the running initial state. In the Figure 5.71 the new generated movement starts in the latent space position of the running codification but as the time passes the DMP attractor moves the action to the walking goal. This is observed in all the dimensions in the latent space.

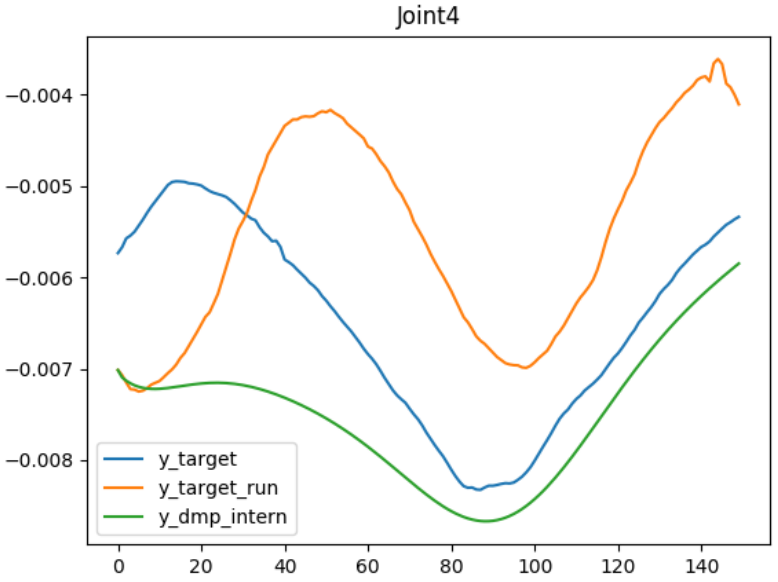


Figure 5.71: Latent space joint 4 generated with the walking of the merged model, while the inital state is the running

Furthermore a problem arise with this structure, when the movement translated to the joint space the values are off. The reason for this problem is the normalization, in order to normalize all the data the mean and the variance are extracted. But when we have 2 sets of different data this values are different and the model tries to generate new movement, during the transformation to the joint space the values are unusual, Figure 5.72.

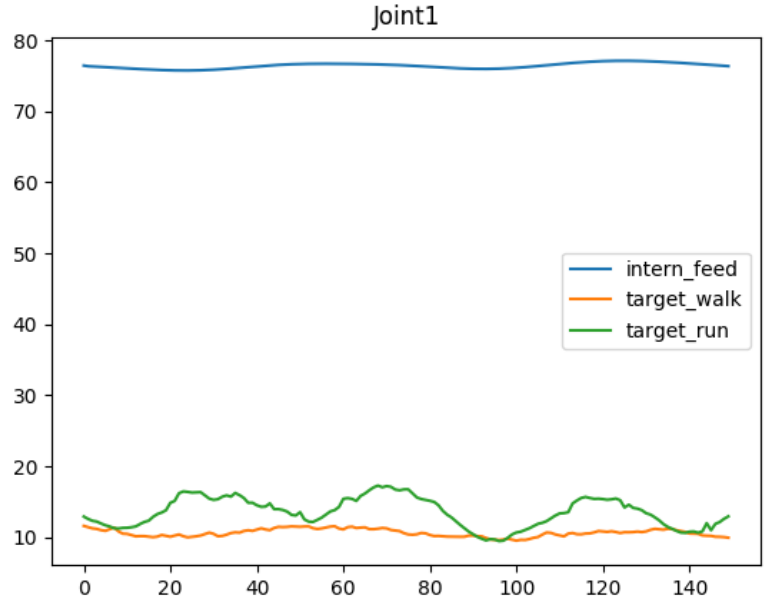


Figure 5.72: Joint space joint 1, processed output.

But if we observe the data before the translation, Figure 5.73, the values are close to the demonstration. As during the training this data is the used for the error calculation the mean and the variance should not affect the solution in the latent space or either the training.

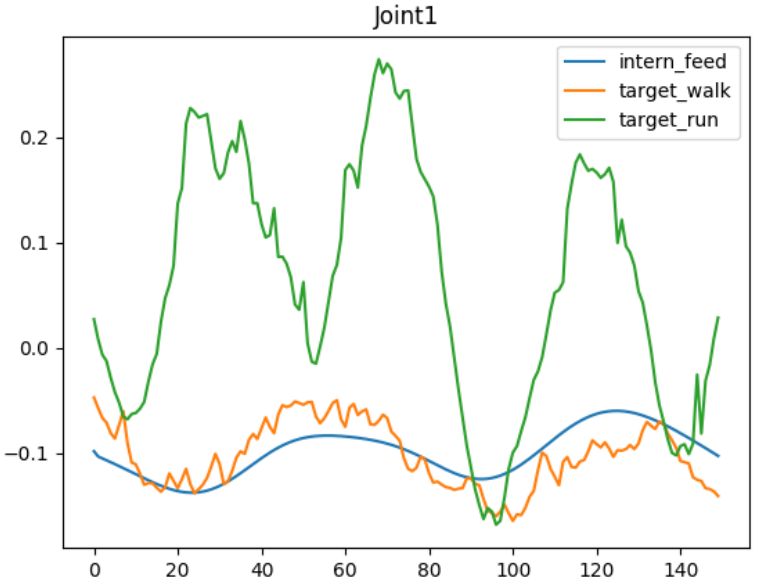


Figure 5.73: Joint space joint 1, non processed output

The running movement is also achieved with considerable exactitude, the model learn the pattern in the target force, Figure 5.74, giving good results in the generation of new movement, Figure 5.75. During individual test running has given the better results, surprisingly being more easy to learn than the walking. Here in the merged model also the running demonstrations perform a bit better than the walking.

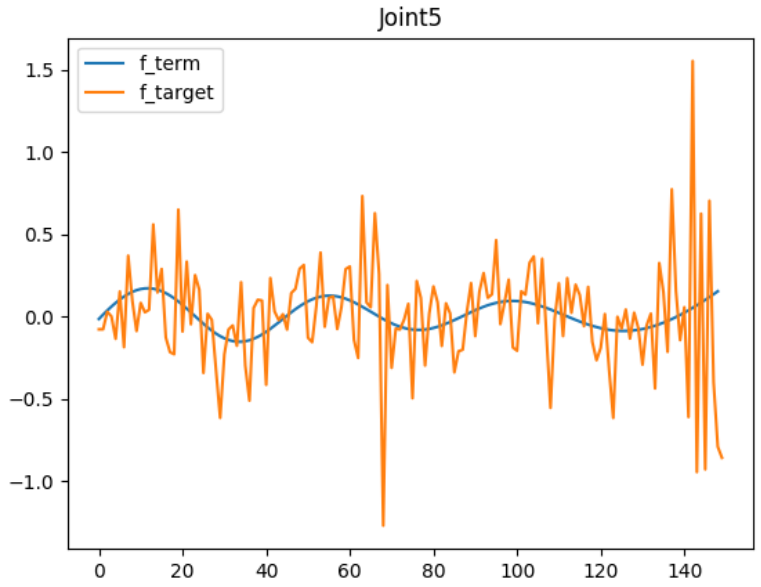


Figure 5.74: Force term of the joint 3 generated with the running of the merged model.

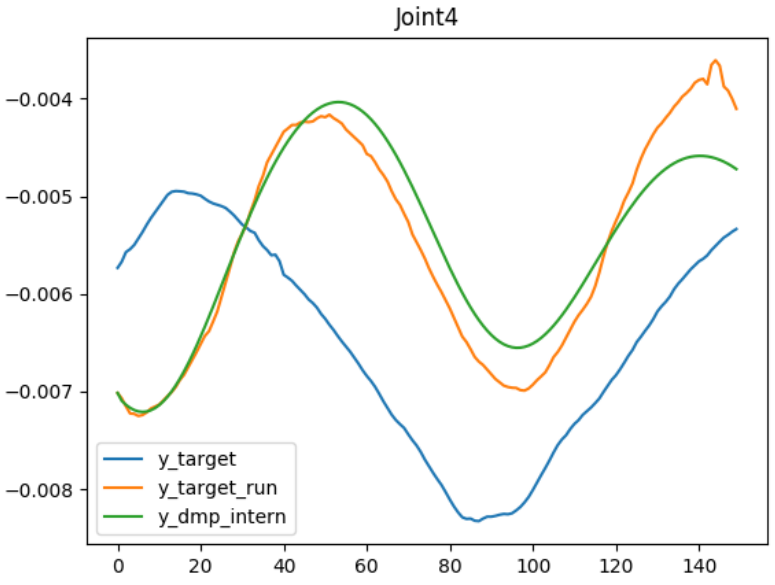


Figure 5.75: Latent space joint 4 generated with the running of the merged model.

If we do the same as done in the walking about changing the initial state but in the reverse way. Now he initial state is the walking and the movement is from the running, Figure 5.76.

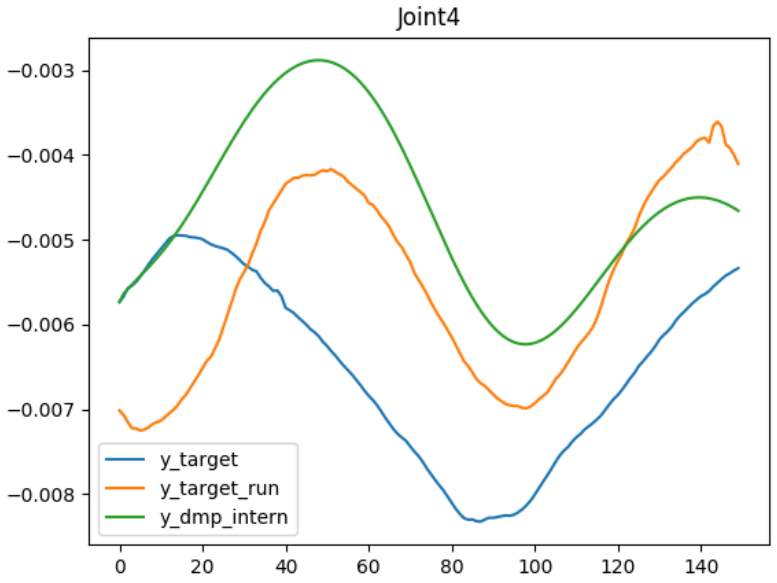


Figure 5.76: Latent space joint 4 generated with the running of the merged model, while the initial state is the walking.

The generated starts in the walking point but follows the movement of running and tries to converge to the running goal.

Movement interpolation:

The next step is the interpolation between the two movements. The interpolation is made with the rules established in the Section 3.3.6. The weights merge is calculated as in the Equation (3.37).

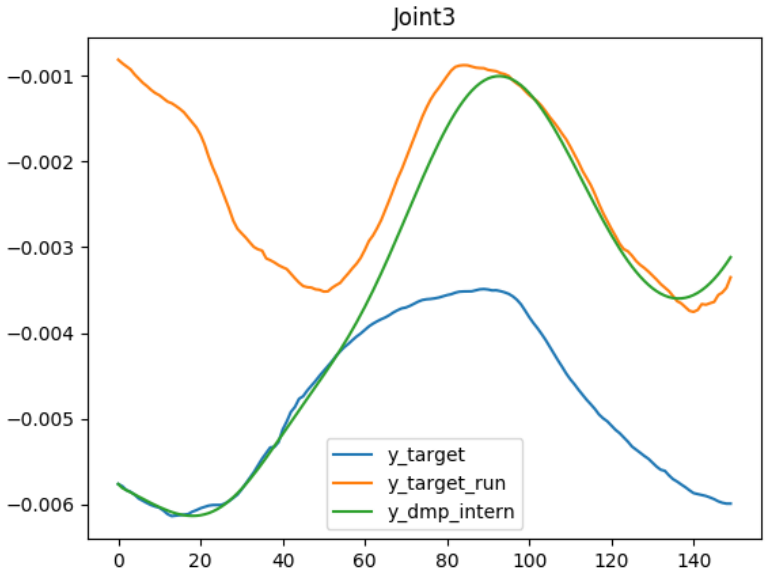


Figure 5.77: Latent space joint 3 generated with the transition between walking and running.

In this case, Figure 5.77, the movement starts to change at the frame 50. It is easy to observe the transition between one movement and the other. While the walking is active, before frame 50 the model follows this path but when the force of the running start to interfere the action shifts to the other movement.

This is also observable in the joints space, Figure 5.78, the generated path changes from one movement to another, with the center point established in the 75 frame. While if we change the center to 25, the model reacts and adapt to the change, Figure 5.79

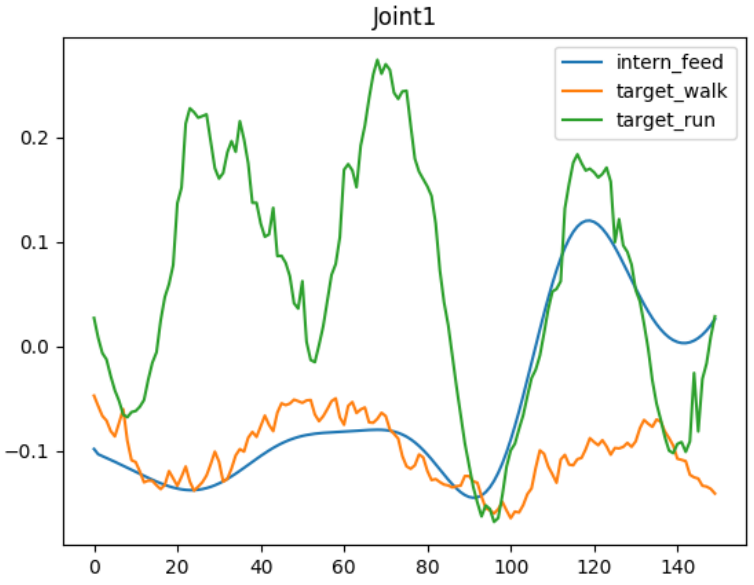


Figure 5.78: Joint space joint 1 generated with the transition between walking and running, centered in the 75 frame.

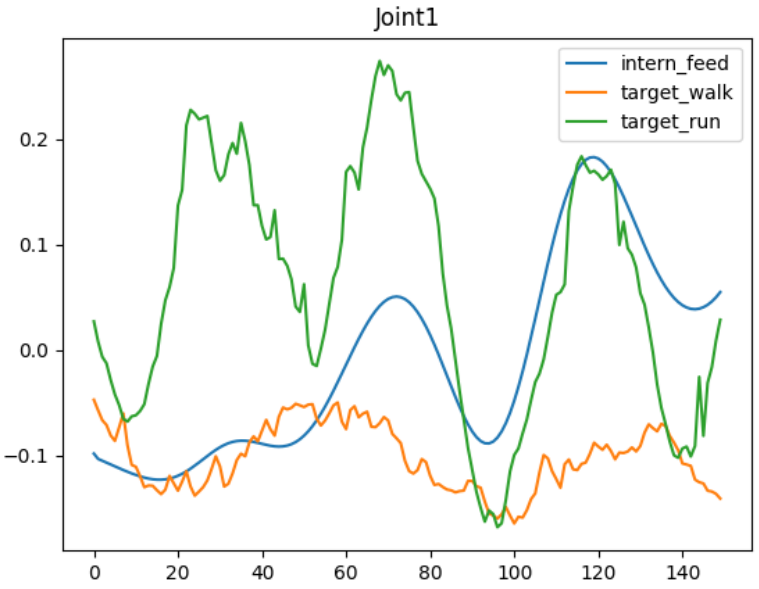


Figure 5.79: Joint space joint 1 generated with the transition between walking and running, centered in the 25 frame.

The model is able to codify both movements but the interpolation or transition between them requires a lot of tuning and parameter adjusting. Although the model reacts perfectly to the changes of this parameters, the sensitivity of them complicates the task. And the unpredictability of the results to this changes creates a big barrier.

CHAPTER 6

CONCLUSIONS

As a conclusion of this thesis I would highlight the following points.

The AE-DMP model presented in [2] is studied and tested. It is compared with the naive implementation of the only DMP applied to all body, Section 3.3.1.1, where the model AE-DMP gives incredible better results. The model performs perfectly in the generation of complex movements, simplifying the body movement to 5 joints providing a great approach. The possibility of modifying this conversion adding more layer or nodes to the AE, or even changing the activation function brings a vast number of options in the transformation, giving the ability to search the best option for each task. This model can be applied to animals, manipulators and mobile robots, with extremely facility. The flexibility and adaptability to different bodies without big changes is a great point for the model.

The AE have passed the test successfully, the system have been analyzed with different structures and interactions with the DMP. The benefits of AE have been proven many times before this thesis, pretraining DNN, classification problems, etc.

The AE-DMP as said preforms much better than the Basic DMP, the movement generated is almost equal to the demonstration and with the adequate training time this difference will be smaller in every training.

Aside the recreation of movements, the DAE-DMP version provides a interesting feature of reconstructing corrupted data. In the event of having any interferences in the information of the body provided or even if it is completely missing, the DAE-DMP has proven, Section 5.2.5, to solve this and reconstruct and generate the movement without major interferences. The DAE-DMP provides a robust model to "control" the body.

Other features as goal changing do not give the expected results. The structure of the AE minimize any intention of modifying the movement and act directly to the latent space if the movement is not trained correctly can lead to unexpected consequences.

The model is able to codify different movements in the latent space as shown in, Section 3.3.5, with the differentiating features. And during the generation of new movement provides good results. the model adapts to the transformation of the main parameters, but a lot of tuning is required to obtain the desirable results. Other problem is the normalization of the data that requires more study and time invested in this features.

All the project as said during the Objectives is produced in Python, and using the TensorFlow library. The knowledge obtained during the development of this thesis is a great value for my future professional trajectory.

Tensorflow is an important part in the state of art in the machine learning sector, and it is forecasted that this importance will increased as the library grows in features and performance. Also, the introduction of portability between different platforms widens the applications of this library.

During the development of the thesis different parts of the machine learning field are inspected and studied as, NN layers and nodes distribution, activation functions, optimization and training methods. Providing a good view of all of them.

About the future projections of the model, the first objective should be to refine the generation of new actions from different movements is the only part in the thesis that requires a review. The implementation in a real robot and the observation of their actions should be the next big step. If this model can be introduced in a real robot successfully it opens the possibility of using widely in future projects in robotics. Currently other more complex methods are used to do the same task, but if this model is refined and applied robustly, the AE-DMP could give a more flexible approach.

Appendices

APPENDIX A

DATA ACQUISITION FUNCTION

```
def readfromamc_func(filepath):
import numpy as np
import re
f = open(filepath,'r')
if f==-1:
    print("failed to load file")
    return -1
data = f.readline()
string = ":DEGREES"
while (data.rstrip('\n') != string):
    data = f.readline()
D=[]
dims =[6,3,3,3,3,3,3,2,3,1,1,2,1,2,2,3,
1,1,2,1,2,3,1,2,1,3,1,2,1];
locations = [1,1,4,7,10,13,16,19,19,22,23,24,26,26,28,28,
31,32,33,35,5,37,40,41,43,44,47,48,50];
frame=1;
data = f.readline()
data = f.readline()
while data != '':
    row = np.zeros(50)
    for _ in range(29):
        linedata = re.split(r'\s+', data)
        if linedata[0] == 'root':
            index = 0
        elif linedata[0] == 'lowerback':
            index = 2
        elif linedata[0] == 'upperback':
            index = 3
        elif linedata[0] == 'thorax':
            index = 4
        elif linedata[0] == 'lowerneck':
            index = 5
        elif linedata[0] == 'upperneck':
            index = 6
        elif linedata[0] == 'head':
            index = 7
        elif linedata[0] == 'rclavicle':
            index = 0
        elif linedata[0] =='rhumerus':
```

```
            index = 9
        elif linedata[0] == 'rradius':
            index = 10
        elif linedata[0] == 'rwrist':
            index = 11
        elif linedata[0] == 'rhand':
            index = 12
        elif linedata[0] == 'rfingers':
            index = 0
        elif linedata[0] == 'rthumb':
            index = 14
        elif linedata[0] == 'lclavicle':
            index = 0
        elif linedata[0] == 'lhumerus':
            index = 16
        elif linedata[0] == 'lradius':
            index = 17
        elif linedata[0] == 'lwrist':
            index = 18
        elif linedata[0] == 'lhand':
            index = 19
        elif linedata[0] == 'lfingers':
            index = 0
        elif linedata[0] ==  'lthumb':
            index = 21
        elif linedata[0] ==  'rfemur':
            index = 22
        elif linedata[0] == 'rtibia':
            index = 23
        elif linedata[0] == 'rfoot':
            index = 24
        elif linedata[0] == 'rtoes':
            index = 25
        elif linedata[0] == 'lfemur':
            index = 26
        elif linedata[0] == 'ltibia':
            index = 27
        elif linedata[0] == 'lfoot':
            index = 28
        elif linedata[0] == 'ltoes':
            index = 29
        else:
            print("Labels are not correct")
            return -1
        if(index != 0):
            location = locations[index-1]
            dim = len(linedata)-1
            row[location-1:location+dim-2] = linedata[1:dim]
        data = f.readline()
    row_T = np.array(row)
    D.append(row_T)
    data = f.readline()
Out = np.array(D)
f.close()
return Out
```

APPENDIX B

DATA WRITING FUNTION

```
def writetoamc_func(filepath,D):
    import numpy as np
    import re
    Data_in = np.squeeze(D)
    f = open(filepath,'w+')
    aux = '#!OML:ASF F:\\VICON\\USERDATA\\INSTALL\\rory3\\rory3.ASF\n'
    f.write(aux)
    aux = ":FULLY-SPECIFIED\n"
    f.write(aux)
    aux = ":DEGREES\n"
    f.write(aux)
    Dummy = [0,17.8934,0,0,0,0]
    locations = [1,1,4,7,10,13,16,19,19,22,23,24,26,26,28,28,
    31,32,33,35,35,37,40,41,43,44,47,48,50];
    dims =[6,3,3,3,3,3,3,2,3,1,1,2,1,2,2,3,1,1,2,1,2,3,1,2,1,3,1,2,1];
    names = ['root','lowerback','upperback','thorax','lowerneck','upperneck','head','rclavicle','
    rhumerus','rradius','rwrist','rhand','rfingers','rthumb','lclavicle','lhumerus','lradius','lwrist
    ','lhand','lfingers','lthumb','rfemur','rtibia','rfoot','rtoes','lfemur','ltibia','lfoot','ltoes'
    ]
    for n in range(Data_in.shape[0]):
        aux = str(n+1)
        f.write(aux+ '\n')
        for i in range(29):
            if(i==0):
                aux = names[i] + ' ' +str(Dummy[0])+ ' ' + str(Dummy[1]) + ' ' + str(Dummy[2])+ ' ' +
    str(Dummy[3]) + ' ' + str(Dummy[4]) + ' ' + str(Dummy[5])
            elif(i==7):
                aux = names[i] + ' ' + str(0) + ' ' + str(0)
            elif(i==12):
                aux = names[i] + ' ' + str(7.12502)
            elif(i==14):
                aux = names[i] + ' ' + str(0) + ' ' + str(0)
            elif(i==19):
                aux = names[i] + ' ' + str(7.12502)
            else:
                aux = names[i]
                for j in range(dims[i]):
                    aux = aux + ' ' + str(Data_in[n,(locations[i]+j-1)])
            f.write(aux+ '\n')
    f.close()
```

APPENDIX C

AE-DMP INTEGRATED TRAINING

```
version = 6.0
from pathlib import Path
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='3' #Used to reduce tensorflow logging

def pause(): #Custom pause function
    programPause = input("Press the <ENTER> key to continue...")

import tensorflow as tf
config = tf.ConfigProto()
config.gpu_options.allocator_type = 'BFC'

import numpy as np
from readfromamc import readfromamc_func
import matplotlib.pyplot as plt
import math
import re
import scipy.signal as signal
#Non tensor ops---
inputs = 50
h_1_neurons = 60
h_2_neurons = 42
h_3_neurons = 25
h_4_neurons = 5
epochs = 100000
file_num = 1
sample_num = 100
#Read data
D = readfromamc_func("Data/joints.amc")

Data_in = np.squeeze(D)
minim = Data_in.shape[0]

#Initialization of placeholder
sess = tf.InteractiveSession()
x = tf.placeholder(tf.float32,shape=[None,inputs],name="Inputs") #Target values
x_corrupt = tf.placeholder(tf.float32,shape=[None,inputs],name="Inputs_corrupted")
Denoised_W_h1 = tf.placeholder(tf.float32,shape=[],name="Input_corruption")

with tf.name_scope('Data_info') as scope:
```

```
    mean_data,std_data = tf.nn.moments(x,[0], keep_dims=True)

with tf.name_scope('AE') as scope: # Input: x_corrupt, output: z_out
    with tf.name_scope('Process_data') as scope:
        x_norm = tf.div(tf.div(tf.subtract(x_corrupt,mean_data),tf.sqrt(std_data)),(math.pi*2))

#Deep Neural Nets build
with tf.name_scope('hidden_1') as scope:
    W_h1 = tf.Variable(tf.random_normal([inputs,h_1_neurons],stddev=0.1),name="Weights_hidden_1")
    b_h1 = tf.Variable(tf.constant(0.1,shape = [h_1_neurons]),name = "Bias_hidden_1")

    h_1 = tf.tanh(tf.matmul(x_norm,(W_h1*Denoised_W_h1))+b_h1)

with tf.name_scope('hidden_2') as scope:
    W_h2 = tf.Variable(tf.random_normal([h_1_neurons,h_2_neurons],stddev=0.1),name="Weights_hidden_2")
    b_h2 = tf.Variable(tf.constant(0.1,shape = [h_2_neurons]),name = "Bias_hidden_2")

    h_2 = tf.tanh(tf.matmul(h_1,W_h2)+b_h2)

with tf.name_scope('hidden_3') as scope:
    W_h3 = tf.Variable(tf.random_normal([h_2_neurons,h_3_neurons],stddev=0.1),name="Weights_hidden_3")
    b_h3 = tf.Variable(tf.constant(0.1,shape = [h_3_neurons]),name = "Bias_hidden_3")

    h_3 = tf.tanh(tf.matmul(h_2,W_h3)+b_h3)

with tf.name_scope('hidden_4') as scope:
    W_h4 = tf.Variable(tf.random_normal([h_3_neurons,h_4_neurons],stddev=0.1),name="Weights_hidden_4")
    b_h4 = tf.Variable(tf.constant(0.1,shape = [h_4_neurons]),name = "Bias_hidden_4")

    y_AE = tf.tanh(tf.matmul(h_3,W_h4)+b_h4)

with tf.name_scope('hidden_g4') as scope:
    W_g4 = tf.Variable(tf.random_normal([h_4_neurons,h_3_neurons],stddev=0.1),name="Weights_hidden_g4"
    )
    b_g4 = tf.Variable(tf.constant(0.1,shape = [h_3_neurons]),name = "Bias_hidden_g4")

    g_4 = tf.tanh(tf.matmul(y_AE,W_g4)+b_g4)

with tf.name_scope('hidden_g3') as scope:
    W_g3 = tf.Variable(tf.random_normal([h_3_neurons,h_2_neurons],stddev=0.1),name="Weights_hidden_g3"
    )
    b_g3 = tf.Variable(tf.constant(0.1,shape = [h_2_neurons]),name = "Bias_hidden_g3")

    g_3 = tf.tanh(tf.matmul(g_4,W_g3)+b_g3)

with tf.name_scope('hidden_g2') as scope:
    W_g2 = tf.Variable(tf.random_normal([h_2_neurons,h_1_neurons],stddev=0.1),name="Weights_hidden_g2"
    )
    b_g2 = tf.Variable(tf.constant(0.1,shape = [h_1_neurons]),name = "Bias_hidden_g2")

    g_2 = tf.tanh(tf.matmul(g_3,W_g2)+b_g2)

with tf.name_scope('hidden_g1') as scope:
    W_g1 = tf.Variable(tf.random_normal([h_1_neurons,inputs],stddev=0.1),name="Weights_hidden_g1")
    b_g1 = tf.Variable(tf.constant(0.1,shape = [inputs]),name = "Bias_hidden_g1")

    z = tf.tanh(tf.matmul(g_2,W_g1)+b_g1)

```

```
    with tf.name_scope('Deprocess_output') as scope:
        z_out = tf.multiply(tf.add(tf.multiply(z,tf.sqrt(std_data)),mean_data),(2*math.pi))  #

with tf.name_scope('Data_info') as scope:
    mean_data_encoder,std_data_encoder = tf.nn.moments(x,[0], keep_dims=True)

with tf.name_scope('Encoder') as scope: # Input: x, output: y
    with tf.name_scope('Process_data') as scope:
        x_norm_encoder = tf.div(tf.div(tf.subtract(x_corrupt,mean_data_encoder),tf.sqrt(
     std_data_encoder)),(math.pi*2))

    with tf.name_scope('hidden_1') as scope:
        h_1 = tf.tanh(tf.matmul(x_norm_encoder,(W_h1*Denoised_W_h1))+b_h1)

    with tf.name_scope('hidden_2') as scope:
        h_2 = tf.tanh(tf.matmul(h_1,W_h2)+b_h2)

    with tf.name_scope('hidden_3') as scope:
        h_3 = tf.tanh(tf.matmul(h_2,W_h3)+b_h3)

    with tf.name_scope('hidden_4') as scope:
        y = tf.tanh(tf.matmul(h_3,W_h4)+b_h4)

# Loss function
with tf.name_scope('Loss_AE') as scope:
    loss_AE = tf.reduce_mean(tf.sqrt(tf.reduce_sum(tf.square(x_norm-z),1)),0)
tf.summary.scalar('Loss_AE', loss_AE)

#DAE training function
with tf.name_scope('Opt_AE') as scope:
    train_step_AE = tf.train.AdadeltaOptimizer(0.1).minimize(loss_AE)

#Non tensor ops---


num = 40
framerate = 120
t_final = minim/framerate
ts = t_final/Data_in.shape[0]
alfax = 1
alfa = 10
beta = alfa/4
tau = 1.0
X0 = 1
dim = h_4_neurons

#Derivative matrix
W_dd = np.zeros([Data_in.shape[0],Data_in.shape[0]])
W_dd[0,0:3]=(1,-2,1) #Eliminate 0 in first row
Aux_array = np.zeros([Data_in.shape[0]])
Aux_array[0:3]=(1,-2,1)
for i in range(1,Data_in.shape[0]-1):
    W_dd[i,:] = Aux_array
    Aux_array=np.roll(Aux_array,1)
W_dd[-1,-3:] = (1,-2,1) # W_dd[Data_reduced.shape[0]-1,0] = 0
W_dd = W_dd/(np.square(ts))

W_d = np.zeros([Data_in.shape[0],Data_in.shape[0]])
```

```
W_d[0,0:2]= (-1,1)
Aux_array = np.zeros(Data_in.shape[0])
Aux_array[0:2]=(-1,1)
for i in range(1,Data_in.shape[0]):
    W_d[i,:] = Aux_array
    Aux_array = np.roll(Aux_array,1)
W_d[-1,-2:]=(-1,1)
W_d = W_d/ts
#---Non tensor ops

with tf.name_scope("Basis_funcs") as scope:
    with tf.name_scope("Kernel_fun_params") as scope:
        mean = tf.linspace(ts,t_final,num)
        mean_distributed = X0*tf.exp(-alfax*mean/tau)
        var_distributed = num/mean_distributed

    with tf.name_scope("Weights") as scope:
        w = []
        for i in range(dim):
            name = "Weights" + str(i)
            scope_name = 'Weights_kernel_' + str(i)
            with tf.name_scope(scope_name) as scope:
                w.append(tf.Variable(tf.truncated_normal([num], stddev=1),name=name))

    with tf.name_scope("Kernel") as scope:
        time = [tf.linspace(0.0,t_final,Data_in.shape[0])]
        aux = tf.zeros([num,1],dtype=tf.float32)
        s_t = tf.transpose(time + aux)

        s = X0*tf.exp(-alfax*s_t/tau)
        internm = -(var_distributed*tf.square(tf.subtract(s,mean_distributed)))
        kernel = tf.exp(internm,name="Kernel_fun")

with tf.name_scope("Forcing_term") as scope:
    f_list = []
    kernel_sum = tf.expand_dims(tf.reduce_sum(kernel,1),1)
    for i in range(dim):
        f_list.append(tf.divide(tf.matmul(kernel,(tf.expand_dims(w[i],1))),kernel_sum))

W_dd_tensor = tf.constant(W_dd,dtype=tf.float32,name="Second_derivative")
W_d_tensor = tf.constant(W_d,dtype=tf.float32,name="First_derivative")

#Non tensor ops---

#Matrix A
A_1 = (tau-np.square(ts)*alfa*beta)/tau
A = np.identity(Data_in.shape[0])
A_aux = np.zeros([1,Data_in.shape[0]])
A = np.vstack((A_aux,A))*A_1
A = np.delete(A,-1,axis=0)
A[0,0] = 1
A_tensor = tf.constant(A,dtype=tf.float32,name="Matrix_A")

#Matrix B
B_1 = (tau*ts-np.square(ts)*alfa)/tau
B = np.identity(Data_in.shape[0])
B_aux = np.zeros([1,Data_in.shape[0]])
B = np.vstack((B_aux,B))*B_1
```

```
B = np.delete(B,-1,axis=0)
B_tensor = tf.constant(B,dtype=tf.float32,name="Matrix_B")

#Matrix C
C_1 = np.square(ts)*alfa*beta/tau
C_aux = np.ones([Data_in.shape[0],1])
C_aux[0] = 0
C_1 = C_1*C_aux
C_1_tensor = tf.constant(C_1,dtype=tf.float32,name="Matrix_C_1")

C_2 = np.square(ts)/tau
C_2 = C_2*C_aux
C_2_tensor = tf.constant(C_2,dtype=tf.float32,name="Matrix_C_2")

#--- Non tensor ops

with tf.name_scope("Goal") as scope:
    goal = tf.transpose(tf.expand_dims(y[-1],axis=1))

goal = tf.stop_gradient(goal)

with tf.name_scope("Forcing_target_term") as scope:
    f_target_list = []
    y_dd_matrix = tf.matmul(W_dd_tensor,y)
    y_d_matrix = tf.matmul(W_d_tensor,y)
    for i in range(dim):
        y_dd = tf.slice(y_dd_matrix,[0,i],[-1,1])
        y_d = tf.slice(y_d_matrix,[0,i],[-1,1])
        f_target_list.append(tf.multiply(tf.square(tf.constant(tau,dtype=tf.float32)),y_dd)-tf.
    multiply(tf.constant(alfa,dtype=tf.float32),tf.subtract(tf.multiply(beta,tf.subtract(goal[0,i],tf
    .slice(y,[0,i],[-1,1]))),tf.multiply(tf.constant(tau,dtype=tf.float32),y_d))))

with tf.name_scope("Pack_tensor_lists") as scope:
    f_target = tf.transpose(tf.squeeze(tf.stack(f_target_list)))
    f = tf.transpose(tf.squeeze(tf.stack(f_list)))

with tf.name_scope("y_computed") as scope:
    y_decode = tf.matmul(A_tensor,y)+tf.matmul(B_tensor,y_d)+tf.multiply(C_1_tensor,goal)+tf.multiply(
    f,C_2_tensor)

with tf.name_scope('Decoder') as scope: # Input: y_decode, output: z_decoded_out
    with tf.name_scope('hidden_g4') as scope:
        g_4 = tf.tanh(tf.matmul(y_decode,W_g4)+b_g4)

    with tf.name_scope('hidden_g3') as scope:
        g_3 = tf.tanh(tf.matmul(g_4,W_g3)+b_g3)

    with tf.name_scope('hidden_g2') as scope:
        g_2 = tf.tanh(tf.matmul(g_3,W_g2)+b_g2)

    with tf.name_scope('hidden_g1') as scope:
        z_decoded = tf.tanh(tf.matmul(g_2,W_g1)+b_g1)

    with tf.name_scope('Deprocess_output') as scope:
        z_decoded_out = tf.multiply(tf.add(tf.multiply(z_decoded,tf.sqrt(std_data_encoder)),
    mean_data_encoder),(2*math.pi))

with tf.name_scope('Target_data') as scope:
```

```
    with tf.name_scope('Process_data') as scope:
        x_norm_target = tf.div(tf.div(tf.subtract(x,mean_data_encoder),tf.sqrt(std_data_encoder)),(
    math.pi*2))

nu = 0.25
mu = 1
with tf.name_scope("Loss_fun_AE-DMP") as scope:
    suma_AE = tf.sqrt(tf.reduce_sum(tf.square(x_norm_target-z_decoded),1))
    loss_AE_scalar = tf.reduce_sum(suma_AE,0)
    suma_DMP = tf.sqrt(tf.reduce_sum(tf.square(f_target-f),1))
    loss_DMP_scalar = tf.reduce_sum(suma_DMP,0)
    spartsity = tf.reduce_sum(tf.abs(y))
    loss_ae_dmp = tf.reduce_sum(suma_AE+nu*suma_DMP,0)+mu*spartsity

tf.summary.scalar('Loss_AE', loss_AE_scalar)
tf.summary.scalar('Loss_DMP', loss_DMP_scalar)
tf.summary.scalar('Loss_AE_DMP', loss_ae_dmp)

with tf.name_scope("Opt_DMP") as scope:
    train_step_DMP = tf.train.AdadeltaOptimizer(0.1).minimize(loss_ae_dmp)

sess.run(tf.global_variables_initializer())

with tf.name_scope("Saver") as scope:
    saver = tf.train.Saver()

#Loading model if exist, else create a new one
my_file = Path("tmp/version.txt")
if my_file.is_file():
    saver = tf.train.Saver()
    print("Loading model")
    saver.restore(sess, "tmp/model.ckpt")
    print("Load sucessful")
else:
    sess.run(tf.global_variables_initializer())
    with tf.name_scope("Saver") as scope:
        saver = tf.train.Saver()

merged = tf.summary.merge_all()

summary_writer = tf.summary.FileWriter("tmp/AE_logs", sess.graph)

x_mse = tf.placeholder(tf.float32,shape=[None],name="MSE_target")

MSE_joint_mean = tf.reduce_mean(tf.squared_difference(z_out[:,0],x_mse))
MSE_joint_variance = tf.sqrt(MSE_joint_mean*Data_in.shape[0])
plt.ion()
DMP_corruptness = 0.05

print("Starting DMP training")

print ("Start DMP training loop")
loop = 0
acc_sparse = 0
trained_epochs = 0
try:
    while trained_epochs<5000000:
        plt.close("all")
```

```
            plt.pause(0.001)
            f_aux = np.transpose(sess.run(f))
            f_target_aux = np.transpose(sess.run(f_target,feed_dict={x: Data_in,x_corrupt: Data_in,
        Denoised_W_h1: 1-DMP_corruptness}))
            print("Sparsity: ", acc_sparse)
            #Every x number plot the actual state of the training
            for i in range(dim):
                plt.figure()
                plt.plot(f_aux[i,:],label="f_term")
                plt.plot(f_target_aux[i,:],label="f_target")
                plt.legend()
            plt.draw()
            plt.pause(0.001)
            for i in range(epochs):
                #Corruption operation
                Matrix_corruption = np.random.rand(Data_in.shape[0],Data_in.shape[1])
                corrupt_data_var = np.copy(Data_in)
                corrupt_data_var[np.where(Matrix_corruption<DMP_corruptness)] = 0
                #Training operation
                sess.run(train_step_DMP,feed_dict={x: Data_in,x_corrupt: Data_in,Denoised_W_h1: 1})
                if (i%10000) == 0: # Each 10000 print the current state
                    summary,acc,acc_AE,acc_DMP,acc_sparse= sess.run([merged,loss_ae_dmp,loss_AE_scalar,
        loss_DMP_scalar,spartsity],feed_dict={x: Data_in,x_corrupt: Data_in,Denoised_W_h1: 1-
        DMP_corruptness})
                    trained_epochs = i+epochs*loop
                    print("Epoch:",trained_epochs," Loss: ",acc, " AE Loss: ",acc_AE, " DMP Loss: ",
        acc_DMP)
                    summary_writer.add_summary(summary, loop)
            loop = loop + 1
    except KeyboardInterrupt:
        pass

    print("Finished training DMP")

    print("Saving data to tmp/model.ckpt")
    #Save model
    save_path = saver.save(sess, "tmp/model.ckpt")


    #Write a file that conserve some parameters for the test, and for the user
    my_file = Path("tmp/version.txt")
    if my_file.is_file():
        f = open("tmp/version.txt",'a')
    else:
        f = open("tmp/version.txt",'w+')
    aux = "Version: " + str(version) + "\n"
    f.write(aux)
    f.write("DMP_training\n")
    aux = "DAE corruption: "+str(DMP_corruptness)+"\n"
    f.write(aux)
    aux = "Nu: "+str(nu)+"\n"
    f.write(aux)
    aux = "Mu: "+str(mu)+"\n"
    f.write(aux)
    aux = "Trained by "+str(trained_epochs)+" epochs\n"
    f.write(aux)
    aux = "Last loss: "+str(acc)+"\n"
    f.write(aux)
```

```
376 f.close()
377 pause()
```

BIBLIOGRAPHY

[1] Heiko Hoffmann-Peter Pastor Stefan Schaal Auke Jan Ijspeert, Jun Nakanishi. Dynamical movement primitives: Learning attractor models for motor behaviors. 2013.

[2] Nutan Chen, Justin Bayer, Sebastian Urban, and Patrick van der Smagt. Efficient movement representation by embedding dynamic movement primitives in deep autoencoders. In *Humanoids*, pages 434–440. IEEE, 2015.

[3] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. 2011.

[4] Michael Elad. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer, 2010.

[5] Yoshua Bengio Guillaume Alain. What regularized auto-encoders learn from the data-generating distributionn. 2014.

[6] Giambattista Parascandolo, Heikki Huttunen, and Tuomas Virtanen. Taming the waves: sine as activation fucntion in deep nerual networks. Technical report, 2017.

[7] Yoshua Bengio Pierre-Antonie Manzagol Pascal Vincent, Hugo Larochelle. Extracting and composing robust features with denoising autoencoders. Technical report.

[8] Yoshua Bengio Pierre-Antonie Manzagol Isabelle Lajoie Pascal Vincent, Hugo Larochelle. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. 2010.

[9] Matthew D. Zeiler. *ADADELTA: AN ADAPTIVE LEARNING RATE METHOD*. Google Inc., USA New York University, USA, 2012.