Reinforcement Learning Toolbox

George Claudiu Andrei  
University of Lorraine, Vandœuvre-lès-Nancy

July 16, 2022

**Abstract:** Reinforcement Learning Toolbox provide functions and blocks for training policies using reinforcement learning algorithms including Deep Q-Network and Deep Deterministic Policy Gradient. These policies can be used to implement controllers and decision-making algorithms for complex systems such as robots and autonomous system. It is also possible to implement the policies using deep neural networks, polynomials or look-up tables. The toolbox give the possibility to train policies by enabling them to interact with environment represented by Matlab or Simulink. The evaluation of the algorithms, experimentation with hyperparameter setting, and monitor training progress is also provided.

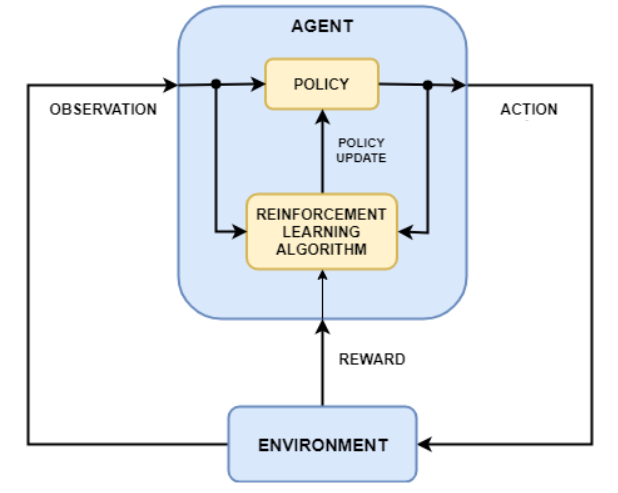
This tutorial paper is organized as follows. The next section presents a briefly introduction to Reinforcement Learning in comparison with traditional Control theory and its main algorithms. Then, a step-by-step Deep Q-Network and Deep Deterministic Policy Gradient algorithm implementation using Reinforcement Learning toolbox on a speed control DC Motor and a Inverted Pendulum will be provided.

**Keywords:** Reinforcement Learning, Deep Deterministic Policy Gradient, Deep Q-Network, Model-free approach.

# What is Reinforcement Learning

Reinforcement learning is a goal-directed computational approach where an Agent learns to perform a specific task by interacting with an unknown dynamic environment. This learning approach makes the Agent able to take a series of decisions in order to maximize the cumulative reward for the task without human intervention and without being explicitly programmed to achieve a specific task.

The following figure shows a general representation of a reinforcement learning scenario with respect to traditional control.



The goal of reinforcement learning is to train an *Agent* in order to achieve a specific task within an unknown *environment.* The agent receives *observations* and a *reward* from the environment and sends *actions* to the environment. The reward is a measure of how successful an action is with respect to completing the task goal.

Here is a brief description of the components involved:

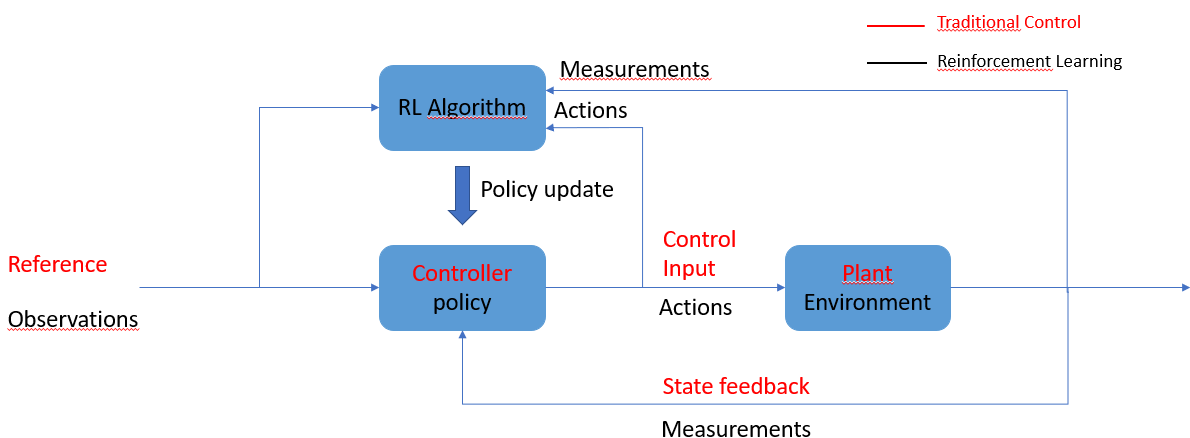
* **Environment** is everything which exists outside the agent. This includes the plant but also any other effects such as a measurement noise, disturbances, filtering and so on.
* **Policy** uses observations and reward signal from the environment and based on these, applies action to the environment. Typically, the policy is a function approximator with tunable parameters, such as a deep neural network.
* **Reward** is one of the most important design function used by Reinforcement Learning to know when the system is approaching its objective or task, e.g.. balancing the pendulum. It is similar to a cost function used in Linear Quadratic Regulator (LQR) based control except a reward function tries to maximize the value and not minimize it.
* **Reinforcement Learning Algorithm**: The Learning Algorithm continuously updates the policy parameters based on the actions, observations, and reward. The goal of the Learning Algorithm is to find an optimal policy that maximizes the cumulative reward received during the task.
* **Actions** are the set of the control signals (e.g. position and velocity of a DC Motor) that are used by the Reinforcement Learning Algorithm, reward function and policy.

In other words, reinforcement learning involves an agent learning the optimal behavior through repeated trial-and-error interactions with the environment without human involvement.

As an example, consider the task of parking a vehicle using automated driving system. The goal of this task is for the vehicle computer (**agent**) to park the vehicle in the correct position and orientation. To do so, the controller collects data from cameras, accelerometers, gyroscopes.

# Reinforcement Learning for Control Systems Applications

The behavior of a Reinforcement Learning policy which is how the policy observes and generates actions to complete a specific task in an optimal manner, is similar to the operation of a controller in a control system- Reinforcement learning can be translated to a control system representation using the following mapping:



|  |  |
| --- | --- |
| **Reinforcement Learning** | **Control Systems** |
| Environment | Everything that is not the controller including the plant, the reference signal, and the error computation. In general, the environment can also include additional elements, such as:   * Measurement Noise * Disturbance Signals * Filters * Analog-To-Digital and Digital-To-Analog converters and so on. |
| Observation | Any measurable value from the environment that is visible to the agent which depending on the available sensors could be error signal, state system from the environment. It is also possible create agents that observe, for example, the reference signal, measurement signal, and measurement signal rate of change. |
| Action | Control Input or manipulated variables |
| Reward | Function of the measurement, error signal, or some other performance matric. For example, it is possible to implement reward functions that minimize the steady-state error while minimizing control effort. |
| Policy | Controller |
| Learning Algorithm | Adaptation mechanism of an adaptive controller |

# Reinforcement Learning Workflow

The general workflow for training an agent using Reinforcement Learning Toolbox includes the following steps:



Figure : RL workflow

1. **Formulate Problem** – Define the task for the agent to learn, including how the agent interacts with the environment from set of observations and actions point of view and any primary and secondary goals the agent must achieve.
2. **Create Environment** – Define the environment within which the agent operates, including the interface between agent and environment and the environment dynamic model.
3. **Define Reward** – Design the reward signal that the agent uses to measure its performance with respect to task goals and how this signal is calculated from the environment.
4. **Create Agent** – Create the agent, which includes defining a policy representation and configuring the agent learning algorithm and its hyperparameters.
5. **Train Agent** – Train the agent policy representation using the defined environment, reward and agent learning algorithm.
6. **Validate Agent** – Evaluate the performance of the trained agent by simulating the agent environment together.
7. **Deploy Policy** – By code generation in order to try the trained agent on a real hardware

# Example: Deep Q-Network on a Linear System

In this section, the implementation of the Deep-Q Network algorithm on a Speed Control DC Motor Linear system will be analyzed step-by-step, following the workflow previously shown. At the end of each step of the workflow the MATLAB script implementation will be also provided.

There are various uses of direct current machines in the industry. DC motors are used in both high and low power applications as well as fixed and variable speed electric drives. For example, their applications range from low power toys, spinning and weaving machines, vacuum cleaners, elevators, electric traction and so on. The speed of a DC motor can be adjusted easily, by changing voltage and current depending on the type of the DC motor used.

## Control Problem

Consider the following state space representation of a DC Motor second-order discrete-time model:

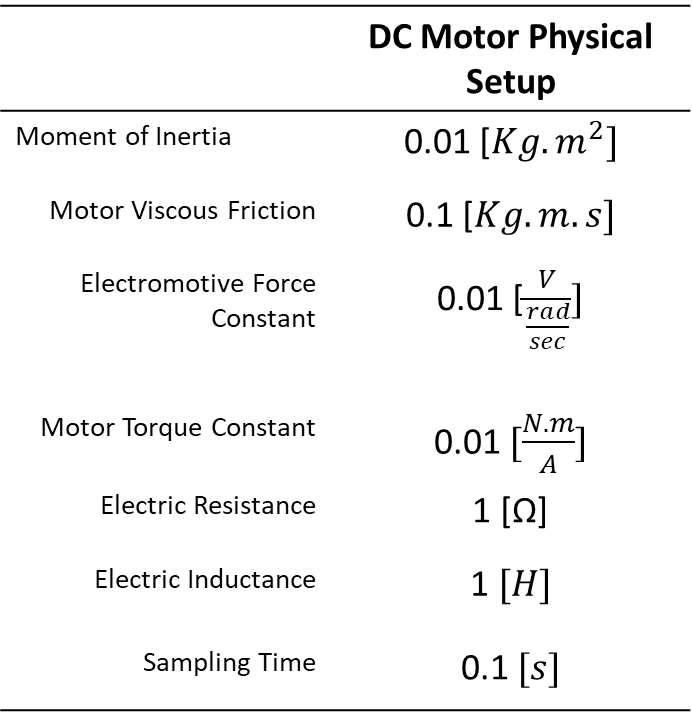
*] V*

Where:

This model was obtained by discretizing a continuous-time model of a real DC Motor. The discretization was performed with the zero-order-hold method using a sampling time of . It is assumed that the DC Motor can only rotate counterclockwise with a saturated range of

For the infinite-horizon LQT problem, the goal is to design an optimal controller for the system which ensures that the output tracks a step reference speed trajectory and acts on the difference between the two, the Control Problem can be seen as regularization of the form:

In the following figure the physical setup of the DC Motor is shown:



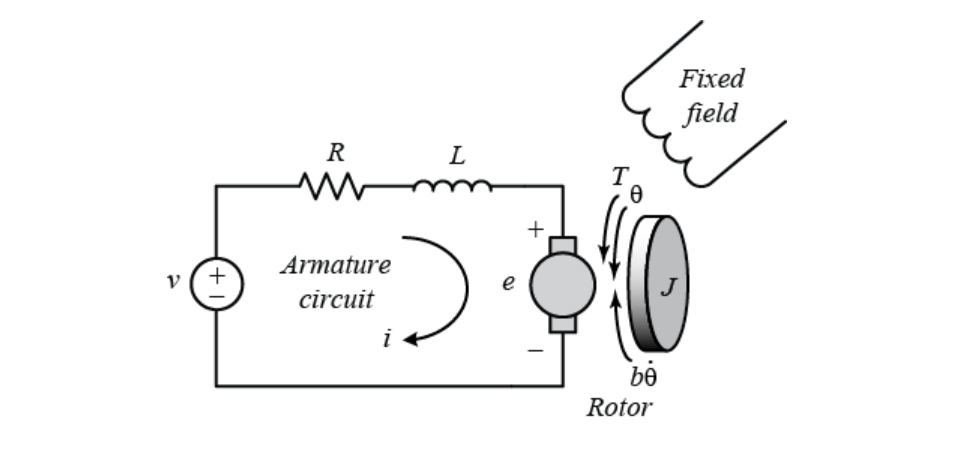
## Create Environment

In a Reinforcement Learning scenario, the environment model the dynamics with which the agent interacts. In particular, the dynamics include the plant modeling and also the disturbances. Practically speaking, the environment is everything except the controller defined by the Reinforcement Learning Algorithm, reward and policy.

Usually what is done to solve a Reinforcement Learning problem is to train in a simulated environment which contains the plant and then improve its behavior by continuing to train the Agent it in the real environment so that it is also able to face off against a real environment characterized by uncertainty and disturbances.

## .1 Modelling of the simulated environment: DC Motor

The electric circuit of the armature and the free-body diagram of the rotor are shown in the following figure:



It is assumed that the control input of the system is the Voltage source applied to the motor’s armature, while the output reached by the sensor is the rotational speed of the shaft. The rotor and shaft are assumed to be rigid. It is also assumed that the friction torque is proportional to shaft angular velocity.

In general, the torque generated by a DC Motor is proportional to the armature current and the magnitude of the magnetic field. In this example, it is assumed that the magnetic field is constant and, therefore, that the motor torque is proportional to only the armature current by a constant factor as shown in the equation below:

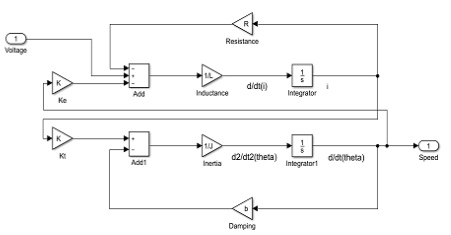
The back electromagnetic force is proportional to the angular velocity of the shaft by a constant factor

This system will be modeled by adding the torques acting on the rotor inertia and integrating the acceleration in order to obtain the speed as state.

First, the integrals of the rotational acceleration and rate of change of the armature current will be modelled:

By applying Newton’s law and Kirchoff’s law to the motor system, the dynamics of the system can be modelled:

The plant modelization in Simulink is shown in the following figure:



Once the dynamics of the plant, which will be defined as the environment of the Reinforcement Learning problem, it is necessary creating the environment MATLAB object which will interact with the Agent by generating the instantaneous rewards signals and observations in response to Agent actions.

The MATLAB environment Object requires first defining the set of Observations and Actions.

## 4.2.2 Action and Observation Signals

Usually what is done to solve a regularization problem is define the observation as where and the action space is made up by the voltage applied to the DC Motor whish is the control input of the system discretize with a step equal to the sample time: :Ts:12] where Ts = sample time = 0.01;

# 4.3 Reward Signals

The reward signal is one of the most important designing function used to guide the learning process of the agent because this signal measures the performance of the Agent with respect to the specific task. In other words, for a given observation or state, the reward measures the effectiveness of taking a particular action. During training, an agent updates its policy based on the rewards received for different state-action combinations in order to maximize the total outcome of the reward function which is the goal of the Reinforcement learning problem.

The regularization control problem can be achieved by using the negative LQR Cost Function as reward function at each time step k:

Where represents the error between the reference signal and the measured one while Q and R are weighted matrices which values depends on the trade-off between speed convergence and control input effort.

## Create Deep Q-Network Agent

The Deep Q-Learning Network (DQN) Algorithm is a Model-Free (knowledge about environment dynamics are unknown), online and off-policy (agent can learn statistically from the observed behaviors including its own past behavior or random exploratory one) Reinforcement Learning method. A DQN Agent is a value-based Reinforcement Learning which trains a critic neural network to estimate the return or future reward

As mentioned in the previous paragraph, DQN Agents can be trained in environment with the following observation and action space:

|  |  |
| --- | --- |
| **Observation Space** | **Action Space** |
| Continuous or discrete – rlNumericSpec for continuous or rlFinitespec for discrete | Discrete - rlFiniteSpec |

DQN Agents use the Critic neural network to estimate the Q-Value function critic based on observation from the environment and Agent action as well. A Q-value function maps an environment state-action pair to a scalar value representing the predicted discounted cumulative long-term reward when the agent starts from the given state and executes the given action. Q-value function can be created using *rlVectorQValueFunction* used only for discrete actions and rlQValueFunction for continuous one. Since the DQN works only with discrete action space, the first one is used. The *rlQValueFunction* instatiates an object which implements a Q-value function approximator which can be used as a critic for a Reinforcement Learning agent. In particular:

* *rlQValueFunction(criticNetwork,Observations,Actions)* = creates the multi-output Q-value function critic with a discrete action space. Here, the criticNetwork is the deep neural network used as an approximator and must have only the observations as input and a single output layer having as many elements as the number of the possible discrete actions. The network input layers are automatically associated with the environment observation channels according to the dimension specifications in ‘Observations’ and ‘Actions’ input arguments, respectively.

Before implementing a Q-value function approximator, it is necessary to design the critic neural network.

## Critic Function Approximator

To estimate a value function, a DQN agent maintains two function approximators:

* Critic the critic, with parameters ϕ, takes an observation and an action as inputs and returns the corresponding of the long-term reward.
* Target Critic used to improve the stability of the optimization, the agent periodically updates the target critic parameters using the latest critic parameter values.

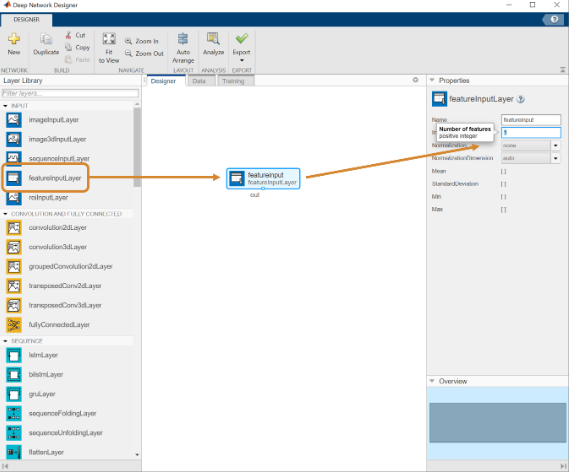
Both critics have the same structure and parameterization.

## Critic Neural Network Design

A Neural Network is represented in MATLAB as ana array, where each element is a variable that represents a layer of the network. Several functions can be used to create different kinds of layers.

There are two different ways to build a neural network:

* Using Network Designer provided by the Toolbox and then make the code exportation to include in the script.

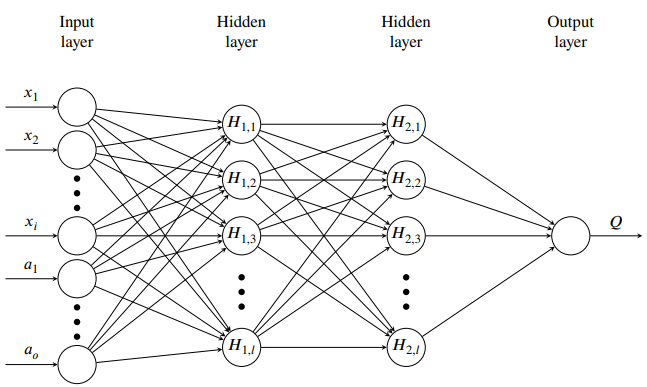
****

* By coding using Matlab functions

In this paper, the second one will be used.

The structure of the Critic Neural Network with the Reward Function will define the performance of the agent in terms of training.

The following figure shows the neural network structures commonly used to train the agent using DQN:



Where:

*i* represents the size of the observation space, *l* number of neurons per layer, *o* size of the action space. The number of hidden layers and neurons per layers are both hypermeters to be tuned based on the complexity of the system and dataset.

The design of the fully connected Critic Neural Network object in MATLAB first requires instantiating the array of layers which will contain the feature input layer as first layer, made by a number of neurons equal to the size of the observation set followed by the hidden layers and lastly, the output layer with a number of neurons equal to the size of the action set.

Once the array containing all the layers of the Critic Network has been created, MATLAB requires to transform it into a *layerGraph* object which let it using predefined properties such as *addLayers* and *connectedLayers*. The last step requires to transform the *layerGraph* object containing all the layers into the initialized *dlnetwork* object which is the one accepted by the train function.

Finally, the Q-Value function representation can be created using the specified deep neural network being able to set the following options in the rlOptimizerOptions object:

* ‘LearnRate’ = Learning rate used in training the function approximator (default value = 0.01)
* ‘GradientThreshold’ = Gradient threshold value for the training of the function approximator (default value = inf)
* ‘L2RegularizationFactor’ = Factor for L2 regularization used in training the function approximator (0.0001 by default)
* ‘Algorithm’ = Algorithm used for training the actor approximator:
  + “ adam “
  + “ sgdm “
  + Rmsprop
* ‘TargetUpdateFrequency’ = Number of steps between target critic updates (1 by default)
* ‘TargetSmoothFactor’ = Smoothing factor (1e-3 by default)
* ‘ResetExperienceBufferBeforeTraining’ = Options for clearing the experience buffer (logical value)
* ‘MiniBatchSize’ = Size of random experience mini-batch (64 by default)
* ‘NumStepsLookAhead’ = Number of future rewards used to estimate the value of the policy (1 by default)
* ‘ExperienceBufferLenght’ = Experience buffer size (10000 by default)
* ‘SampleTime’ = Sample time used in SIMULINK (1 by default)
* ‘Discount Factor’ (0.9 by default)

Before creating the agent, first specify the DQN agent options using rlDQNAgentOptions object:

* ‘UseDoubleDQN’ = Flag for using double DQN for value function target updates, specified as a logical value (0 by default)
* EpsilonGreedyEploration = Options for epsilon-greedy exploration, specified as an with the following properties:
  + Epsilon = Probability threshold to either randomly select an action or select the action that maximizes the state-action value function. A larger value of Epsilon means that the agent randomly explores the action space at a higher rate (1 by default)
  + EpsilonMin = Minimum value of Epsilon (0.01 by defauòt)
  + EpsilonDecay = Decay rate

At the end of each training time step, if Epsilon is greater than EpsilonMin, then it is updated using the following formula according to an exponential behavior:

The following script shows how the observation and action set as environment interface with the Simulink model, the Critic Network and DQN agent was created on MATLAB:

-----------------------------------------------------------------------------------------------------------------------

%% Save path of the model and agent as a string.

mdl = 'Motor\_Model\_leadlag'

agent = [mdl '/RL Agent'];

%% Observation and Action state

act = (0:Ts:12); % Discretized the Action Space Vector between u\_max and u\_min.

obsInfo = rlNumericSpec([2 1]); % Defining the continuous Observation Vector (e\_k,y\_meas)

actInfo = rlFiniteSetSpec(act); % Defining the discretize Action State Vector

%% Create the Environment interface

env = rlSimulinkEnv(mdl,agent,obsInfo,actInfo);

%% Create DQN agent

%% Designing Neural Network

nO = nO = numel(actInfo.Elements); % Saving the dimension of the discretized action space

dnn = [

featureInputLayer(2,"Name","InputLayer\_observations","Normalization","none")

fullyConnectedLayer(32,"Name","fullyConnectedNetwork\_1")

leakyReluLayer(0.01,"Name","leakyrelu")

fullyConnectedLayer(32,"Name","fullyConnectedLayer\_2")

leakyReluLayer(0.01,"Name","leakyrelu")

fullyConnectedLayer(32,"Name","fullyConnectedLayer\_3")

leakyReluLayer(0.01,"Name","leakyrelu")

fullyConnectedLayer(nO,"Name","Discretization Values of the Action Space (Voltage)")];

%% Creating Layer Graph Object

dnn = dlnetwork(dnn);

%% View Network Configuration

figure;

plot(layerGraph(dnn));

%% Critic Specifications

%Specify options for the critic optimizer using rlOptimizerOptions for the Agent.

criticOptions = rlOptimizerOptions('LearnRate',1e-4,'GradientThreshold',1,'L2RegularizationFactor',1e-4);

%{

Create the critic representation using the specified deep neural network and options.

We must also specify the action and observation info for the critic, which you obtain from the environment interface.

%}

ritic = rlVectorQValueFunction(dnn,obsInfo,actInfo);

% To create the DQN agent, first specify the DQN agent options using rlDQNAgentOptions.

agentOptions = rlDQNAgentOptions(...

'DiscountFactor',0.99,...

'SampleTime',Ts,...

'TargetSmoothFactor',1e-4,...

'CriticOptimizerOptions',criticOptions,...

'ExperienceBufferLength',2e6,...

'MiniBatchSize',128);

agentOptions.EpsilonGreedyExploration.Epsilon = 0.9;

agentOptions.EpsilonGreedyExploration.EpsilonDecay = 1e-5;

agentOptions.EpsilonGreedyExploration.EpsilonMin = .02;

agent = rlDQNAgent(critic,agentOptions);

criticNet = getModel(getCritic(agent));

plot(layerGraph(criticNet))

Notice that the Critic neural network is made up by a input layer with the dimension of the observation state and then follows three hidden fully connected layers with their respective activation LeakyRelu function in order to take into account also the negative values of the observation state.

# Train Agent

DQN Agents use the following training algorithm:

1. Inizialize the Critic neural network with a randon parameter values ϕ and initialize the target critic parameters with the same values:
2. For each time step defined by the Sampling Time:
   1. For the current observation select a random action with probability ϵ. Otherwise, select the action for which the critic Q-Value function is greatest over all the action space A:
   2. Execute action and observe reward and next observation
   3. Store the experience
   4. Sample a random M mini-batch size experiences from the experience buffer. To specify the mini-batch size M, use the MiniBatchSize option.
   5. If is a terminal state, set the value function target to otherwise set to:
   6. Update the critic parameters by one-step minimization of the loss L across all sampled experiences:
   7. Update the target critic parameters depending on the target update method (see below)
   8. Update he probability threshold ϵ for selecting a random action based on the decay rate specified in the EpsilonGreedyExploration option.

DQN Agents update their target critic parameters using one of the following target update methods:

1. Smoothing = Update the target parameters at every time step using smoothing factor τ. To specify the smoothing factor, use the TargetSmoothFactor option:
2. Periodic = Update the target parameters periodically without smoothing based on the update period specified in TargetUpdateFrequency parameter.
3. Periodic Smoothing = Update the target parameters periodically with smoothing.

To train the agent, first specify the training options using rlTrainingOptions:

- MaxEpisodes = Maximum number of peisodes to train the agent

- MaxStepsPerEpisode = Maximum number of steps to run per episode, specified as a positive integer. In general, you define episode termination conditions in the environment. This value is the maximum number of steps to run in the episode if other termination conditions are not met.

- ScoreAveragingWindowLength = Window lenght for averaging

- StopTrainingCriteria = Training termination condition

o "AverageSteps" — Stop training when the running average number of steps per episode equals or exceeds the critical value specified by the option StopTrainingValue. The average is computed using the window 'ScoreAveragingWindowLength'.

o "AverageReward" — Stop training when the running average reward equals or exceeds the critical value.

o "EpisodeReward" — Stop training when the reward in the current episode equals or exceeds the critical value.

o "GlobalStepCount" — Stop training when the total number of steps in all episodes (the total number of times the agent is invoked) equals or exceeds the critical value.

o "EpisodeCount" — Stop training when the number of training episodes equals or exceeds the critical value.

- StopTrainingValue = critical value of training termination

- Plots = plot of average reward, episode reward and expected reward during training phase.

Finally, the agent will be trained using train function.

The following script shows shows how to train the DQN Agent:

%% Train the Agent

maxepisodes = 500;

opt = rlTrainingOptions(...

'MaxEpisodes',maxepisodes,...

'MaxStepsPerEpisode',1000,...

'Verbose',false,...

'Plots','training-progress',...

'StopTrainingCriteria','EpisodeReward',...

'StopTrainingValue',0,...

'SaveAgentCriteria','EpisodeReward',...

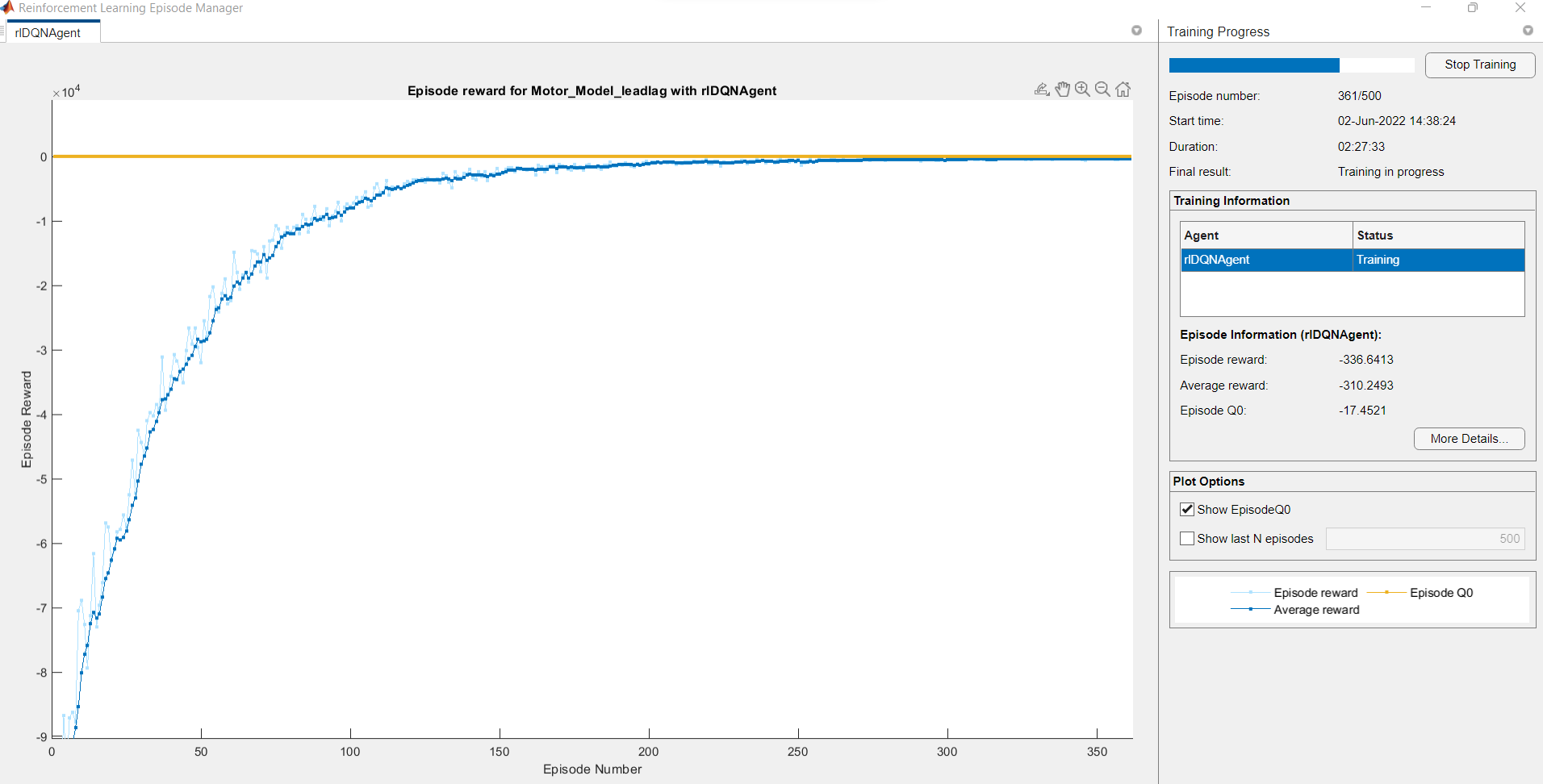
'SaveAgentValue',-2.5);

trainResults = train(agent,env,opt);

Notice that the max steps per episode was setted to 1000. In other words, if this quantity is multiplied by the sample time we obtain the duration of a single episode which is 100s.

## Episode Manager

By default, calling the train function opens the Reinforcement Learning Episode Manager, which lets you visualize the training progress:



The Episode Manager plot shows the reward for each episode (*EpisodeReward*) and a running average reward value (*AverageReward*).

For agents with a critic, Episode Q0 is the estimate of the discounted long-term reward at the start of each episode, given the initial observation of the environment. As training progresses, if the critic is well designed and learns successfully, Episode Q0 approaches in average the true discounted long-term reward, which may be offset from the *EpisodeReward* value because of discounting.

## Strategies for Improving Training

- Observe the agent's behavior during training, such as with scopes or other visualization blocks in the Simulink model. With this, you should be able to see evolutions in the policy. Is the agent getting stuck in a bad policy? Is it learning to exploit the reward in unintended ways?

- Training takes time. Agents can go through periods of better and worse performance as they try different policies. Even if your agent is not yet performing well, unless it's clearly no longer learning anything useful, let it keep training. If you reach the maximum number of training episodes while the agent is still making progress, increase the number of episodes.

1. Exploration is critically important. If an agent doesn't explore enough, it will settle on a poor policy. If your agent seems to have stopped learning, try experimenting with the exploration options to promote better exploration.

- Ultimately, your agent's learning is dictated by the reward function. Check that your agent isn't learning to exploit a “loophole” in your reward, such as the robot driving into the shelves to terminate the episode early to avoid negative rewards. Try shaping your reward function to guide your agent towards desirable states. Relying only on sparse rewards (such as a bonus when a task is successfully achieved) can make training difficult because the agent may never achieve the reward through random exploration.

- If your learning rate is too low, training may take a long time. However, a learning rate that is too high may cause unstable learning. Try to use as large a learning rate as you can, but if your agent's policy seems to be changing randomly without any improvement to the average reward, your learning rate may be too high.

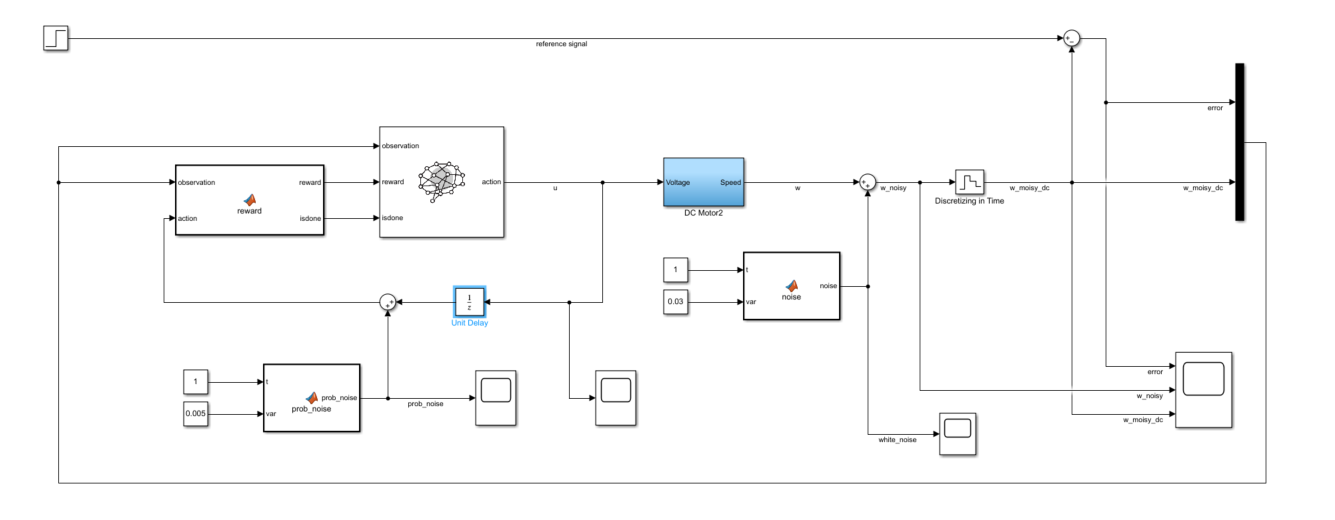
1. With enough neurons, a network can represent an extremely complicated function. But more neurons means more parameters, which requires more training. Start with a simple network – use the default network or copy the architecture from a similar example. But if your agent doesn't seem to be able to learn, no matter what else you try, you could try increasing the number of hidden-layer neurons in your networks.
2. Normalization of the action and observation state in the same scale range helps the neural network to improve its performance and time from a training point of view.

In this example a min-max normalization was implemented for both observation and action state in order to improve the speed and the performance of the training

where:

1. is the original data with no normalization.
2. is the normalized data.
3. are respectively the maximum and minimum values of the quantity to be normalized
4. are respectively the upper and lower values of the new range for the normalized data: .

The overall Simulink model containing the agent, environment, normalization of observation and action state is shown in the following figure:



Notice that the action at each time step from the controller is passed back to the reward function matlab function (via a unit delay to align them with the new state observations they cause). This means they can be included in the reward calculation. In addiction, a white noise was designed and added to the agent action in order to simulate the tolerance of the actuator and to improve the exploration of the Agent itself.

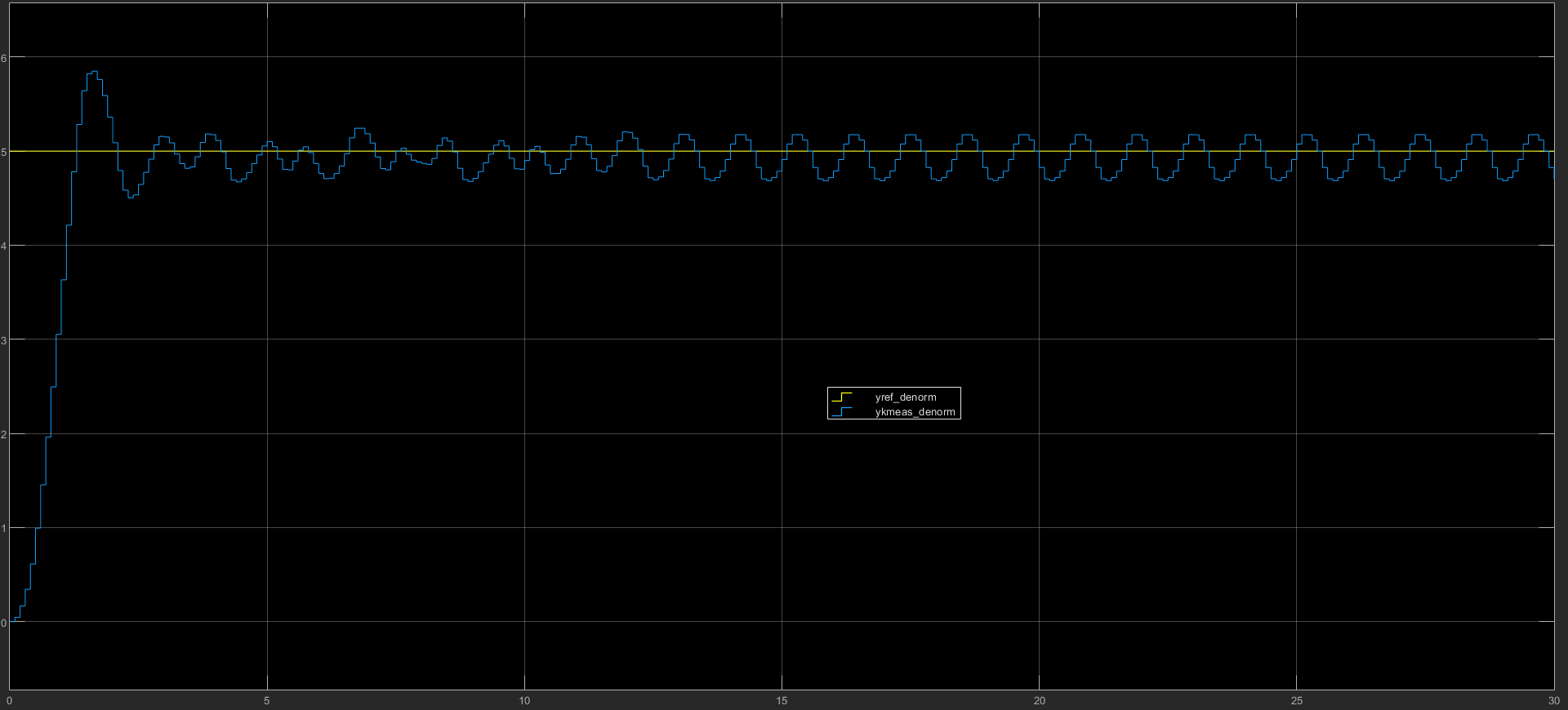
## **Validate Trained Policy**

To validate your trained agent, you can simulate the agent within the training environment using the sim function. To configure the simulation, use rlSimulationOptions.

When validating your agent, consider checking how your agent handles the following:

* Changes to simulation initial conditions — To change the model initial conditions, modify the reset function for the environment
* Mismatches between the training and simulation environment dynamics — To check such mismatches, create test environments in the same way that you created the training environment, modifying the environment behavior.

## Results



The plot shows how the trained agent was able to track the trajectory without knowing the dynamics of the DC Motor but with and oscillating behaviour.

# Example: Deep Deterministic Gradient Policy on a Linear System

In this section, the implementation of the Deep Deterministic Gradient Policy (DDPG) algorithm on a Motor Speed Control system characterized by the same physical setup and dynamics (so the same state space representation) as the previous example with the only difference that the discretization of the systems is not necessary since DPPG works in continuous observation and action space.

This choice was made in order to make a comparison in terms of performances between DDPG algorithm characterized by a continuous action space and DQN algorithm.

## Control Problem

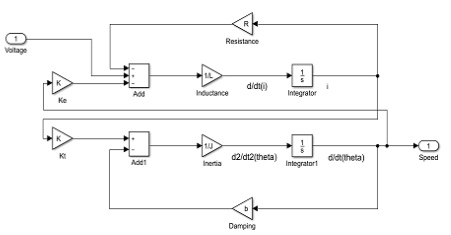
Since in the previous example the trained Agent was able to track with good performances the step reference speed only with the amplitude of the trained value, here the regularization control problem is set with respect to a step function which changes its amplitude in time of the form:



The design of the step function reference signal which different amplitudes was carried out in order to allow the neural network, and therefore the agent to become aware of all the frequencies present in its operating range and to verify after training whether the agent was able to track any step function or other shapes such as sin and ramp with different amplitudes until saturation.

## Create Environment

Since in a RL scenario, the environment model the dynamics with the agent interacts, the same model environment will be used as the previous case:



# 5.2.2 Action and Observation Signals

Differently from the previous implementation, in this case the continuous observation set is augmented in such a way as to make it possible for the agent to be aware of all the frequencies present in the reference signal: . The action set on the other hand, is defined as continuous since the DDPG can be trained in continuous set, is defined by the action range of the voltage applied to the DC Motor: which will be normalized in the same scale as the observation set.

## Reward Signal

The LQR Cost Function was used as reward function as in the previous one with an extra term in order to decrease more the oscillatory behavior by penalizing it:

+ D

Where:

;

D is a flag in order to incentivize low steady state error values of the following form:

if (e<=0.02)&&(e>-0.02)

D = 100;

else

D = 0;

end

## Create a Deep Deterministic Gradient Agent

The Deep Deterministic Policy Gradient Algorithm is a Model-Free, online and off-policy RL method as DQN one. A DPPG Agent is an actor-critic Reinforcement Learning Agent which searches for an optimal policy which maximizes the expected cumulative long-term reward. DPPG can be trained in environments with the following observation and action spaces:

|  |  |
| --- | --- |
| **Observation Space** | **Action Space** |
| Continuous or discrete – rlNumericSpec for continuous or rlFiniteSpec for discrete | Continuous or discrete – rlNumericSpec for continuous or rlFiniteSpec for discrete |

In this implementation the Agent is trained using continuous spaces both for observations and action.

DDPG Agents use the following actor and critic:

|  |  |
| --- | --- |
| Critic | Actor |
| Q-value function critic , which can be created using rlQValueFunction as in the previous implementation (see link) | Deterministic policy actor , which can be created using rlContinuousDeterministicActor |

The rlContinuousDeterministicActor object implements a function approximator to be used as deterministic actor within a reinforcement learning agent with a continuous action space.

In particular:

* rlContinuousDeterministicActor(actorNetwork,Observations,Actions) = creates a continuous deterministic actor object using the deep neural network net as underlying approximator. The network must have a single output layer with the same data type and dimensions as the number of actions. The function takes as entries also the continuous observations and action space.

## .1 Actor and Critic approximators

To estimate the policy and value function, a DDPG agent maintains four function approximators:

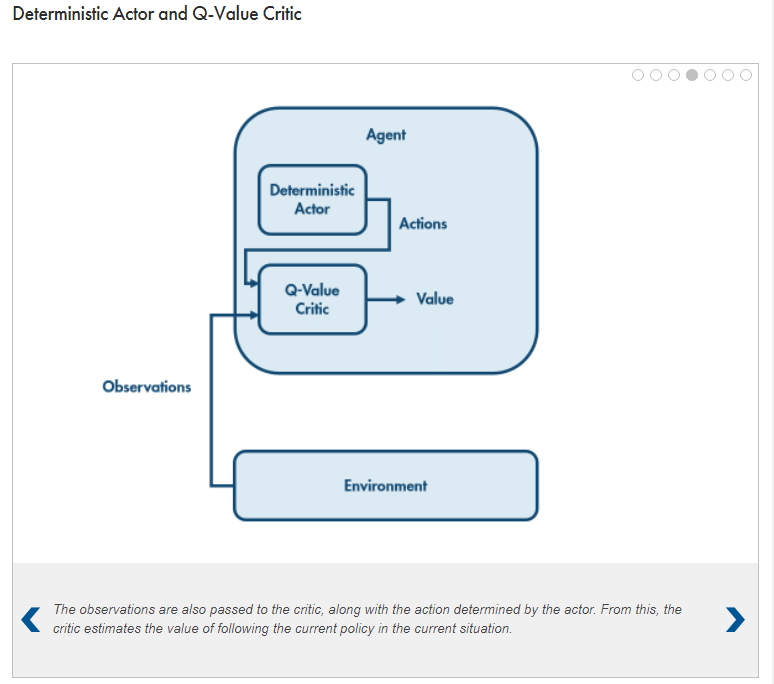
* Actor – The actor, with parameters takes the observation and returns the corresponding action that maximizes the long-term reward.
* Target actor – To improve the stability of the optimization, the agent periodically updates the target actor parameters using the latest actor parameter values
* Critic – The critic with parameters , takes observation at time step t and action and returns the corresponding expectation of the long-term reward.
* Target critic – To improve stability of the optimization, the agent periodically updates the target critic parameters using the latest critic parameter values.

Both the critic and the target critic have the same structure and parameterization, and both actor and target actor have the same structure and parameterization.

# Critic and Actor Neural Networks Design

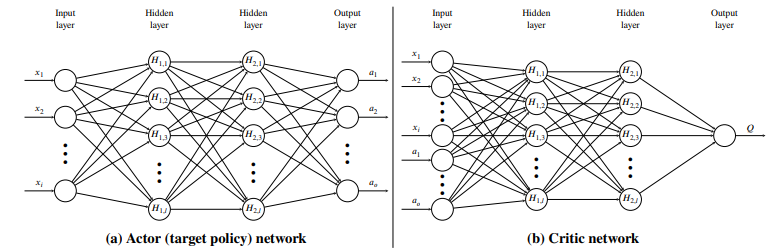
The actor uses the observed state to determine a new action. This action and the observed state are passed to the critic, which determines the value. That is, the critic estimates how much reward the agent will receive in the future from this situation. Combining this with the observed reward gives an updated estimate of the value of being in the original state. The original estimate and the updated estimate are used by the training algorithm to update both the actor and the critic. The critic is updated to produce estimates closer to that achieved with the actual rewards. The actor is updated to produce actions that will lead to states with higher values.

The relationship between the actor and critic is represented in the following figure:



The structure of both neural networks with the reward function will define the performance of the agent training.

The following figure shows the neural networks structures commonly used to train the DDPG Agent:



Where:

i represents the size of the observation space, l number of neurons per layer, o size of the action space. Here, the critic network will use the output action of the actor with the observated state to estinate the Q-Value function. The number of hidden layers and neurons per layers are both hypermeters to be tuned based on the complexity of the system and dataset.

The design of the fully connected Critic Neural Network object in MATLAB follows the same procedure as the previous implementation with the only difference that the entry action will be picked from the actor output using *addLayers* function in order to add the number of layers necessary for the action in the critic network which will be filled up by the output of the actor using the function *additionLayers*.

On the other side, the design of the fully connecter Actor Neural Network object is done using the same steps as the critic one with the only difference that for continuous actions, the final layer of a deterministic actor network should have one neuron for each action.

Critic and Actor options could be set up using rlOptimizerOptions with the same options as shown in the previous implementation.

The critic representation among the specified neural network and options can be created by rlQValueFunction.

In the same way, the deterministic actor representation among the specified neural network and options can be created using rlContinuousDeterministicActor.

The critic representation among the specified neural network and options can be created by rlQValueFunction as the previous example.

In the same way, the deterministic actor representation among the specified neural network and options can be created using rlContinuousDeterministicActor.

# 4.4.3 Actor exploration law

A major challenge of DDPG application is the generation of continuous action in exploration phase. The exploration of DDPG algorithm is independent from learning. In this example, the exploration action is constructed by adding a random noise from an Ornstein–Uhlenbeck (OU) process to the exploration policy

The noise is sampled randomly from OU process and since is a random exploration law, the training is easily to fail if is too small or too large. As a result, in order to reduce the failure rate of training, the variance of the noise should be between 1% and 10% of the maximum control input action, in this case 5 V. At each sample time step t, the standard deviation of the noise dacays in time.

The variance of the noise and the decay rate of the standard deviation could be set up in rlDDPGAgentOptions object among all the other properties discussed in the previous example.

The DDPG agent is created using rlDDPGAgent object which takes as input the critic and actor neural networks with the options defined previously.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

%% Environment Interface

obsInfo = rlNumericSpec([4 1]); % Defining the normalized continuous Observation Vector of the Augmented state [y\_meas, y\_ref, y\_dot\_meas, y\_dot\_ref]

actInfo = rlNumericSpec([1 1],'LowerLimit',-1,'UpperLimit',1);% Defining the contimuois Action State Vector which will be denormilized in the simulink schema;

%% Create Environment

env = rlSimulinkEnv(mdl,agent,obsInfo,actInfo);

%% Designing Critic

obsPath = [featureInputLayer(4,'Normalization','none','Name',"Obs")

fullyConnectedLayer(16,"Name","fc1")

leakyReluLayer(0.5,"Name","leakyrelu1")

fullyConnectedLayer(16,"Name","fc2")

additionLayer(2,"Name","add")

leakyReluLayer(0.5,"Name","leakyrelu2")

fullyConnectedLayer(16,"Name","fc3")

leakyReluLayer(0.5,"Name","leakyrelu3")

fullyConnectedLayer(1,"Name","fc4")];

actPath = [featureInputLayer(1,"Normalization","none","Name","act")

fullyConnectedLayer(16,"Name","fc5")];

criticNetwork = layerGraph(obsPath);

criticNetwork = addLayers(criticNetwork,actPath);

criticNetwork = connectLayers(criticNetwork,'fc5','add/in2');

criticNetwork = dlnetwork(criticNetwork);

criticOptions = rlOptimizerOptions('LearnRate',1e-4,'GradientThreshold',1,'L2RegularizationFactor',1e-4);

critic = rlQValueFunction(criticNetwork,obsInfo,actInfo);

%% Designing Actor

actnet = [featureInputLayer(4,"Normalization",'none',"Name","Observations")

fullyConnectedLayer(32,"Name","fca1")

leakyReluLayer(0.5,'Name','leakya1')

fullyConnectedLayer(32,"Name","fca2")

leakyReluLayer(0.5,'Name','leakya2')

fullyConnectedLayer(32,"Name","fca3")

leakyReluLayer(0.5,'Name','leakya1')

fullyConnectedLayer(1,"Name","fca5")

tanhLayer("Name","tanh1")];

actorNetwork = dlnetwork(actnet);

actorOptions = rlOptimizerOptions('LearnRate',1e-5,'GradientThreshold',1,'L2RegularizationFactor',1e-4);

actor = rlContinuousDeterministicActor(actnet,obsInfo,actInfo);

%% Defining the DDPG Agent

agentOptions = rlDDPGAgentOptions(...

'DiscountFactor',0.9,...

"TargetSmoothFactor",1e-3,...

'SampleTime',Ts,...

'ActorOptimizerOptions',actorOptions,...

'CriticOptimizerOptions',criticOptions,...

'MiniBatchSize',128,...

'ExperienceBufferLength',1e6);

agentOptions.NoiseOptions.Variance = 1\*0.5/sqrt(Ts);

agentOptions.NoiseOptions.VarianceDecayRate = 1e-6;

agent = rlDDPGAgent(actor,critic,agentOptions);

---------------------------------------------------------------------------------------------------------------------------------------------------------------

It can be noticed that the normalization is applied to both observation and action stet and the step function which varies in time its amplitude was built with the signal builder with a duration of 60 sec.

# 5.3 Train Agent

DDPG Agents use the following training algorithm, in which they update their actor and critic at each time step t:

1. Initialize the critic with random parameter values ϕ and initialize the target critic parameters with the same values .
2. Initialize the actor with a random parameter value *θ* and initialize the target actor parameters .
3. For each training time step t:
   1. For the current observation select action where *N* is stochastic noise from the noise model (see link)
   2. Execute action . Observe the reward and next observation .
   3. Store the experience in the experience buffer. The
   4. Sample a random mini-batch of *M* experiences from the experience buffer.
   5. If is a terminal state, set the value function target to . Otherwise, set it to:

The value function target is the sum of experience reward and the discounted future reward.

To compute the cumulative reward, the agent first computes a next action by passing the next observation from the sampled experience to the target actor. The agent finds the cumulative reward by passing the next action to the target critic.

* 1. Update the critic parameters by minimizing the loss *L* across all sampled experiences:
  2. Update the actor parameters using the following sampled policy gradient to maximize the expected discounted reward.

where

Here, is the gradient of the critic output with respect to the action computed by the actor network, and is the gradient of the actor output with respect to the actor parameters. Both gradients are evaluated for observation .

* 1. Update the target actor and critic parameter (both ) with the same methods shown in the previous chapter

After setting the training options with rlTrainingOptions objects the agent could be train using train function:

maxepisodes = 1000;

maxsteps = ceil(500/Ts);

opt = rlTrainingOptions(...

'MaxEpisodes',maxepisodes,...

'MaxStepsPerEpisode',600,...

'Verbose',false,...

'Plots','training-progress',...

'StopTrainingCriteria','EpisodeReward',...

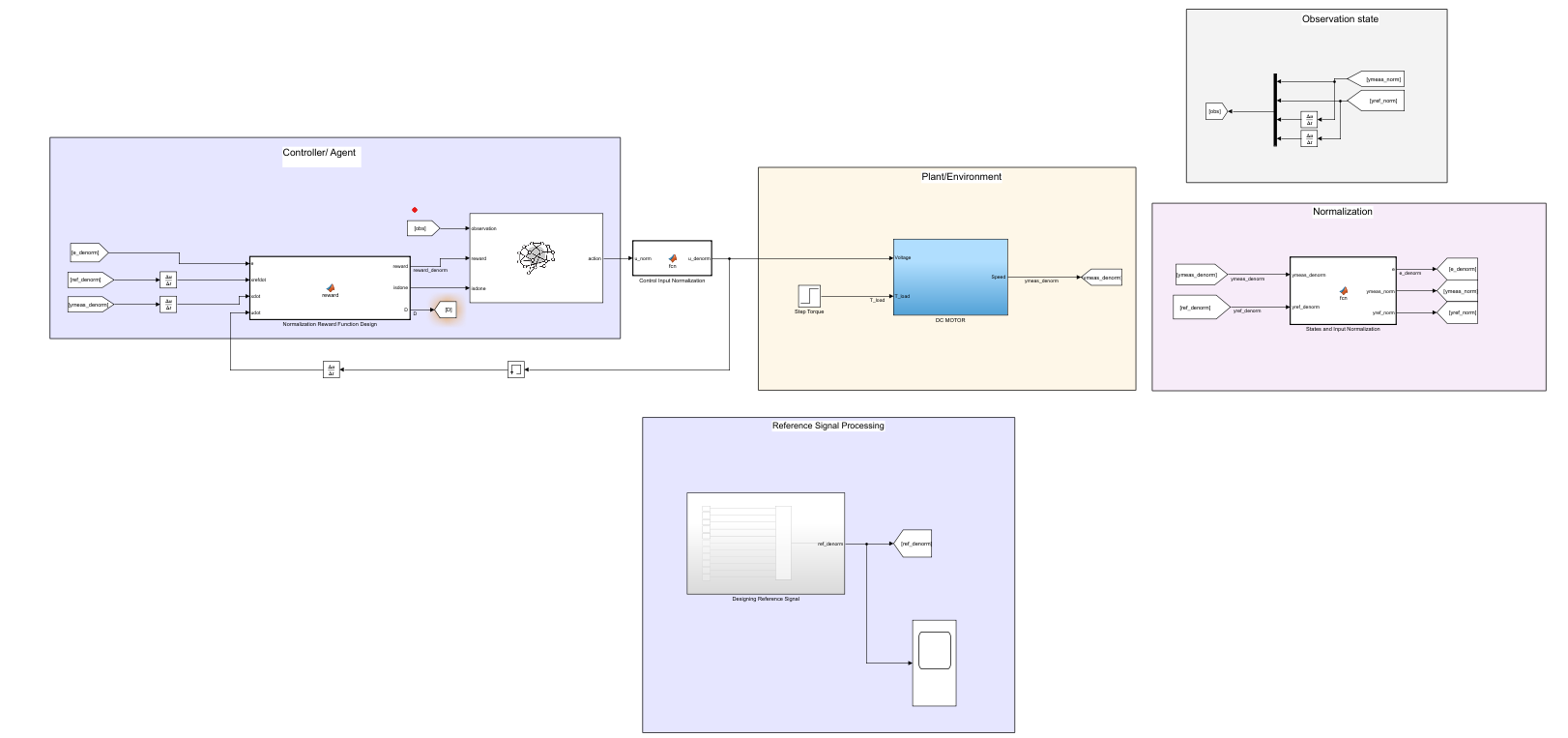
'StopTrainingValue',-1,...

'SaveAgentCriteria','EpisodeReward',...

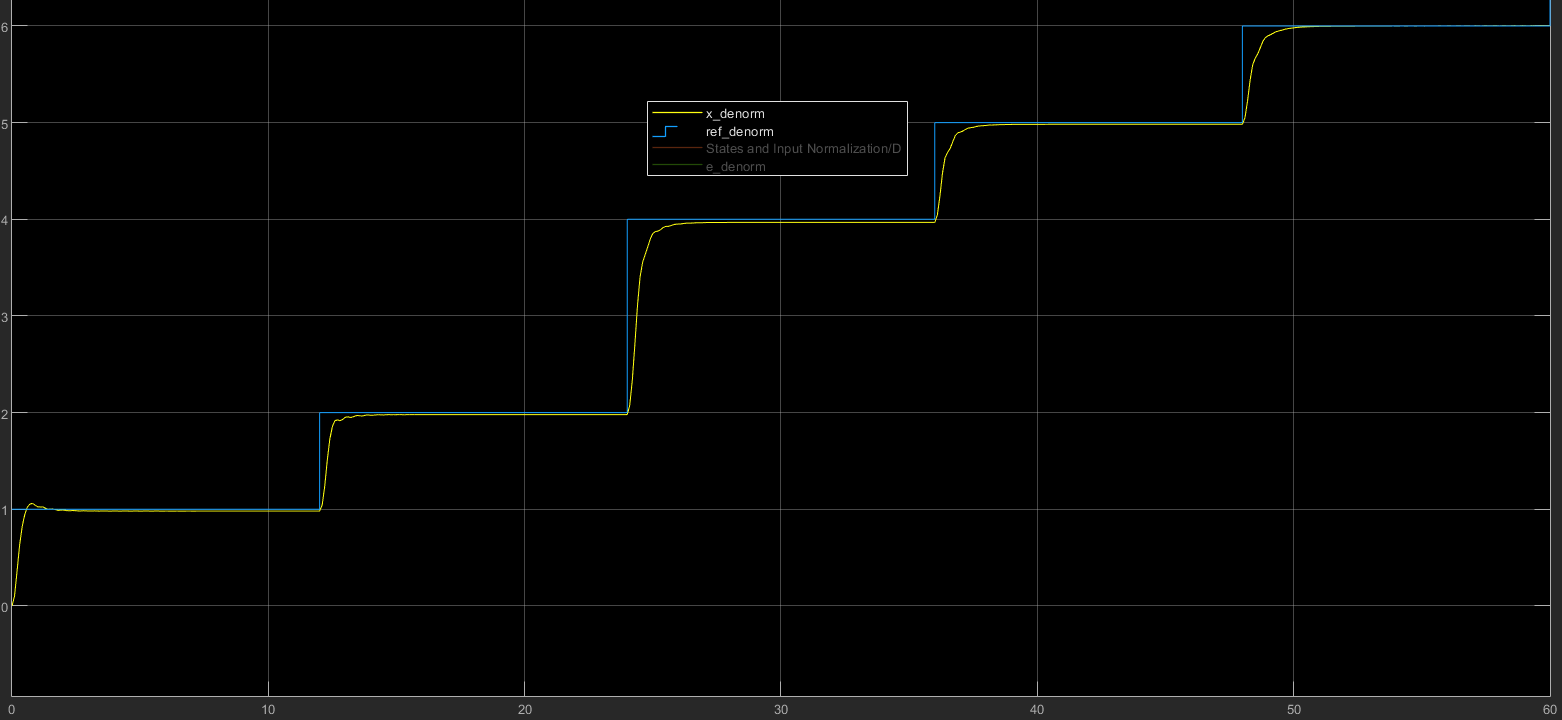
'SaveAgentValue',-2.5);

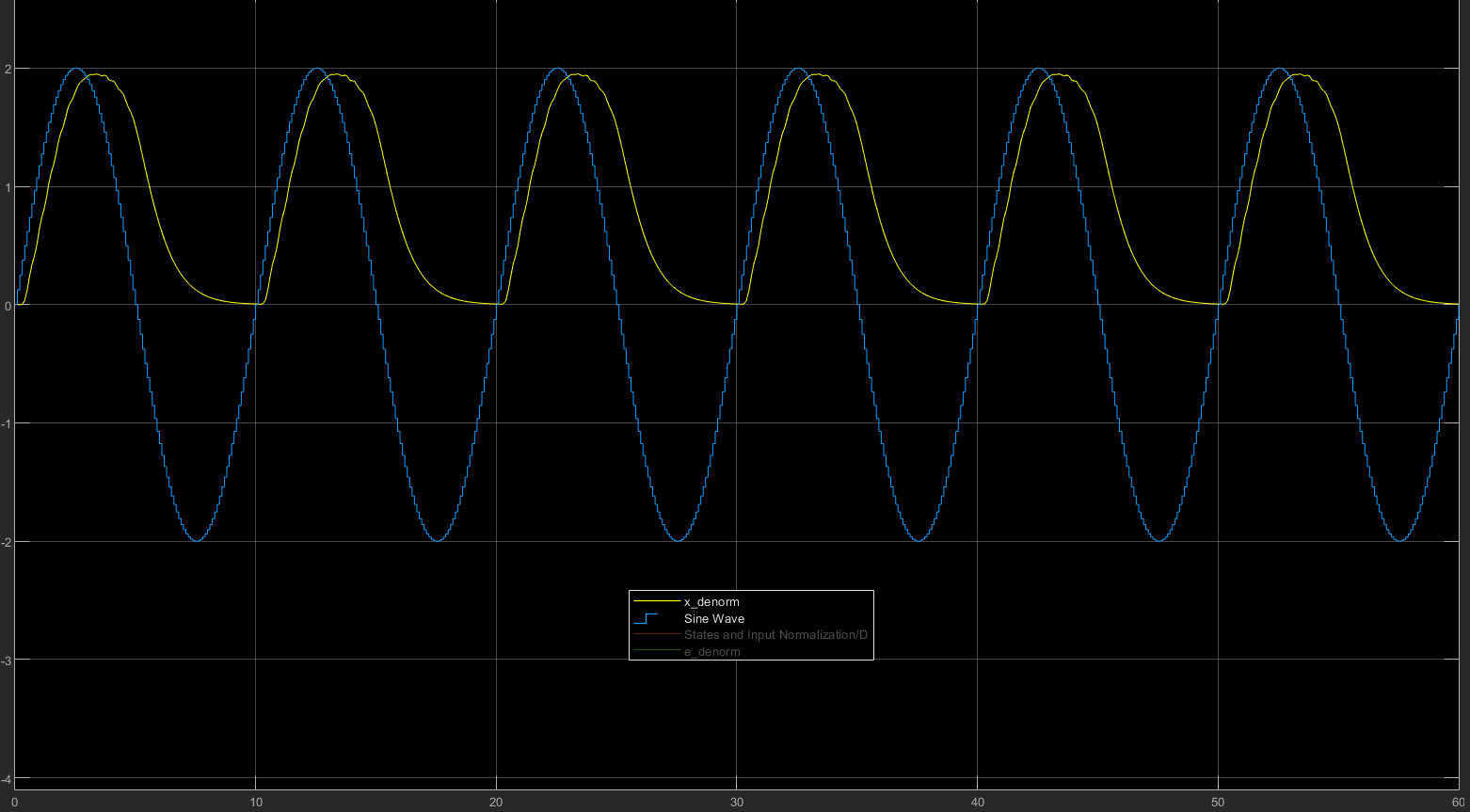
trainResults = train(agent,env,opt);

The overall Simulink model containing all the RL problem ingredients is shown in the following figure:



# 5.5 Results





The plots show how the agent was able to learn the dynamics of the system and track the the designed reference signal with a raise time of 0.3 sec, managing to rotate the motor following the reference signal with a steady state error for each phase of amplitude smaller than that assigned by the flag and with an acceptable oscillator behavior with a zero mean. The agent is also able to track a sin wave reference signal which has different dynamics as it was trained.

# Example: Deep Deterministic Gradient Policy on a Non Linear System

In this section, the implementation of the DDPG algorithm on a Non-Linear system first is analyzed. The agent first will train in a simulated environment and then the goal will be to apply the optimal policy and the system dynamics learned during training on the real hardware system in order to verify if the agent will be able to adapt to the dynamics of the real system and in non-simulated conditions such as air friction or with some physical variables which change in time such as mass.

The Non-Linear system will be the QUBE-Servo 2 Inverted Pendulum system, shown below, which has two encoders to measure the position of the rotary arm (i.e., the DC motor angle) and the pendulum link and a DC motor at the base of the rotary arm.



## Control Problem

The control of a pendulum has been one of fundamental problems in control field. As a control strategy to stabilize at the up-right position, it is well known that a linear quadratic technique is effective. The goal of the implementation of the model-free DDPG algorithm is aiming not only to make an agent learn its dynamics but also to stabilize it and also to swing up the pendulum in the vertical position.

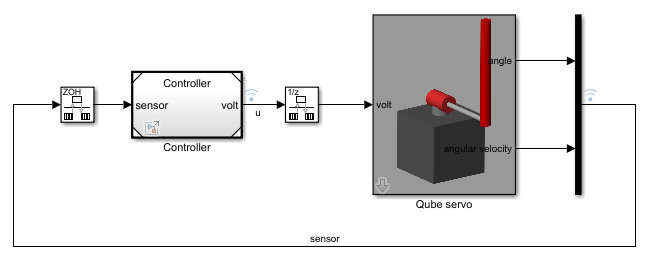
## Create the Environment

The environment used to train the agent is a nonlinear dynamic model of the rotational inverted pendulum QUBE-Servo 2 system and is defined in the Simulink QUBE-Servo 2 Pendulum Model block provided by MATLAB which contains all the real physical parameters of the hardware. The system consist of a motor arm, which is actuated by a DC servo motor, with a swinging pendulum arm attached to its end. This system is challenging to control because it is underactuated, highly non linear and non minimum phase.

In the following figure the physical parameters used are shown:

%% Plant parameters

R\_m = 21.7; % Motor resistance [Ohm]

mu\_m = 3.08e-6; % Damping of motor shaft [Nm/rad/s]

K\_t = 0.042; % Torque constant [Nm/A]

K\_b = 0.0392; % Back EMF contant [V/rad/s]

K\_b\_2 = 0.182; % Back EMF contant [V/rad/s]

J\_m = 4e-6; % Motor shaft inertia [kg\*m^2]

L\_m = 4.98e-3; % Motor Inductance [H]

rod\_rad = 0.003; % Arm rod radius [m]

p\_rad = 0.0045; % Pendulum radius [m]

L\_p = 0.126; % Length of Pendulum [m]

L\_r = 0.103; % Length of Arm rod [m]

m\_p = 0.024; % Mass of Pendulum [kg]

m\_r = 0.095; % Mass of Arm rod [kg]

J\_p = m\_p\*(p\_rad^2/4+L\_p^2/12) + m\_p\*(L\_p/2)^2; % Inertia of Pendulum [kg\*m^2]

J\_r = m\_r\*(rod\_rad^2/4+L\_r^2/12) + m\_r\*(L\_r/2)^2; % Inertia of Arm rod [kg\*m^2]

D\_r = 0.001; % Damping of Arm rod [Nm/rad/s]

D\_r\_2 = 1.88e-04; % Damping of Arm rod [Nm/rad/s]

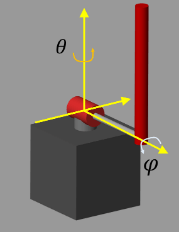
D\_p = 8e-6; % Damping og Pendulum [Nm/rad/s]

n = 1; % Gera ratio

g = 9.81; % Gravity [m/s^2]

As it can be noticed, the pendulum system is modeled in Simulink using Simscape Electrical and Simscape Multibody components. For this system:

* is the motor arm angle and is the pendulum angle.
* The motor arm angle is 0 radians when the arm is oriented horizontal and forward as shown in the diagram (counterclockwise is positive)
* The pendulum angle is 0 radians when the pendulum is oriented vertically downwards (counterclockwise is positive)
* Input of the plant model is a DC voltage signal for the motor. The voltage values range from -12 to 12 V.
* The pendulum and motor angles and angular velocities are measured by sensors



In the SIMULINK model, a change of reference system was applied in order to have angle 0 in vertical equilibrium position in order to treat the problem as regularization control problem and to allow the agent not to have to take into account the angle signs in the point vertical unstable equilibrium position.

# 6.2.2 Action and Observation Signals

For the rotary inverted pendulum there are six continuous observation signals: ]. On the other side, the continuous action set is the voltage applied to the DC Motor between -12 and 12 V. Both the observation and action state are normalized between -1 and 1 in the same way as the previous examples (link)

The choice of setting the sin and cos of the angles as observations and not setting directly the angles coming from the sensor reading was made to speed up the training phase. While it would be possible for the neural network to learn the sin and cos observations as part of its learning, in this way will be costless and can remove some overhead from the process.

## Reward Signal

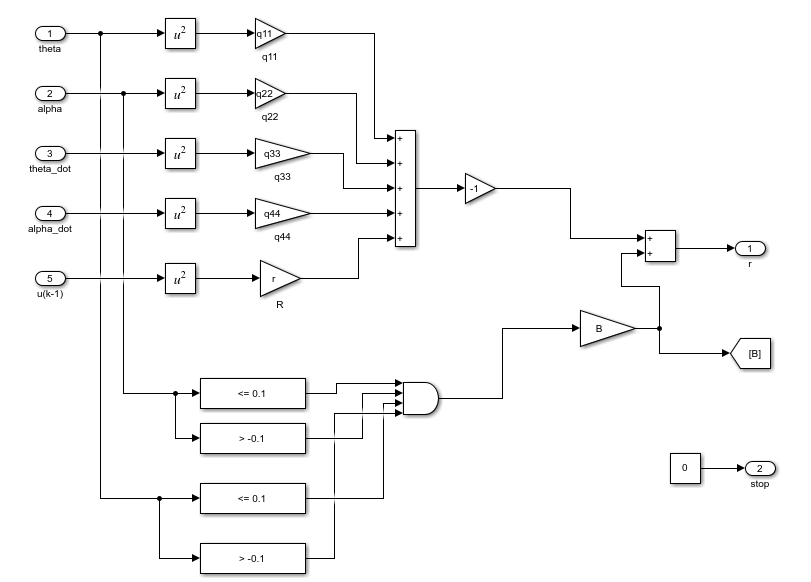
The reward signal was designed as follows:

The agent's purpose is to maximize the reward function. In other words, the weighted quadratic function rewards the agent when the rotary arm stays within the 0 radians for both the angles. The terms concerning both rotational speeds and control input effort are introduced in order to penalize high oscillations and the control motor voltage does not go too high.

It was found that quadratic-based reward signals are easier to tune and increase the likelihood of training a successful policy with this system.

A flag has also been inserted in the reward function which allows to obtain a value of +100 if the system configuration is in the vertical equilibrium position (.

In this implementation, the reward function was modelled using Simulink blocks as follows:



Notice that the isDone flag is setted to 0 for the training in the simulated environment in order to let the Agent train for the entire episode while in the real environment will be set to 1 when the agent will try to apply a voltage over the saturation band for safe conditions.

The entire scripts looks as follows:

%% Open the model

mdl = 'pend\_simscape'

agent = [mdl '/RL Agent'];

%% Sample Time

Ts = 0.01;

%% Environment Interface

obsInfo = rlNumericSpec([8 1],'LowerLimit',[1; -1; -1; -1; 1; -1; -1; -1],'UpperLimit', [1; 1; 1; 1; 1; 1; 1; 1]);

actInfo = rlNumericSpec([1 1],'LowerLimit',-1,'UpperLimit',1)

%% Create the Environment

env = rlSimulinkEnv(mdl,agent,obsInfo,actInfo);

%% Designing Critic

obsPath = [featureInputLayer(8,'Normalization','none','Name',"Obs")

fullyConnectedLayer(100,"Name","fc1")

leakyReluLayer(0.5,"Name","leakyrelu1")

fullyConnectedLayer(100,"Name","fc2")

additionLayer(2,"Name","add")

leakyReluLayer(0.5,"Name","leakyrelu2")

fullyConnectedLayer(100,"Name","fc3")

leakyReluLayer(0.5,"Name","leakyrelu3")

fullyConnectedLayer(1,"Name","fc4")];

actPath = [featureInputLayer(1,"Normalization","none","Name","act")

fullyConnectedLayer(300,"Name","fc5")];

criticNetwork = layerGraph(obsPath);

criticNetwork = addLayers(criticNetwork,actPath);

criticNetwork = connectLayers(criticNetwork,'fc5','add/in2');

criticNetwork = dlnetwork(criticNetwork);

criticOptions = rlOptimizerOptions('LearnRate',1e-4,'GradientThreshold',1,'L2RegularizationFactor',1e-4);

critic = rlQValueFunction(criticNetwork,obsInfo,actInfo);

%% Designing Deterministic Actor

%{

For continuous actions, the final layer of a deterministic actor network should have one neuron for each action.

actnet = [featureInputLayer(8,"Normalization",'none',"Name","Observations")

fullyConnectedLayer(200,"Name","fca1")

reluLayer("Name","relua1")

fullyConnectedLayer(200,"Name","fca2")

reluLayer("Name","relua2")

fullyConnectedLayer(200,"Name","fca3")

reluLayer("Name","relua3")

fullyConnectedLayer(200,"Name","fca4")

reluLayer("Name","relua4")

fullyConnectedLayer(1,"Name","fca5")

tanhLayer('Name','tanh1')];

actorNetwork = dlnetwork(actnet);

actorOptions = rlOptimizerOptions('LearnRate',1e-4,'GradientThreshold',1,'L2RegularizationFactor',1e-4);

actor = rlContinuousDeterministicActor(actnet,obsInfo,actInfo);

%% Defining the DDPG Agent

%% Agent Option

agentOptions = rlDDPGAgentOptions(...

'DiscountFactor',0.95,...

"TargetSmoothFactor",1e-3,...

'SampleTime',Ts,...

'ActorOptimizerOptions',actorOptions,...

'CriticOptimizerOptions',criticOptions,...

'MiniBatchSize',128,...

'ExperienceBufferLength',2e6);

agentOptions.NoiseOptions.Variance = 1/sqrt(Ts);

agentOptions.NoiseOptions.VarianceDecayRate = 1e-6;

agent = rlDDPGAgent(actor,critic,agentOptions);

maxepisodes = 1000;

maxsteps = ceil(500/Ts);

opt = rlTrainingOptions(...

'MaxEpisodes',maxepisodes,...

'MaxStepsPerEpisode',2000,...

'Verbose',false,...

'Plots','training-progress',...

'StopTrainingCriteria','EpisodeReward',...

'StopTrainingValue',10000,...

'SaveAgentCriteria','EpisodeReward',...

'SaveAgentValue',-2.5);

trainResults = train(agent,env,opt);

## Results



The plots shows the trained agent learnt the dynamics of the system making him stabilize in the unstable equilibrium point after 5 sec and being able to maintain this position for the entire episode.