A brief analysis of intelligent control techniques for robotic manipulators

Norman Di Palo, Tiziano Guadagnino
Sapienza University of Rome

Abstract

In this work we analyze the performances of different intelligent control algorithms, applied to trajectory tracking on a 2R and 3R robotic manipulator, whose dynamical parameters and structure are unknown. We developed and compared genetic neuro-controller, two main Reinforcement Learning algorithms (Q-Learning and Actor-Critic), and a neural network with trigonometric basis functions that learns the inverse dynamics by error backpropagation.

1 Introduction

Intelligent control is the branch of control theory that studies algorithms based on artificial intelligence and machine learning techniques. These fields have widely evolved in the recent years, in large part thanks to the breakthrough of Deep Learning, that obtained remarkable results in fields like computer vision and natural language processing. It's possible to use techniques from these fields in robotics too, especially when dealing with uncertainties in various parameters or unmodeled effects. In this work, we mainly focused on three different classes of algorithms: the first one are genetic or evolutionary algorithms, a class of AI algorithms used for optimization and inspired by natural evolution. The second are reinforcement learning algorithms, that allow an agent to learn an optimal policy/behaviour from an autonomous interaction with an unknown environment. The last algorithm is based on inverse dynamics learning with neural networks, in which the network is trained by error backpropagation in a more supervised learning-fashion. These controllers were used in parallel to a classic PD controller to perform trajectory tracking with a robotic manipulator, with unknown physical parameters. We analyzed the performances and behavior of these algorithms in term of total error in position and velocity along all the trajectory, as well as the magnitudes of torques used. A priority for us was to make these algorithms learn is a short time, thus making them feasible for real world robots and applications, while often many learning algorithms are run for many hours or days on different and parallel simulations. This was a strong bottleneck that required us to rethink and tune all the algorithms to converge in under a hundred of runs.

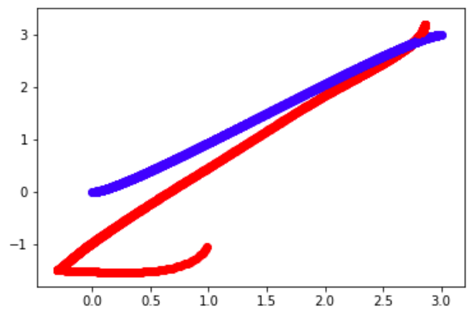
2 Proposed Methods

2.1 Evolutionary Algorithms

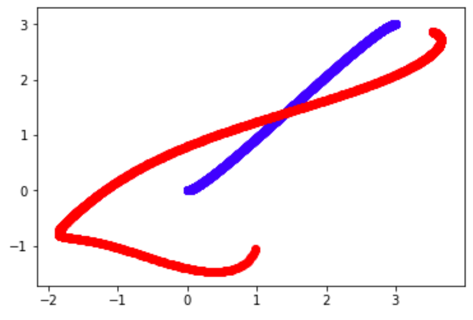
Evolutionary and genetic algorithms are a class of artificial intelligence algorithm, inspired by the concept of Darwinian evolution and selection, and used mainly for optimization problem, especially in cases where a classic solution is difficult to obtain. The main idea underneath these algorithms in to find a solution for and optimization problem by randomly evolving and combining the parameters of the best performing solutions, based on a fitness function $f(\theta)$, selecting the best ones and continuing to evolve them. These family of algorithms are widely used, with applications in robotics too. We designed an evolutionary algorithm to evolve a neurocontroller that was run in parallel to a PD controller. The neurocontroller is made of a neural network that takes as input the state of the robot, i.e. the position and velocity of each joint, as well as the errors in position and velocity. The outputs of the network are the torques to apply to each joint. The activations functions are tanh for each layers, except for the output layer that is linear. We explored different topologies for the network, tyring to minimize as much as possible the number of parameters to make both the learning phase quicker and to make it computationally light. Our final model was made of two hidden layers, with 150 total parameters.

The evolutionary algorithm was applied to the parameters of the network, evolving the best performing solutions based on the total final error in position and velocity, integrated at each time step. These algorithms are best-fitted to cases where we can execute multiple runs in parallel, since we can quickly evaluate the best solutions, evolve them and select the next generation of solutions. In our case, as mentioned in the introduction, we designed it to run on a single robot, trying to obtain good results in as little iterations as possible. We did so by optimizing the way the evolution occurred: at each step, the best performing parameters were updated by a vector of gaussian white noise of the same dimension of the weights vector, multiplied by a learning step. The new set of weights were then evaluated on a new run. If the overall loss was lower than the lowest loss, we continued to evolve the weights towards the same direction sampled before. Otherwise, the original weights were updated by adding the dame vector, but with half the learning step. Furthermore, in the first phase we used a higher learning step to escape local minima, while we decremented it exponentially as the learning went on.

These steps, along with the right topology of the network, allowed the algorithm to find quite interesting solutions, improving the performance of the PD controller, obtaining half the total error after around 70 iterations. The final behavior of this kind of controller is quite unpredictable, but it was interesting to see the final trajectory performed by the robot under a PD controller and then with neurocontroller + PD. The trajectories we tried are shown in the images, that plot the joint space. We started the robot with an error both in position and velocity to see how the controllers recovered from those, obtaining interesting solutions.



```
Out[102]:  4190.2400026753594
```



```
Out[101]:  8291.6808815984077
```

Figure 1: Comparison of trajectory tracking in the joint space. Up: neurocontroller + PD. Down: PD alone. Note how the total integrated error is about an half with the neurocontroller.

Algorithm 1 Evolutionary algorithm applied to a neurocontroller

```
1: Initialize network weights $w_{try}$=$w_{best}$, lowest error $err_{low}$ to inf, $improved = false$, $direction = 0$
2: for each episode do
3:     $err_{episode}$=0
4:     Copy best achieving weights $w_{try}$=$w_{best}$
5:     if $improved == True$ then
6:         $improved = False$
7:         Evolve $w_{try} + = direction$
8:     else
9:         Sample $direction$ from White Noise Gaussian
10:        Evolve $w_{try} + = direction$
11:    end if
12:    while Episode is not finished do
13:        $err_{tot} + = |err_{position}| + |err_{velocity}|$
14:    end while
15:    if $err_{tot} < err_{low}$ then
16:        $err_{low} = err_{tot}$
17:        $w_{best} = w_{try}$
18:        $improved = True$
19:
```

2.2 Reinforcement Learning

Reinforcement Learning had a renaissance in recent years, driven by the evolution of Deep Learning. In the recent years, RL has obtained various unexpected results, such as AlphaGo that beat the world champions of Go, Atari DQN, and finally breakthroughs in the field of control of humanoids and quadrupeds. New algorithms have been proposed in just a few years that have set new state-of-the-art results in different environments, from video game playing [1] to continuous control. In this work, we analyzed two main RL algorithms, trying to apply them to the trajectory tracking task. One common aspect of the aforementioned works is the training time and computational power needed: those algorithms can require even weeks of training with hundreds of thousands of iterations. On the contrary, as mentioned in the introduction, our goal was to develop algorithms that were capable of obtaining satisfactory results in a short time, with a hundred of runs at most. We thus had to rethink those algorithms to fit our requirements. The first algorithm used is Deep Q-Learning, based on the famous Q-Learning work by Bellman. This algorithm was used to dynamically tune the gains of the PD controller in real time, to enhance the performance of it with respect to a PD with fixed weights. The second algorithm is Actor Critic, another widely used class of algorithm, particularly fitted for continuous control. In this case, the goal was to develop a policy that mapped the continuous state of the robot into torques for the joints.

2.2.1 Q-Learning for Dynamical PD Tuning

We were interested by the idea of dynamically tuning the gains of the PD controller in real time, based on the behavior of the robot and on the evolution of the errors in position and velocity, so we designed a Q-Learning algorithm to achieve this task. In the framework of RL, Q-learning is a family of algorithms with the central idea of approximating and exploiting the Q function [2]. The Q function in defined as a function of states and actions, that returns the expected total reward from state s, doing action a and then following the optimal policy. Thus, for discrete actions, if the Q function is correctly approximated, we can choose in each state the action with the highest Q value. In our architecture, the Q function is approximated by a neural network that have as input the state of the robot (i.e. the joints velocity, position, and respective errors) and the action taken in that particular time step. The actions that we designed, in a discrete fashion, are finite increments of position and velocity gains independently. In particular, the increments are +5, +3, 0, -3, -5 for the position gain and +1,+0.5,0,-0.5,-1 for the velocity gain.

The reward is computed as:

$$r(s) = e^{-\|e_p\|^2} + e^{-\|\dot{e_p}\|^2} \tag{1}$$

From the definition of the *Q-function*, we know that:

$$Q(s,a) = r(s) + \gamma \cdot max_{a'} Q(\delta(s,a), a^{'}) \tag{2}$$

Where γ is the discount factor and $\delta(s,a)$ is the transition model that, given the current state and action, return the next state of the robot. In particular, developing this recursive formula, the *Q-function* can be also defined as the sum of discounted reward :

$$Q(s,a) = \sum_{i=0}^{\infty} \gamma^i \cdot r(s) \tag{3}$$

In our algorithm we truncate this summation to the first five terms, giving the result as target for the neural network. Essentially we approximate the *Q-function* in the next five step of the tracking.

It is well known in the RL literature how these algorithms, when using function approximators, can hardly converge. Thus, in the years, various techniques to stabilize the learning phase have been proposed. One of the most successful is episodic memory: it's based on storing all the steps of the agent in a data structure, and then randomly sampling past experiences from it to update the networks. This technique can help decorrelate the various training examples, that otherwise would be highly correlated and not i.i.d, as required for training a neural network. We thus implemented this architecture in our algorithm to help the learning phase converge.

Algorithm 2 Q-based dynamic PD

```
1: for each episode do
2:     $s \leftarrow s_0$
3:     take action $a = (a_p, a_d)$, as $a = \arg\max_{\tilde{a}} Q(s, \tilde{a})$
4:     $Kp \leftarrow Kp + a_p, Kd \leftarrow Kd + a_d$
5:     initialize the $Q$ estimation : $\tilde{Q} = 0$
6:     for $i = 0$ to 4 do
7:         $\tilde{Q} \leftarrow \tilde{Q} + r(s)$
8:         $u = Kp \cdot e_p + Kd \cdot \dot{e_p}$
9:         $\ddot{q} = B(q)^{-1} \cdot (u - c(q, \dot{q}) - g(q))$
10:        double integration of $\ddot{q} \rightarrow (q, \dot{q})$
11:        $\tilde{s} = (q, \dot{q}, e_p, \dot{e_p})$
12:        $s \leftarrow \tilde{s}$
13:    end for
14:    Store in the experience buffer $\leftarrow (s_0, a, \tilde{Q})$
15:    Drawn a random batch of experience from the buffer
16:    Update the parameter of the neural network
17:    $s_0 \leftarrow s$
18: end for=0
```

After various experiments and different parameters, we noticed how the algorithm learn to constantly increase the gains. This concept is known for improving the performance of the controller, that the controller learned without any prior knowledge, but can lead to high torques and power consumption. We tried to obtain more unpredictable behaviours but the final results often converged towards the mentioned solution.

2.2.2 Actor Critic for Continuous Control

Actor-Critic methods are one of the most widely used classes of RL algorithms in the recent times. Various improvements allowed these family of algorithms to achieve outstanding results in the control of complex systems, such as humanoids and quadrupeds, as show in those works. AC algorithms are based on the optimization of a policy (the actor) with respect to the expected total return. They achieve these task exploiting the information given by the Critic, a function approximator that learn to simulate the *Value function* $V(s)$ (usually via the Bellman equation, as in Q-learning). This allows the Actor to receive frequent updates based on the feedback of the Critic, optimizing the policy at each step. The main theoretical results behind these ideas are the Policy Gradient theory, both Stochastic than the more recent Deterministic PG. Our work is based on recent research on the application of AC to robotic manipulators, such as this thesis [3]. Its goal is quite similar to ours, with the main difference that there the robot is controlled in velocity, with a low level controller that injects torques based on those desired velocities. Thus, we explored the performance of these kind of architecture on a robot controlled in acceleration. We implemented an

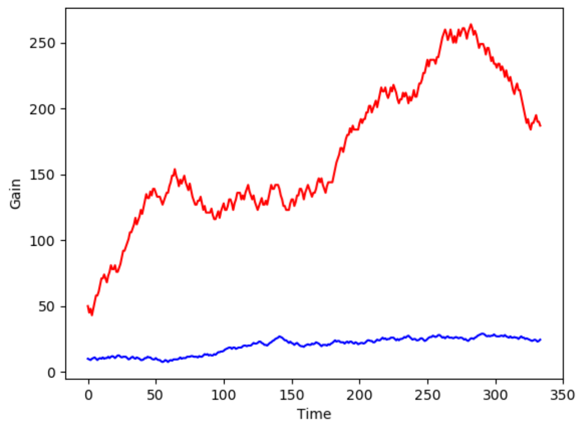


Figure 2: Gain Behavior: Kp in red, Kd in blue

independent AC architecture for each joint of the robot. As suggested in the previous works, this help the algorithm to scale to robot with high DOFs while keeping a short training time. The architecture is the following: in order to have an easier update step, our function approximator is linear in the set of differentiable parameters. We designed a 2D array of Gaussian Radial Basis Functions. These functions are equally distributed in space and with the same variance. The input to this function approximator is the error in position and velocity of the joint. The output is the weighted sum of the output of these RBFs, first normalized and then multiplied by the parameters theta. This value becomes the torque input for the joint in case of the actor, or the approximated $V(s)$ function in case of the critic. The general expression for this function approximator is :

$$f(x) = \sum_{i=1}^{N} \psi_i \cdot e^{\frac{-(x_i-\mu_i)}{2\cdot\sigma^2}} \tag{4}$$

Where μ is the mean and σ^2 is the variance of the RBF kernel. In particular we use the same reward of the previous algorithm (eq.(1))

As mentioned in the last section, instability is a major problem for these kind of algorithms, along with the need for long training times. Thus, tuning the algorithm to perform well and converge to a satisfying policy in a short time is not trivial. We managed to obtain interesting results for a 1 or 2R robot, but we were unable to scale this algorithm to larger robots while keeping it stable and performing under a hundred training runs. We noticed interesting behaviors in the case of a 1R robot, where in just a few training runs the AC controller learns to be in phase with the PD controller that runs in parallel.

We suspect that we were unable to reproduce the performance of the refer-

Algorithm 3 Actor-Critic

```
1: for each episodes do
2:     Generate initial action $a = a_0$
3:     $\omega_0 = 0$
4:     $s_0 = (q_0, \dot{q_0})$
5:     Compute initial reward $r(s_0)$
6:     repeat
7:         Apply $u = a_k + Kp \cdot e_{pk} + Kd \cdot \dot{e_{pk}}$
8:         Measure $s_{k+1}$ and compute the reward $r(s_{k+1})$
9:         Compute next action $a_{k+1} = \pi(s_{k+1}, \phi_k) + \delta_u$ with $\delta_u \approx \mathcal{N}(0, \tau^2)$
10:        $e_{td} = r(s_{k+1}) + \gamma \cdot V(s_{k+1}, \theta_k) - V(s_k, \theta_k)$
11:        $\omega_{k+1} \leftarrow \gamma\omega_k + \frac{\partial V(s,\theta)}{\partial \theta}\Big|_{s=s_k,\theta=\theta_k}$
12:        $\theta_{k+1} \leftarrow \theta_k + \alpha_c e_{td} \omega_{k+1}$
13:        $\phi_{k+1} \leftarrow \phi_k + \alpha_a e_{td} \delta_u \frac{\partial \pi(s,\phi)}{\partial \phi}\Big|_{s=s_k,\phi_k}$
14:        $k \leftarrow k + 1$
15:    until terminal state
16: end for
```

enced previous work [3] due to the fact that we control the robot at the acceleration level, thus directly dealing with all the inertias and dynamical coupling, while that work is based on velocity commands given to a low level controller built in on the robot.

Many approaches and architectures have been proposed to overcome these kind of instabilities, but we didn't explore this approach further due to the remarkable results that we obtained at the same time with the inverse dynamics learning via error backpropagation that we describe in the next section.

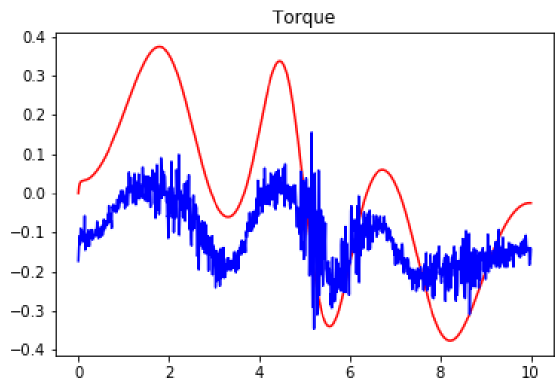


Figure 3: Torque behavior for a 1R robot, in red the PD and in blue the ouput of the actor

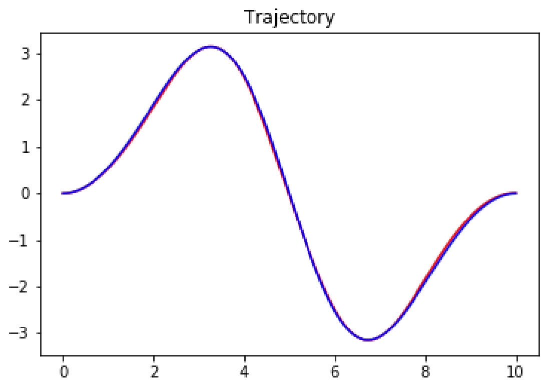


Figure 4: Trajectory obtained for the 1R robot, in red the one followed by the robot, in blue the desired

2.3 Inverse Dynamical Model Learning using Feedback Error

The last method that we implemented was, as anticipated, the best performing, lowering drastically the errors both in position and velocity after just a few runs, and even lowering of a considerable amount the feedback torques, thus making the whole trajectory tracking accurate and energy efficient. Even more, this method has shown remarkable generalization properties: even if trained on a particular trajectory, this controller can accurately command the robot even on unseen trajectories, enhancing considerably the performances with respect to a

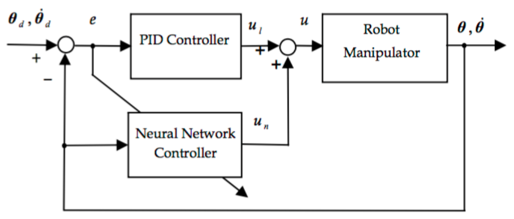


Figure 5: High level block diagram of the implemented controllers.

feedback only control. This technique is based on previous works, such as [4], [5].

The overall architecture is quite similar to the previous proposed methods: we implemented a parallel of a classic PD controller and a neural network controller, that outputs torques for each joint receiving as input the desired joint positions and velocities. The main difference, here, is that the network is used to learn online the dynamical model of the robot, thus allowing an efficient feedforward application of torques, based on the desired trajectory. Another main difference from the previous architectures is the use of various basis functions, based on different trigonometric functions that resemble the usual terms appearing in the dynamical model of a 3DOF antropomorphic robot manipulator. This initial kernels-layer allows the network to rapidly shape the output function to resemble the actual dynamical model of the robot. The network, that can be analyzed in detail in the code of this work, outputs torques for each joint. The training phase is quite different from an usual neural network, that usually computes the error of its ouput with a reference ground truth, and then adjusts its weights via differentiation and backpropagation. In this case, the backpropagated error is directly the error in position and velocity, computed at the next step, used to make a step of stochastic gradient descent on the weights. Thus, the netork isn't based on any ground truth torques, as it would be in an imitation learning setting.
This method allows the network to rapidly converge, learning the dynamical model and consequently computing the adequate torques for the robot to follow the desired trajectory and minimize the errors. The plots in the figures show the error decreasing over time, as well as the various feedforward and feedback torques for each joint.

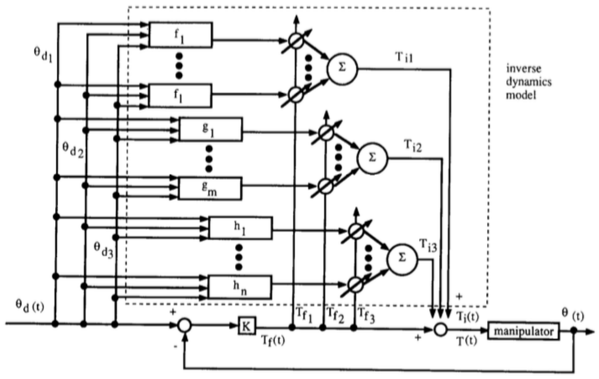
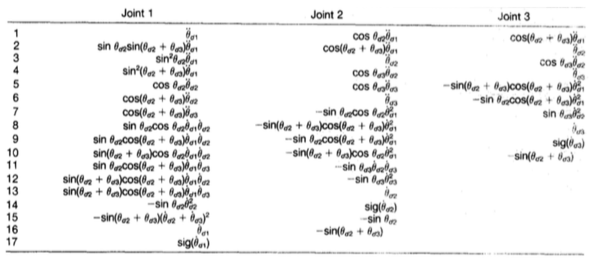
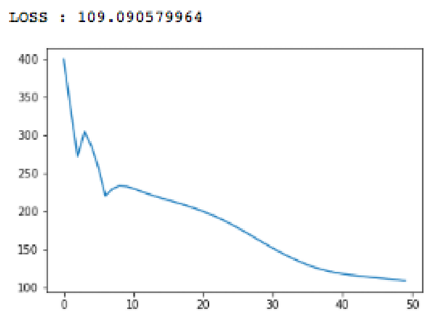


Figure 6: Original network architecture from the paper [4].



	Joint 1	Joint 2	Joint 3
1	$\ddot{\theta}_{d1}$	$\cos\theta_{d2}\ddot{\theta}_{d1}$	$\cos(\theta_{d2}+\theta_{d3})\ddot{\theta}_{d1}$
2	$\sin\theta_{d2}\sin(\theta_{d2}+\theta_{d3})\ddot{\theta}_{d1}$	$\cos(\theta_{d2}+\theta_{d3})\ddot{\theta}_{d1}$	$\ddot{\theta}_{d2}$
3	$\sin^2\theta_{d2}\ddot{\theta}_{d1}$	$\ddot{\theta}_{d2}$	$\cos\theta_{d3}\ddot{\theta}_{d2}$
4	$\sin^2(\theta_{d2}+\theta_{d3})\ddot{\theta}_{d1}$	$\cos\theta_{d3}\ddot{\theta}_{d2}$	$\ddot{\theta}_{d3}$
5	$\cos\theta_{d2}\ddot{\theta}_{d2}$	$\cos\theta_{d3}\ddot{\theta}_{d3}$	$-\sin(\theta_{d2}+\theta_{d3})\cos(\theta_{d2}+\theta_{d3})\dot{\theta}_{d1}^2$
6	$\cos(\theta_{d2}+\theta_{d3})\ddot{\theta}_{d2}$	$\ddot{\theta}_{d3}$	$-\sin\theta_{d2}\cos(\theta_{d2}+\theta_{d3})\dot{\theta}_{d1}^2$
7	$\cos(\theta_{d2}+\theta_{d3})\ddot{\theta}_{d3}$	$-\sin\theta_{d2}\cos\theta_{d2}\dot{\theta}_{d1}^2$	$\sin\theta_{d3}\dot{\theta}_{d2}^2$
8	$\sin\theta_{d2}\cos\theta_{d2}\dot{\theta}_{d1}\dot{\theta}_{d2}$	$-\sin(\theta_{d2}+\theta_{d3})\cos(\theta_{d2}+\theta_{d3})\dot{\theta}_{d1}^2$	$\dot{\theta}_{d3}$
9	$\sin\theta_{d2}\cos(\theta_{d2}+\theta_{d3})\dot{\theta}_{d1}\dot{\theta}_{d2}$	$-\sin\theta_{d2}\cos(\theta_{d2}+\theta_{d3})\dot{\theta}_{d1}^2$	$\mathrm{sig}(\dot{\theta}_{d3})$
10	$\sin(\theta_{d2}+\theta_{d3})\cos\theta_{d2}\dot{\theta}_{d1}\dot{\theta}_{d2}$	$-\sin(\theta_{d2}+\theta_{d3})\cos\theta_{d2}\dot{\theta}_{d1}^2$	$-\sin(\theta_{d2}+\theta_{d3})$
11	$\sin\theta_{d2}\cos(\theta_{d2}+\theta_{d3})\dot{\theta}_{d1}\dot{\theta}_{d3}$	$-\sin\theta_{d3}\dot{\theta}_{d2}\dot{\theta}_{d3}$	
12	$\sin(\theta_{d2}+\theta_{d3})\cos(\theta_{d2}+\theta_{d3})\dot{\theta}_{d1}\dot{\theta}_{d2}$	$-\sin\theta_{d3}\dot{\theta}_{d3}^2$	
13	$\sin(\theta_{d2}+\theta_{d3})\cos(\theta_{d2}+\theta_{d3})\dot{\theta}_{d1}\dot{\theta}_{d3}$	$\dot{\theta}_{d2}$	
14	$-\sin\theta_{d2}\dot{\theta}_{d2}^2$	$\mathrm{sig}(\dot{\theta}_{d2})$	
15	$-\sin(\theta_{d2}+\theta_{d3})(\dot{\theta}_{d2}+\dot{\theta}_{d3})^2$	$-\sin\theta_{d2}$	
16	$\dot{\theta}_{d1}$	$-\sin(\theta_{d2}+\theta_{d3})$	
17	$\mathrm{sig}(\dot{\theta}_{d1})$		

Figure 7: List of basis functions from the paper [4].



```
LOSS : 109.090579964
```

Figure 8: Total integrated error evolving after each episode with online learning.

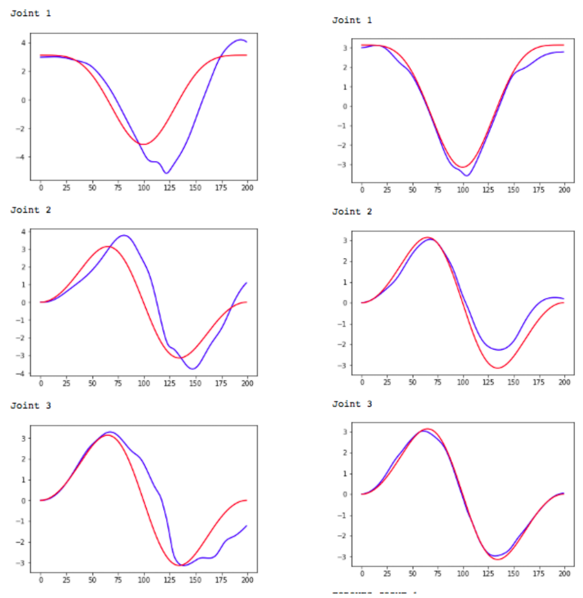
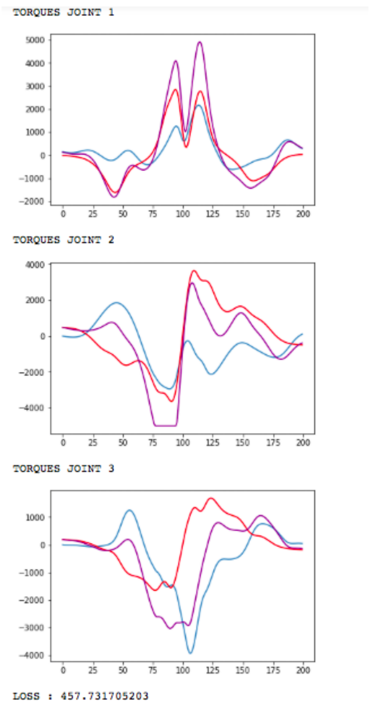


Figure 9: Comparison of joints trajectories. Desired trajectory in red, performed trajectory in blue. Left: PD Controller. Right: PD+neurocontroller after learning.

3 Conclusion

In this work we designed, implemented and compared different algorithms and intelligent controllers for robotic manipulator trajectory tracking. The evolutionary algorithm applied to the neurocontroller showed remarkable results and interesting trajectories, eventhough it is a derivative-free algorithm with no form of error propagation. The small and flexible network architecture played an important role in making the algorithm work well. The reinforcement learning algorithms showed interesting results but didn't scale to a 3DOF robot when training on under 100 episodes. As the literature suggest, many techniques can be used to alleviate the instability, but most of them require long learning times and parallelism. Finally, the dynamical model learning through feedback-error showed the most impressive results, obtaning a quite accurate model of the robot and thus lowering dramatically the total error in the trajectory after 40-50 runs. This method also generalized on unseen trajectories, making the robot accurate and energy efficient.



```
TORQUES JOINT 1
TORQUES JOINT 2
TORQUES JOINT 3
LOSS : 457.731705203
```

Figure 10: Comparison of joints torques. Blue: PD Torques. Red: NN feedforward torques. Purple: sum of torques with threshold.

References

[1] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[2] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[3] Pane Yudha, Prawira. Reinforcement learning for tracking control in robotics. *For the degree of Master of Science in Systems*, 1, 2015.

[4] Miyamoto, H. et al. Feedback-error-learning neural network for trajectory control of a robotic manipulator. *Neural Networks* 1988.

[5] Katic, D. and Vukobratovic, M. Intelligent Control of Robotic Systems *Springer* 2003.