Introduction

This is the tidied-up version of the GitHub repository used for the summer 2022 Control theory and Reinforcement learning UROP. The repository is split into the two parts described by the title. This document provides a brief overview of the work done and the files associated with it.

Control Section

The project uses the example of a pendulum attached to a sliding cart (henceforth cartpole problem) in order to trial and implement different control strategies.

Pendulum.py

Program that contains the state-space representation of the cart-pole system. The 4 coupled first-order non-linear ODEs describing the motion are stored here.

RK4solver.py

The system is solved using the RK4 numerical method with a fixed step size. The inbuilt python numerical solver is not used because its step size is not fixed which causes some issues when working with the reinforcement learning program.

Nonlinearctrl.py

Body of the control program. Takes in the initial conditions of the system, and then simulates how they evolve with time under the effect of different strategies. The one implemented at the moment is non-linear energy-based control followed by a Linear Quadratic Regulator (LQR) in a sufficiently small region. The purpose of the control strategies is to stabilise the pendulum at the upright (unstable) fixed point. The program prints out a graph of the change of the kinematic variables with time. The LQR is from an imported pre-existing control module, whereas the energy control has been implemented manually.

non_linear.py

This module implements energy-based control. The idea is to raise the pendulum into a homoclinic orbit, where it has just about enough energy to cycle between the two fixed points. This is done by taking the energy difference between the current pendulum state and the desired pendulum state and then multiplying it by the current velocity (in order to obtain the correct direction of the control force application). This then gives the desired control torque, which in turn can be expressed in terms of the actual control input, i.e the force applied on the cart. Under this force application, the unstable fixed point also becomes attractive.

PID.py

Introduction to the UROP, this was the first control strategy tried. It was found that a full PID controller is not needed, as there is no constant source of error that needs to be adjusted for. A PD controller does fine enough. Not currently used

Reinforcement Learning section

Deep Neural Network

The neural network used here has 2 hidden layers of 32 neurons each. Policy Gradient methods are used to train the neural network.

RL_Program.py

RL_Program.py is the main body of the program used for running and training the machine learning algorithm and for keeping track of the changes made to the weight arrays in each epoch. Owing to limitations in the code, the size of the neural network is hard-coded at the moment.

The user is capable of choosing the sample size for each epoch and the number of epochs. The function parameters and initial conditions for the simulation are then repeatedly passed to the SimPend function.

The SimPend function runs the cart-pole pendulum simulation for a given length of time. For every complete simulation in each epoch, an array of the force applied to the pendulum at each time step, the input and output activations and the reward function are saved. These are then used in order to run backpropagation on the weights of the NN. No biases are used due to programmer oversight.

The array of the final reward functions for each completed simulation is then normalised, and every reward function that is then above average is marked as “successful” and vice-versa. “Successful” simulations then have all of their “decisions” (forces applied) passed into the backprop algorithm, and their cost function is minimised. Decisions belonging to failed simulations then have their cost function maximised (grad(f) of the weight array is subtracted rather than added).

The program currently does not work, owing to the fact that the complicated expression in the NN output was not accounted for in the derivation of the backprop equations.

NonLinPend.py

Contains the SimPend function which is the main body of the cart-pole environment program. It accepts the simulation parameters from Master_Program.py, whereupon it uses RK4 in order to simulate the dynamical behaviour of a pendulum mounted on a sliding cart (cartpole problem). At each time step the current state space vector of the pendulum system is fed into the neural network which encodes the force to be applied to the pendulum as a normal distribution (one output node encodes x_bar and one encodes sigma). Both outputs are given as a number between 0 and 1, and that is the format in which they are returned to the function.

Both numbers are then passed through the logit function in order to convert them into a format that can be used by the simulator for control. The idea is to then record both the input activations (state space variables), the outputs and the reward function at that point. A correction to prevent exponential growth is added. The above arrays are then returned back to the original function.

better_backp.py

Backpropagation algorithm that uses tensors in order to ensure fast multiplication. The state space representation of the system at a given point is fed into the function, aswell as the actual force that was applied (selected from normal distribution) and the array of weights used by the neural network at that time. The program then finds the grad of a quadratic cost function to determine how each

individual weight can be adjusted in order to most efficiently change the final activations. This is semi-supervised learning as we do not actually know what the activations should be, we can only determine the direction in which we'd like the values to go. The backpropagation algorithm outputs a suggestion of the changes to be done to each item in the weight arrays.