ECE4424: Midway Project Progress Update

A Neural Network Based Real-Time SISO Controller

Mihir Savadi | 25th April 2021

mihirsavadi1@vt.edu

Please see the project proposal for deeper technical context, located at the link here – https://drive.google.com/file/d/1U9vPFnOtadhH-pVDtQEv7QSINVUuMfTP/view?usp=sharing.

To summarize the project, I am attempting to build an architecture and approach based on Neural Networks (NN) that will allow for a control-feedback structure to tune itself over time. See sections 1 and 2 in the link above for a quick and easy introduction into what control-feedback loops are and what PID controllers are. For a quick example: a car's cruise control needs to constantly observe vehicle speed and manage throttle and braking in real time in order to keep the car at a set speed no matter the influences from the environment. A controller would use current and historical information on the error (the cars actual speed minus its intended speed) to make decisions as to how to set the car's (also called the 'plant') throttle or brake inputs. The parameters that govern this controller however are mostly always manually human tuned before the plant is allowed to go out into the real world, in order to ensure that the plant's behavior is always perfectly damped. In order to for my NN based implementation to achieve perfect damping, it will follow a pseudo-supervised approach whereby there will be no training data to start with. Instead, it will create its own database of labelled data based on an objective function that quantifies plant damping as a function of historical data observed in the circuit: the responses of the plant; the control inputs of the controller; and the errors generated. See section 4 of mentioned in the first paragraph for a more in-depth explanation of this. At every given time interval, the NN will undergo a round of gradient descent based on the database it has built up so far in order to calculate new weights, whilst the current NN concurrently works on its old weights as new input data comes in in real time. Then, at some point shortly after the gradient descent step is completed, the weights of the NN are updated. This entire time the database will constantly be appended to, and this cycle will repeat until the NN controller achieves adequate damping. In order for this NN based controller to generalize good performance, it needs a large amount of variance in input data – this variance can be constrained based on the typical application space of the plant in question. I have created long time-series training input sequences that are randomized based on some constraints, in order to achieve a good amount of variance.

In large part the original plan as described in my proposal is still underway and has been largely unchanged. Because of the novel nature of this project, sound underlying infrastructure had to be created in order to allow an environment for easy, quick, modular, and reliable testing, so that results and optimizations can be made efficiently and with clarity – a sort of 'testbench' if you will. This took a much longer time than I expected – creating the test bench code-base, planning its architecture, and ensuring it was well tested. The default example I used was a to emulate a run-of-the-mill PID to plant circuit setup, according to figure 1 below, with a simple stepped input, and a plant based on the FOPDT model with an arbitrary set of parameters. The PID was poorly tuned just for the sake of clearly illustrating an example of poorly damped output, which would provide intuitive insight into what total system behavior would look like in general – this is shown in figure 2 below, notice the oscillations in the error 'e' curve. Note

that figure 2 was automatically generated via the aforementioned testbench, which will be explained in the next paragraph.

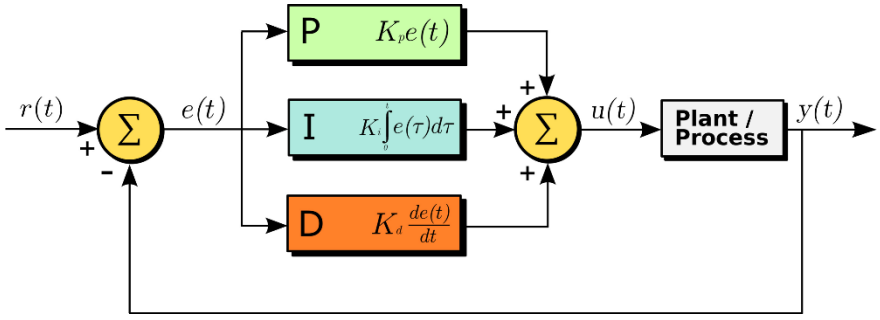

Figure 1

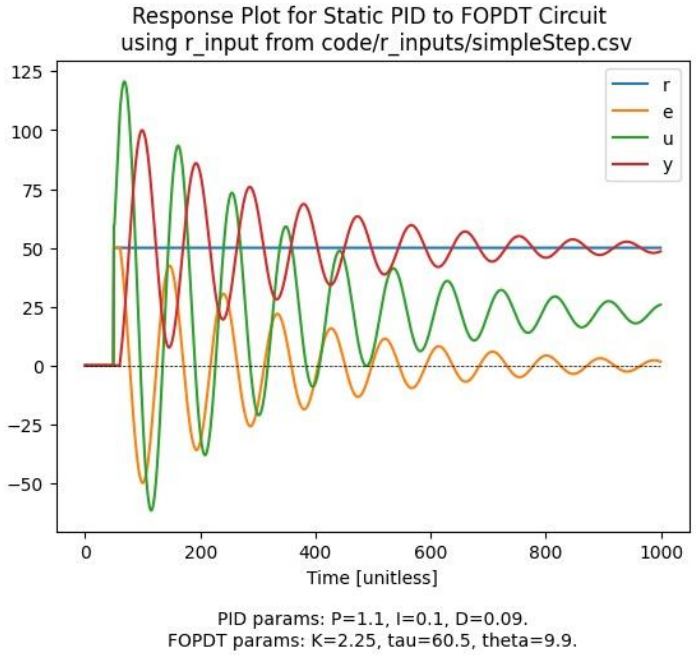

Figure 2

I plan on releasing a public GitHub repository with my entire codebase (including the testbench), as well as various forms of documentation including ECE4424 deliverables. The testbench is very modular and employs a very object orientated approach, which heavy use of both class inheritance (to enforce inter-class communication consistency without compromising modularity) and composition. The testbench is fundamentally based on the concept of actors, whereby each element any given circuit (like the one in figure 1 for example) is treated as a black box, or an actor, whose basic inputs and outputs are standardized by an abstract actor base class, and whose internal calculations are abstracted away. Actors can be controllers, plants, or any other influencing element of a circuit. Actors are then wrapped up in a circuit class, which contains standardized inputs and outputs, and abstracts away the interconnections between whatever actors may be in a particular circuit. These standardizations are again enforced by an abstract circuit base class. Circuit classes contain a single function to update actor outputs according to its defined interconnections for each time step, as well as store historical data for each part of the circuit. Finally, a schedular class is what the user interacts with in order to utilize the testbench – it contains an instance of a circuit class, which is determined on initialization of a scheduler object; it provides a

function to get inputs into the circuit, either by time-step by time-step polling or by parsing an external text file; it provides a function to probe the circuit to get information on its stored current and historical data; and it provides a function to generate a convenient plot to quickly visualize circuit behavior (an example of this is in figure 4).

The user can create arbitrary actor classes – controllers, plants, etc. – as well as circuit classes and they will function predictably with the aforementioned scheduler, as long as the respective abstract base classes are adhered to. Thus, a user can use the scheduler class of this testbench to automate testing and analysis of a variety of controllers, plants, and circuits, in an arbitrarily simulated real-time environment, which is pretty cool. While I have not completed this step yet, I do plan on wrapping the __main__.py part of the script such that the testbench can be used as a stand-alone CLI tool, employing actor and circuit classes have been established in the code base. As you can expect, polishing up all of this took quite a bit of time. The value of all of it is that now I am enabled to quite easily and reliably test any arbitrary control system I choose to implement, which will make research, design, and testing of my novel NN based controllers a lot easier.

As for the actual architecture of the controller, one of the ideas I was going to first attempt would be described as follows: a gradient descent 'box' would sit in parallel to the PID controller, that would be able to adjust the P, I, and D parameters of said PID controller. A cost function would be defined by equation 1 below – please refer to figure 1 for what each variable refers to. And a gradient vector would be established by analyzing the simple PID equation shown in equation 2 below, coupled with the cost function in equation 1. At a pre-defined number of time steps, an average cost function output would be calculated using historically collected data, and using a pre-set learning rate the PID parameters would be adjusted. The historically collected database would then be wiped and appended to from fresh, after which at then end of the another set of the pre-defined set of time steps, this cycle would repeat, and would continue until either no more input data is available or the cost function has achieved an adequately low result – i.e., the plant achieves a good low level of damping.

$$\frac{1}{z+1}\sum_{i=0}^{z}(r[i]-y[i])^2$$

where $0 \leq x \leq n$ and n is the number of time steps prior to the current time step

Equation 1

$$u[t] = K_P e[t] + K_I \sum_{i=0}^{n} e[t-i] + K_d(e[t]-e[t-1])$$

where $e[t] = r[t] - y[t]$ and $n \geq 0$ and n is the number of time steps prior to the current time step

Equation 2

Another approach I was going to try was to create a simple feed forward NN, completely replacing the PID controller box, where the input layer would be some vector $\{r[t-k], u[t-i], y[t-i]\}$ where $0 \leq k \leq n$ and $1 \leq i \leq n$, and the output layer would be a single output $u[t]$. Using equation 1 to calculate a value at every pre-determined interval of time steps, and then averaging a given number of these values for a period of time, a loss function value can be used to achieve gradient descent and a periodic update of the weights of the NN until good damping of the plant is achieved. This, along with the explanation in the previous paragraph, still needs to be thoroughly tested before submission of the final report.