Imperial College London
Department of Computing

Machine Learning with Chaotic Recurrent Neural Networks

Thomas Fonlladosa (tcf12)

September 2013

Supervised by Professor Murray Shanahan

Submitted in part fulfilment of the requirements for the
MSc Degree in Advanced Computing of Imperial College London

Abstract

The aim of this project is to use some machine learning methods with continuous time recurrent neural networks that spontaneously exhibit chaotic behaviours and explore different applications. We first apply the training algorithms to learn the spontaneous or inputs-depending generation of complex periodic patterns.
Secondly, these results are applied to control a two joints robot arm. We also extend them to the post-learning generation of new untaught patterns that matches new inputs by interpolation of the trained behaviours.
Thirdly a robust 2-bits working memory is implemented.
At last we train a network to generate complex aperiodic control signals that could be used to control a drone trajectory on one dimension. In this part, a similar interpolation process leads to the generation of new untaught signals.

Acknowledgements

I would like to thank Professor Murray Shanahan for the precious suggestions and guidelines he gave me during this project, as for the enthusiasm he showed about our results.
Thanks also to Thomas, Nicolas, Hugo, Diana, Nick, George, Marc-Antoine, Guillaume and their flatmates for hosting me during the summer and thus making my work easier.
Thanks to Julien-James my friend and proof-reader for his useful corrections.
Thanks to my friend Suzanne for her priceless moral support during the last weeks of this project.

Contents

List of Figures

Introduction

Recurrent neural networks are different from feed-forward networks because their recurrent connectivity give them internal states allowing to exhibit dynamical behaviours. The dynamic of such networks depends on the inputs they receive as well as on their previous state. This particularity of recurrent networks to be able to produce a wide range of dynamical behaviours has a lot of applications. They are also a source of interest because some models of recurrent neural networks can be a good representation of biological brains.
Two recent publications ([1] and [2]) describe how recurrent neural networks that spontaneously exhibit chaotic behaviour can be trained with different learning algorithms to produce some computational structures such as a working memory or to generate some complex periodic patterns. Those two papers are our main source of inspiration for this project: in this thesis, after having replicated some of their results, we tried to go further and explore new areas of application of such networks.

The first chapter of this report presents the background of the thesis.
The second chapter presents formally the network architectures and the learning algorithms that are applied to them in this project.
The third chapter gives some details of the matlab implementation.
The fourth chapter describes the first results, ie the generation of periodic patterns.
The fifth chapter explains the application of the first results to a two-joints robot arm, and the interpolation process which is introduced to learn new patterns.
The sixth chapter is about the implementation of a robust 2-bits memory.
The seventh chapter deals with the learning of complex aperiodic control-signals within the framework of an application to a drone controller.
The eighth chapter finally describes the future perspectives that could follow this project regarding the previous chapter in particular.

1 Background

Labyrinthe sans clef ! ...

1.1 Recurrent Neural Networks

In this project, we are using a generic network of N neurons who are sparsely randomly recurrently connected by excitatory and inhibitory synapses. This general architecture with sparse and asymmetric connections has often been used to model the dynamic activity of recurrent biological neural networks ([1], [3]). The neurons can also receive external inputs. Some read-out units perform a linear combination of the neurons activity to generate an output. The network output(s) is sometime fed back to the network, directly or via a second feedback network.

1.2 Leaky integrator Model

Each network unit i is a leaky integrator. In this model, the membrane potential x_i of each neuron i follows a variant of this first order differential equations, which depends on the particular network architecture that is used (cf 2.1 for the details of the different network architectures):

$$\tau \dot{x}_i(t) = -x_i(t) + \lambda \sum_{i=1}^{N} W_{ij}^{rec} r_j(t) + \sum_{i=1}^{M} W_{ij}^{in} I_j(t) + \sum_{i=1}^{L} W_{ij}^{z} z_j(t)$$

where

- τ is the time constant (we can assume it is the same for all units)
- x_i is the neuron's membrane potential
- λ is a scaling factor on the recurrent connection whose value can introduce chaotic dynamic in the network.
- W_{ij}^{rec} is the synaptic weight from neuron j to i for the recurrent network
- $r_j = tanh(x_j(t))$ is the firing rate of the j^{th} neuron.
- W_{ij}^{in} is the weight of the incoming current $I_j(t)$ in i.
- W_{ij}^{z} is the weight applied to the read-out unit j when fed back to neuron i.
- z_j is the output of the readout unit j.

In fact, each neuron can been seen as a RC circuit, and the differential equations are derived from Kirchoff's current law that says that the total current flowing toward any node of an electrical circuit is zero [4]. This model of integrate and fire neurons ignores inter-neuron propagation times. There exists some more biologically plausible neuron models like Izhikevich or Hodgkins-Huxley models that have a larger repertoire of behaviours but are also more computationally expensive.

1.3 Chaotic activity

The dynamic of such neural networks, also called continuous-time recurrent neural networks has been studied in details, and they have a lot of applications [5]. Because of delayed effects implied by the use of recurrent connections and feedback loops, such networks can spontaneously exhibit chaotic activity.
The transition from a stationary to a chaotic state occur at a critical value of the gain parameter λ [6]. At the transition between chaos and stability, neural networks can produce complex computational tasks. [7].
Experimental results showed that spontaneously chaotic networks are easier to train and produce more accurate and robust outputs than non chaotic networks. The more a network is chaotic the better is the training, however with an upper bound ([1]).

1.4 Reservoir Computing

Reservoir computing is a framework that gives a way of designing and training recurrent neural networks. It is a good model for biological networks. The main characteristic of reservoir computing as describe in [8] are mainly:

- the use of a reservoir which is a large randomly connected recurrent neural network that can be excited by input signals. The reservoir maps the input to a higher dimensional space because each neuron has its own non linear activity that will depend on the input.
- an readout mechanism to produce an output signal from the reservoir's neuron activity, usually a linear combination of the neurons' activities.
- a learning mechanism that can be used to train the reservoir output, for example by linear regression.

Echo-state networks [9] and Liquid State Machines ([10]) are the two main types of reservoir computing.

According to [8] reservoir computing has now become a paradigm for neural computation, both as a computational technique for technical applications and as an explanatory model for processes in biological brains.

1.5 Plasticity and learning in the biological brain

Brain performance can be enhanced in a wide range of functions with training. Training results in changes in the synaptic connectivity of the cortex. For example training on motor [11] and perceptual tasks [12] in animals leads after hundreds of trials to enhanced performances, with concomitant changes in synaptic connectivity in both sensory and motor areas. Plasticity has also been demonstrated in some animals' pre-frontal cortex which is responsible for cognitive functions [13].
Working memory (WM) capacity is the ability to retain and manipulate information during a short period of time [14]. This ability is regarded as closely related to cognitive abilities. Working memory can also be trained, and the training is associated with changes in the brain activity in frontal and parietal cortex and basal ganglia and in the dopamine receptor density [15].
Legenstein et al. [16] showed that reinforcement learning methods applied to artificial neural networks could explain these brain network reorganisations.
The reinforcement learning methods are thus biologically plausible because synaptic plasticity is observed in the brain, as well as the presence of neuro-modulators like dopamine who can act as a learning reward. In fact the learning reward acts as a modulator that supervises the spike-timing-dependent plasticity (STDP) or other forms of learning that depend on post-synaptic and pre-synaptic activity in the tradition of Hebbian learning [17].

1.6 Learning in Recurrent Neural Networks

1.6.1 FORCE learning

Susillo and Abbott [1] developed a learning procedure that changes chaotic activity of a recurrent network into a wide range of activity patterns. This supervised training procedure called FORCE learning differs from common learning methods because the output error is small from the beginning but the number of modifications needed to maintain this error small is reduced instead. By maintaining a small output error, this method prevents learning failure that can occur when erroneous outputs are fed back, making the network activity diverge from the expected output. Another advantage of FORCE learning is that it allows to train networks without restricting the synaptic strength modifications to the output neurons, making the network architecture more biologically plausible. At last, by performing strong and quick synaptic modifications, this procedure achieve training in chaotic networks which provides advantages mentioned previously 1.3.
Feeding back an output close to the desired output but still different is crucial for the network stability because it allows the network to sample instabilities and deal with them.
The FORCE learning procedure has been compared to Jaeger and Hass' echo-state learning where the desired output f is fed back with noise to the network during training. Noise is introduced in echo-state learning to ensure the network's stability after training even in the presence of small fluctuations in the feedback loop. Results shows that echo-state learning converge less often and with larger error than FORCE learning.

1.6.2 Reward modulated learning

Hebbs's 1949 learning rule states that "neurons that fire together wire together", ie the connections strength between two neurons should increase when the neurons fire simultaneously.
Legenstein et al [16] introduced an exploratory Hebbian rule were a global reward signal and neuronal noise are used to perform a Hebbian weight update. In contrast with previous node-perturbating learning rules [18], their approach do not need to separate the exploration noise from the output signal. Their learning rule is biologically plausible and reproduced the results that were found during brain-computer learning experiments. A zero-mean noise is added to the firing-rates of the readout neurons and allow the exploration of alternative behaviours. This noise can be interpreted as spontaneous activity or input from other brain regions. A modulatory factor contains a precise information on how much the system performance has recently increased or decreased. This learning rule is a 3 factors Hebbian learning rule because it depends on the correlation between presynaptic and postsynaptic activity and a modulatory third factor).
Hoerzer, Legenstein and Maas [2] investigate a variation of this exploratory Hebbian rule where the modulatory third factor contains the minimum amount of information. They use a binary third factor that only indicates whether the network's performance has recently improved or not. This way of training a network without a fully supervised learning rule like FORCE learning is more biologically plausible.

1.6.3 Applications

FORCE learning is used by Sussillo and Abbott in [1] to train the network to produce a wide variety of periodic functions as output. Training works successfully with different kinds of network architectures described in 2.1 and typically converges, ie find a set of fixed-synaptic weights that keep producing the desired output after training, in about 1000τ, where τ is the basic time constant of the system. A very large dynamic range of outputs can be produced.
Hoerzer et al. [2] obtain similar results when they replace the supervised FORCE learning rule by the reward-modulated Hebbian learning rule. They also test the system under 45 different parameter settings to investigate the influence of the frequency components of the target output, the update interval of the weights and the modulatory signal, and the time constant of the exploration noise added to the output. Frequency components must be sufficiently slow otherwise the readout

unit cannot adapt its output quickly enough. However, when frequency components are too high, the performance can be improved by using a smaller time constant τ for the network. The modulatory signal's update frequency mustn't be too low compared to the system evolution in order to adapt the output to the target quickly, especially when the frequency component's of the target are high. Regarding the exploration noise in the readout units, performance are enhanced when the noise is not time correlated.
In both method, the activity of the output neuron has a strong influence on the internal network activity which also become periodic during and after training due to the influence of the feedback loop. However the gain parameter λ must be carefully chosen: the network must initially exhibit chaotic dynamics to generate the target function but if the network activity is too chaotic, the feedback loop cannot bring the network to a stable state.

2 Theory

... question sans réponse,

2.1 Networks architectures

Three different recurrent network architectures were used in this project.

2.1.1 Architecture A

The first model we used is a generic neural network of N_G neurons who are sparsely randomly recurrently connected by excitatory and inhibitory synapses. This network is called the generator network and his defined by its connectivity matrix J_G.
The network can receive external inputs I.
Some readout units perform linear combinations of the neurons' firing rates r to generate what are called the network outputs z. These outputs are fed back to all the neurons in the network.

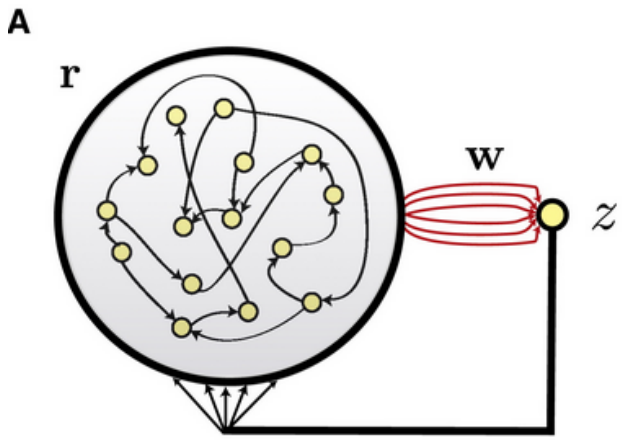

Figure 2.1: Network architecture A, taken from [1]

In this model, the membrane potential x_i of each neuron i follows the following first order differential equations:

$$\tau \dot{x}_i(t) = -x_i(t) + g_G \sum_{j=1}^{N_G} J_{ij}^G r_j(t) + \sum_{j=1}^{N_{in}} J_{ij}^{GIn} I_j(t) + g_z \sum_{j=1}^{N_{out}} J_{ij}^z z_j(t) \tag{2.1}$$

where

- τ is the time constant (we can assume it is the same for all units).
- x_i is the membrane potential of the i^{th} neuron of the generator network.
- $r_j = tanh(x_j(t))$ is the firing rate of the j^{th} neuron.
- I_j is the j^{th} input applied to the network.

- $z_j = \sum_{i=1}^{N_G} W_{ij} r_i$ is the j^{th} output (W_{ij} is the weight applied to the firing rate r_i in the weighted sum z_j).
- g_G is a scaling factor on the recurrent connection within the generator network, whose value can introduce chaotic dynamic in the network.
- g_z is a scaling factor on the feedback loop. The feedback connections are usually made stronger so that the feedback has a strong enough effect on the network chaotic activity to allow learning.
- J_{ij}^G is the synaptic weight from neuron j to i for the recurrent generator network.
- J_{ij}^{GIn} is the weight applied to incoming current $I_j(t)$ in neuron i.
- J_{ij}^z is the weight applied to the read-out unit j when fed back to neuron i.
- N_G, N_{in}, N_{out} are respectively the numbers of neurons, inputs and outputs.

During training the modifications are applied to the weights (W_{ij}) to make the network produce the desired output autonomously.

2.1.2 Architecture B

The second model we used is a variant of the first one where the network's outputs are not directly fed back to the generator network. Instead, a second network of N_F neurons, called the feedback network, is used. The feedback network receives inputs from the generator network and feeds back its activity to the generator network.

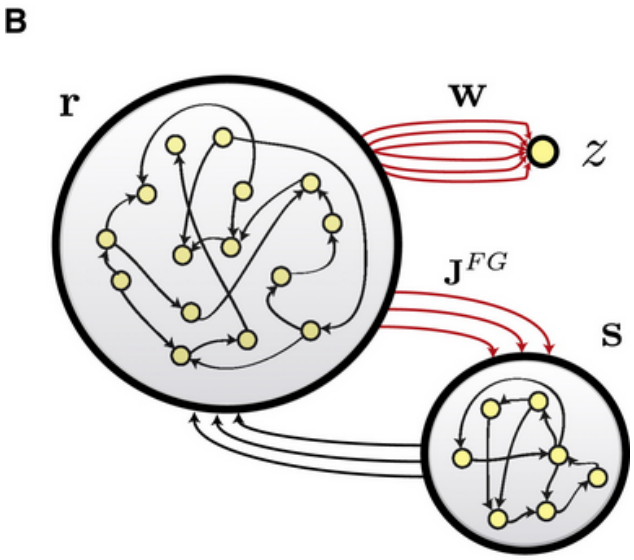

Figure 2.2: Network architecture B, taken from [1]

The network architecture now evolve according to two differential equations:

For the generator network:

$$\tau \dot{x}_i(t) = -x_i(t) + g_G \sum_{j=1}^{N_G} J_{ij}^G r_j(t) + \sum_{j=1}^{N_{in}} J_{ij}^{GIn} I_j(t) + g_{GF} \sum_{j=1}^{N_F} J_{ij}^{GF} s_j(t) \tag{2.2}$$

For the feedback network:

$$\tau \dot{y}_i(t) = -y_i(t) + g_F \sum_{j=1}^{N_F} J_{ij}^F s_j(t) + \sum_{j=1}^{N_{in}} J_{ij}^{FIn} I_j(t) + g_{FG} \sum_{j=1}^{N_G} J_{ij}^{FG} r_j(t) \tag{2.3}$$

where the notations are the same as in (2.1), plus:

- y_i is the i^{th} feedback neuron's membrane potential.
- $s_j = tanh(y_j(t))$ is the firing rate of the j^{th} feedback neuron.
- g_F is the scaling factor on the recurrent connexion within the feedback network.
- J_{ij}^{GF}, J_{ij}^{FG}, g_{GF} and g_{FG} are respectively the synaptic strength of the connexion from the feedback network to the generator network and vice versa and the associated scaling factors.
- J_{ij}^{FIn} is the weight applied to incoming current $I_j(t)$ in feedback network's i^{th} neuron.

The outputs of this architecture are still some weighted sums of the generator network's activity ($z_j = \sum_{i=1}^{N_G} W_{ij} r_i$ for the j^{th} output). During training, modifications are applied to both the weights (W_{ij}) and the synapses from the generator to the feedback network (J_{ij}^{FG}).
This architecture is more biologically plausible because the output is not directly fed back to all the neurons. Each neuron of the feedback network has a different activity which is fed back to different neurons.

2.1.3 Architecture C

In the third model, there is no explicit feed back of the generator network activity. Because the generator network is recurrent we can consider that, in a sense, it produces its own feedback.

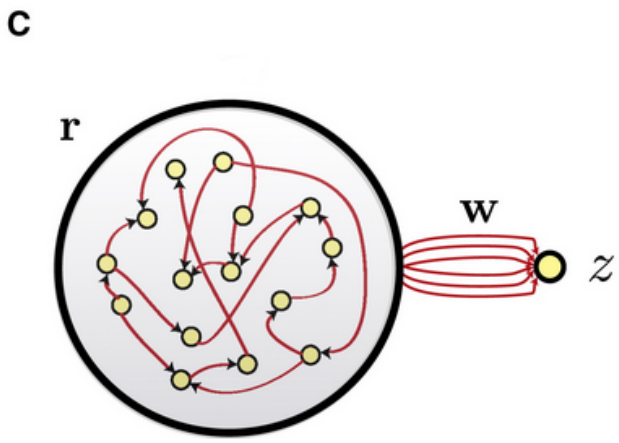

Figure 2.3: Network architecture C, taken from [1]

The equation that governs the network's activity is a simplification of (2.1) with the same notations:

$$\tau \dot{x}_i(t) = -x_i(t) + g_G \sum_{j=1}^{N_G} J_{ij}^G r_j(t) + \sum_{j=1}^{N_{in}} J_{ij}^{GIn} I_j(t) \tag{2.4}$$

In this case during training, the modifications are applied to the internal recurrent connections themselves (J_G^{ij}) and to the weights of the readout units W_{ij}.

2.2 Learning Algorithms

In this part we present a formal description of the learning algorithms presented in 1.6

2.2.1 FORCE learning

2.2.1.1 Architecture A

To satisfy the requirement of FORCE learning (quickly reducing the output error and keeping it small while looking for a set of fixed weights that will maintain it small), Sussillo and Abbott used

the recursive least square algorithm (RLS) from Haykin's Adaptative Filter Theory (2002).
Let $W(t)$ be the vector containing the weights connecting the neurons to the output and $r(t)$ be the vector containing the neurons' firing rates at time t.
The RLS modification is given by the following equation:

$$W(t) = W(t - \Delta t) - e_{-}(t)P(t)r(t) \tag{2.5}$$

where

- $e_{-}(t)$ is the error between the current and the desired output $f(t)$ at time t before the weights update:
$$e_{-}(t) = W(t - \Delta t)^T r(t) - f(t) \tag{2.6}$$
- $P(t)$ is an NxN matrix updated with the following rule:
$$P(t) = P(t - \Delta t) - \frac{P(t - \Delta t)r(t)r(t)^T P(t - \Delta t)}{1 + r(t)^T P(t - \Delta t)r(t)} \tag{2.7}$$
$$P(0) = \frac{1}{\alpha} Id_N \tag{2.8}$$

α is a constant parameter acting as a learning rate, whose value should be chosen according to the target function, subject to the constraint $\alpha << N$. A small value of α implies a fast learning but can sometime lead to stability problems, whereas if alpha is too large learning can fail.
Sussillo and Abbott shows that the RLS rule satisfies the FORCE learning constraints: if $\alpha << N$ then the error is small since the first update and the weights converge to a fixed value while the error is reduced.
The time between two weights updates Δt can be different (larger) from the integration time used for network simulation.

2.2.1.2 Architecture B

Another advantage of FORCE learning is that it can be applied to the network architecture described in 2.1.2 where the feedback pathway is separated from the network's linear output.
In this case, the read ou weights$W$ but also the synaptic weights connecting the generator network to the feedback network $J^F G$ are updated by using the RLS rule with the same error term $e_{-}(t)$ coming from the readout neurons:

$$J_{ai}^{FG}(t) = J_{ai}^{FG}(t - \Delta t) - e_{-}(t) \sum_{j \in A(a)} P_{ij}^{a}(t) r_j(t) \tag{2.9}$$

where

- $A(a)$ is the list of neurons from the generator network that are presynaptic to the neuron a in the feedback network.
- P^a is a square matrix of size the length of $A(a)$ updated with the following equations:
$$P_{ij}^{a}(t) = P_{ij}^{a}(t - \Delta t) - \frac{\sum_{k \in A(a)} \sum_{l \in A(a)} P_{ik}^{a}(t - \Delta t) r_k(t) r_l(t) P_{lj}^{a}(t - \Delta t)}{1 + \sum_{k \in A(a)} \sum_{l \in A(a)} r_k(t) P_{kl}^{a}(t - \Delta t) r_l(t)} \tag{2.10}$$
$$P^a(0) = \frac{1}{\alpha} Id \tag{2.11}$$

In this configuration, it is interesting to notice that FORCE learning works even if the neurons do not all receive the same feedback. Moreover, training works in the feedback network, although the

connections from the generator to the feedback network are sparse. A small sample of the generator network activity can contain enough information to allow training. This rely on the accuracy of randomly sampling a large system described by Sussillo in [19].

2.2.1.3 Architecture C

Similarly, FORCE learning can be used with the architecture C described in2.1.3 where there is only a generator network and no explicit feedback loop. The modifications are applied to the recurrent connexions using the RLS rule with again the same error term $e_-(t)$ that is used to modify the read out weights :

$$J_{ij}^G(t) = J_{ij}^G(t - \Delta t) - e_-(t) \sum_{k \in B(i)} P_{jk}^i(t) r_k(t) \tag{2.12}$$

where

- $B(i)$ is the list of neurons from the generator network that are presynaptic to the neuron i in the generator network.
- P^i is a square matrix of size the length of $B(i)$ updated with the following equations:

$$P_{jk}^i(t) = P_{jk}^i(t)(t - \Delta t) - \frac{\sum_{l \in B(i)} \sum_{m \in B(i)} P_{jl}^i(t - \Delta t) r_l(t) r_m(t) P_{mk}^i(t - \Delta t)}{1 + \sum_{l \in B(i)} \sum_{m \in B(i)} r_l(t) P_{lm}^i(t - \Delta t) r_m(t)} \tag{2.13}$$

$$P^i(0) = \frac{1}{\alpha} Id \tag{2.14}$$

2.2.2 Reward Modulated Learning

The second learning algorithm used in this project was a reward-modulated Hebbian learning rule using a weak third factor described in 1.6.3.

2.2.2.1 Exploration Noise

This algorithm is used with the network architecture A described in 2.1.1.
The notations are the same except that some noise is added to the neurons firing rates r and to the network outputs z which is fed back to the network:

$$r_j = tanh(x_j(t)) + \xi_j^{state}(t) \tag{2.15}$$

$$z_j(t) = \sum_{i=1}^{N_G} W_{ij} r_i(t) + \xi_j(t) \tag{2.16}$$

where

- $\xi_j^{state}(t)$ is a zero mean noise drawn from a uniform distribution in the interval $[-0.05, 0.05]$
- $\xi_j(t)$ is the exploration noise drawn from uniform distribution in the range [-0.5, 0.5].

Contrary to FORCE learning, the exploration noise applied to the output during training is here necessary to make the learning possible.

2.2.2.2 Performance measure

To apply a reward-modulated learning rule, the system performance must be measured. The measure performance P is the sum of the mean squared errors of the network outputs:

$$P(t) = -\sum_{i=1}^{N_{out}} (z_i(t) - f_i(t))^2 \tag{2.17}$$

where

- $z_i(t)$ is the output produced by the i^{th} read out unit.
- $f_i(t)$ is the desired output for the i^{th} read out unit.

To measure the network recent performance a second variable $\overline{P}$ is introduced. It consists in a low-pass filtered version of P:

$$\overline{P}(t) = q\overline{P}(t - \Delta t) + (1 - q)P(t) \tag{2.18}$$

The usual value for q in this project is 80%.

2.2.2.3 Third factor

A binary third factor $M(t)$ is used, indicating whether the system performance has recently improved or not:

$$M(t) = \begin{cases} 1 & \text{if } P(t) > \overline{P}(t) \\ 0 & \text{if } P(t) \leq \overline{P}(t) \end{cases} \tag{2.19}$$

2.2.2.4 Update rule

The update rule of the output synaptic weights is given by:

$$W_{ij}(t) = W_{ij}(t - \Delta t) + \eta(t)(z_i(t) - \overline{z}_i(t))M(t)r_j(t) \tag{2.20}$$

where

- $r(t)$ is a vector containing the firing rates of the network's neurons,
- and $W_i(t)$ contains the corresponding synaptic weights from these neurons to the readout neuron i.
- $\eta(t)$ is a learning rate that can be constant or decay in time as learning saturates.
- $\overline{z}_i(t)$ is a low-pass filtered version of the noisy output $z_i(t)$.

3 Implementation

Songe qui s'évapore, ...

3.1 Organisation

The implementation was made using Matlab. Different scripts were written corresponding to the different experiments that were performed.
In each script the following variables must be defined:

- the time step dt used to run the networks (usually $dt = 1ms$).
- the network parameters (number of neurons, time constant τ, scaling factors, spareness parameters, numbers of inputs/outputs...)
- the learning and simulation durations (in millisecond).
- the different inputs time series that will be used for training and simulation.
- the target functions (ie the desired outputs) used during the training.

Then some functiosn are called to train et simulate the neural network. During the training and the simulation, the neurons activity is updated according to the differential equations presented in 2.1 that are solved with a first order Euler approximation.

3.2 Create Network

Three functions were implemented to create the networks corresponding to the three different architectures that are used during the project. These functions take the network specifications into parameters and return the structure *net* that contains the network's parameters. During this process the connectivity matrices are initialized according to their spareness parameters.
The three functions have the following signatures:

```
function net = CreateNetworkA(N,p,tau,K,N_in,g,g_z,dt)
function net = CreateNetworkB(Ng,Nf,pg,pf,pgf,pfg,tho,K,N_in,g_g,g_f,g_gf,g_fg,g_z,dt)
function net = CreateNetworkC(Ng,pg,tau,K,N_in,g,dt)
```

K is the number of readout units, i.e. the number of outputs. The other parameters correspond to those used in 2.1.

3.3 Train Network

3.3.1 FORCE learning

Four training function were implemented to apply FORCE learning to each of the three network architecture. The fourth function apply FORCE learning to the network of architecture C, in the particular case when the generator network is fully connected. In this case, the training algorithm is highly simplified because all the neurons share the same matrice P.
The four functions *TrainNetworkForce*, *TrainNetworkForceB*, *TrainNetworkForceC* and *TrainNetworkForceCall2all* have the following sinature:

```
function [ net, traindata, P ] = TrainNetwork( net, P, F_targets, input)
```

The struct *traindata* returned after training contains different time series taken during training: the network output Z (matrix of size $(K, learning_time)$) , the neurons' activities X (size $(N, learningtime)$) and the norms of each readout unit's weights update $||dw||$ (size $(K, learningtime)$).

3.3.2 Reward modulated learning

The reward modulated learning algorithm is implemented in the function:

```
function [ net, traindata, l] = TrainNetworkReward( net, F_targets, input,l_0, Tc )
```

where l_0 is the initial learning rate and Tc the learning rate decay constant.
The learning rate η is updated as follow:

$$\eta(t) = \frac{\eta_0}{1 + \frac{t}{T_c}} \tag{3.1}$$

The struct traindata returned by the function contains the same fields than for FORCE learning.

3.4 Run Network

Several functions were implemented to run the different networks architectures.

3.4.1 Run network A-B-C

The three functions *RunNetwork*, *RunNetworkB*, *RunNetworkC* that simulate the neural networks for the different architectures have the following signature:

```
function [net, rundata ] = RunNetwork( net, input, time )
```

The struct *rundata* returned by the functions contains the network outputs Z, the neurons' activities X.

3.4.2 Drone simulation

For the two last experiments with the drone simulation, we define the functions *RunNetwork2*, *RunNetwork3* in which two variants of *RunNetwork* are implemented. *RunNetwork2* run the simulation for a network of architecture A but the input that is given to the network is the difference between the drone position and the usual input (which is th eposition of a target teh drone tries to follow). The drone position is updated at each time step using euler approximation, the speed being given by the network output.

3.5 Inputs/Outputs Definition

N_{in}, the number of network inputs, must be at least equal to 1 to avoid dimension mismatch issues. If there is no input we simply set the input value to 0.

3.5.1 Static inputs

When some static (stationary) inputs are applied to the Network architecture, the constant values are stored in a vector of size N_{in}

3.5.2 Dynamic inputs

When the Network is fed with continuous dynamic inputs, the inputs' continuous values is stored in a matrix of size $(N_{in}, time)$ where each line correspond to the value of one of the input over time.

3.5.3 Target functions

The desired outputs that we want the network to produce through the read out units are stored in a matrix of size $(K, learningtime)$ where the i^{th} line correspond to the desired value for the i^{th} read out unit over the learning time.

4 Generating periodic patterns

... étincelle qui fuit !

The first experiments consisted in training the networks with the two algorithms described in 2.2 to produce different periodic patterns. In this part we used some networks of architecture A with respectively 500 and 1000 neurons with FORCE learning and reward modulated learning, and a rewiring probability of 0.1 with both algorithms.

4.1 No input

We first consider some neural networks without incoming current. The self-sustained activity is then due to the recurrent architecture.

4.1.1 Single output

4.1.1.1 Simple sinusoid

We trained the network to produce a simple sinusoid function. The figure 4.1 shows the network output during the training. We can see that it matches perfectly the target from the beginning of learning. As expected the norm of the weights update decreases over time to stay close to 0 after 5 seconds.

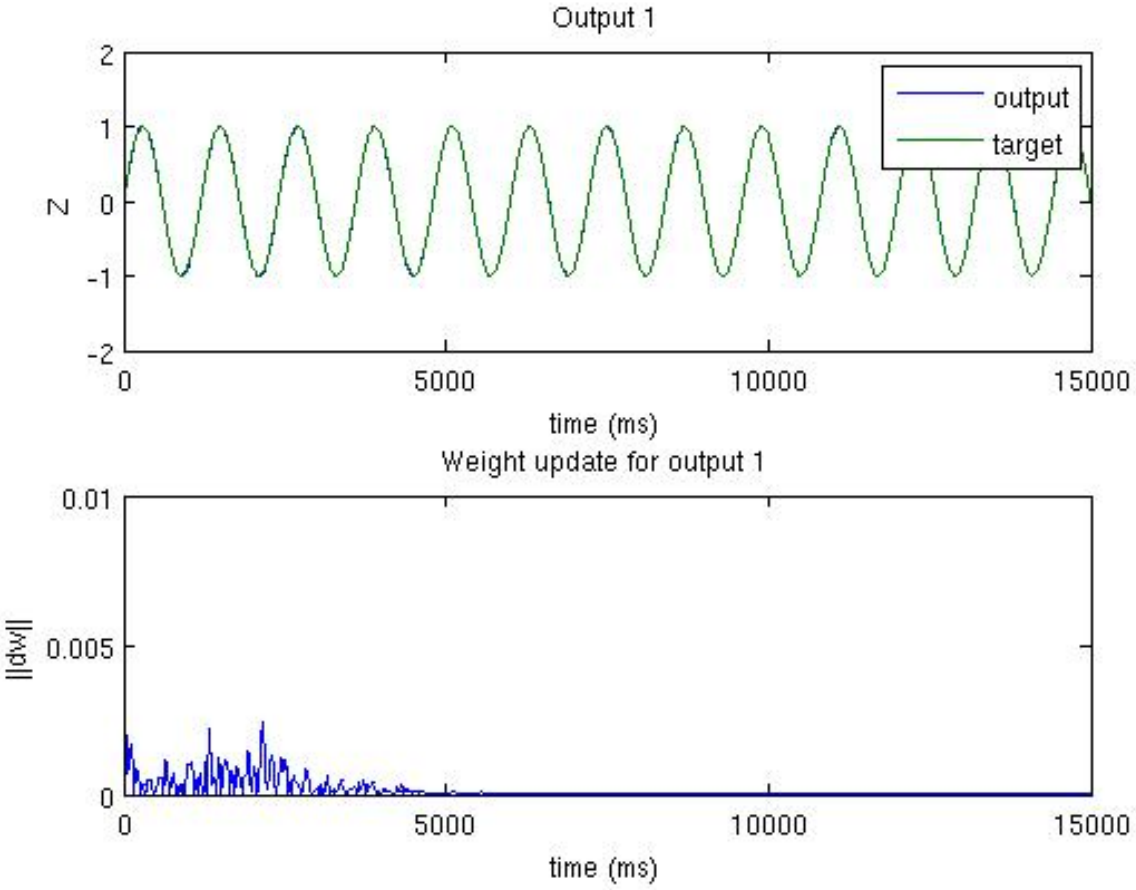

Figure 4.1: FORCE Learning of a sinusoid pattern

The figure 4.2 shows the network output after learning. Before the simulation the neuron's membrane potential are reinitialized to random values. The output (in blue) quickly converge to the desired periodic pattern. The error cannot be easily measured because of the phase difference between the real output and the desired target function.

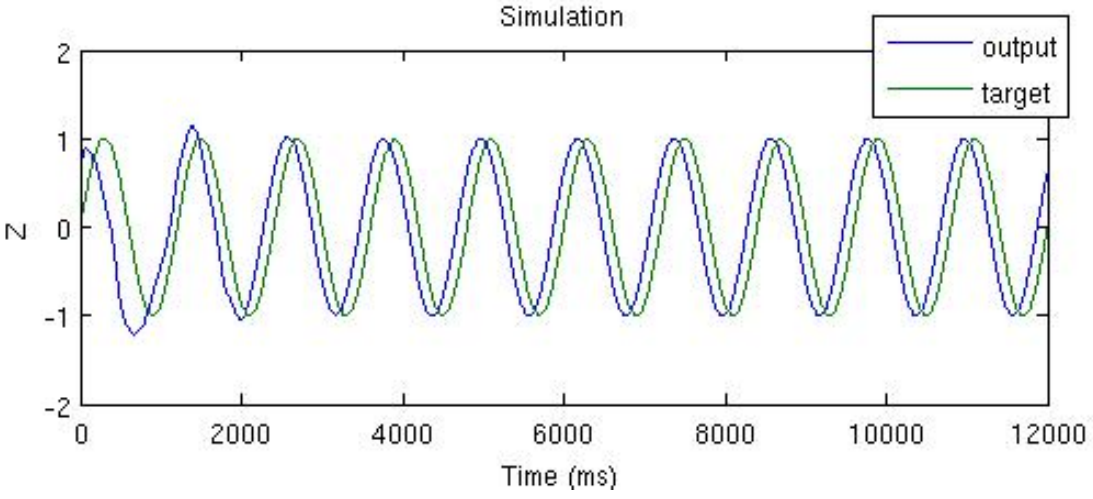

Figure 4.2: Post-learning simulation

4.1.1.2 More complex target

The second step consisted in training the network to produce a more complex periodic function. The target function was defined according to one of the experiment in [2].

$$f(t) = sin(2\pi t)\frac{1.3}{1.5} + sin(4\pi t)\frac{1.3}{3} + sin(6\pi t)\frac{1.3}{9} + sin(8\pi t)\frac{1.3}{3}; \tag{4.1}$$

FORCE learning
Applying FORCE learning to the match this more complex desired output leads to similar results than with the simple sine wave. The only difference being that it take twice the time (10 seconds) for the weight update to converge to zero as shown on figure 4.3.
After the training and the neuron's potential random reinitialization, the network output converge to the desired pattern as shown on figure4.4.

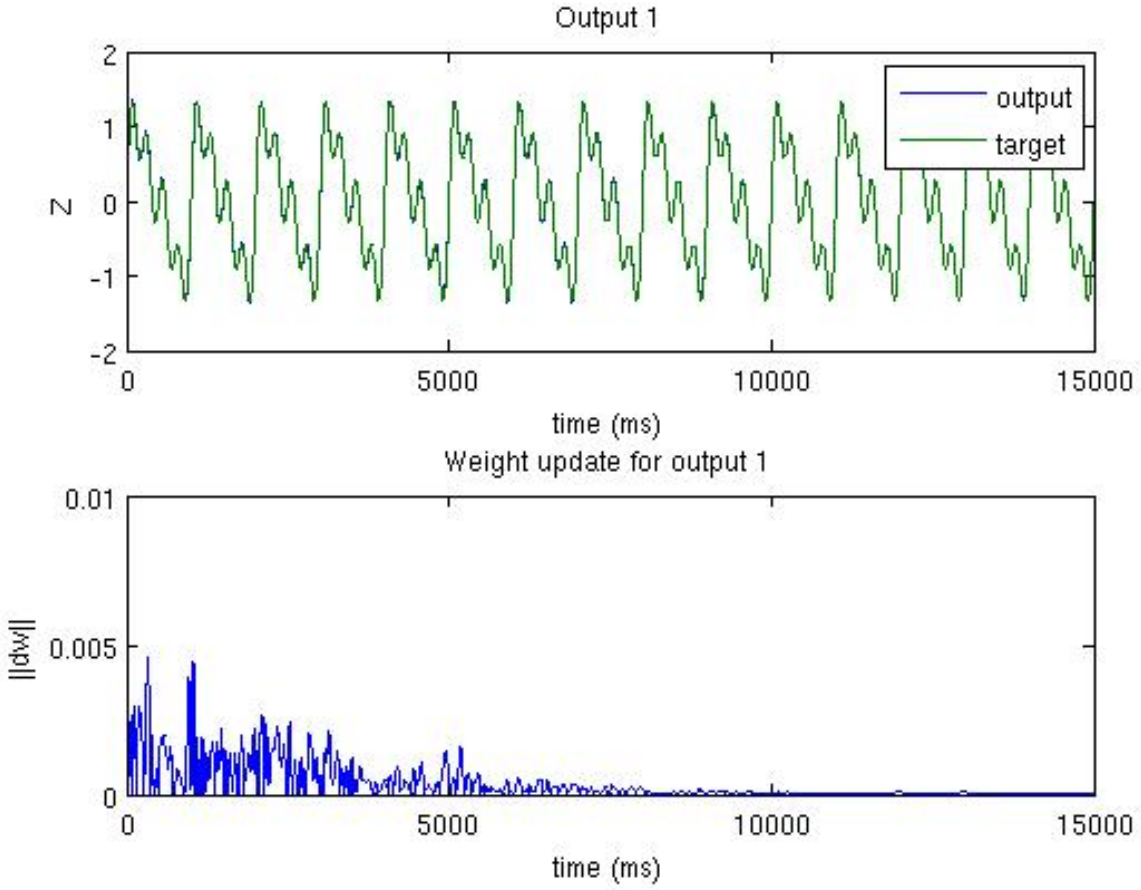

Figure 4.3: FORCE learning of a complex periodic pattern

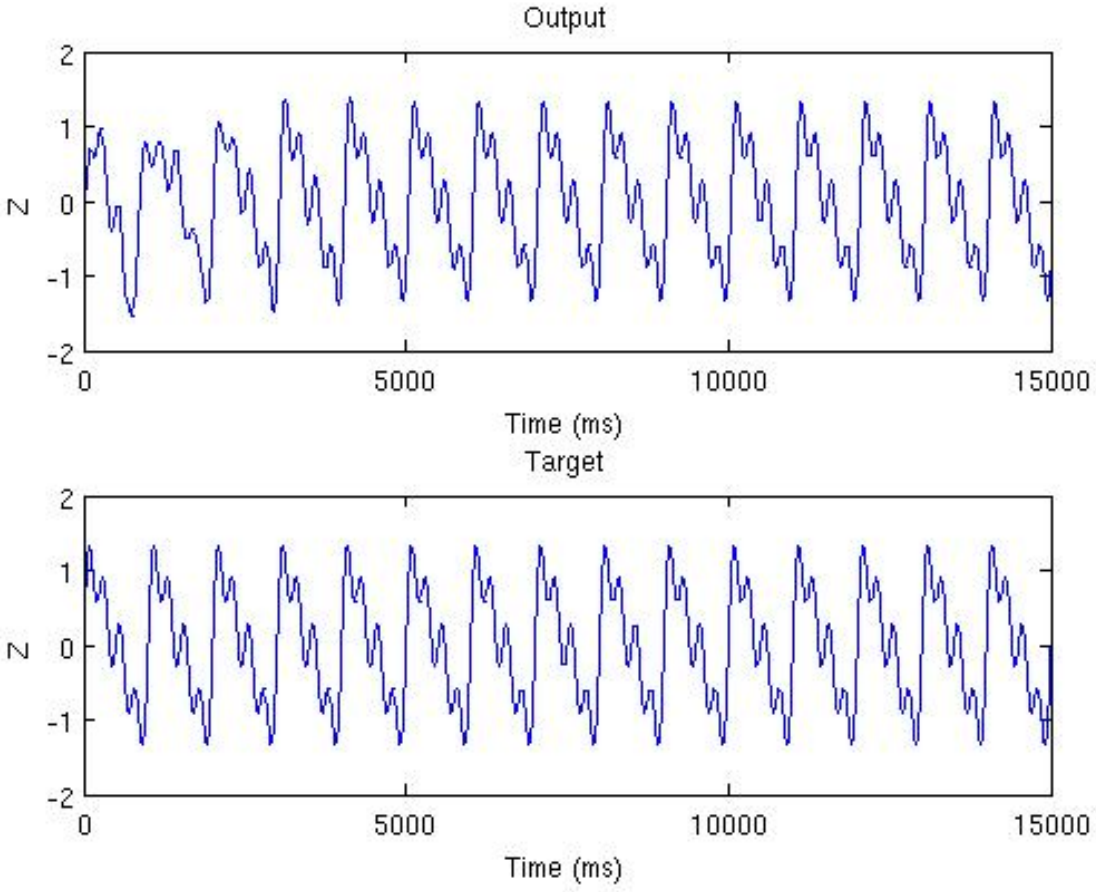

Figure 4.4: Post-learning simulation of a network trained with FORCE learning to produce a complex pattern

Reward learning

We also tried to apply the reward modulated learning rule to learn the same complex pattern. The figure 4.5 shows the output evolution during training. The first two plots represent the output during the 5 first and last seconds of learning. We can see that the output progressively follows the desired target function. The training duration is however longer than with FORCE learning (a learning of at least 100 s was necessary to obtain reasonable results in this case).

The figure 4.6 shows the exploration noise which is added to the feedback loop only during training. The network was tested after learning without the neurons' potentials being reinitialized. In this case the network follows the desired pattern but with a slight difference in the signal frequency which explains the separation of the two lines that represent the real and the desired outputs in figure 4.7.

When we tried the run the network after reinitializing the membranes potential. The network often fail to converge to the desired output as shown on figure 4.8 where it converges to a similar but still different pattern. The figure 4.9 shows the case where the network succeed to converge to the desired pattern.

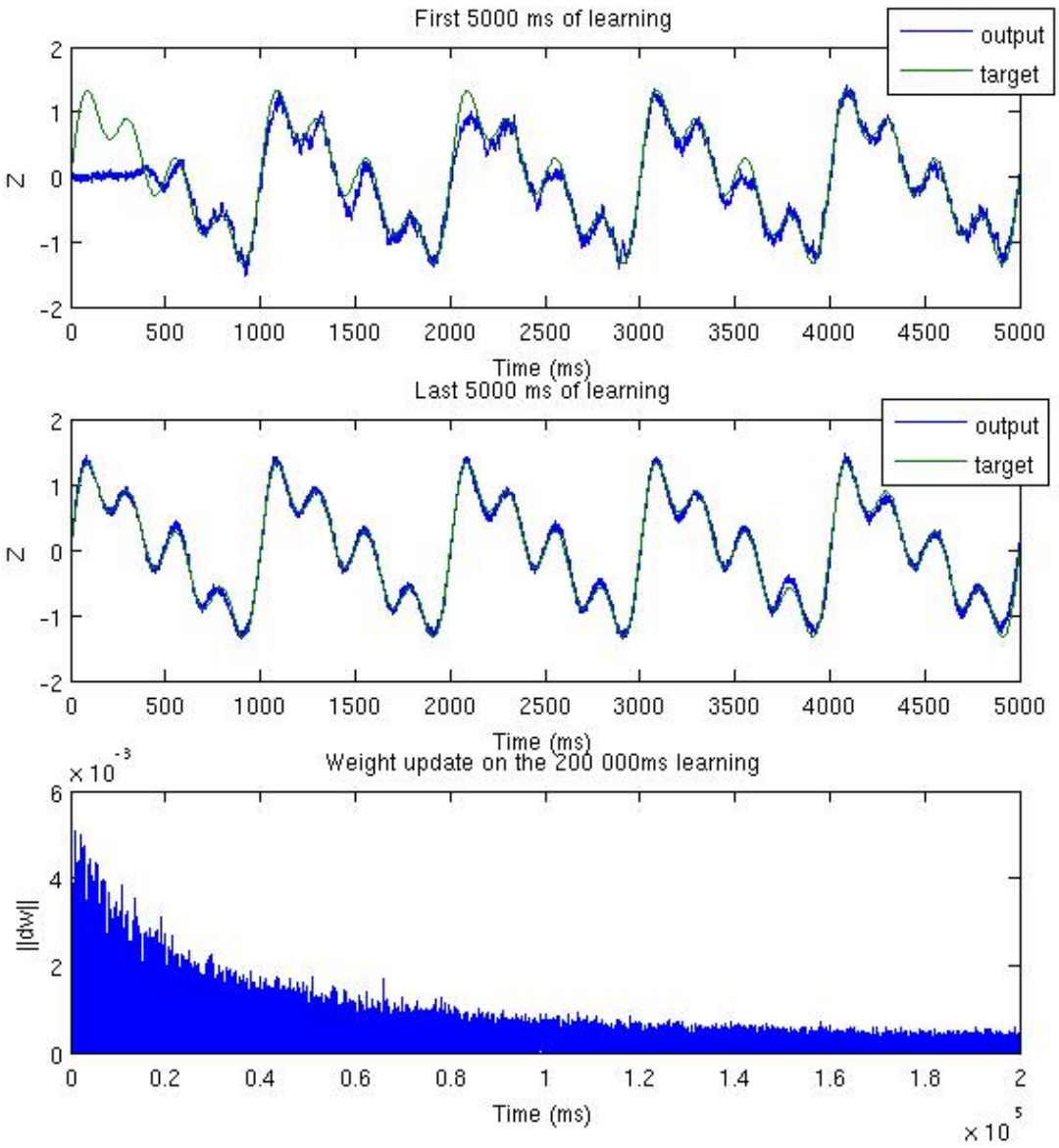

Figure 4.5: Reward modulated learning of a complex periodic pattern

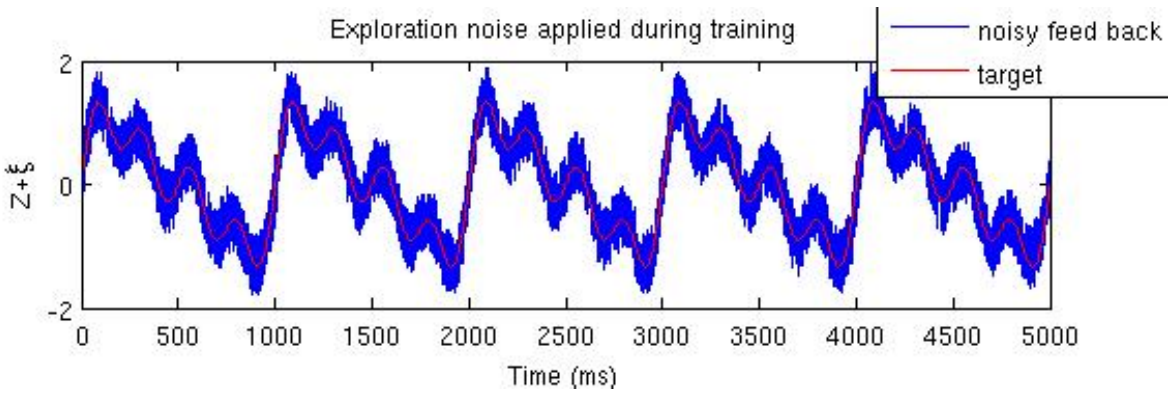

Figure 4.6: Exploration noise added to the feedback loop during reward modulated learning

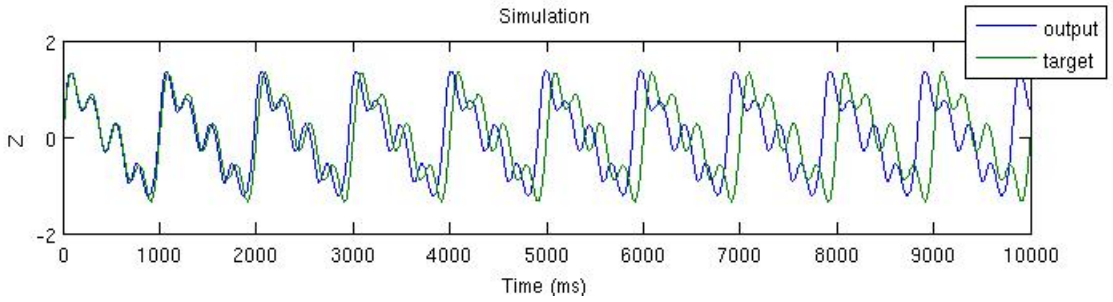

Figure 4.7: Post-learning simulation of a network trained with reward learning to produce a complex pattern, without reinitialization of the neurons potentials

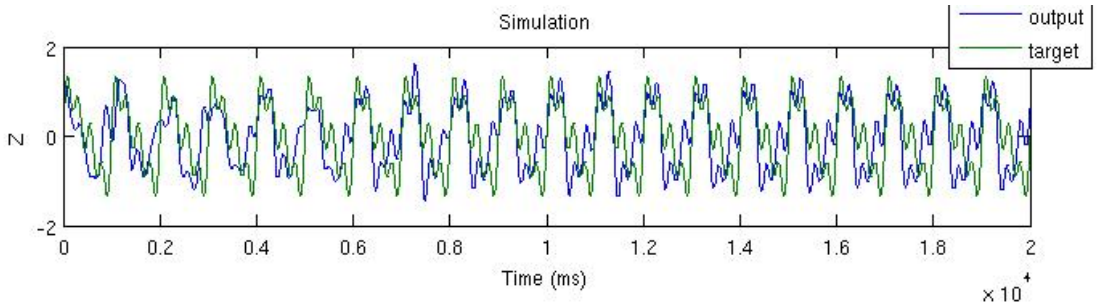

Figure 4.8: Post-learning simulation of a network trained with reward learning. Convergence failure

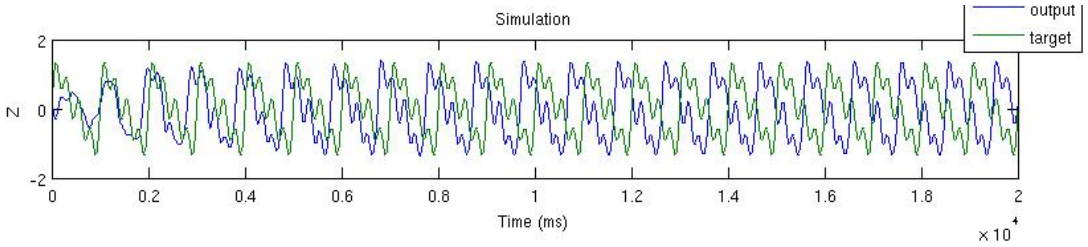

Figure 4.9: Post-learning simulation of a network trained with reward learning. Convergence success

4.1.2 Multiple outputs

The next step of our experiments was to train the network to produce multiple outputs at the same time with the FORCE learning algorithm, still with no incoming current. Each output is associated to a different readout unit whose weights are independently updated (they share the same matrix P but different error terms e in the equation 2.5.
We trained the network with the two following target functions:

$$f1(t) = 1.5sin(2\pi t) \tag{4.2}$$
$$f2(t) = 0.7cos(5\pi t) \tag{4.3}$$

The figures 4.10 and 4.11 shows the results of the training which are similar to the single output case. Both outputs match their target functions during the training and converge quickly to it during during the simulation.

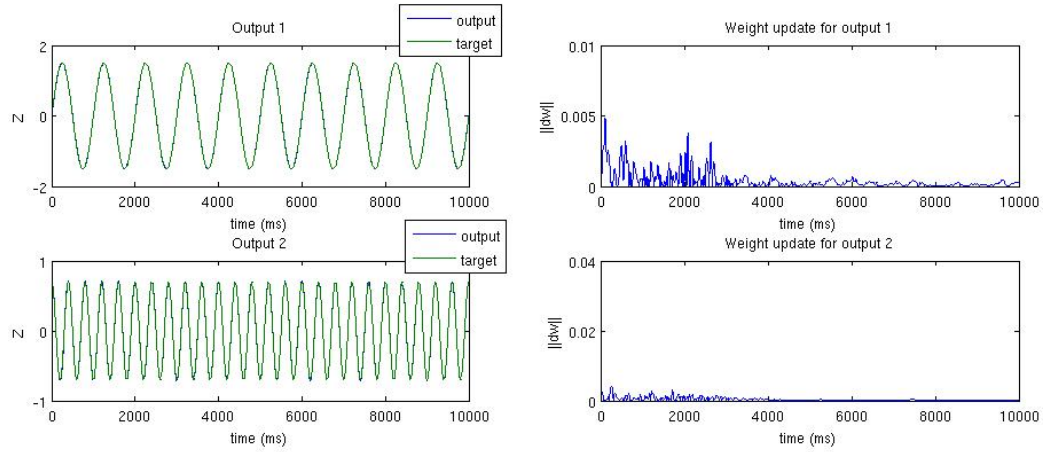

Figure 4.10: FORCE learning of 2 outputs simultaneously

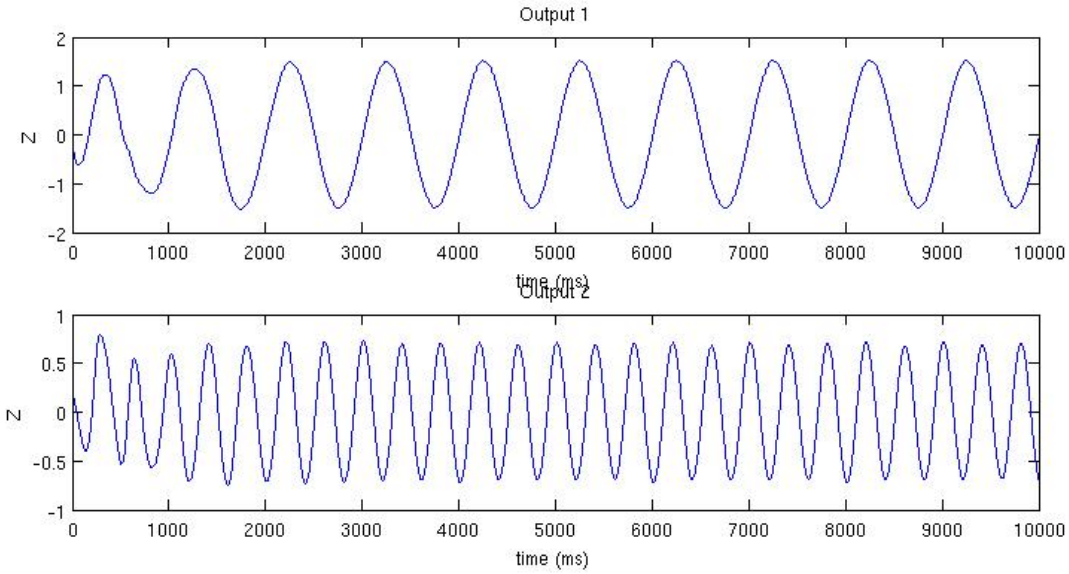

Figure 4.11: Post learning simulation of the 2 outputs network trained with FORCE learning

4.2 Input/output matching

In the last experiment of this chapter we consider a network of 1000 neurons with two read-out units that receive an incoming current *input*. We trained the network using FORCE learning in order to match two different behaviours with two different input values.
The input values $i1$ and $i2$ are randomly generated in the range $[-1, 1]$. For the learning to succeed the two values must however be significantly different. Each input was associated with the two desired outputs as follow:

when $input = i1$:

$$f1(t) = 1.0sin(\frac{5}{3}\pi t) \tag{4.4}$$

$$f2(t) = 0.5cos(5\pi t) \tag{4.5}$$

when $input = i2$:

$$f1(t) = 1.5cos(\frac{5}{2}\pi t) \tag{4.6}$$

$$f2(t) = 0.6sin(\frac{5}{4}\pi t) \tag{4.7}$$

We applied the training algorithm once for each input/outputs configuration with a learning time of 5 seconds.
The figures 4.12 to 4.15 shows the results of this experiment.

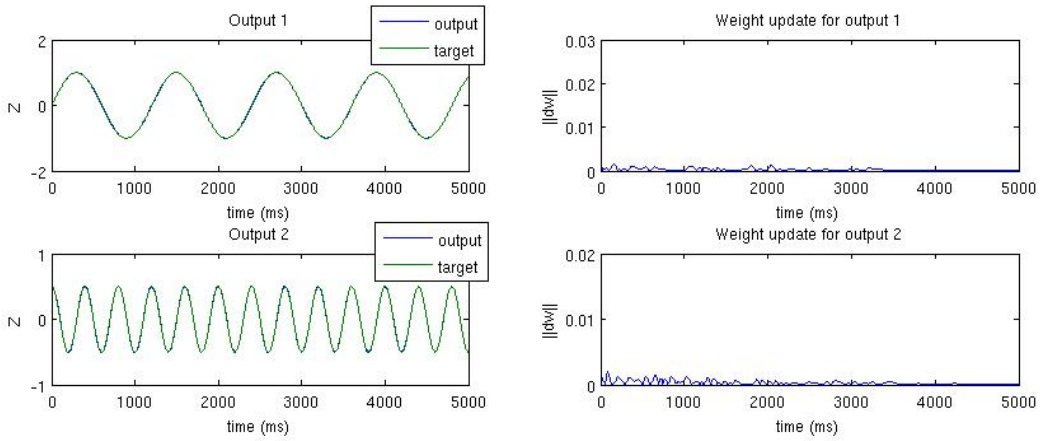

Figure 4.12: First training ($input = i1 = 0.5$)

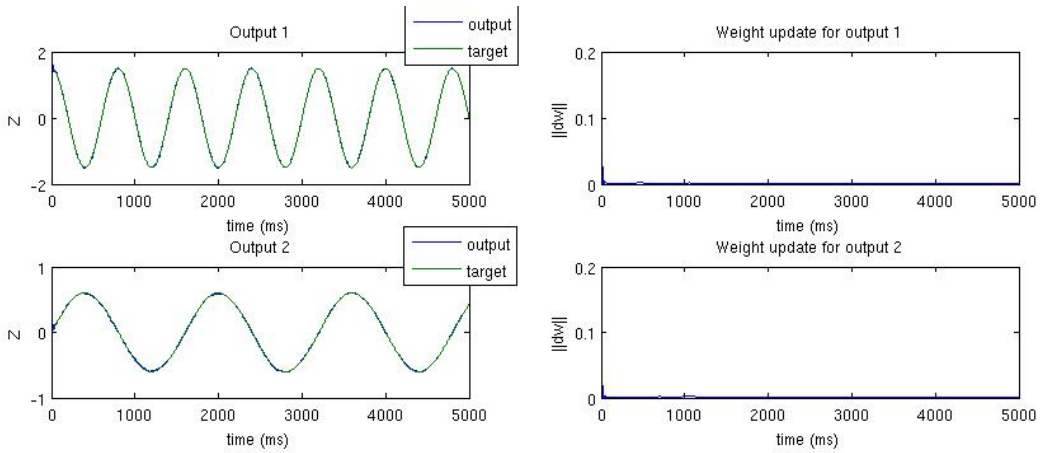

Figure 4.13: Second training ($input = i2 = -0.8$)

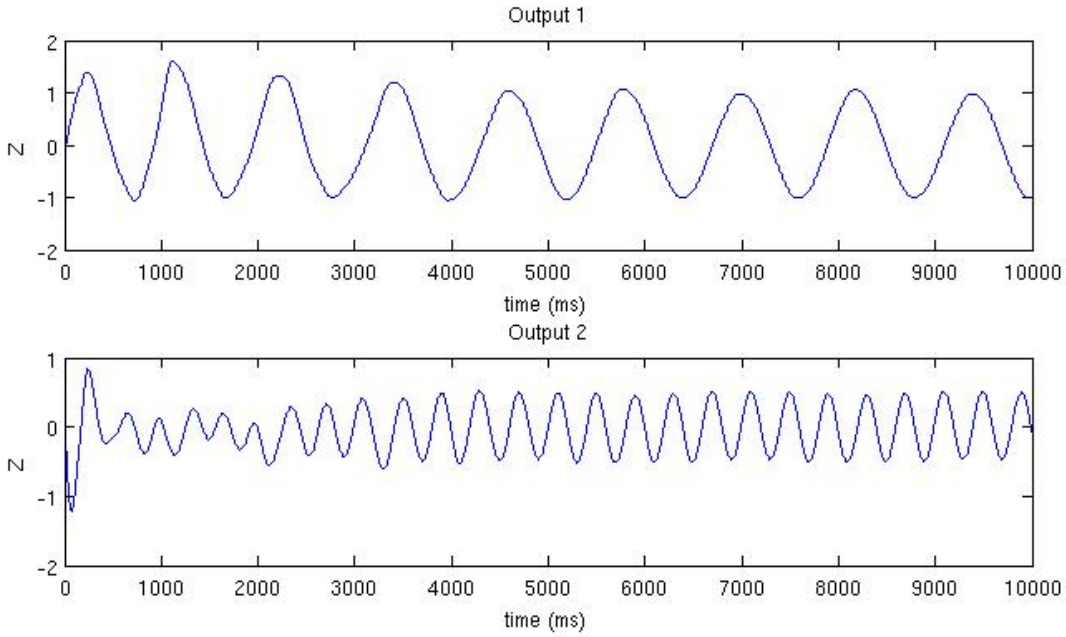

Figure 4.14: Post-learning simulation of input/outputs matching ($input = i1 = 0.5$)

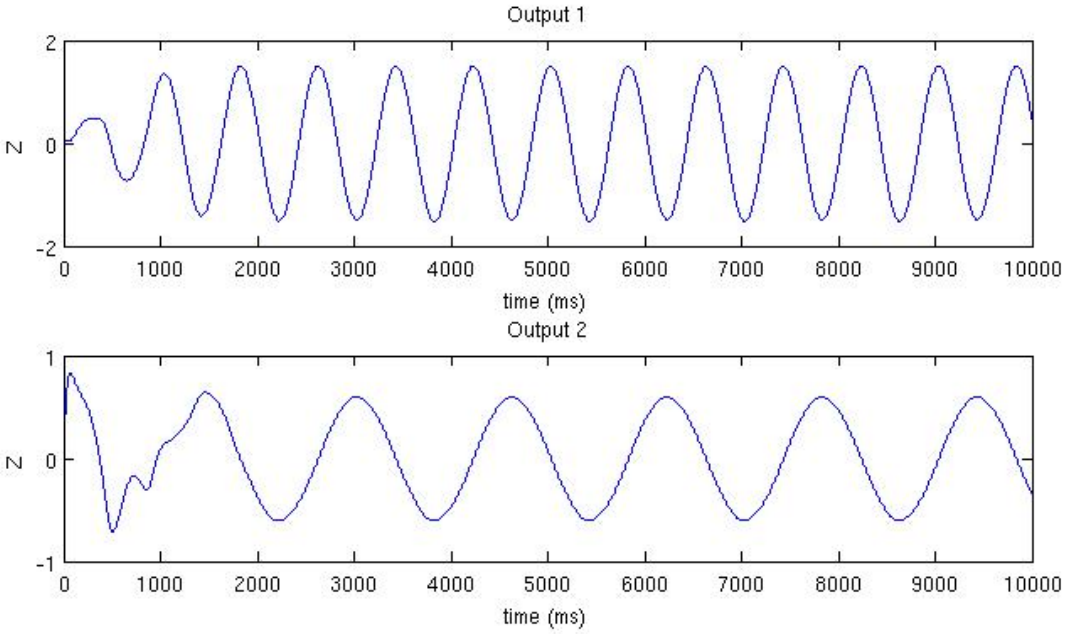

Figure 4.15: Post-learning simulation of input/outputs matching ($input = i2 = -0.8$)

4.3 Conclusion

This part presented the first replications of the results of [1] in the chronological order of their implementation. The most interesting results are the one of the last experiment that a fortiori imply the previous ones. It shows that FORCE learning can be applied to the generation of multiple non similar outputs(different frequency and amplitude) and that inputs can be used to switch between different behaviours.
Regarding the reward-modulated learning, although the first results (4.1.1.2) are convincing we encountered some trouble when it came to the multiple output generation. We focus on the following chapters on FORCE learning algorithms.

Regarding the two other network architectures presented in 2.1.2 and 2.1.3, we tried to obtain the same results as those presented above with more or less success. In those two network configurations the training algorithms are really slower than with the architecture A because of the numerous matrices that need to be updated at each iteration (respectively N_G and N_F matrices P^i for architectures B and C). That's why in the following part we only used the network of architecture A (see Figure 2.1).

5 Application to a robot arm

Éclair qui sort de l'ombre ...

5.1 Robot description

In this chapter we consider a robot arm with two joints. Each arm section is defined by its length ($L1$, $L2$) and the first joint is fixed in 2D space in which the robot evolves (the first joint coordinates are $(0, 0)$). The degree of freedom of each angle allow the robot to describe a wide range of movements in the 2D space. We name v and w the angles between the two joints and the horizontal line.
The figure 5.1 shows the robot arm with the two angles.
The position (x, y) of the robot arm's end evolve according to the following equation:

$$x = L1.cos(v) + L2.cos(v + w) \tag{5.1}$$
$$y = L1.sin(v) + L2.sin(v + w) \tag{5.2}$$

To inverse this equation and obtain the angle from the position we use the following equations:

$$k = \frac{x^2 + y^2 - L1^2 - L2^2}{2.L1.L2} \tag{5.3}$$
$$w = atan2(\sqrt{1 - k^2}, k) \quad [2\pi] \tag{5.4}$$
$$v = atan2(y, x) - atan2(L1 + L2.cos(w), L2.sin(w)) \quad [2\pi] \tag{5.5}$$

In this chapter we consider a neural network of 1000 neurons with 2 readout units. The idea is to associate each joint angle with one of the network output. By learning periodic patterns we can make the robot arm describe some periodic movement.

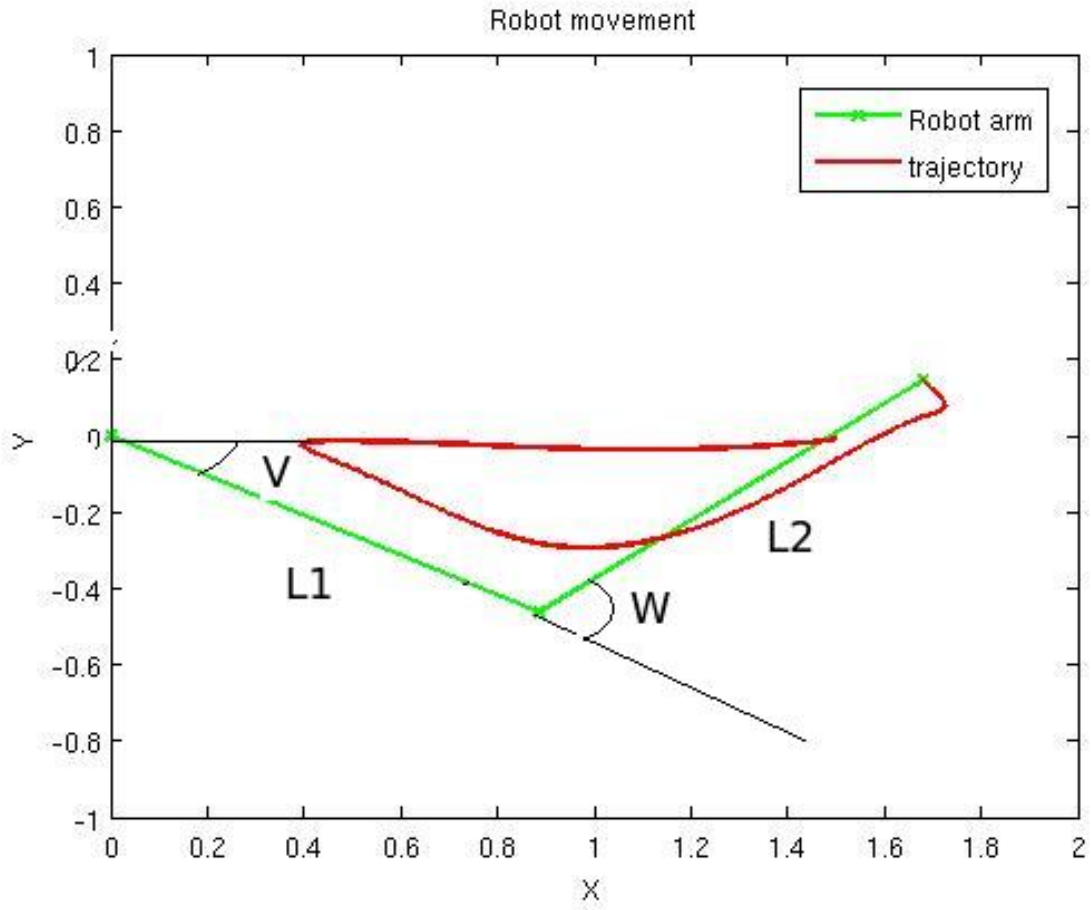

Figure 5.1: Robot arm description

5.2 Matching inputs with different movements

In the first experiment we train the network to produce 3 pairs of outputs depending on one input that take 3 different values ($i1$,$i2$,$i3$) that are randomly generated in the range ($[-1, 1]$). This experiment is similar to the one in 4.2, except that we learn 3 pairs of output instead of two, and that the target functions are more complex.
The 3 pairs of outputs that are trained correspond to 3 trajectory: a square, a circle and a loop of different amplitude and centred on the same point. The target functions are obtained from the desired trajectory with equations 5.3 and 5.5. After the simulation, the trajectory that is effectively followed by the arm is obtained from the network outputs with equations 5.1 and 5.2.
The figures 5.2 to 5.4 show the training data, and the figures 5.5 to 5.7 show the results of the simulation after the training. Figure 5.8 shows the actual trajectory during the post-learning simulation. We can see that when the input changes (different color on the figure) the robot converge to the desired movement that it then repeated until the input changes again. By using a different number of neurons we can get more precise result but the learning will be longer. For example it is possible to improve the square trajectory,especially at the corners.

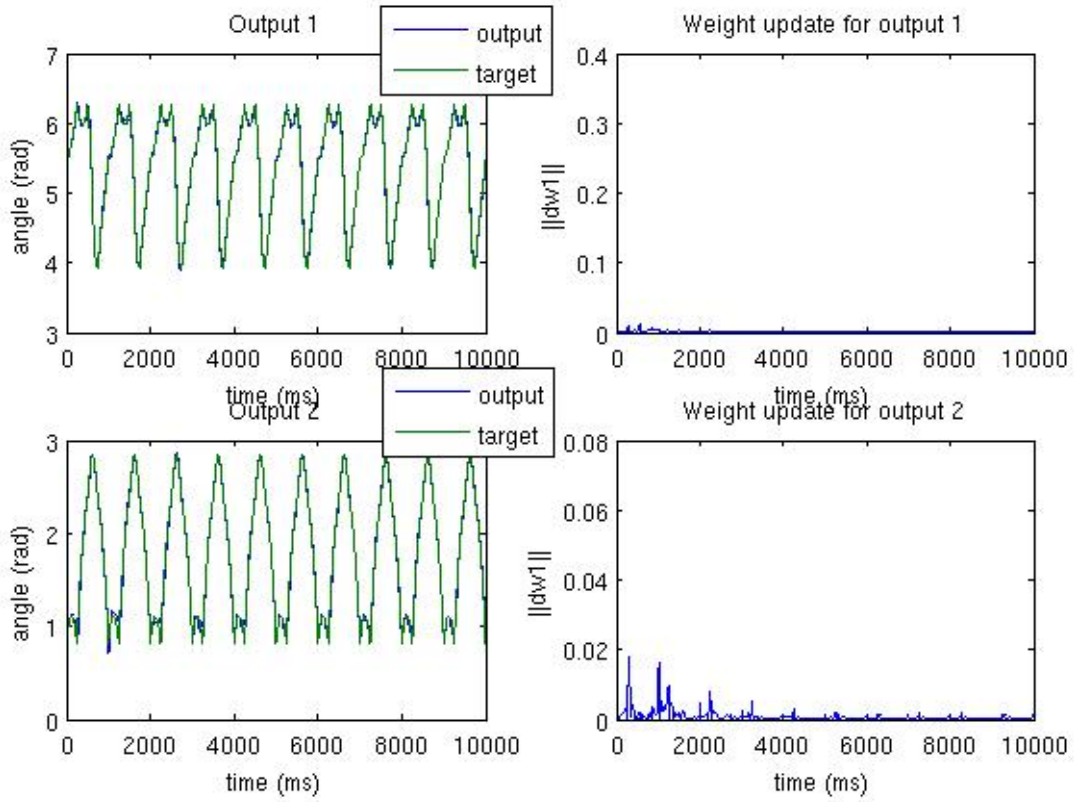

Figure 5.2: FORCE learning of joint angles v and w to produce a squared movement

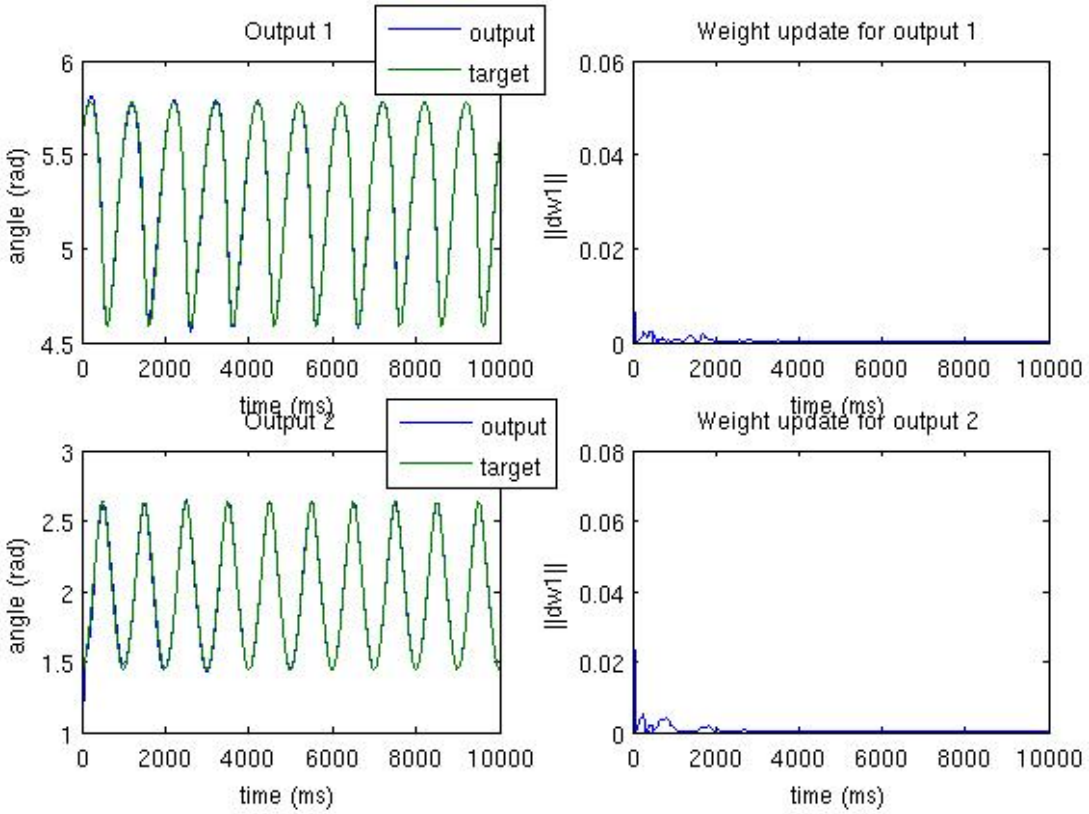

Figure 5.3: FORCE learning of joint angles v and w to produce a circular movement

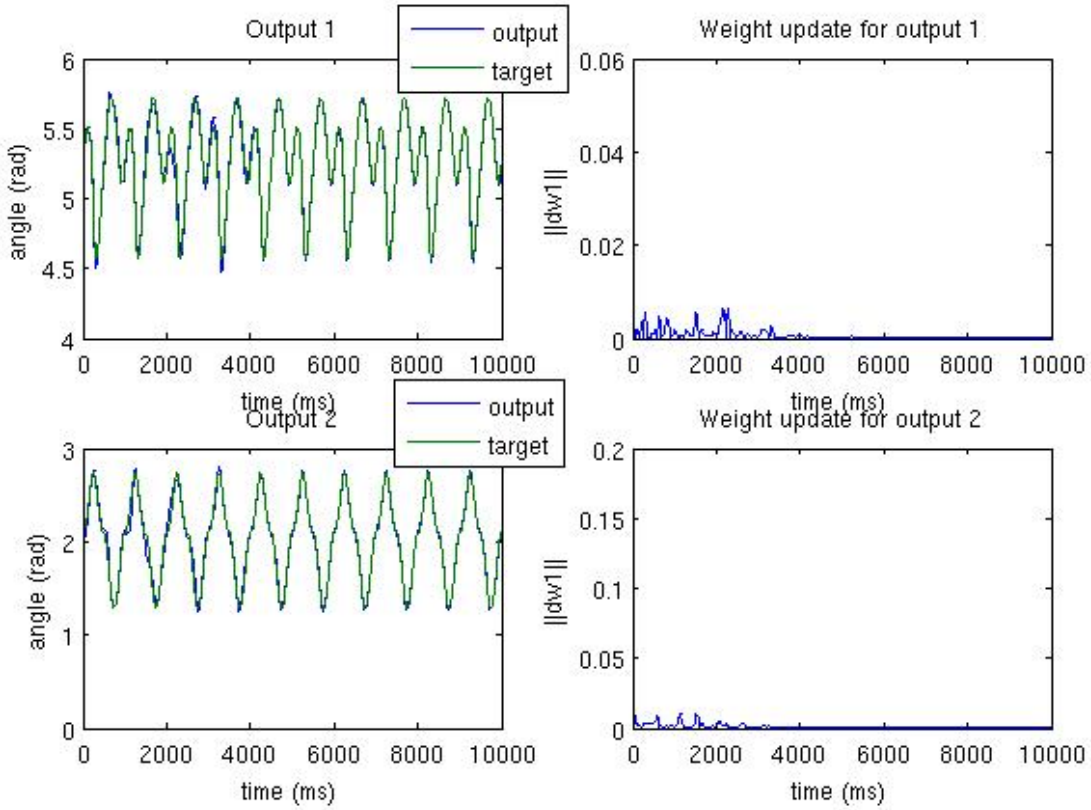

Figure 5.4: FORCE learning of joint angles v and w to produce a loop movement

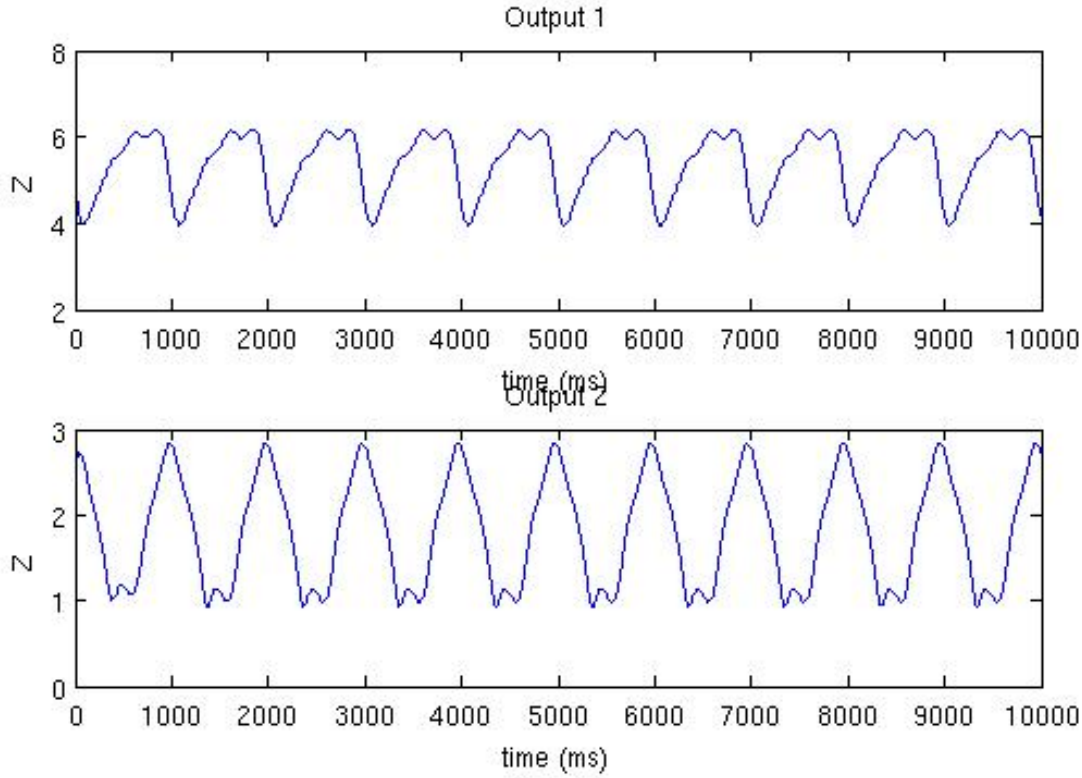

Figure 5.5: Post learning simulation ($input = i1 = -0.5$)

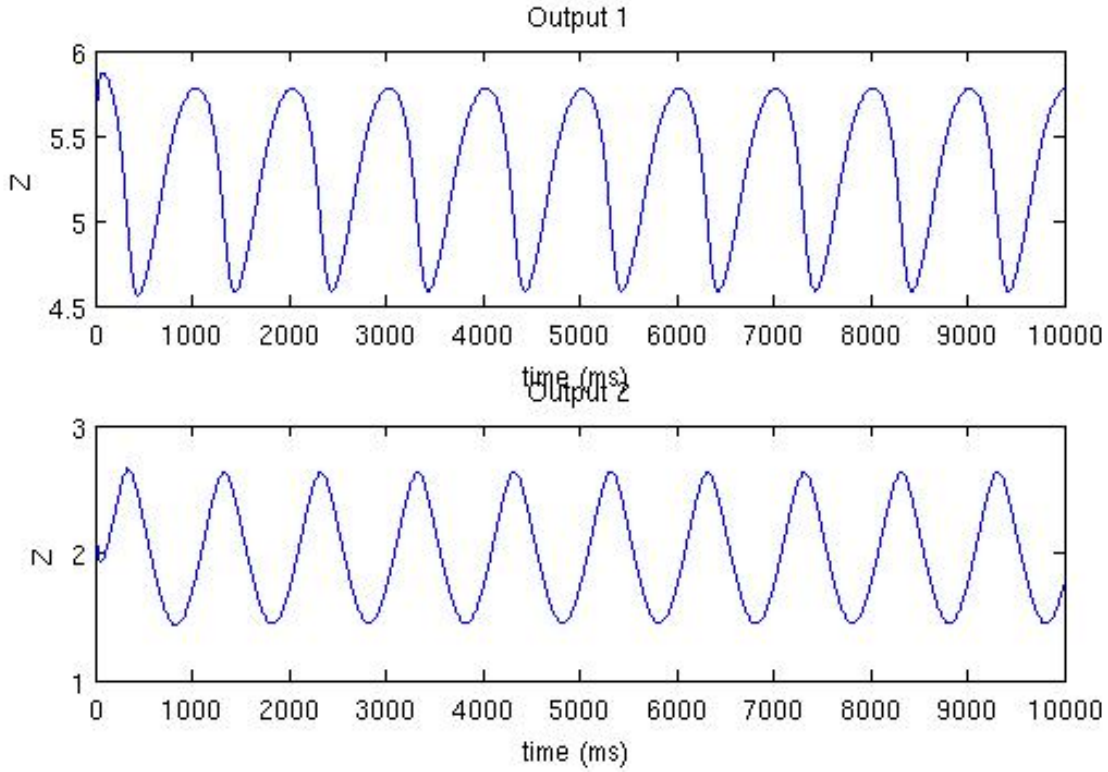

Figure 5.6: Post learning simulation ($input = i2 = 0.6$)

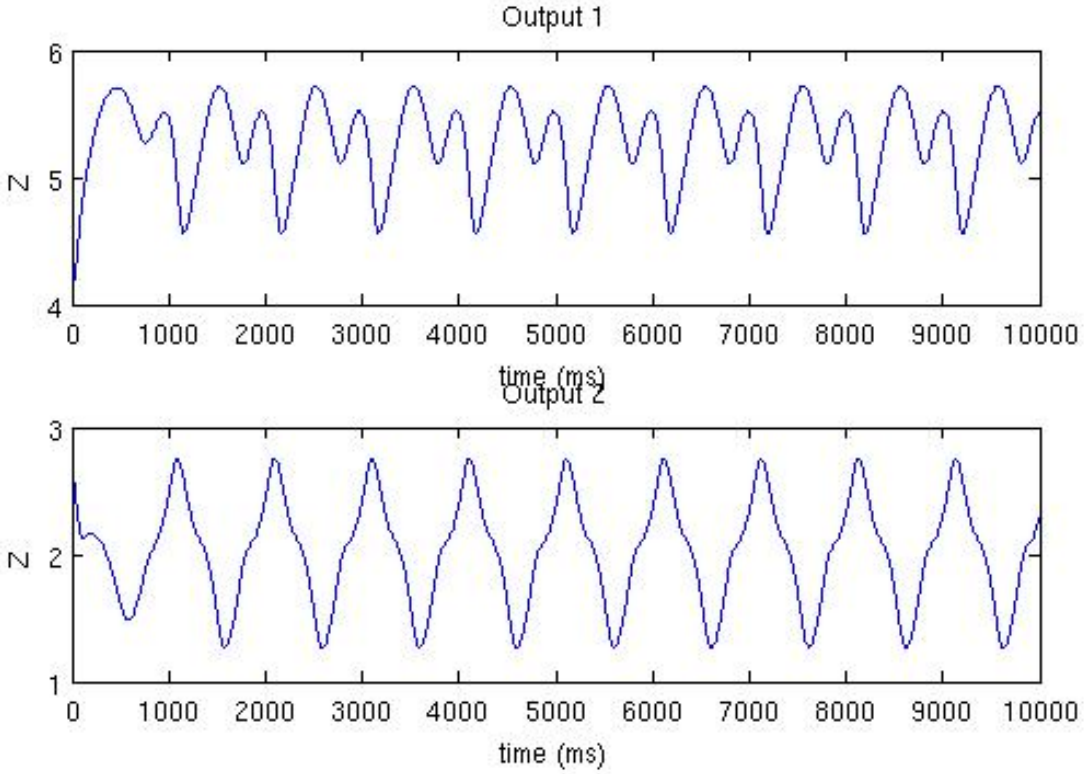

Figure 5.7: Post learning simulation ($input = i3 = 0.2$)

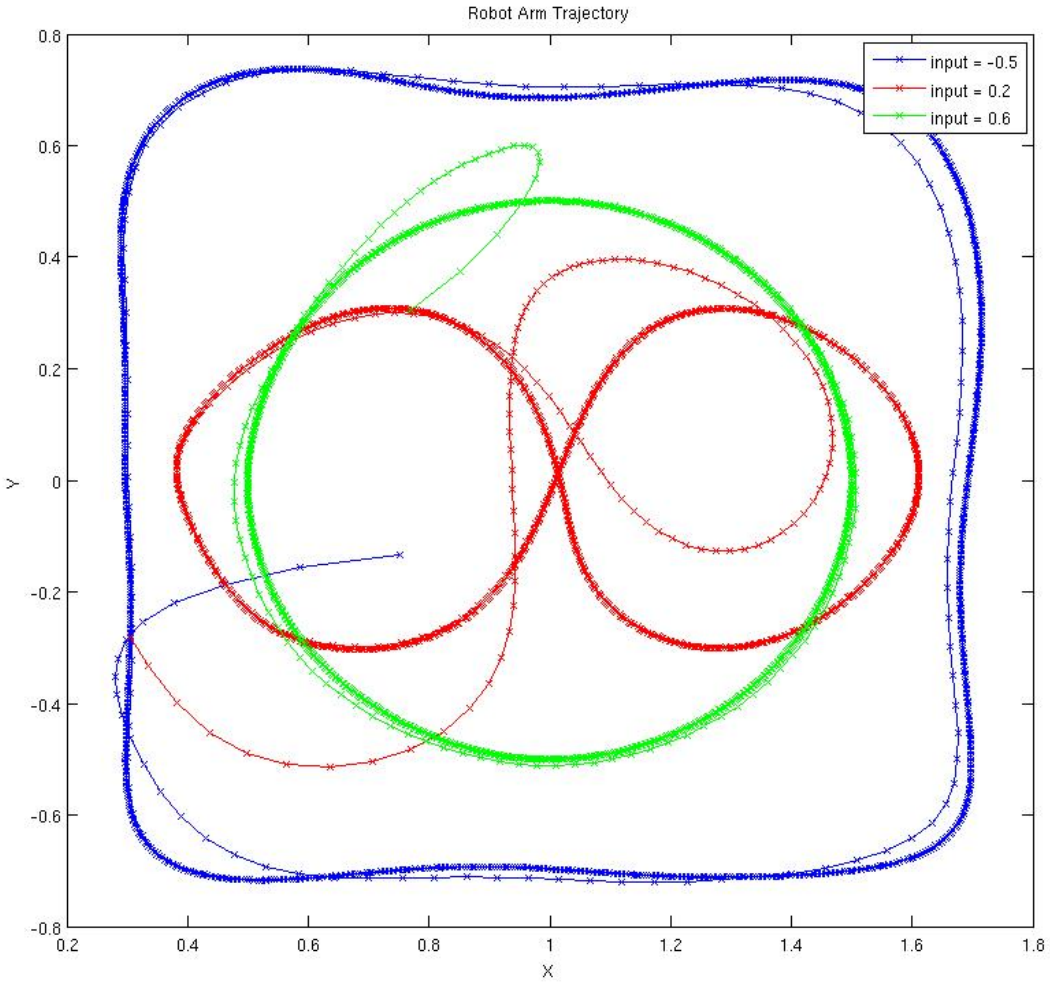

Figure 5.8: Robot arm trajectory during the simulation

5.3 Inputs interpolation to produce new movements

In the second experiment, we focused on one shape and trained the network to produce circles of different amplitudes depending on the input. In this case we trained the network with 4 values that were linearly spaced and no longer randomly generated. During training, the inputs were associated with the desired amplitudes that were also linearly spaced. By doing this we managed to produce new desired movements that hadn't been taught during training by running the network with new inputs whose value were between the values used during training. It shows that the network learns to produce a circle whose radius is proportionate to the input.
To make things clearer the table 5.1 presents the input values that were used during training and simulation and the associated desired radius. The first 4 values correspond to the one that are used during the training where the desired radius is known via the provided target functions. The last three values are the one used during the simulation. In the last three cases the associated radius is the radius that we expect to obtain but is unknown to the network.

Input	-0.3	-0.1	0.1	0.3	-0.2	0	0.2
Radius	0.2	0.4	0.6	0.8	0.3	0.5	0.7

Table 5.1: Input and associated circle radius

The figure 5.9 shows the result of the simulation. The circles in blue were learned during training. On the contrary the three others circles are produced when the network is run with a new input value. In these cases the new circles are still proportionate to the input. The figure 5.10 shows only these new circles with the expected radius (in blue).

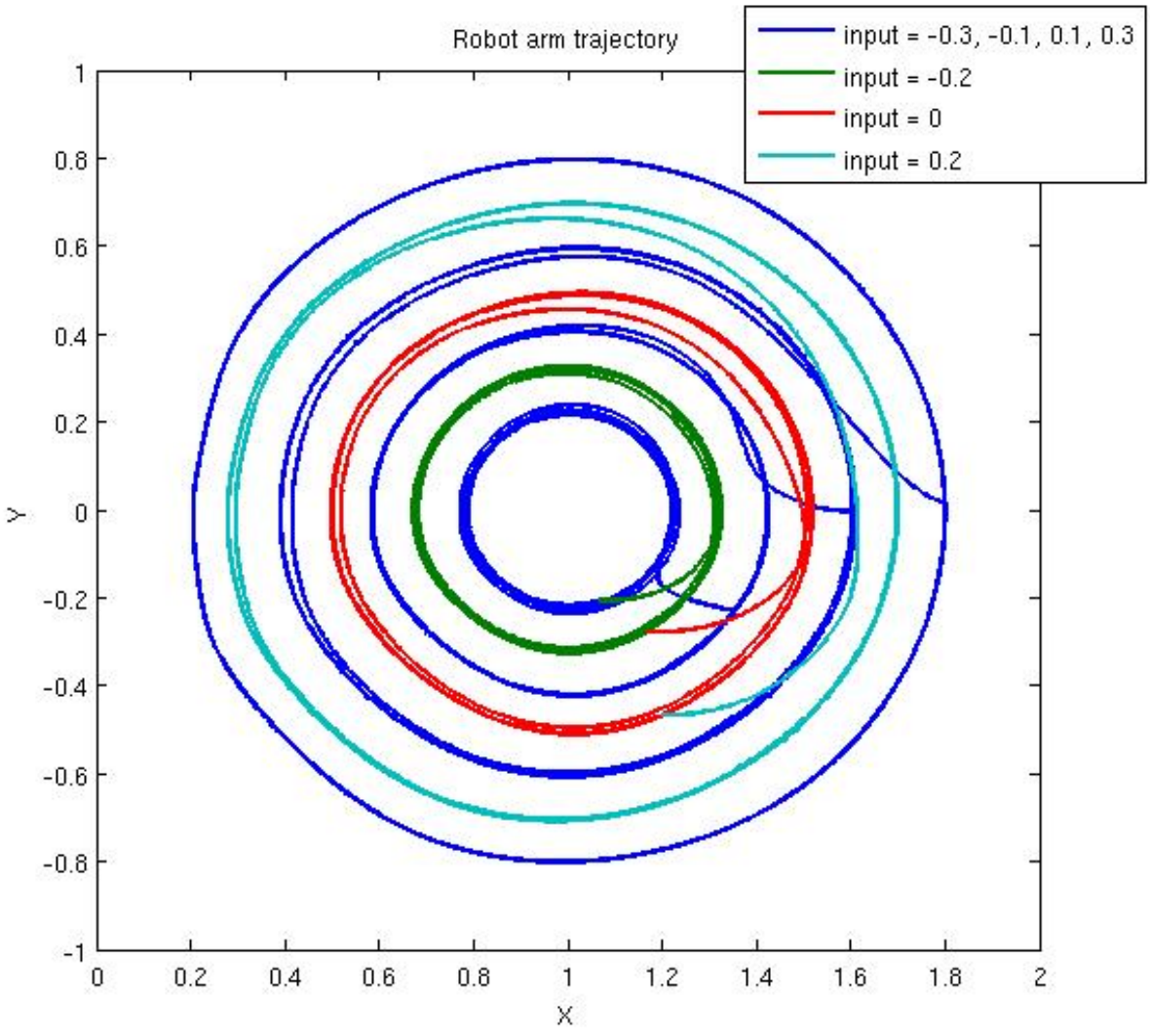

Figure 5.9: Interpolation to produce new circles

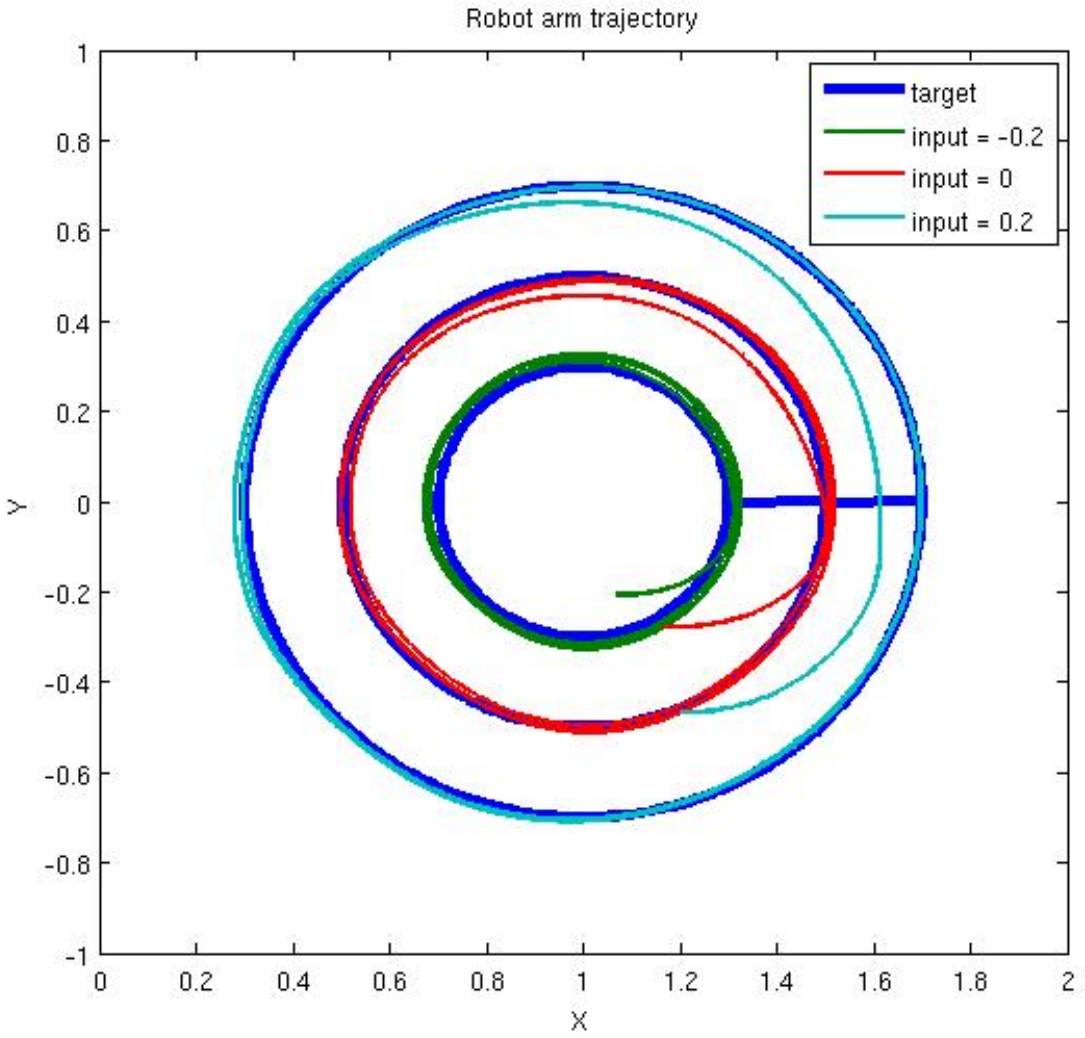

Figure 5.10: Producing new circles with the radius depending on the input

After obtaining these results we tried to reproduce the experiment with a different (more complex) shape. The table 5.2 similarly shows the inputs that were used with the associated/expected square side lengths. The figure 5.11 shows the shapes that are obtained against the expected one (in blue). Although the results are not as accurate as with the simple circular movement, the network still succeed in producing a new movement close to the expected one when run with a new input.

Input	-0.3	-0.1	0.1	0.3	-0.2	0	0.2
Side length	0.2	0.3	0.4	0.5	0.25	0.35	0.45

Table 5.2: Input and associated square side length

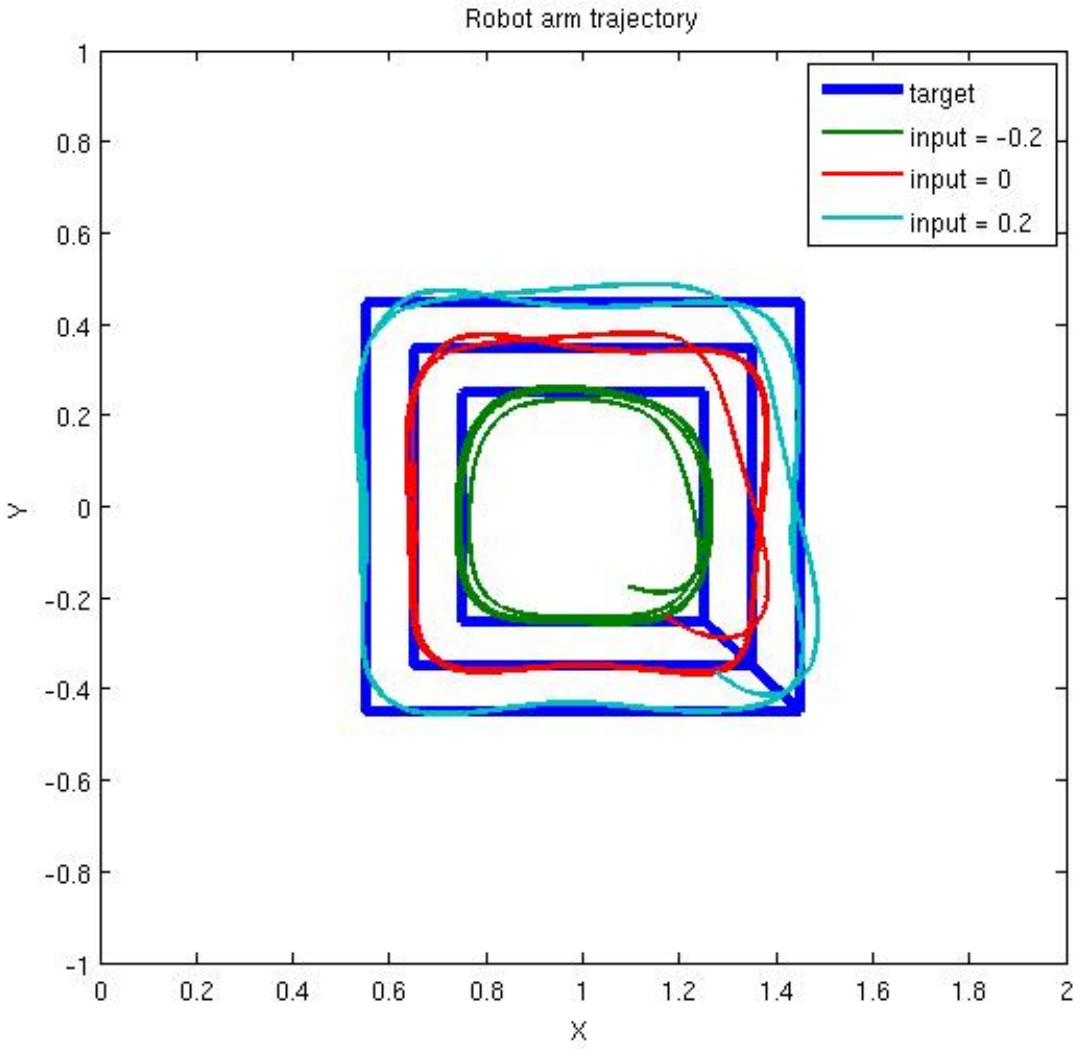

Figure 5.11: Producing new squares with the side length depending on the input

At last, we try to associate the input with both the circles' radius and centres. The circles' center are also linearly spaced during training. The table 5.3 shows the values that are used and the figure 5.12 shows the simulation results with the new inputs. Here again, when we run the network with unseen inputs, it manages to follow the new expected trajectory.

Input	-0.3	-0.1	0.1	0.3	-0.2	0	0.2
Radius	0.2	0.4	0.6	0.8	0.3	0.5	0.7
x_c	0.8	0.9	1	1.1	0.85	0.95	1.05
y_c	-0.2	-0.1	0	0.1	-0.15	-0.05	0.05

Table 5.3: Input and associated circle radius and center coordinates

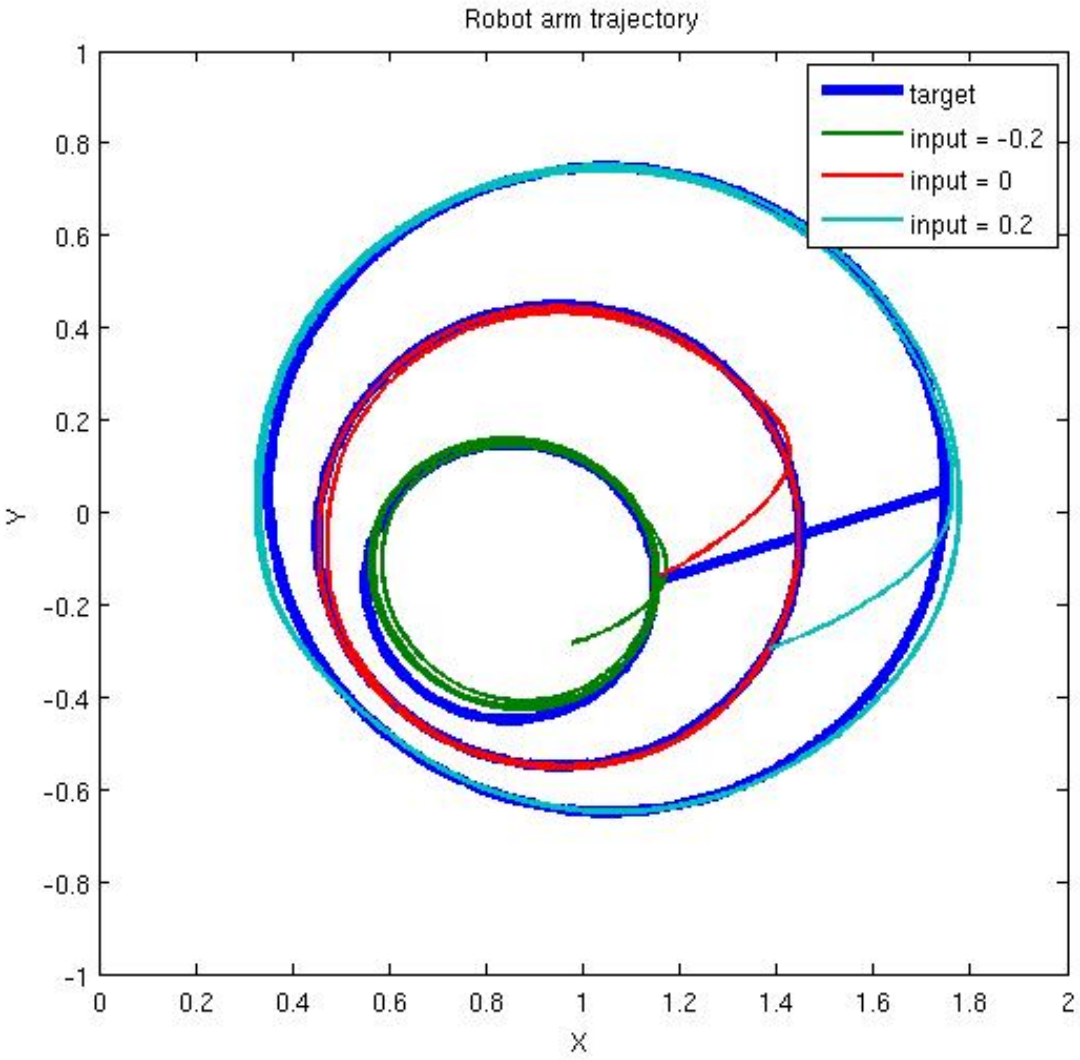

Figure 5.12: Producing new circles with the radius and the center depending on the input

5.4 Conclusion

In this part we first managed to train the network to produce 3 different pairs of complex outputs that depend on the input value.
Secondly we realize that the input/output matching can be extended to new unseen inputs and then produce some new desired outputs. This result that can seem obvious afterward but it wasn't.

6 Working Memory

... et rentre dans la nuit,

Until now, the network have been used to generate periodic outputs that are analogous to motor outputs. In this chapter we generate a more complex pattern where the network produces two outputs, each of them being associated with two inputs. By doing this we manage to generate a 2-bits memory: each output keeps record of which one of its two associated inputs had the latest pulsation.

6.1 Input/Output Generation

The inputs are held to zero and at each time step (ie every milisecond) a pulsation of 100 ms occurs with a probability of 0.0005. We first create the binary inputs and for each pair of input we create the associated binary target function. The binary target switches to 1 when a pulse occurs on its first input, called the ON input, and to -1 when a pulse occur on its second input called the OFF input. We then use the Matlab function *filter* with different parameters to transform the binary inputs and targets into continuous valued functions.
The figure 6.1 shows two inputs with their associated output after filtering.

To generate the inputs

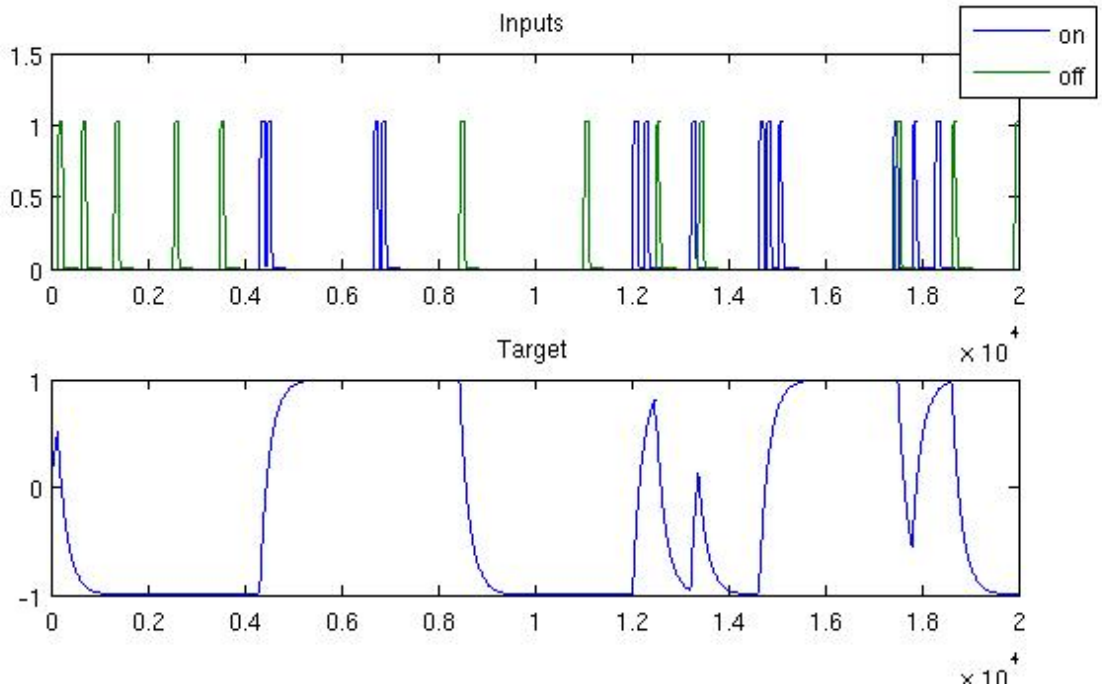

Figure 6.1: Inputs/output generation

6.2 Results

After having generated two pairs of inputs values and two associated target functions of length 100000 ms as described above, we train the network with these data. The figure 6.2 shows the

training results.
We then generate 4 new inputs with the same random process and run the neural network with these new inputs. The figure 6.3 shows the result of this simulation. We can see that each output is associated with two inputs and switches between 1 and -1 when a pulsation occurs respectively on the first (in blue) or the second (in green) input. For more clearness we also plot in red the target function corresponding to each pairs of input during the simulation although they are not used in the algorithm. This task is memory-dependent because the network needs to reminds the recent input history.

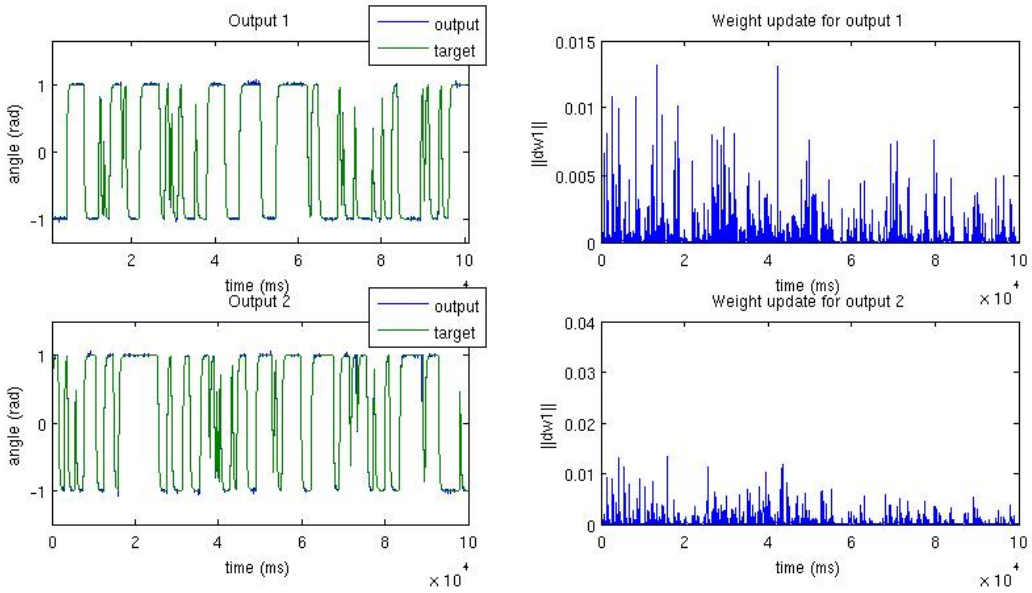

Figure 6.2: 100 s working memory training

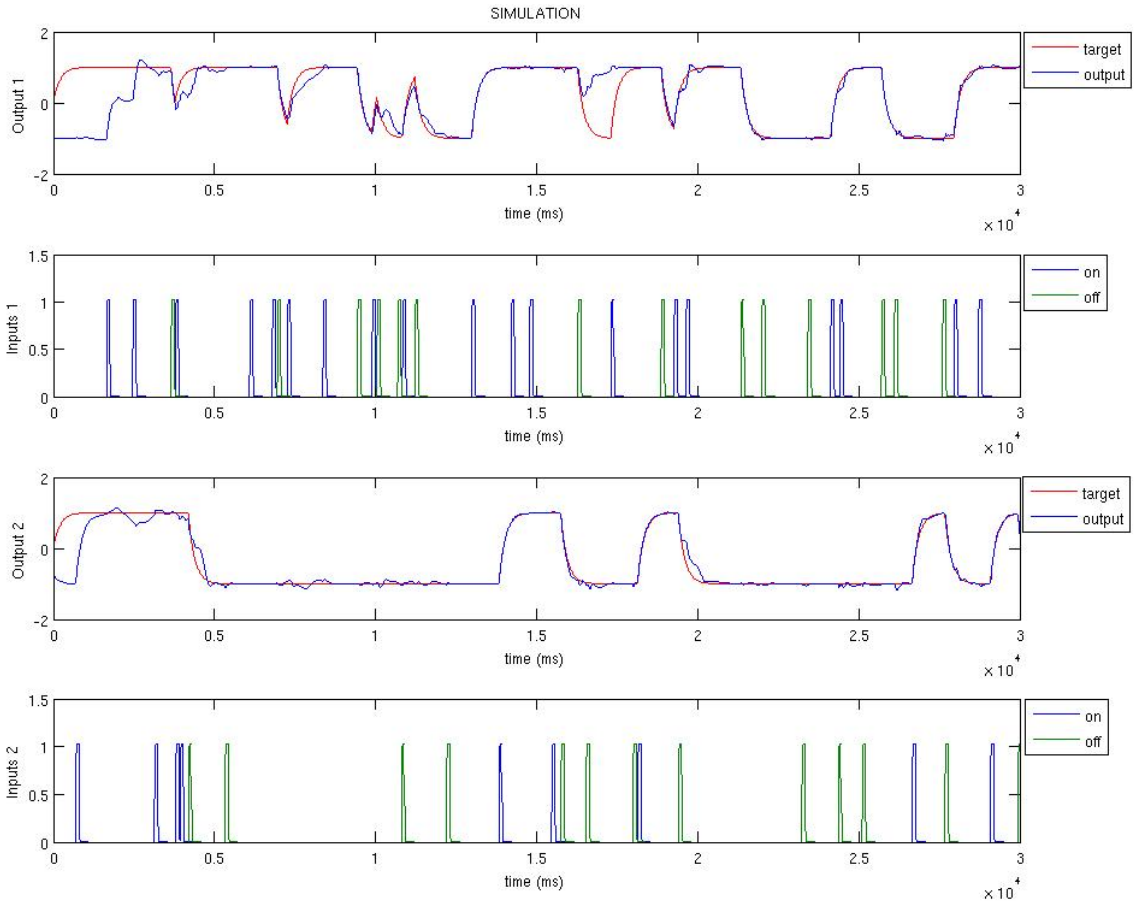

Figure 6.3: Working-memory post-learning simulation

6.3 Conclusion

We can also see that the outputs are robust to input noise because one output is only affected by the two outputs it is associated with, despite the random recurrent connexions of the network.
Generating a working memory, even a small and simple one, is a point of great interest because many cognitive abilities in the brain require a working memory and the way this process is achieved in the brained has not been completely understood yet.
This shows that the networks can learn autonomously a computational rule and develop the required working memory to hold information on its recent input history.

7 Application to a Drone controller

Minute que le temps ...

In this last chapter we tried to apply some results of the previous chapters to produce outputs that could be used as control signals for a drone to follow a target. We reduced our problem to a one dimension problem. The idea would be for the drone to move on the left-right axis in order to come in front of target. We only simulated the drone trajectory on Matlab, but we had in mind to apply our results to a real ARDrone2.0. This drone is equipped with a frontal camera who could be used to detect the target position. We experimented two different approaches that are presented in the two following sections.

7.1 Following a slow target

Our first approach was to produce an output that would make the drone follow a target which is slowly moving by producing a speed which is directly proportionate to the distance between the drone and the target. To do so we train the network to produce an output which is equal to the input with 4 linearly spaced inputs in the range $[-1, 1]$ as seen on figure 7.1. Then when we run the network with any new input in the same range, it will produce an output equal to the input. Starting from that we can run the network with the input being the difference between the target position and the drone position at each time step. The output will be equal to this difference. The drone position is updated at each time step using Euler approximation with the network output being used as the speed, i.e. the derivative of the position. The three plots of figure 7.2 shows the result of the simulation. The first plot shows the target position (in green) and the drone position (in blue) over time. We can see that the two positions stay really close to each other. The second plot shows the absolute difference between the two positions. The third plot shows the network output (in green) and the derivative of the target position (in blue).

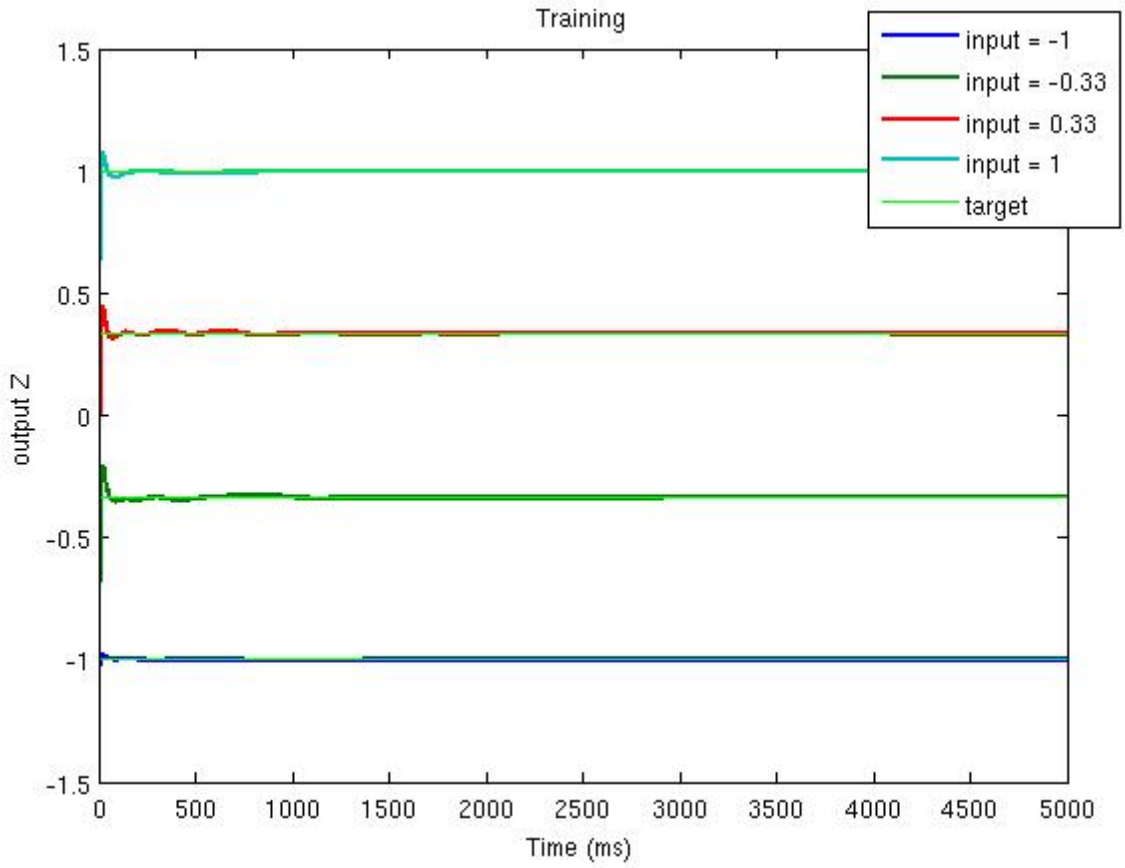

Figure 7.1: Post learning simulation

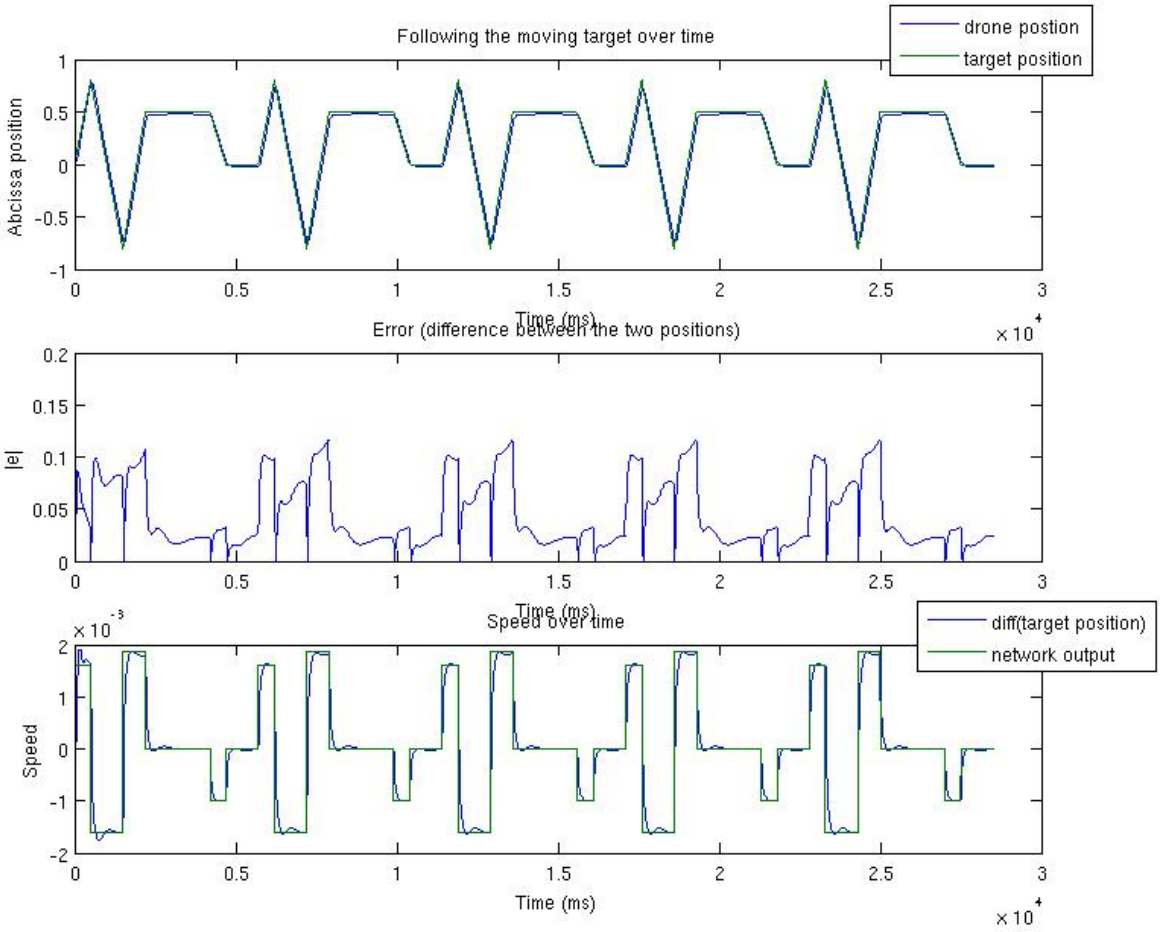

Figure 7.2: FORCE learning application to track a moving target

7.2 Following fast target

Our second approach consisted in learning some more complex control signals to make the drone reach a target as quick as possible. Whereas in the first approach, the drone is initially close to the target and follows it as it moves slowly, in this case the target directly appears at a certain distance and the drone try to reach it quickly.

7.2.1 input/ouput generation

Every 1500 ms the input which represents the target position changes its value which is randomly generated in the range [-2.5:0.25:2.5]. We try to define empirically what seems to be the optimal trajectory to reach this point. To produce the trajectory we use the following equation which is the solution of a second order differential equations with constant coefficients:

$$trajectory(d,c,T) = c+(d-c)(1-e^{-\zeta.\omega_0.T}.cos(\omega_0.\sqrt{1-\zeta^2}.T)-\frac{\zeta}{\sqrt{1-z^2}}.e^{-\zeta.\omega_0.T}.sin(\omega_0.\sqrt{1-\zeta^2}.T)) \tag{7.1}$$

where

- ζ is the system damping coefficient ($\zeta > 0$)
- ω_0 is the system natural frequency
- d is the desired position
- c is the current position

To generate the control signal associated with the input, we differentiate the desired continuous trajectory that we obtained from the inputs. The figures 7.3 and 7.4 shows the results of the process with different damping coefficients values. On each figure the first plot shows the input (in blue), i.e. the target position, and the associated trajectory to reach this position (in green). The second plot shows the derivative of the trajectory that we want the neural network to generate.

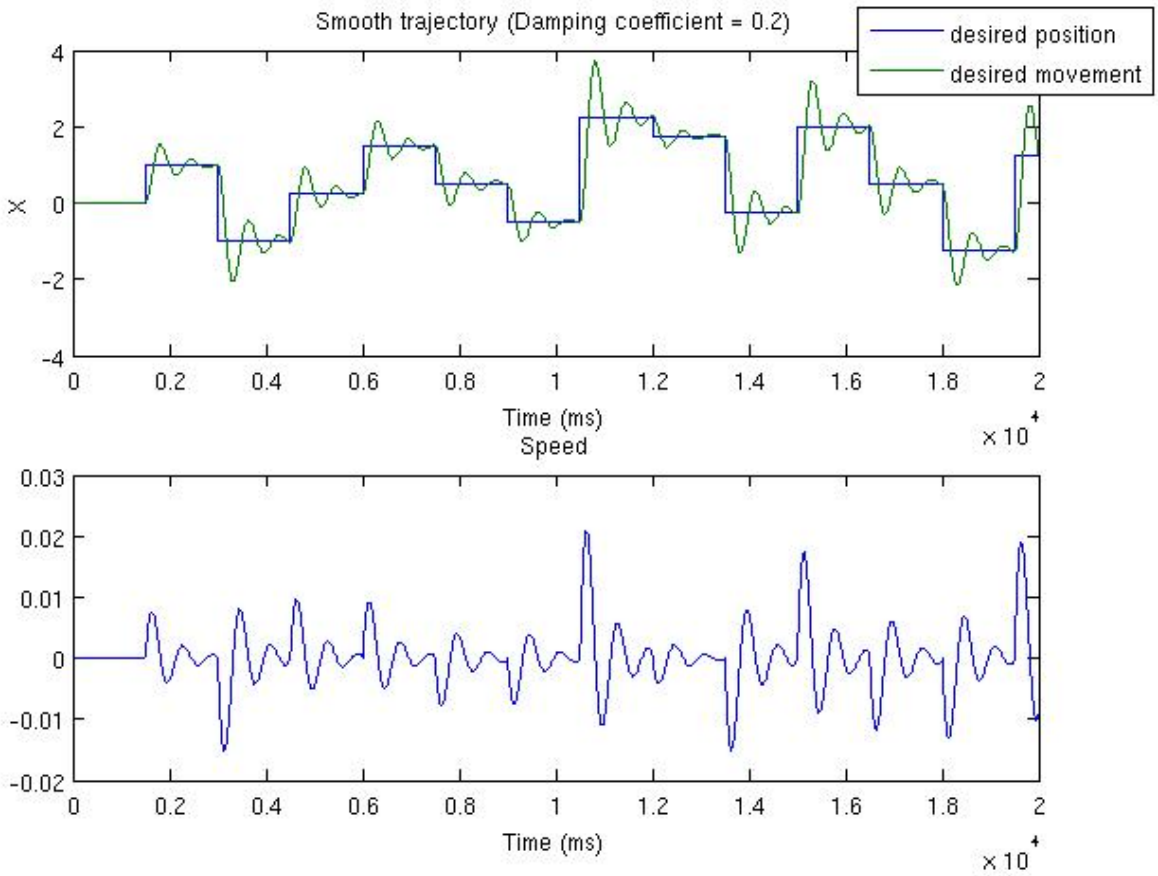

Figure 7.3: Input and corresponding desired output (damping coefficient $\zeta = 0.2$)

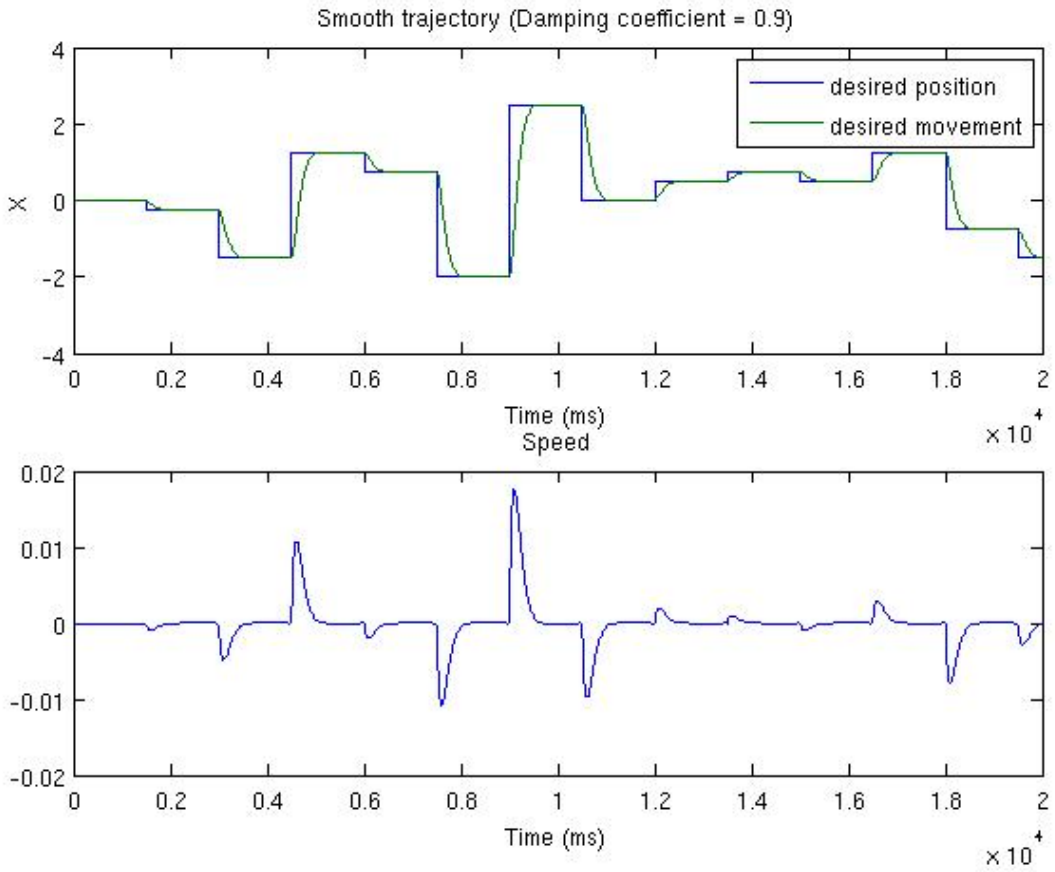

Figure 7.4: Input and corresponding desired output (damping coefficient $\zeta = 09$)

7.2.2 Learning the aperiodic control signal

After having generated some 100000 ms long time series of inputs and the associated outputs, we train the network with these data. The figure 7.5 shows the training results.

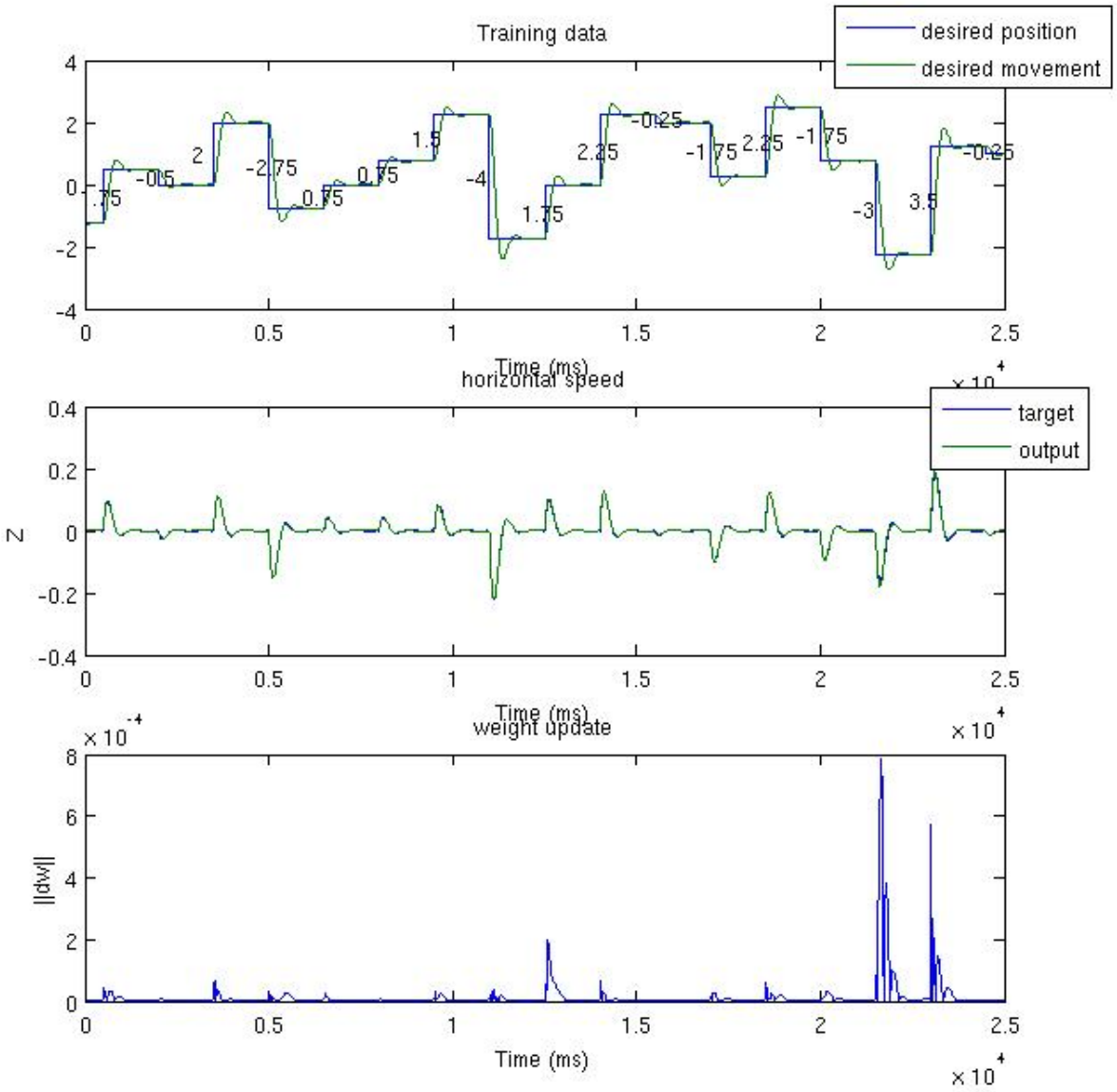

Figure 7.5: 25 s of FORCE learning of the drone control signal

We then generate a new time series of inputs, which is now taken in the range $[-2.5 : 1.667 : 2.5]$ i.e. the difference between the input value at different times is a multiple of 0.1667 instead of 0.25 like in the training. We also generate the smooth trajectory and the control signal associated with the new input. When the network is run with this new input time series it produces the expected control signal as seen on figures 7.6 and 7.7. On these figures, the first plot shows the desired output and the one produced by the network which are almost the same (the second plot shows the absolute difference between the actual and the expected control signal). The third plot shows the input with the step size every time the input changes. The fourth plot shows the trajectory obtained by integrating the network output, together with the desired trajectory.
Because we sum up the control signal to obtain the trajectory, the error also sums up over time which explains the deviation from the expected trajectory. However in practice the deviation from the desired trajectory would induce a change in the input which represents the distance between our drone and the target, so the error would be corrected over time.
These results are very interesting for two reasons.
First, we manage to produce a signal that does not depend on the input absolute value but on its relative value. The control signal amplitude is proportionate to the size of the difference in the input value.

The second interesting results is that the network can be run with new input values (more precisely with new input step size), that haven't been seen during training. In this case the network still produce the correct control signal that it hadn't been trained to generate.

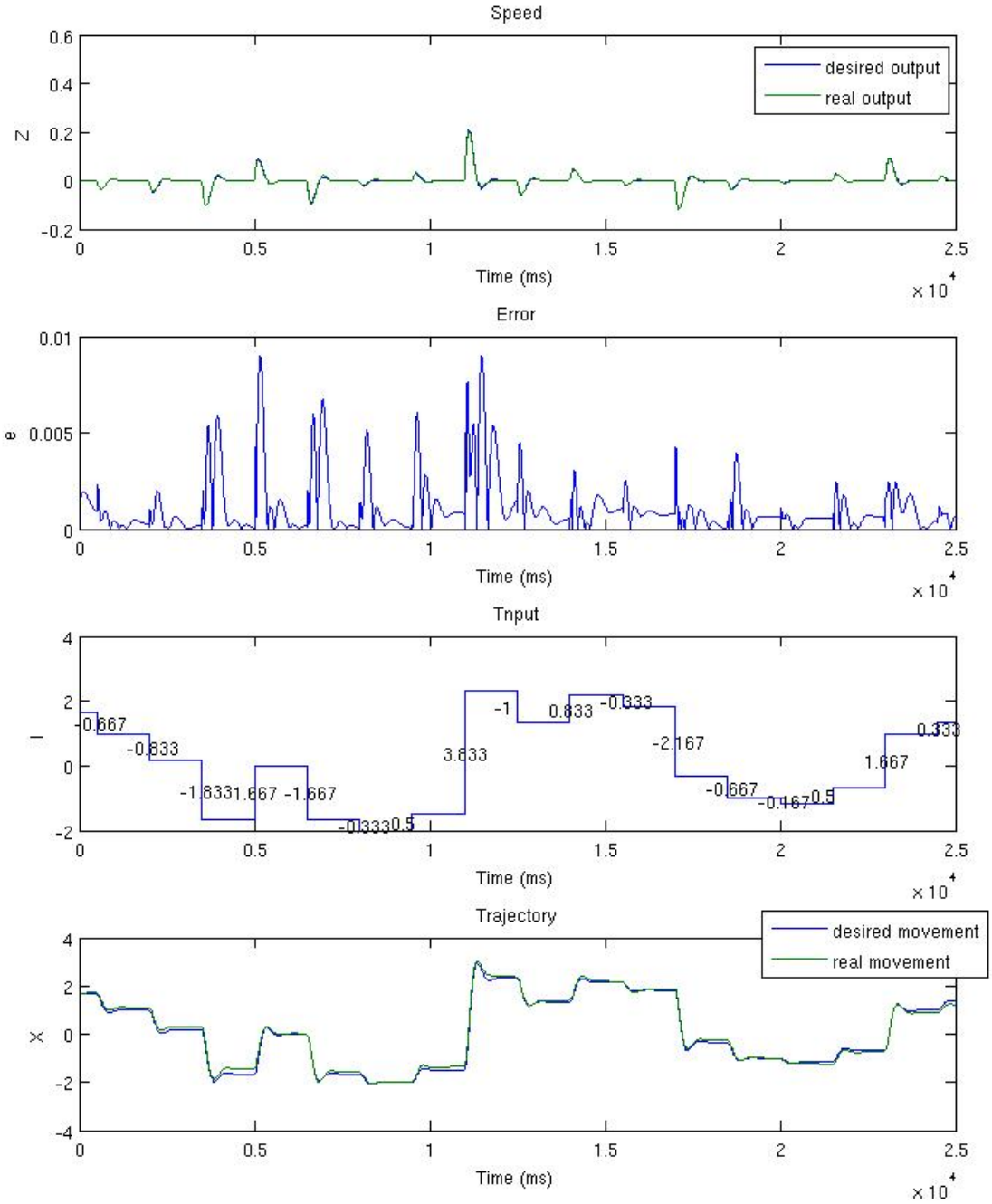

Figure 7.6: Network simulation on unseen input values (the desired trajectory and the associated control signal are produced with a damping coefficient $\zeta = 0.5$)

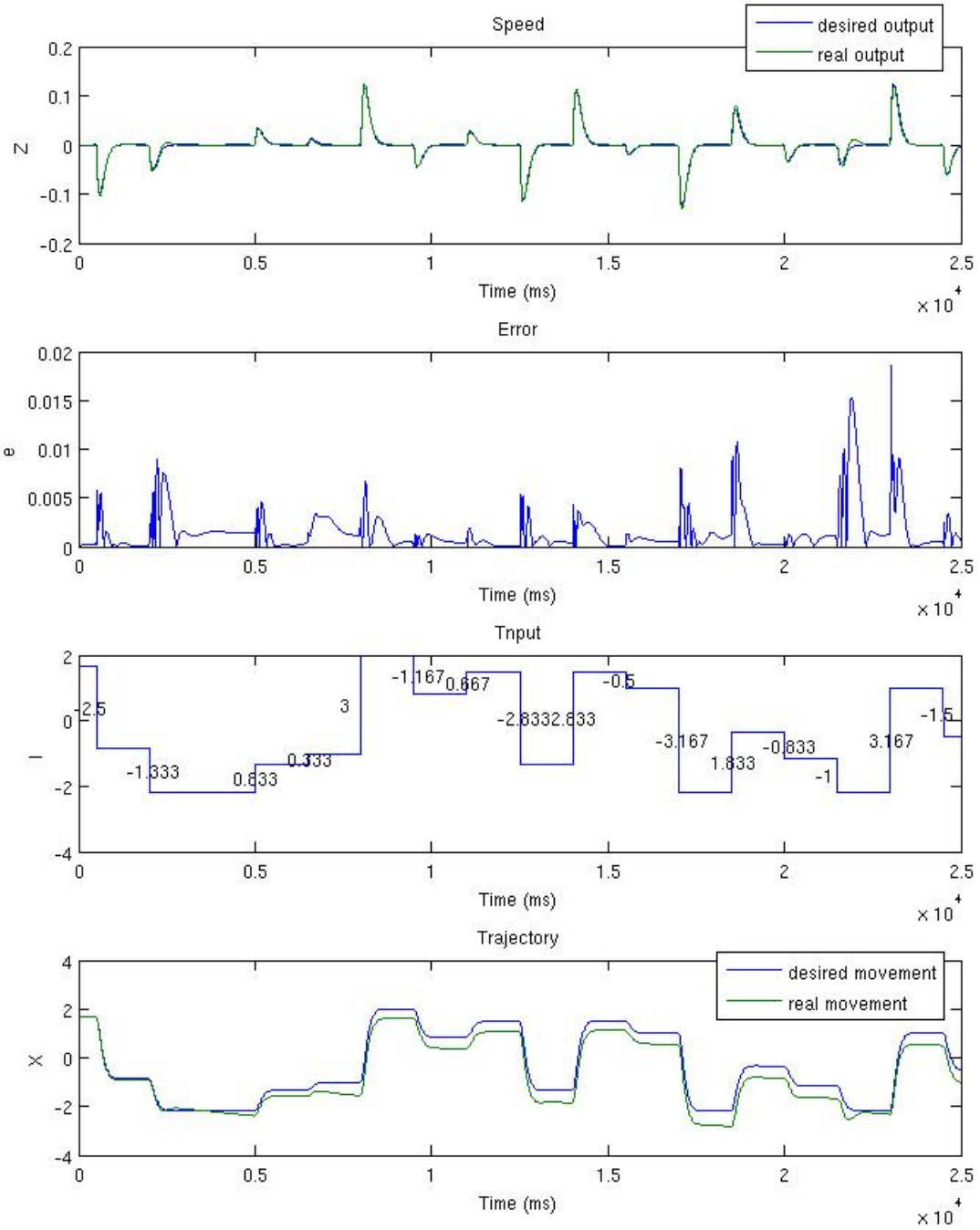

Figure 7.7: Network simulation on unseen input values (the desired trajectory and the associated control signal are produced with a damping coefficient $\zeta = 0.9$)

7.3 Conclusion

In this part we managed to extend the result we found in a part 5.3 where we managed to produce a circle of any radius by simply learning to produce 4 different circles. Here we learn a small repertoire of aperiodic control signals but managed to produce a wide range of different signals by using new inputs after the learning. This phenomena is really interesting and even if we had the intuition it might happen it wasn't guarantee at all. To our knowledge, this kind of input interpolation to

produce new outputs with reservoir computing hasn't been done before and it might be interesting to test it further or to think of other applications.
Regarding the drone control itself, the next chapter describes what could be the future steps to reach our goal.

8 Future perspectives

... prête et retire à l'homme.

Lamartine

8.1 Collecting real training data

In section 7.2 of the last chapter we assumed that the best strategy to quickly reach a target was to reach it with a high speed, even if that means going a little too far and then going back a bit. A more relevant approach would consist in collecting real data on the drone to use a more accurate control signal.
An expert could pilot the real drone and reach some points at different distances as quick as possible. We tried to run such an experiment with an ARDrone but the results were not convincing because the data we collected were very noisy. However the experiment could be run again in a better environment and hopefully provide some data that could be used to train the neural network.
An other solution to collect training data would be to use a proper drone simulator.

8.2 Further training

In the last chapter, we train the network to produce the desired speed to apply to the drone. However, although the drone has a high acceleration, the control signal to apply to the Drone might not be exactly equal or proportionate to the desired speed because of the drone inertia. A future thing to be done would be to explore other shapes of control signals to apply to the drone to obtain the desired speed and trajectory.
We limited our study on the drone to a one dimension movement. A following step would be to extend our results to several dimensions. We could have two inputs, each controlling the drone trajectory on one dimension. As we saw in chapter 6 where we implemented a working memory, it is possible to make an input having an effect on one output without affecting the others outputs. However some problem occur when we want the input to evolve in a wide range of continuous and previously unseen values like it was the case for the drone. In this case when an input take a different value, the network might not be able to know which input has changed. Moreover if several inputs changes at the same time, each neuron receive the weighted sum of the inputs ($\sum_{j=1}^{N_{in}} J_{ij}^{GIn} I_j(t)$) it can not know which input as changed and of how much.
We could also imagine training the network to produce a large repertoire of behaviour and movement and then use some properly speaking reinforcement learning techniques to teach the drone how to use a combination of these different behaviour to reach any point.

Conclusion

The first parts of this project confirmed the results of some experiments that had already been made. Reproducing these results from scratch was a really good way to get familiar with the learning algorithms and the use of recurrent neural networks before going further. We managed to generate several complex periodic patterns and to realize some inputs/outputs matching in a wide range of frequencies. We also reproduced a two-bits working memory which is a complex computational structure. Being able to train a chaotic recurrent network to generate this behaviour can shed a new light on how such operations are achieved in the brain which is still an area of research.

However the main interest of this project is the new kind of training we realized and that, as far as we know, hadn't been done before. By training the network with only four different input values, we can produce a complex periodic output that will be proportionate to any new input taken in the range of the inputs used during training. For example, in our case, we managed to produce a circle of any radius in a certain range whereas only four circles had been learned during training. This is quite challenging because usually the input that were used during training were the same that were used during the simulation. It wasn't obvious at all that such an interpolation would succeed and that we would be able to generate new correct outputs by using intermediate input values. The extension of these results to the generation of more complex aperiodic patterns is also very promising. Although we are not sure whether this could be used to completely pilot a drone, this question deserve to be investigated. Moreover there might other areas were this kind of learning could be very useful.

To conclude this project was an amazing opportunity to become more familiar with the very interesting field of recurrent neural network. It was interesting and challenging to be free to explore different applications of such networks to finally arrive to a more precise goal step by step.

Bibliography

[1] D. Sussillo and L.F. Abbott - Generating coherent patterns of activity from chaotic neural networks. 2009

[2] G.M. Hoerzer, R. Legenstein and W. Maas - Emergence of Complex Computational Structures From Chaotic Neural Networks Through Reward-Modulated Hebbian Learning. 2012

[3] Hopfield - Neurons with graded response have collective computational properties like those of two-states neurons. 1984

[4] Haykin - Neural networks: a comprehensive foundation. 1999

[5] Beer - On the dynamics of small continuous-time recurrent neural networks. 1995

[6] Sompolinsky, Crisanti and Sommers - Chaos in random neural networks. 1988

[7] Bertschinger and Natschlger - Real-time computation at the edge of chaos in recurrent neural networks. 2004

[8] http://reservoir-computing.org/.

[9] H. Jaeger - The "echo state" approach to analysing and training recurrent neural networks - with an Erratum note. 2001

[10] T. Natschlager, W. Maass, and H. Markram - The "Liquid Computer": A Novel Strategy for Real-Time Computing on Time Series. 2002

[11] Nudo et al. - Use-dependent alterations of movement representations in primary motor cortex of adult squirrel monkeys. 1996

[12] Rainer and Miller - Effects of visual experience on the representation of objects in the prefrontal cortex. 2000

[13] Recanzone et al. - Topographic reorganization of the hand representation in cortical area 3b of owl monkeys trained in a frequency-discrimination task. 1992

[14] Klingberg, Forssberg, Westerberg - Training plasticity of working memory in children with ADHD. 2002

[15] Klingberg - Training plasticity of working memory. 2010

[16] Legenstein, CHase, Schwartz, Maas - A reward-modulated Hebbian learning rule can explain experimentally observed network reorganization in a brain control task. 2010

[17] Frémaux, Sprekeler, Gerstner - Functional Requirements for Reward-Modulated Spike-Timing-Dependent Plasticity. 2010

[18] Fiete, Seung - Gradient Learning in Spiking Neural Networks by Dynamic Perturbation of Conductances. 2006

[19] Sussillo - Learning in chaotic recurrent networks. 2009