Technion – Israel Institute of Technology

Project: Self balancing an inverted pendulum using Reinforcement Learning

Ido Leffel & Yotam ishay

Supervised by Ayal Taitler

October 2017

Abstract

In this work, we have implemented a version of the celebrated DQN algorithm in order to construct an intelligent controller for self-balancing an inverted pendulum, both in simulation and reality.

Contents

1. Introduction

1.1. The inverted pendulum problem

Balancing an inverted pendulum is a classic experiment in the area of control systems. The inverted pendulum setup consists of a cart driven by a DC motor. The motor can steer the cart left and right on a track. On the cart, a pendulum is mounted such that it can freely rotate around an axis that is perpendicular to the direction of motion of the cart. The objective is to control the motion of the cart such that the pendulum is balanced in its upright position, and the cart is located in a specified location on the track. The schematic diagram in Figure 1 shows the construction of the system including the relevant parameters and variables.

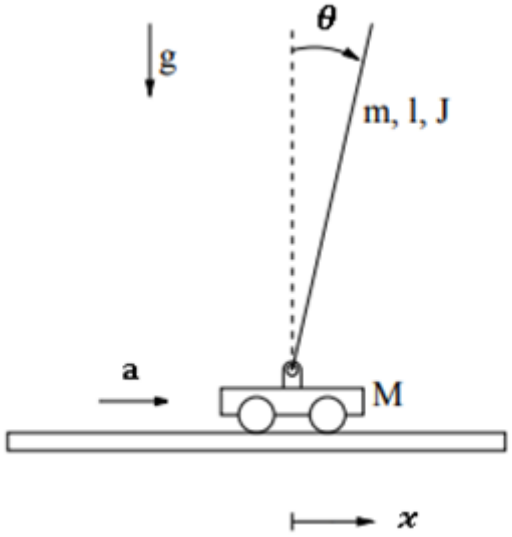

Figure 1: Schematic drawing of the inverted pendulum

1.2. Traditional control

The traditional method for constructing the desired controller is to devise a mathematical model of the system above according to its dynamics, and then use one of several specific techniques for designing a controller. After the design is completed, the controller's performance is tested on the real system.

Due to the relative complexity of the problem, a popular approach is to devise two 'simpler' controllers – a non-linear "swing-up" controller to swing the pendulum to a near upright position,

and a linear controller for stabilizing the pendulum in its upright position (while the pendulum is near its upright position).

The approach above yields excellent results. Its main downside is the necessity to construct a suitable mathematical model and solving its dynamics – potentially a hard task due to the non-linearity of the problem (and an extremely difficult task in more complex real systems). For this reason, the question of designing a controller for a system without using a model is a fairly important one.

1.3. Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning inspired by behaviorist psychology, concerned with how a 'software agent' (essentially a controller) ought to take actions in an environment so as to maximize some notion of cumulative reward.

RL methods essentially deal with the solution of optimal control problems using on-line measurements. The following diagram depicts the standard RL setup in which the mentioned 'agent' interacts with the environment:

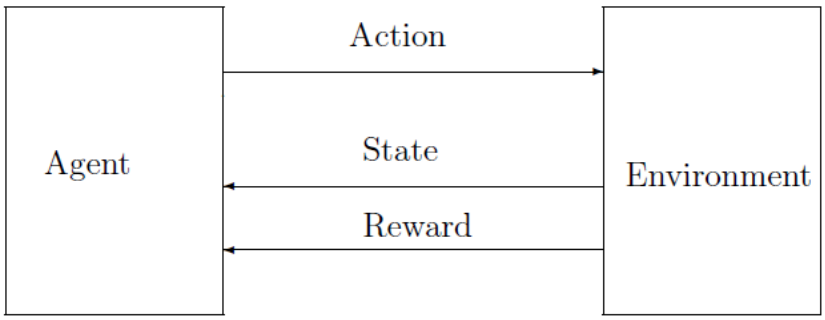

Diagram 1: standard RL setup

Let us assume that the agent is interacting with the environment in discrete time steps. At each time step k, the agent observes the state s_k which represents the current physical state of the system. Then, the agent takes an action a_k and receives a reward $r(s_k, a_k)$. The action taken by the agent affects the environment and advances the state s_k to a new state s_{k+1} according to the state

transition matrix $P(s_{k+1}|s_k, a_k)$, and so on. Let us also assume that both the action-space (A) and state-space (S) are discrete.

We define the cumulative reward (or 'return') $R = \sum_{k=1}^{N} r(s_k, a_k)$, where N is the number of discrete time steps of the above process. The goal of the agent is to learn a policy $\pi: S \rightarrow A$ that maximizes the expected return $E[R]$.

The main approaches for learning in this context can be classified as follows:

- Indirect learning: Estimate an explicit model of the environment and compute an optimal policy for the estimated model.
- Direct learning: The optimal control policy is learned without first learning an explicit model.

In this work, our focus will be on the second approach.

1.4. DQN

In recent years, several new RL algorithms have been devised. Specifically, a new Direct-learning algorithm named DQN (Deep Q-Network) (Mnih et al. 2015)

An important concept in the context of DQN is that of a Q-function (or a state-action value function). $Q^{\pi}: S \text{ x } A \rightarrow R$ depicts the expected return after taking an action a from state s, and thereafter following the policy π. The optimal Q-function, denoted by $Q^*(s, a)$, depicts the expected return after following *an optimal policy* π^*. $Q^*(s, a)$ obeys the celebrated Bellman's equation of optimality:

$$Q^*(s_k, a_k) = E_{s_{k+1}}[\, r(s_k, a_k) + max_{a'} Q^*(s_{k+1}, a') \mid s_k, a_k \,] \qquad (1)$$

In this context, our goal is to find $Q^*(s, a)$ which guarantees optimal control of the pendulum. In most real world cases, it is no feasible to compute it with brute force according to the above recursive equation (either due to the large domain of Q which is of size $|SxA|$, or since we do not possess perfect information of the RL framework). Therefore, it is common practice to try to approximate the value of $Q^*(s, a)$ with different kinds of functions.

The DQN algorithm uses a deep neural network (a 'Q-network' denoted by Q_θ) for function approximation in the following way:

The training of the network is done by optimizing the weights of the network (θ_i) in a way which minimizes the expected Temporal Difference (TD) error of Bellman's equation:

$$L(\theta) = E_{s_k, a_k, r_k, s_{k+1}}[||Q_\theta(s_k, a_k) - y_k\,||_2^2] \qquad (2)$$

Where

$$y_k = \begin{cases} r(s_k,a_k) & , \quad s_{k+1}\ is\ terminal \\ r(s_k,a_k) + max_a Q_{\theta_{target}}(s_{k+1},a) & , \quad s_{k+1}\ is\ not\ terminal \end{cases} \qquad (3)$$

In each training step k, the tuple $< s_k, a_k, r_k, s_{k+1} >$ is stored in an Experience Replay Buffer D, from which samples are drawn uniformly in order to reduce time correlations to train the network.
The DQN maintains two separate Q-networks: one current Q-network with parameters θ and another target Q-network with parameters θ_{target}.
Once every fixed number of training steps, DQN sets θ_{target} to θ, to avoid frequent updates of the target and therefore noisy learning.
The weights θ_i can then be trained by stochastic gradient descent (SGD) and back-propagation

$$\theta_{i+1} = \theta_i + \alpha \nabla_{\theta_i} L_i(\theta_i) \qquad (4)$$

Where α is the learning rate.

The max operator in DQN (3) uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, it is possible to decouple the selection from the evaluation. This is the idea behind Double Q-learning (Double DQN), in which the following equations replace eq. (3):

$$y_k = \begin{cases} r(s_k,a_k) & , \quad s_{k+1}\ is\ terminal \\ r(s_k,a_k) + Q_{\theta_{target}}(s_{k+1},a_{k+1}) & , \quad s_{k+1}\ is\ not\ terminal \end{cases} \qquad (5)$$

$$a_{k+1} = argmax_a Q_\theta(s_{k+1}, a) \qquad (6)$$

In the following sections, our version of the DQN algorithm for self-balancing an inverted pendulum will be discussed in more detail.

1.5. Simulation vs. Reality

The initial goal of the project was to devise a controller (a DQN agent) to balance a real inverted pendulum. After building the environment and establishing communication with the system (further details will be provided in section 3), we have written the code for the DQN agent. As will be shown below, using the DQN algorithm requires setting values for a relatively large number of parameters (both for the Q-network and for the algorithm itself) in addition to devising a good reward function. Testing and evaluating performance for each choice of parameters and reward takes quite a long time on the real system, and for some parameter values can even destroy the system setup without proper care.

After considerable effort and little success, the notion of working with a computer simulation of the inverted pendulum had been suggested. The motivation for simulating the inverted pendulum environment was to test different parameters and rewards in a quicker and more efficient way (five to ten times faster using a GeForce GTX Titan X GPU) , and hopefully acquire better understanding both of general qualities of the system and of specific numeric values for the parameters and reward function. An additional idea was to train the agent in the simulated environment and then load the resulting Q-network to the agent of the real environment, hoping to shorten the learning time and eventually achieve better performance of the DQN agent.

2. Simulation

2.1. Setup

The learning environment – a simulated version of the inverted pendulum - is a custom built simulation based on the 'Cartpole' simulation of OpenAI gym (a toolkit for developing and comparing reinforcement learning algorithms).

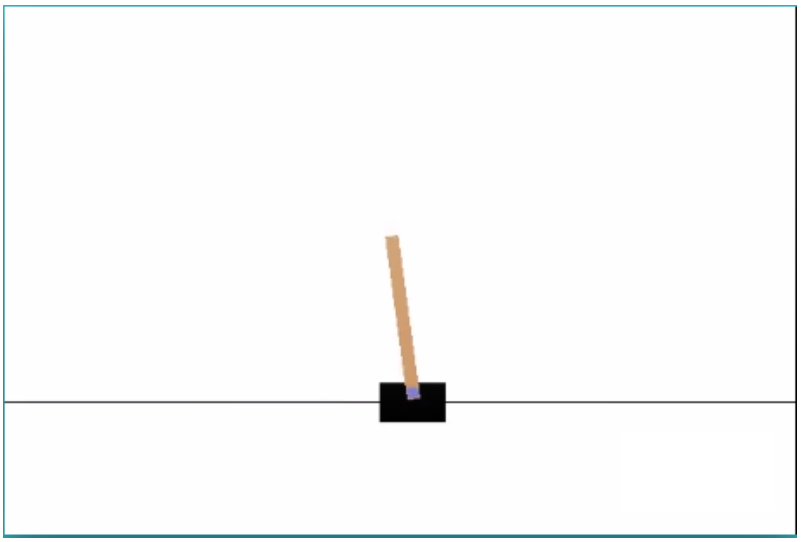

Figure 2: The inverted pendulum simulation environment

We have made a few modifications to the original environment for the purpose of modeling our real inverted pendulum environment better.

The simulation model takes into account the following parameters: (numeric values were taken from measurements of the real pendulum)

Parameter	Symbol	Value	Units
Cart mass	M	0.496	Kg
Pendulum mass	m	0.231	Kg
Pendulum length	l	0.328	m
Force magnitude	F	10	N
Gravity	g	9.8	$\frac{m}{s^2}$
Time step (control loop time)	T	0.035	sec

Table 1: Simulation parameters

The simulation model is a discrete time second order dynamics of the system's equations of motion. The equations can be easily derived using the Lagrangian of the system

$$L = \frac{1}{2}(M+m)\dot{x}^2 - ml\dot{x}\dot{\theta}\cos(\theta) + \frac{1}{2}ml^2\dot{\theta}^2 - mglcos(\theta) \qquad (7)$$

Euler-Lagrange equations and some algebra lead to the following systems equations of motion:

$$(M+m)\ddot{x} = F + ml\ddot{\theta}\cos(\theta) - ml\dot{\theta}^2\sin(\theta) \qquad (8)$$

$$l\ddot{\theta} = \ddot{x}cos(\theta) + gsin(\theta) \qquad (9)$$

2.2. The DQN controller

In order to train a DQN agent to perform in the simulated environment we have used an implementation of the Double DQN algorithm in python with Tensorflow (written by Ayal Taitler). We simulated each attempt to self-balance the pendulum as an independent episode comprised of discrete time steps. At the beginning of each episode, the pendulum was placed at a random initial state. Each episode was terminated upon passing a constant maximum number of steps, or when the cart's distance from the center exceeded a pre-defined value. At each step, the environment returned a reward (as will be discussed in the next section).

The DQN controller is a non-linear neural controller, a fully connected Multi-Layer Perceptron with 3 layers (2 hidden layers and an output layer). The first hidden layer is of 40 units and the second layer is of 100 units. The output layer is of 5 units (which correspond to the possible 5 discretized actions we've allowed the controller to take), and the input layer is of 5 units as well (which correspond to the representation of each state of the system as the following 5-tuple $< x, \dot{x}, \cos(\theta), \sin(\theta), \dot{\theta} >$).

All activations are the linear rectifier $f(x) = \max(0, x)$.

The DQN controller is essentially a map between states s_k (the controller input) and actions a_k (the controller output). As was mentioned before, the Q-network $Q_\theta(s_k, a_k)$ is the function approximation of the optimal Q-function $Q^*(s_k, a_k)$, and from there the transition to the action is immediate and given by

$$a_k = argmax_{a'}(Q_\theta(s_k, a')) \quad (7)$$

To explore the environment, we have used the ϵ-greedy heuristic: in each step, the DQN controller chooses an action according to eq. (7) with probability $1 - \epsilon$ and a random action with probability ϵ.

2.3. DQN hyper-parameters and reward definition

The use of the DQN algorithm requires setting values for several additional hyper-parameters. The list of those parameters used throughout the simulation (which led to the best simulation results) is given in table 2.

Parameter	Value	Description
α	0.000025	Learning rate
ϵ	0.1	ϵ-greedy exploration probability
D	200000	Replay buffer size
N	300	Episode's maximum steps
γ	0.999	Discount factor
Mini-batch	32	Mini batch size sampled from buffer
C	50	Update rate of Q target network

Table 2: Hyper-parameters used by the learning algorithm

As was mentioned before, at each step of each episode, the state of the system is observed. After much deliberation, the state representation was chosen to be the following 5-tuple:

$$< x, \dot{x}, \cos(\theta), \sin(\theta), \dot{\theta} >)$$

The reason for using a sinusoidal function of θ (instead of θ itself) was to account for the periodicity of the pendulum's angle – a state with angle θ should be considered identical to a state with angle $\theta \pm 2\pi k, k \in Z$. The angle values in the simulation were not confined to a domain with length 2π hence the need for sinusoidal functions. Moreover, we have used both $\cos(\theta)$ and $\sin(\theta)$ in the state representation in order to account for both magnitude and sign of θ, thereby showing a complete and injective representation of the state to the learning agent.

There are 5 discretized actions the agent can take at each step (which correspond to the force magnitude of the action in $[N]$): $(-10, -5, 0, 5, 10)$. The sign indicates the direction of the applied force. The reason for choosing the above action space was to give the controller the possibility of using both 'hard' (10) and 'soft' (5) actions in each direction, in addition to no action at all (0).

The learning is episodic and there are no terminal states. The DQN agent receives the following reward in each step:

$$r(s_k, a_k) = -n * (q(0) * |x_k| + q(1) * |\dot{x_k}| - q(2) * (2 - |1 - \cos(\theta_k)|) + q(3)|\dot{\theta}_k| + r * |a_k|)$$

The reward parameters are listed in Table 3.

Parameter	Value	Description
n	0.1	Scaling factor for the reward
q	[5,0.1,7.5,0.1]	Weight vector for the state variables
r	0.01	Weight of the control effort

Table 3: Reward parameters

The following issues were addressed while contemplating several reward functions and eventually lead to the above final reward definition:

- *The reward function shall include all state variables*: since the goal is to balance the pendulum in its upright position and place the cart at the center of the track, our first reward functions included only θ and x (and excluded the velocities and control effort). We figured that since the state representation for the controller includes the full state of the system, the controller will have the same information and will be able to learn just as well while being rewarded only for its θ and x values (the values we wish to control). Experimenting with different rewards revealed that including all state variables improves the learning substantially – the full state rewards lead to much better performance.
- *Changing the dynamic range of the reward can greatly improve learning*: In our first tries, we normalized the rewards to a fixed domain $r(s_k, a_k) \in [-c, c]$ for some constant c. We thought that the normalization will help the learning process for the neural network since the gradients are affected by the reward function's value, according to eq. (4) (due to the propagation of gradients during the back-propagation algorithm, the network weights could take on extreme values when the reward values are extreme). However, when we stopped normalizing the reward (but still used some scaling coefficient n to control its magnitude), the controller performed much better. The final domain that lead to the best performance was $r(s_k, a_k) \in [-\mathrm{a} * 10^4, \mathrm{b} * 10^2]$ for small positive constants a, b. Intuitively, this domain suggests that the agent is

gravely penalized when it is far from the desired location and is rewarded less as it approaches the desired location.

- *The type of norm used in the reward function is of great importance to the learning process:* We had decided to use a reward function that measures the distance of the current state from the desired state. In order to account for the relative importance of the state variables (angle is the most important, then the position, velocities and control effort) we had decided to use a weighted norm (a norm that involves multiplication by a particular weight function, in our case a vector of the chosen state weights) and specifically the l_p norm. The choice of weights, as well as the choice of p, had significant impact on the performance of the controller.

2.4. Results

We show the performance of the DQN agent for different control loop times (Figure 3) and different action discretization (Figure 4).

Figure 3: Simulation results for different control loop time T

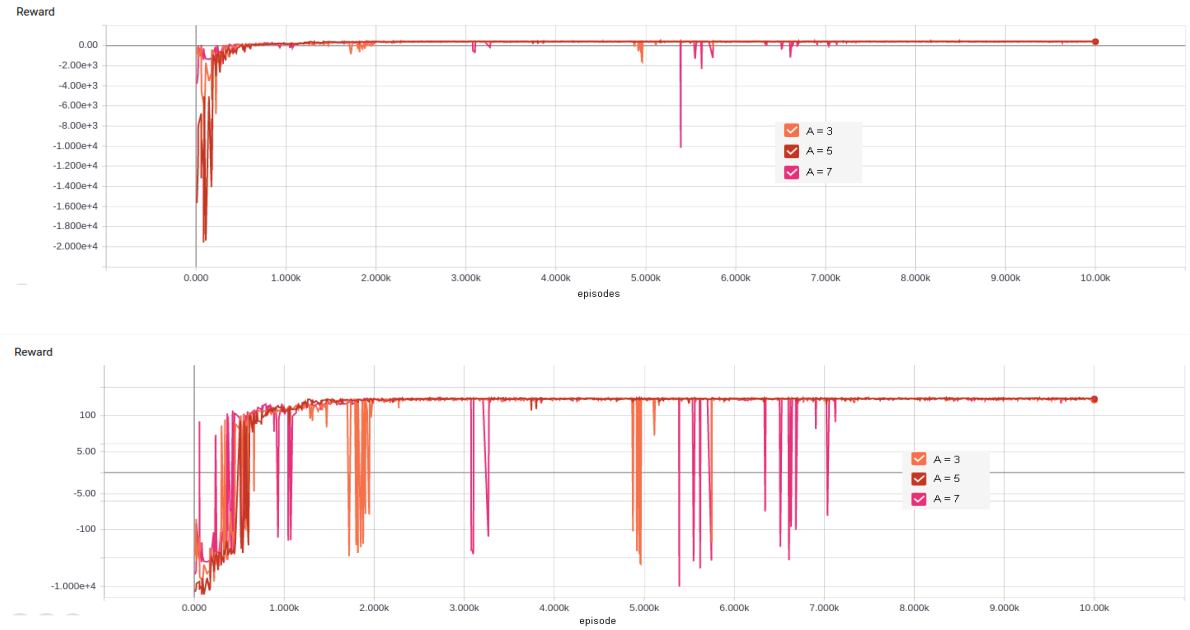

Figure 4: Simulation results for different action discretization A

Both figures 3 and 4 show two graphs – those are the same graphs with different reward scales.

It can be seen from figure 3 that the DQN reward converges for all three (slow, intermediate and fast) values of T. As T increases, the reward rises faster since the agent explores better for a constant number of steps (the time length for each episode increases with T hence the agent reaches 'further' steps at earlier times). However, for high enough T the DQN agent will fail to reach to the above results and will not be able stabilize the pendulum in its upright position, since the control time will be too slow for maintaining the pendulum in this unstable stationary point. All three DQN agents in figure 3 performed nicely and achieved the goal of stabilizing the pendulum in its upright position while keeping the cart at the center of the track.

In figure 4 we have shown the performance of three DQN agents with different action capabilities. All three agents eventually achieved satisfactory results and succeeded in balancing the pendulum and cart. As can be seen in figure 4, the 5 action discretization agent (which corresponds to using one 'hard' (10) and one 'soft' (5) action in each direction, in addition to no action at all (0)) had risen the fastest and maintained near constant plateau during the rest of the training episodes, outperforming the other two agents for the above constant number of training episodes.

3. Reality

3.1. Setup

The workspace is comprised of the following 5 key elements:

- PC – Intel core i7 4770 @ 3.4 GHz, 8 GB RAM on a Windows 7 64-bit environment, working in Matlab.
- National Instrument Multifunction I/O device (NI PCIe-6321) – used as an interface between the PC (software) and the amplifier and motor (hardware).
- UPM 1503 amplifier – for amplifying the analog signal coming from the NI device to the motor.
- Globe Motors 537A354 DC motor (equipped with an encoder for measuring the required signals) – for driving the cart.
- A pair of incremental encoders – for acquiring the position x and angle θ of the cart and pendulum respectively.

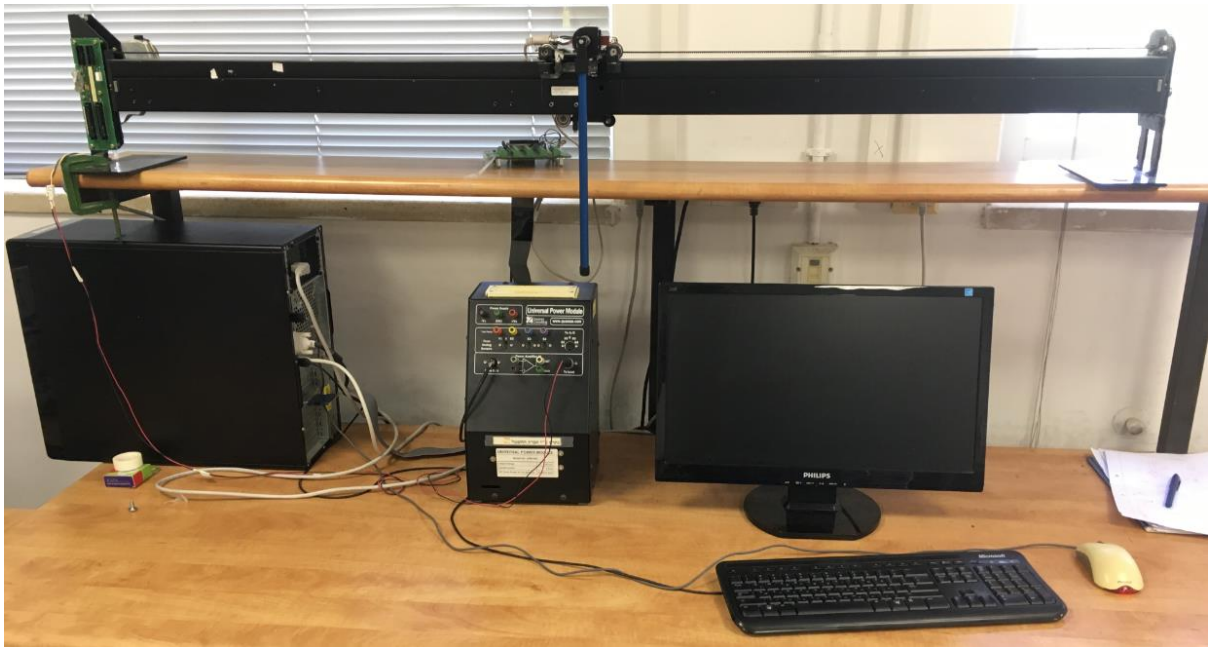

Figure 5: The inverted pendulum workspace

Figure 6 depicts the block diagram of the system's operation:

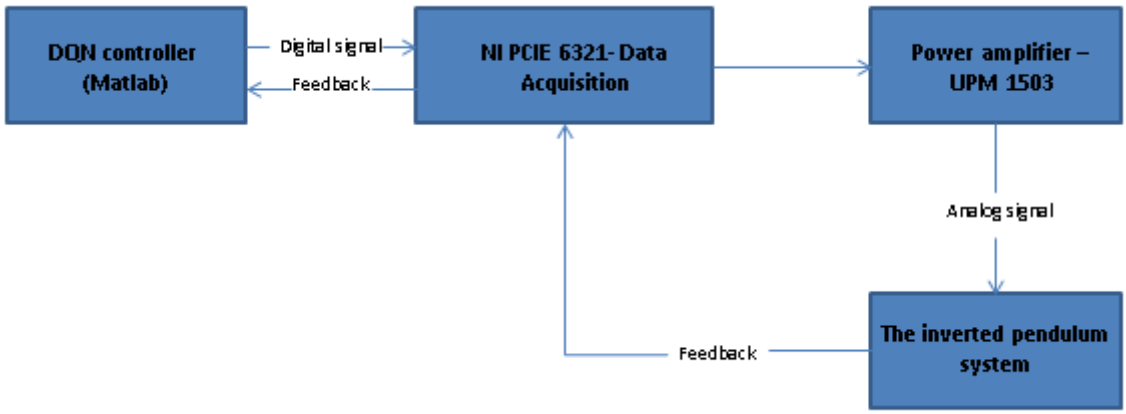

Figure 6: Workspace block diagram

The DQN controller output is an action for the DC motor. It is a digital signal (the action's value is a discretized voltage in Volts). The signal is transmitted to a D\A converter located in the NI PCIe-6321 device. Then, the analog signal goes through the UPM 1503 amplifier and from there it reaches the motor.

As was mentioned before, two incremental encoders are installed for acquiring the position x and angle θ of the cart and pendulum respectively. The encoders' measurements are transmitted to the NI PCIe-6321 device and from there they reach the DQN controller. The encoders are connected to the NI device according to Tables 4 and 5.

Pin number (NI)	Pin number (Encoder)	Wire color	Description
14	2	Brown	VCC
53	3	Blue	DGND
37	6	Green	A
45	8	Black	B

Table 4: Encoder 1 (measures x) - connects to CTR1 input channel of the NI device

Pin number (NI)	Pin number (Encoder)	Wire color	Input (NI)
8	4	Black	VCC
15	1	Blue	DGND
42	3	White	A
46	5	Orange	B

Table 5: Encoder 2 (measures θ) - connects to CTR2 input channel of the NI device

The Ni device is connected to the amplifier according to Table 6.

Pin number (NI)	Description	Input (NI)
55	Shield	AGND
21	Signal	Vout

Table 6: Analog outputs

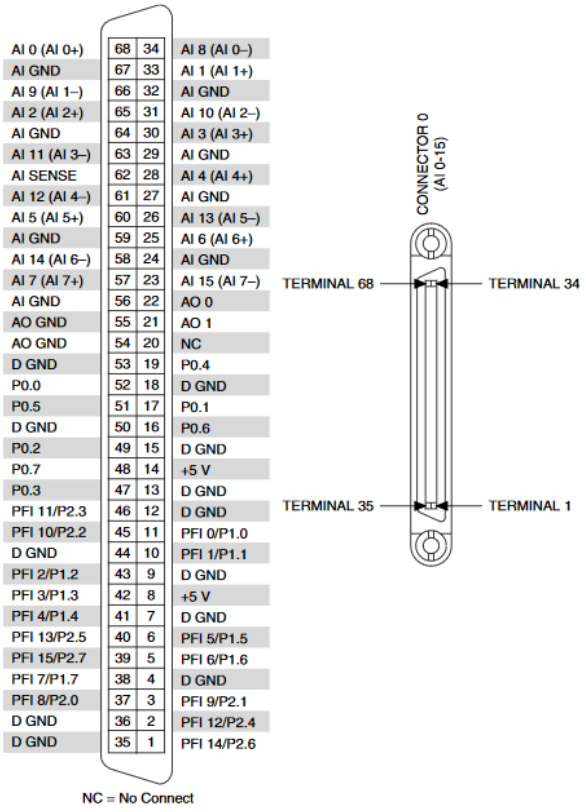

Figure 7: NI PCIe-6321 pinouts

The system parameters are listed in table 7. As can be seen in table 7, all parameters that were part of the simulation model have the exact same values. The parameters that were not part of the simulation model are listed below.

Parameter	Symbol	Value	Units
Cart mass	M	0.496	Kg
Pendulum mass	m	0.231	Kg
Pendulum length	l	0.328	m
Force magnitude	F	10	N
Gravity	g	9.8	$\frac{m}{s^2}$
Motor Torque constant (and back emf constant in SI units)	Km	0.0767	$\frac{Nm}{Amp}$ $\frac{V}{rad/s}$
Motor Armature Resistance	Rm	2.6	Ω
Gearbox ratio	Kg	3.71	N/A
Motor pinion # teeth	NA	24	Teeth
Motor pinion diameter	r	0.00635	m

Table 7: System parameters

3.2. The DQN controller

The DQN controller is implemented in Matlab. The implementation in Matlab instead of python was due to problems we encountered in the data acquisition part - there is an API in python for interfacing with the NI-DAQmx driver (the driver for the NI PCIe-6321 device) named 'nidaqmx'. However, it is relatively new and contains bugs (specifically in the area of reading data from encoders). Interfacing with Matlab was better documented and easier to achieve.

As was done in the simulation part, each attempt to self-balance the pendulum was an independent episode comprised of discrete time steps. Each episode was terminated upon passing a constant maximum number of steps, or when the cart's distance from the center exceeded a pre-defined value. At each step, the reward was given according to the measurements from the encoders.

The DQN hyper-parameters and reward definition are exactly the same as in the python simulation. Certain adjustments had to be made since the system was real (as opposed to simulated):

- *Forcing a constant control loop time (i.e. time step)*: Setting each step of each episode to last a constant time period (by pausing the progression to the next loop iteration until

the end of the desired time) has proven to be extremely important in training the pendulum. When we first implemented the DQN algorithm (without forcing constant time steps), each loop iteration lasted a different amount of time and the controller's performance barely improved with time. This action had allowed us to model the pendulum dynamics as an MDP (Markov Decision Process) instead of an SMDP (Semi-Markov Decision Process), which is a more complicated process (SMDPs model stochastic control problems arising in Markovian dynamic systems where the sojourn time in each state is a general continuous random variable). As a result, we could apply standard reinforcement learning methods for MDPs (specifically DQN).

- *Setting a control loop time that will be long enough to capture the* $action \rightarrow state\ observation \rightarrow\ reward$ *causation*: initially we had set the control loop time to be 20 milliseconds (which led to good performance in the simulation part). We later discovered that the control loop time in Matlab (for the real pendulum) is substantially longer , and were finally able to reduce it to 35 ms. While working with a 20 ms time step, we hadn't kept the $action \rightarrow state\ observation \rightarrow\ reward$ causation hence the data used by the controller was corrupt, and the controller's performance failed to improve with time.
- *The training of the Q-network parameters is performed at the end of each episode (instead of during each training step)*: After each episode, 300 32-sized mini-batches are randomly collected from the Experience Replay Buffer and then used to train the Q-network parameters (θ). In addition, the update of θ_{target} to θ (which happens each C episodes) is performed after the episode ends.

  The above changes to the original DQN algorithm had allowed us to decrease of the control loop time from over 100 ms to below 35 ms.

3.3. Results and Discussion

We show the performance of the DQN agent for different control loop times (Figure 7) and for different action discretization (Figure 8).

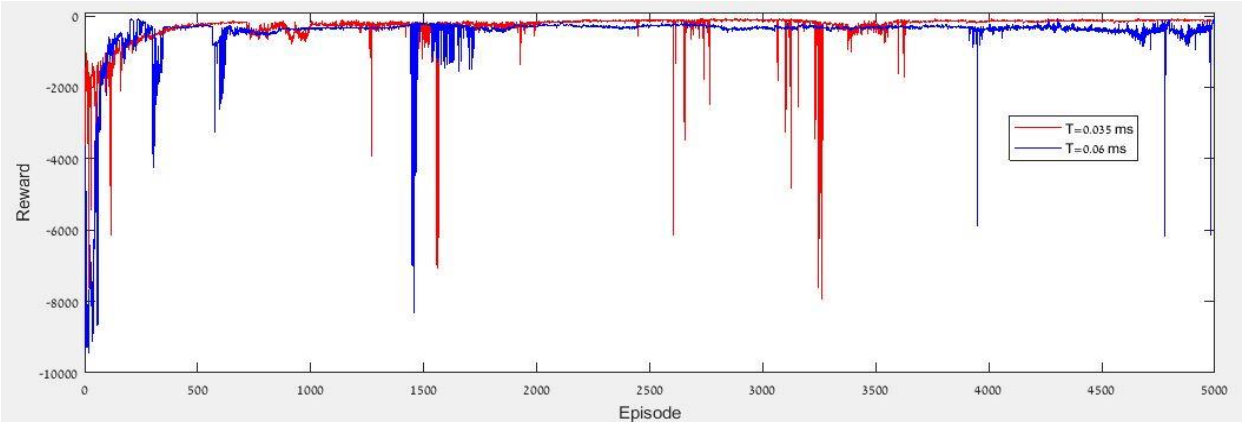

Figure 7: Reality results for different control loop time T

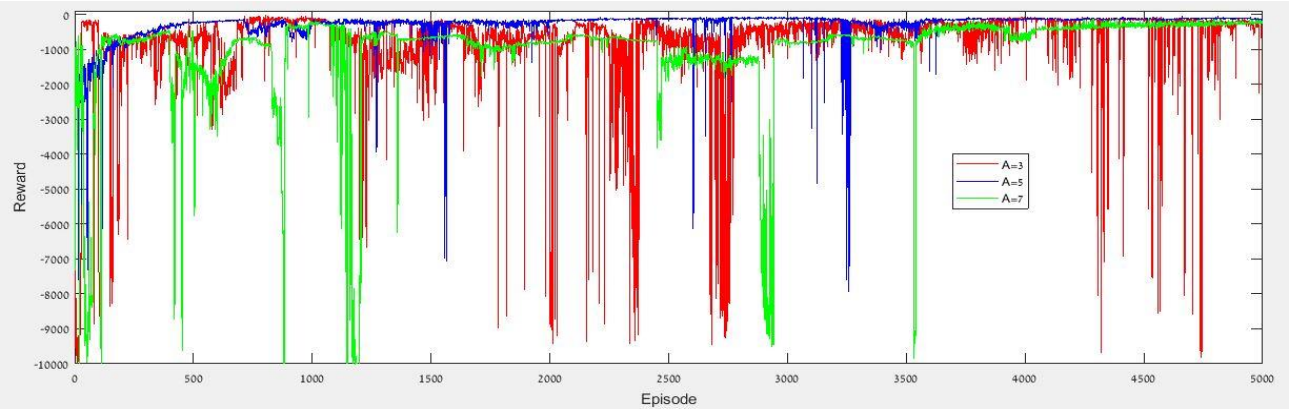

Figure 8: Reality results for different action discretization A

It can be seen from figure 8 that the DQN controller is learning with time and the reward converges to a near constant range for both control loop times (action discretization is 5 in both cases). However, the reward doesn't reach positive values (as it did in the simulation part) and the controller fails to balance the pendulum in its upright position.

From figure 8 and from observing the pendulum in real time we can deduce that the 3-actions controller performs poorly and hardly manages to better with time, while the 5-actions controller performs best and shows improvement with time.

The DQN controller (in all its versions) ultimately failed to balance the real pendulum in its upright position. During our attempts to achieve this unachieved goal we had tried several ideas and reached several conclusions:

- *The simulated model is not a good enough model for the real system*: As was shown in tables 1 and 7, the simulated model does not take into account several parameters of the real system. In addition, it does not account for several other constraints and effects of the real physical system and environment such as friction, action and data acquisition delays, measurement noise and other disturbances. More importantly, the simulated model is *deterministic* whereas the real system is *stochastic and uncertain* in that future trajectories cannot be precisely predicted. We previously stated that one of the goals of the simulation was to train the agent in the simulation and then load the resulting Q-network to the agent of the real system. In practice, this attempt failed and the agents had actually shown worse results when their Q-networks were initialized with the simulation Q-network weights and biases. In conclusion, a less simplified model is needed for achieving better results in reality (which in turn can direct us to a more suitable network architecture and hyper parameters).
- *Exploration is probably not the problem but can help nonetheless:* Another benefit of the simulation is that it is easy to start each episode in a completely random state. This fact, in addition to the relative ease in swinging the simulated pendulum over the real pendulum, led us to believe that the real pendulum fails to learn properly because it does not explore the environment as good as the simulated pendulum. For better exploring the real environment we had used two methods:
  (1) Increasing the exploration rate at the beginning and decreasing it gradually.
  (2) Using prior knowledge during the learning and directing the algorithm to the more interesting region of the state and action spaces.

  Method (1) helped in achieving faster convergence of the reward, but did not lead to higher final rewards. In method (2) we have devised a very basic swinging controller and used its policy instead of the controller in every fixed number of episodes, in order to direct the algorithm to the states in which the pendulum is closer to its upright position. This method helped a little but also didn't improve the reward sufficiently. Using a smarter controller to guide the algorithm can presumably help achieve better results.

- *The control loop time is significant*: Even though the simulation converged with different (long) control loop times, we noticed that in reality, the control loop time is much more

important to the performance of the controller. Further decrease of the control time (both of our version of the DQN algorithm and of our implementation in Matlab) can help.

- *Decaying the learning rate and exploration rate is important:* Intuitively, as the agent betters its performance and converges toward optimal control, its learning rate should decrease in order to fully converge to the optimal policy. The exploration rate should also decrease with time for the same reason – as it converges toward optimal control, we wouldn't want it to take random actions (while using the ϵ-greedy for example).

4. Summary and Conclusions

In this work we have addressed the task of balancing an inverted pendulum using a non-linear neural controller trained using a version of the DQN algorithm. We have tried to do so both to a computer simulated pendulum and to a real live pendulum using different tools to account for the difference between simulation and reality.

While the controller in the simulation is shown to learn near optimal policy, the controller for the real pendulum fails to learn a sufficiently good policy for balancing the pendulum. Nevertheless, several important conclusions were drawn and the differences between simulation and reality were investigated.