ASSIGMENT REPORT

May 28, 2020

Gansheng TAN
Shanghai JiaoTong University
Department of Mechanical Engineering

Contents

Chapter 1

Introduction

This report consists of the solutions to assignment 2 of intelligence control. The notations are the same as those in course slides unless specification. I contributed to the several machine learning package in Python including mne, neuroKit and keras. For convenient purpose, the machine learning parts are developed with Python. The accompanying codes written in python are jupyter notebooks that can run with jupyter lab or jupyter notebooks. For readers who do not want to try their own parameters, the source codes are also transformed into HTML pages for readers.

Chapter 2

Solutions

2.1 Question 1

2.1.1 Induction

In my understanding, this training data-set for desired bp-neural networks (bp-NNs). The pimpliest way to adjusting the weights is to train an initialized neural network with the help of keras [3]. If time permitted, the system is approximated using a personalized bp-NNs to familiarize the back-propagation algorithms and improve its performance in a low level. In view of the miss knowledge of the initial value of y (i.e. y[0]), it is set to 0 for subsequent analysis.

2.1.2 Solutions

The command signal u and system output y are sketched in Fig 2.1. The first trial is to use a simplest neural network to approximate the system (i.e. the perceptron). The three layers are fully connected. The cost function is mean square error while the optimization method is set to 'adam"s method which is an adaptive iterative update of weights. The network output after 50 epochs (i.e. 50 iterations) is plot in Fig 2.3 where the network does not recognize the non-linear nature of the system. The designed network was further refined by adding a second hidden layer and increasing numbers of epochs. As shown in Fig 2.4, the refined network can depict the system dynamics.The weights are initialized using He's method [4].

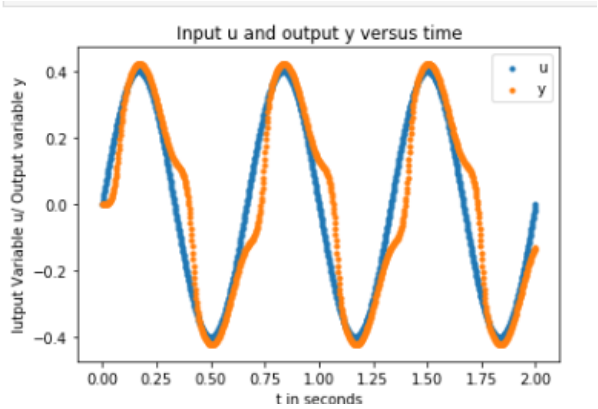

Figure 2.1: Visualization of the input signal 'u' and system output 'y'

```
import keras
model = keras.Sequential()
# initialize the weights using he's method
model.add(keras.layers.Dense(10, input_dim=1, activation='relu', kernel_initializer='he_uniform'))
model.add(keras.layers.Dense(1))
model.compile(loss='mse', optimizer='adam')
model.fit(u, y, epochs=50, batch_size=100, verbose=0)
yhat = model.predict(u)
```

Figure 2.2: Network architecture: the sequential networks with one hidden layer with 10 nodes

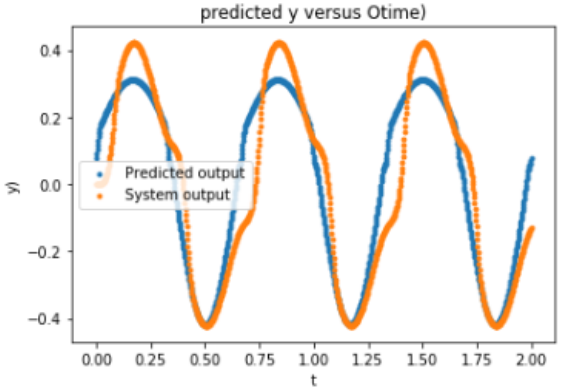

Figure 2.3: Network prediction with 50 epochs

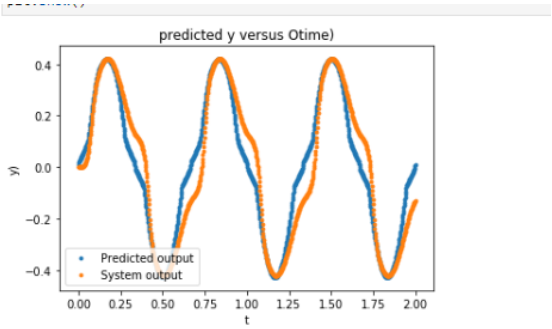

Figure 2.4: Network prediction after 1000 epochs' training

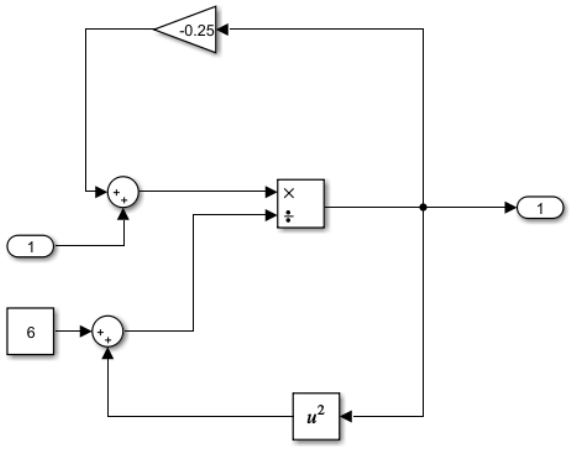

Figure 2.5: Block diagram of the given system in question 2

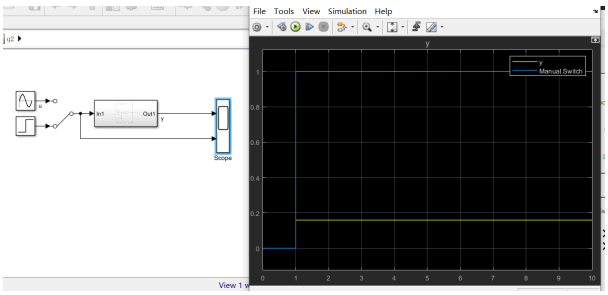

Figure 2.6: Open loop response with step command

2.2 Question 2

There is no easy way to derive the transfer function of the given system. The system is modelled is Simulink with blocks shown in Fig 2.6. From Fig 2.6 and 2.7, the given system has steady state error. After the insertion of a PI controller, the closed loop system can track the step input but with a large settling time. The P and I of the PI controller are respectively 2 and 1. The N is set to 0. The designed system output for the saturated sin input is shown in Fig 2.9

$$C(s) = P + I\frac{1}{s} + D\frac{N}{1 + N\frac{1}{s}} \tag{2.1}$$

An RBF network was designed to improve the PID performance. The problem is reformulated as follows: optimise three parameters P, I, and D with the help of a RBF network. There three parameters are equivalent to K_c, τ_D and τ_D in

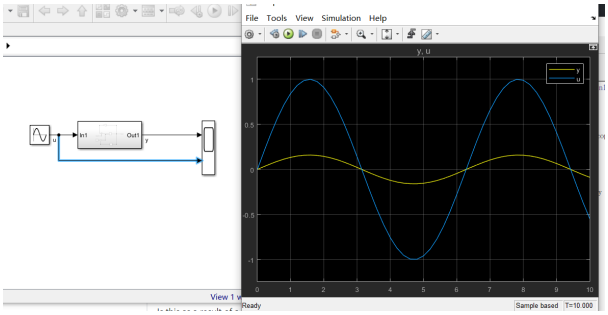

Figure 2.7: Open loop response with sinusoidal command

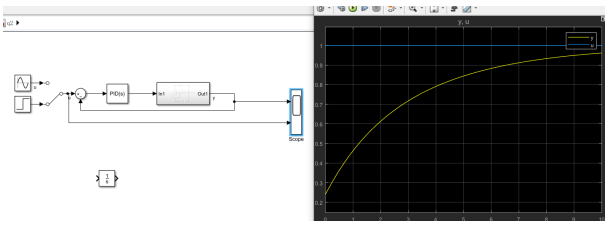

Figure 2.8: Step response with PID controller

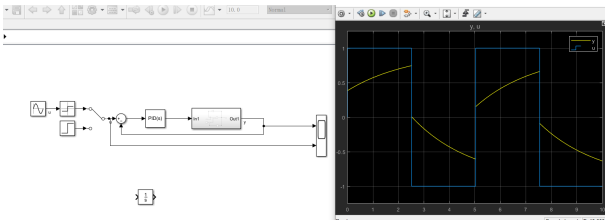

Figure 2.9: System output for saturated sinudoidal input with PID controller

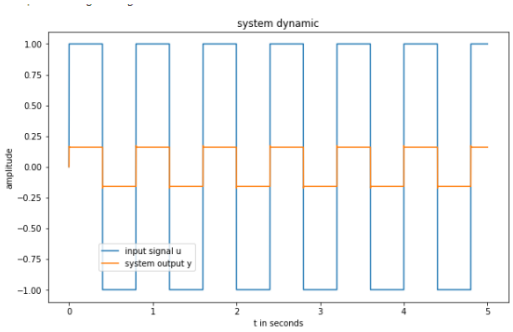

Figure 2.10: Open loop system output on Python

Equation 2.2. The RBFNN-system architecture is shown in Fig 2.12, which again is inappropriate in my point of view. Such system ignore the RBF nature and is merely a 'masterpiece' of combination [10] [2]. To understand the RBF in a wider sense, please refer to discussion section part. T

$$u(t) = u_{bias} + K_c e(t) + \frac{K_c}{\tau_I}\int_0^t e(t)dt - K_c\tau_D\frac{d(e(t))}{dt} \tag{2.2}$$

where $e(t)$ is the difference between command signal and the measured output and ubias was set to u(t) to have a smooth form when $e(t) = 0$ (i.e. the PID is turned on)

To Properly set up the work space in Python, the system and PID controller were re-implemented as shown in Fig 2.10 and 2.11. The system performance was greatly improved when τ_D was set extremely small. he RBF network that identifies the system take u_t. y_{t-1} and y_t as inputs. The RBFNN above PID controller is supposed to give out $\frac{\alpha y}{\alpha u}$, which is used to tune PID with incremental PID controller method. Since this variation of RBF is out of norm, it should be implemented from scratch. The RBF-NN consists of 1 hidden layer of 6 neuros and takes the RBF transform of $u(t-1)$ and $y(t-1)$ as inputs. The RBF transform applies k-mean algorithm to estimated the center vector and uses $\frac{d_{max}}{\sqrt{2*k}}$ as the denominator where d_{max} is the maximum distance between centroids and k represents the number of RBF neurons. The RBF-NN updates its weights via adam's gradient descend method. Please note that the hidden layer and the output can be viewed as a linear regression problem. But the inverse of matrix often encounter rank problem, thus is not our primary option. The training samples are white noises which risks not representing the input space. The performance of RBF-NN is shown in Fig 2.13. For the saturated sinusoidal

[14]:
```
# closed loop behavior with PID
import numpy as np
from IPython.core.debugger import set_trace
def systemOutput(ytminus1,utminus1):
    return (-0.25*ytminus1+utminus1)/(6+ytminus1**2)
Kc = 4
tauI = 0.01
tauD = 0
# command signal
sfreq=1e3
tmin = 0
tmax = 5
t = np.linspace(tmin,tmax,int((tmax-tmin)*sfreq))
u = np.sign(np.sin(2.5*np.pi*t))

y = np.zeros(len(t))
e = np.zeros(len(t))
de = np.zeros(len(t))
ie = np.zeros(len(t)) # integral of error
PID_op = np.zeros(len(t)) #PID output
for t_i in range(len(t)-1):
    e[t_i+1] = u[t_i] - y[t_i]
    de[t_i+1] = (e[t_i+1] - e[t_i])*sfreq
    ie[t_i+1] = ie[t_i] + e[t_i+1] / sfreq
    PID_op[t_i+1] = Kc*e[t_i+1] + Kc/tauI*ie[t_i+1] - Kc * ta
    y[t_i+1] = systemOutput(y[t_i],PID_op[t_i+1])
#     set_trace()
#Visualization
from matplotlib import pyplot as plt
fig = plt.figure(figsize = [10,6])
ax1 = fig.add_subplot(111)
ax1.plot(t,u,label='input signal u')
ax1.plot(t,y,label='system output y')
ax1.set_title('system dynamic')
ax1.set_xlabel('t in seconds')
ax1.set_ylabel('amplitude')
ax1.legend(loc='lower left', bbox_to_anchor=(0.1,0.1))
```

[14]:
```
<matplotlib.legend.Legend at 0x2170665c518>
```

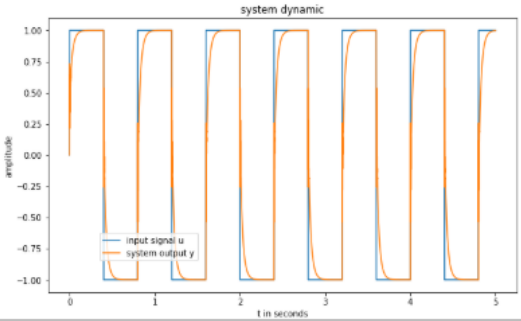

Figure 2.11: System output with $Kc = 4$ and $tau_D = 0.01$

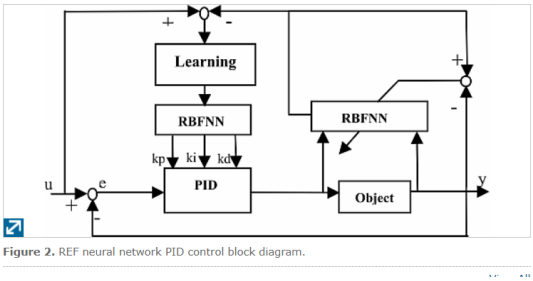

Figure 2.12: RBFNN-base PID tuning system [10]

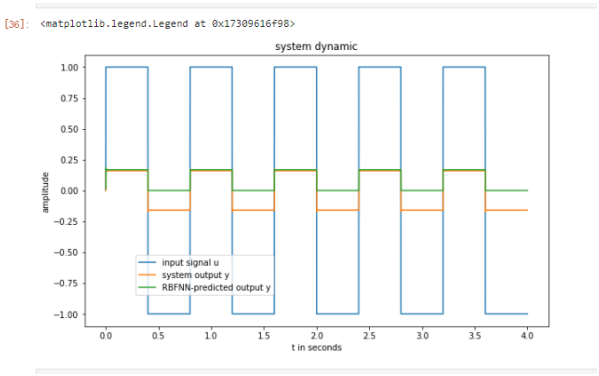

Figure 2.13: RBF-NN prediction

inputs, the RBF-NN can mimic the system when the input is positive, but fails in other circumstance. This occurs commonly is practice, we herein accept the imperfection instead of artificially' tuning the RBF-NN.

The next step is to make use of the smooth gradient $\frac{dy}{du}$ deduced from the RBF-NN. The command signal u for a discrete closed-loop can be written as:

$$u(t-1) = K_p error(t-1) + K_i \sum_{j=0}^{t-1} error(j) + K_d[error(t-1) - error(t-2)] \tag{2.3}$$

$$\begin{aligned} \Delta u(t) &= u(t) - u(t-1) \\ &= K_p(error(t) - error(t-1)) + K_i error(t) \\ &+ K_d(error(t) - 2error(t-1) + error(t-2)) \end{aligned} \tag{2.4}$$

Thus,

$$\begin{aligned} u(t) &= \Delta u(t) + u(t-1) \\ &= u(t-1) + K_p(e(t) - e(t-1)) + K_i e(t) + K_d(e(t) - 2e(t-1) + e(t-2)) \end{aligned} \tag{2.5}$$

The three input to the PID controller is:

$$\begin{aligned} xc(1) &= e(k) - e(k-1) \\ xc(2) &= e(k) \\ xc(3) &= e(k) - 2e(k-1) + e(k-2) \end{aligned} \tag{2.6}$$

where $e(k) = u(k) - y(k)$

Here we take MSE cost for subsequent optimization: $J(k) = E(k) = \frac{1}{2}e^2(k)$ The three PID parameters are optimised according to gradient descend method.

$$\begin{aligned} \Delta k_p &= -\eta \frac{\partial E}{\partial k_p} \\ &= -\eta \frac{\partial E}{\partial y}\frac{\partial y}{\partial \Delta u}\frac{\partial \Delta u}{\partial k_p} \\ &= \eta e(k)\frac{\partial y}{\partial \Delta u} xc(1) \\ \Delta k_I &= -\eta \frac{\partial E}{\partial k_l} \\ &= -\eta \frac{\partial E}{\partial y}\frac{\partial y}{\partial \Delta u}\frac{\partial \Delta u}{\partial k_l} \\ &= \eta e(k)\frac{\partial y}{\partial \Delta u} xc(2) \\ \Delta k_D &= -\eta \frac{\partial E}{\partial k_D} \\ &= -\eta \frac{\partial E}{\partial y}\frac{\partial y}{\partial \Delta u}\frac{\partial \Delta u}{\partial k_D} \\ &= \eta e(k)\frac{\partial y}{\partial \Delta u} xc(3) \end{aligned} \tag{2.7}$$

where η is the learning rate. In the RBF network, the network output y_m can be written as

$$y_{NN}(k) = w_1 h_1 + w_2 h_2 + \ldots\ldots + w_m h_m \tag{2.8}$$

h_m is the RBF kernel. In our case, the RBF kernel remains while the weights updates during the learning iterations. The $\frac{\partial y}{\partial \Delta u}$ can be obtained by the RBF-NN

$$\begin{aligned} \frac{\partial y(k)}{\partial u(k)} \approx \frac{\partial y_{NN}(k)}{\partial u(k)} &= \sum_{j=1}^{m} \frac{\partial y_{NN}(k)}{\partial h_j(k)}\frac{\partial h_j(k)}{\partial u(k)} \\ &= \sum_{j=1}^{m} w_j h_j \frac{c_{ji} - u(k)}{b_j^2} \end{aligned} \tag{2.9}$$

```
    H = rbf_list(np.array([[u[t_i],y[t_i]]]),centroids,std_list)
    dydu =np.sum(H*W*(centroids[:,0]-np.repeat(np.array([u[t_i]]),
    e[t_i+1] = u[t_i] - y[t_i]
    ie[t_i+1] = ie[t_i] + e[t_i+1] / sfreq
    PID_op[t_i+1] = Kp*e[t_i+1] + KI*ie[t_i+1]
    y[t_i+1] = systemOutput(y[t_i],PID_op[t_i+1])
    # update Kp and KI
    Kp = Kp+eta*e[t_i+1]*dydu*(e[t_i+1]-e[t_i])
    KI = KI+eta*e[t_i+1]*dydu*e[t_i+1]
#Visualization
from matplotlib import pyplot as plt
fig = plt.figure(figsize = [10,6])
ax1 = fig.add_subplot(111)
ax1.plot(t,u,label='input signal u')
ax1.plot(t,y,label='RBF-NN based PID control system output y')
ax1.set_title('system dynamic')
ax1.set_xlabel('t in seconds')
ax1.set_ylabel('amplitude')
ax1.legend(loc='lower left', bbox_to_anchor=(0.1,0.1))
```

```
<matplotlib.legend.Legend at 0x1730cd17978>
```

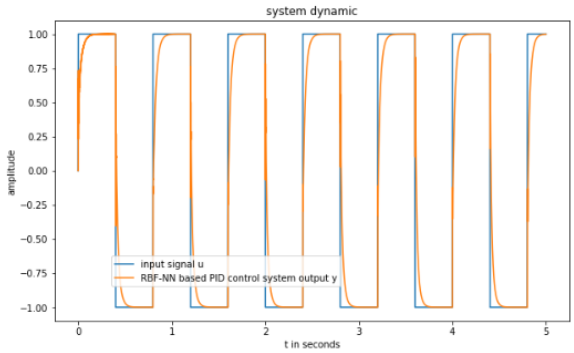

Figure 2.14: RBFNN-base PID tuning system performance

In implementation, we have seen PI control yields satisfactory result, therefore the derivative will not be considered in RBF-NN based PID control. The results and comparison are shown in Fig 2.14 and 2.15. It is noticed that the RBF-NN PID control is superior even thought the system dynamic is not fully learnt by the RBF-NN

2.2.1 Final thought

Although RBF-NN does bring about performance improvement to the control, I dislike the way that it is incorporated in PID control. Such improvement is expected due to the incremental PID control theory.

2.3 Question 3

There are two constrain-handling techniques for genetic algorithm: applying penalty functions [6] and VCH (Violation Constraint-Handling) method [1]. In

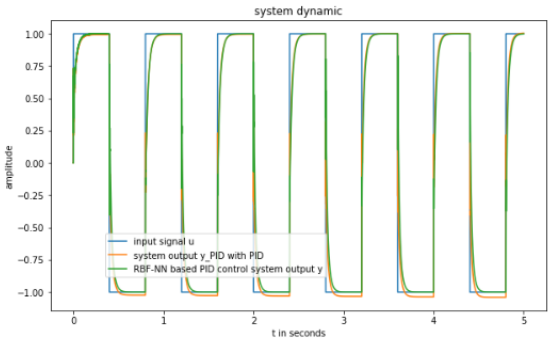

Figure 2.15: comparison between RBF-NN PID control and PID control prediction

this question, we prefer the VCH approach as it require less parameter tuning. The key step occurs during the population sorting (i.e. selecting the parents). The population is sorted according to a set of rules to evolve the population at each generation:

1. If one individual is infeasible and the other is feasible, the winner is the feasible solution

2. If both individuals are feasible, the winner is the one with the highest fitness value

3. If both individuals are infeasible, the winner is the one with the lowest Number of Violations (N.V)

4. If both individuals are infeasible with the same (N.V), the one with the lowest Constraints Violation (C.V) value wins

To calculate N.V. and C.V., equality constrains are transformed to inequality constraints using the 2.10 expression:

$$\begin{aligned} h(\vec{x}) &= 0 \\ |h(\vec{x})| - \epsilon &\leq 0 \end{aligned} \tag{2.10}$$

As for the genetic algorithm (GA) parameters. The crossover probability (CP), also called recombination probability, is the probability that a genetic operator used to combine the genetic information of two parents to generate new offspring. In view of implementation, when a normally distributed variable is larger than CP, the crossover occurs between two parents, otherwise, the offspring will have the exact same gene as one of their parents. To make the

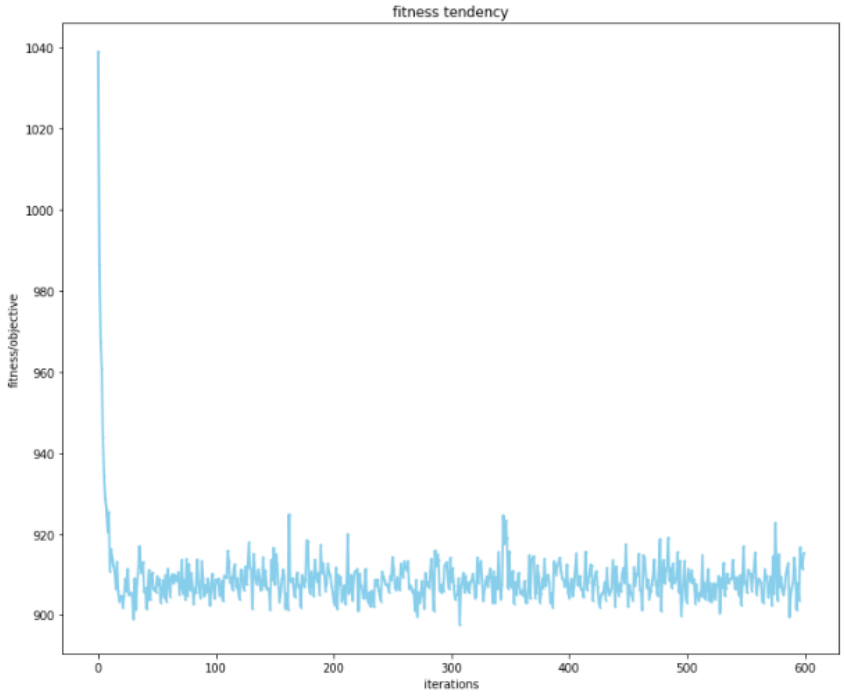

Figure 2.16: Evolution of the fitness value

GA more coherent with natural selection theory, the mutation can happen on both parents and crossover offsprings. Thus the pseudo code can be written as follows,

(i) initialize the population with size of 'pop_size'

(ii) natural selection based on the above rules

(iii) the selected parents are recombined with the probability CP. Please note that the offspring size is set to the difference between population size and the number of parents because parents are regarded as survivors after the natural selection.

2.3.1 Results and discussion

The evolution of the fitness value (i.e. the objective function) are shown on Fig 2.16, 2.17 and 2.18. The fitness value converge to a minimum value about 902. However, N.V. and C.V. do not ave the converging tendency. Two main reasons accounts for this phenomenon. First it's the selection bias. The C.V. and N.V,

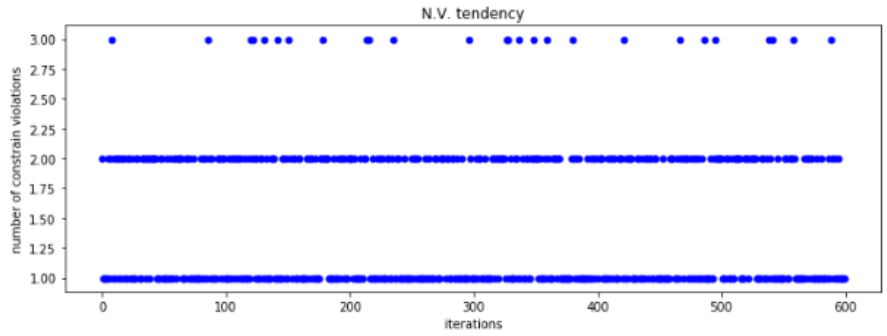

Figure 2.17: Evolution of the N.V.. The C.V. is the one of the individual having smallest fitness value in each generation

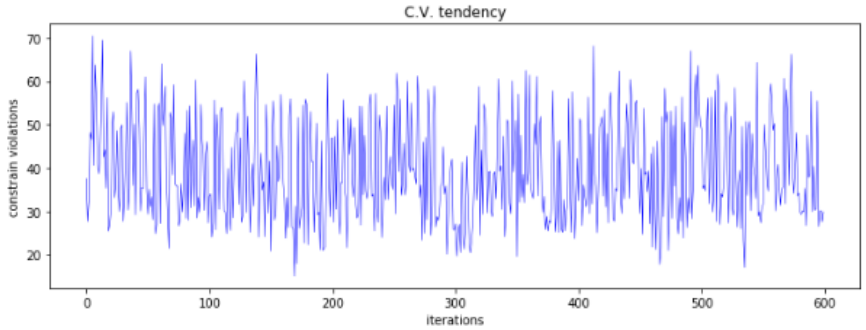

Figure 2.18: Evolution of the C.V.. The C.V. is the one of the individual having smallest fitness value in each generation

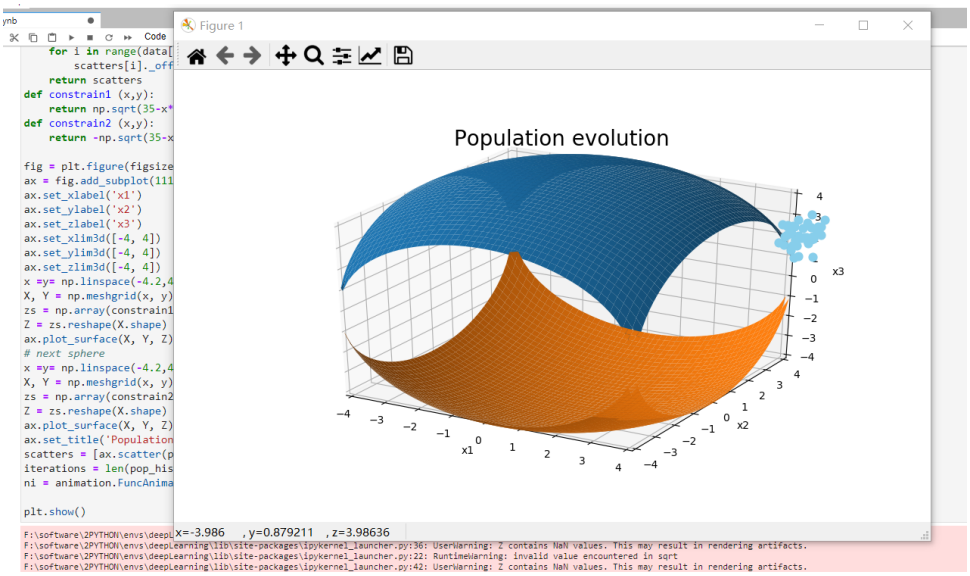

Figure 2.19: Population evolution animation screenshot (1)

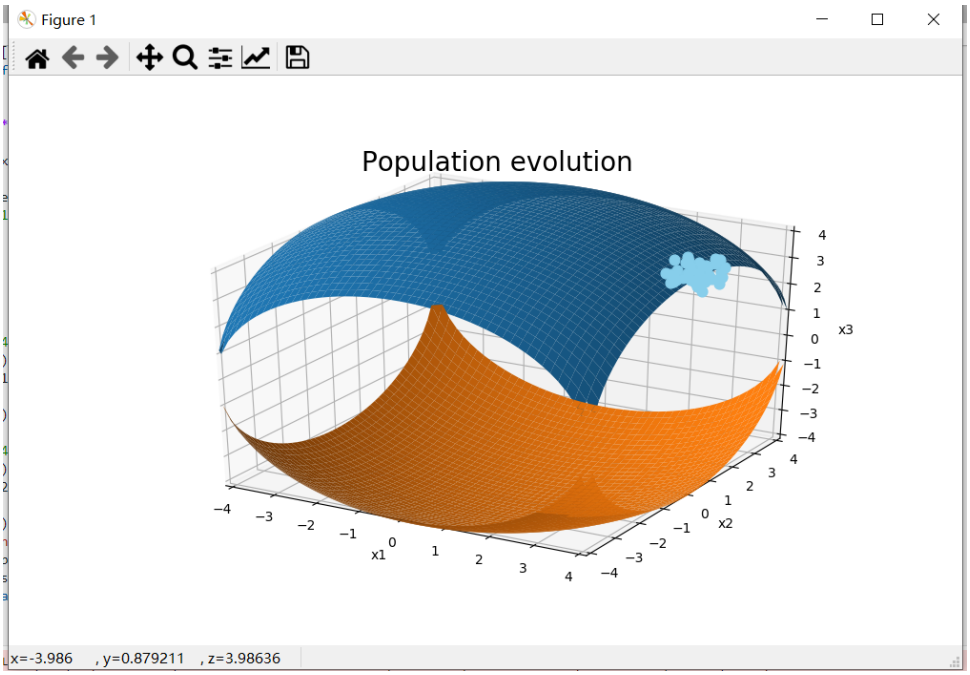

Figure 2.20: Population evolution animation screenshot (2)

are those of the individual having smallest fitness value in each generation. Further, the mutation occurring both on parents and offspring results in the population jumping out of the local minimum. The evolution model can be improved by tuning the parameters: population size, crossover probability (CP) and mutation probability (MP). Fig 2.19 and 2.20 show the animation screenshots of how the population evolve. One of the constrain is depicted as the 3Dsurface. Readers are recommended to open the corresponding notebook to see the animation.

2.4 Question 4

2.4.1 Literature Review

We recall that in Question 2, the term $\frac{\partial y(k)}{\partial u(k)}$ is obtained by the RBF-NN and was used to update the PID controller parameters. In this question, I would like to implement a different type of neurocontroller, RNN. In the first place, I summarize the incorporation method of neural network in control systems. [9]. Multilayer neural networks have been applied successfully in the identification and control of dynamic systems. Three typical neural network controllers are

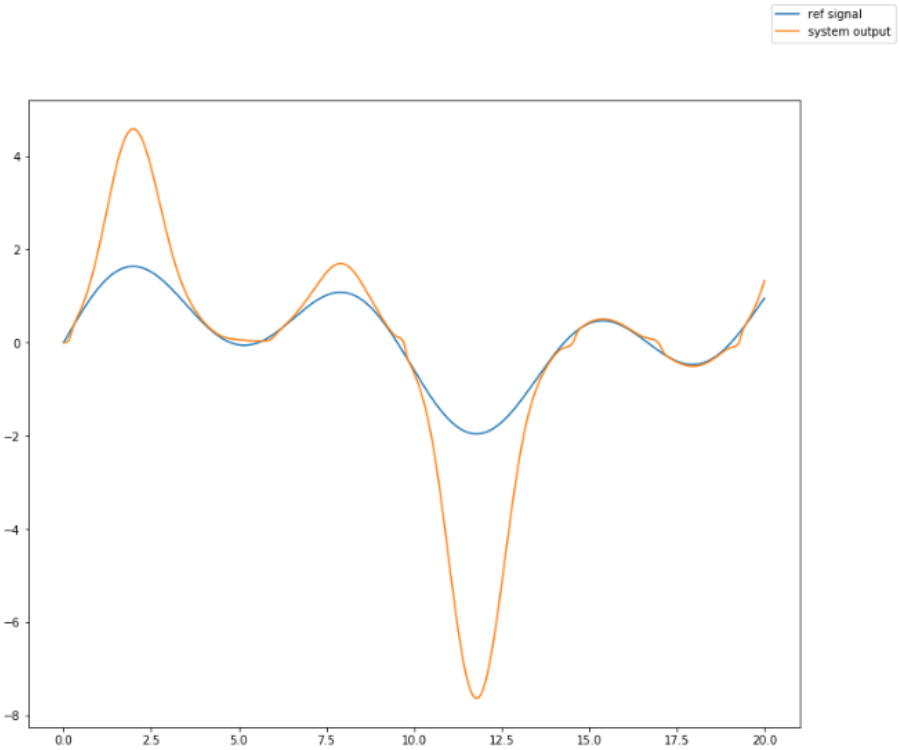

Figure 2.21: System evolution with reference input

model predictive control[8], NARMA-L2 control[7], and model reference control [5]. Two steps are often involved when using neural networks for control: system identification and control design. For the question 4, I would like to implement a novel gradient-based approach for direct control. Parts of the main ideas comes from Xu's work [9]. The work presented in this report refines Xu's model and its physical interpretation is commented.

2.4.2 Main idea

The system dynamic is presents in Fig 2.21. The system behaviors are similar to the reference signal, however the system fails to track the reference signal.

The main philosophy is to design a neural network that identifies a part of the non-linear dynamic. Besides, the optimisation of this network will lead to eventual minimum tracking error. The neural network architecture that I choose for this question is recurrent neural network (RNN). But as readers continue reading, they will discover the real-time tuning will make this RNN a pseudo-RNN. Let us assume that the system output $y(k)$ can be modelled as

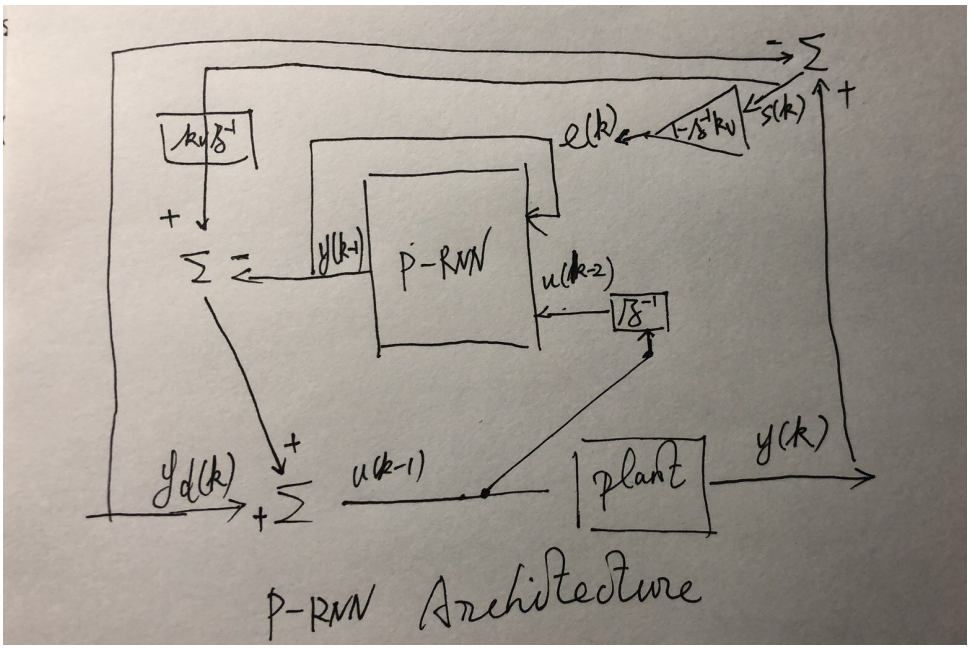

Figure 2.22: Pseudo-RNN architecture

follows:

$$y(k) = f^*(k-1) + u(k-1) \tag{2.11}$$

$$\widehat{f^*(k-1)} = W_2 * \sigma(W_1 x(\vec{k-1})), \tag{2.12}$$

where σ is the sigmoid activation. $f^*(k-1)$ is the non-linear dynamic that we want to approximate with a pseudo-RNN. W_1 and W_2 are the weights of input-hidden layers and hidden-output layer. The network input $x(k-1)$ = [u(k-2),y(k-1)] where the name pseudo-RNN comes. Combining with the following definition of $u(k-1)$, the minimum RNN error function is identical to minimum tracking error.

$$u(k-1) = -y(k-1) + d(k) = k_v s(k-1) \tag{2.13}$$

We can here in show that to minimize the RNN network error $e(k)$ (i.e. to train RNN to learn $f^*(k-1)$) is to minimize the tracking error.

$$\begin{aligned} e(k) &= f^*(k-1) - y(k-1) \\ &= y(k) - u(k-1) - y(k-1) - d(k) + d(k) \\ &= s(k) - k_v s(k-1) \end{aligned} \tag{2.14}$$

k_v is set to be a value between 0 and 1, ensuring the $s(k) = s(k-1)$ would not happen. The block diagram of the neural-control system is presented in Fig 2.22. Now let's focus on the gradient. We have two layers' weight to be tuned, W_1 and W_2. The cost function $E(k)$ is the MSE, which equals to $\frac{1}{2}e^2$.

$$\begin{aligned} W_2(k+1) &= W_2(k) - lr * \frac{dE(k)}{dW_2(k)} \\ &= W_2(k) + lr * e(k)\sigma\prime(k) \end{aligned} \tag{2.15}$$

We specify here that the$\sigma\prime(k)$ is the derivative of the sigmoid function at the hidden layer value of k^{th} iteration (time). Please note that the online tuning rendres the p-RNN no longer an RNN, thus the gradient for $W_1(k)$ only depends on the value of k^{th} iteration.

$$\begin{aligned} W_1(k+1) &= W_1(k) - lr * \frac{dE(k)}{dW_2(k)} \\ &= W_1(k) + lr * e(k)\sigma'(k)W_2(k)e(k)x^T(k) \end{aligned} \tag{2.16}$$

The pseudo-code is illustrated below as an auxilary tool to understand the p-RNN architecture:

1.

2.4.3 Results and Further improvement

The implementation was done with loops, and a module called p-RNN is created for further testing. The performance of the neurocontroller with $lr = 0.2$ is shown in Fig 2.23. It is expected the controller perform randomly at the begining because the p-RNN is randomized and does not understand $f^*(k-1)$. But the system performance is optimized after several iteration. Besides, the neuro-controller was expected to have a slight delay because its k^{th} output depends on $(k-1)^{th}$ reference signal and the plant output. This is a minor problem as it can barely perceived in Fig 2.23. The main advantage of such architecture is its adaptivity to any disturbance and reference signal theoretically if the sampling frequency is higher enough. The lr parameter can set higher for disturbance with dramatic fluctuation and lower with quasi-static periodical reference signal.

The optimization algorithm has not been refined because p-RNN does not

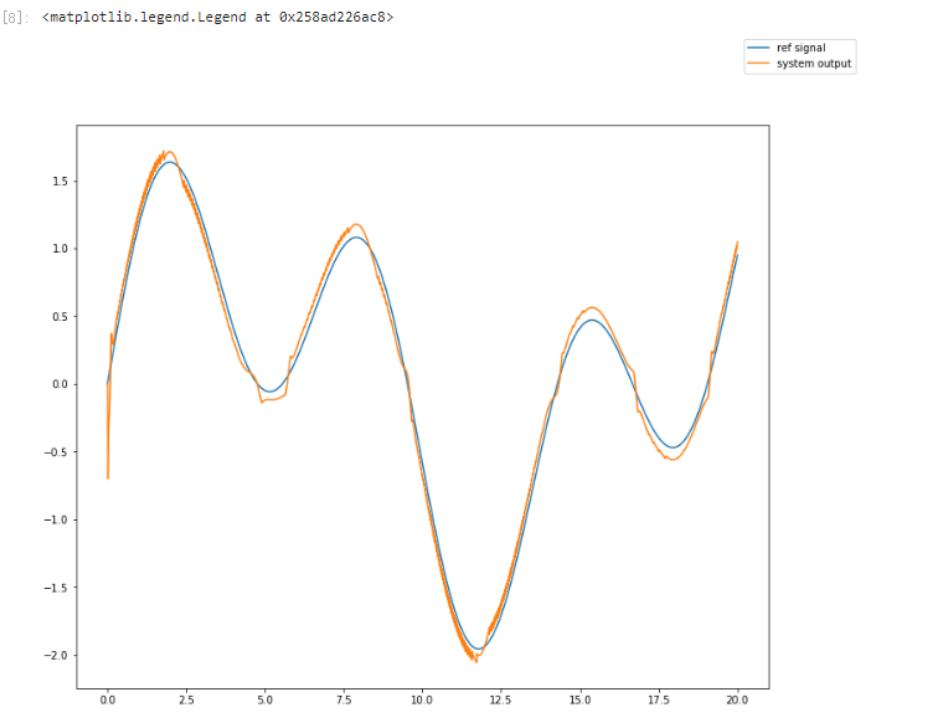

Figure 2.23: Performance of neurocontroller

have problems such as vanish gradient. But for a general problem, such architecture might require attentions on gradient descend algorithm. Suggestions on tuning algorithm will be published on GanshengT's Github, readers are welcomed to follow if interested.

2.5 Question 5

In short, this article is more suitable for educational purpose than illustrating the authors' original idea. To better understand what a single-index model is, I additionally consult [1]. If I understand correctly, the single-index model defined in this paper suggest the single-layer neural network. That is, the response variable y can be represented by the weighted sum of variable Z and smoothed variable X and the residual term ϵ.

The aim of the article is to estimate the predictive variable. It provides detail background information about fuzzy neural network, scatterplot smoother and particle swarm optimisation. I prefer to call the scatterplot soother regression. I appreciate the various illustrative examples that give out the comparison

between concerning algorithms. The fuzzy neural network is no other than replacing the weight with fuzzy parameters. As an experienced machine learning researcher, I depreciate its contribution in data estimation techniques. On one hand, I recall that if enough samples are provided, the relationship between input variable and predictive variable can be obtained via the conventional neural network, which is stated as "universal approximation property" in this article. However, this theorem does not touch upon the algorithmic learnability of network parameters. Besides, a single layer neural could not present the data architecture in realistic problems [2]. The proposed model is not proved solid in the circumstance of scarce samples. On the other hand, this paper does not contain enough original idea and work. The proposed model is a combination of fuzzy neural network, activation function estimation particle swarm optimisation which are respectively extensively developed.

Finally, I comment the advantage of fuzzy neural network that it provides a reasonable and intuitive initial configurations of the network. A good initialization that lies on the concave loss landscape of the neural network would avoid the vanishing gradient in network optimisation and increase computational efficiency. Further work should be focus on the potential of initializing the neural network based on linguistic variable.

Chapter 3

Discussion and acknowledgement

I would like to express my gratitude to Professor Guanglin Shi and my classmates in Intelligent control course for their guidance and support. The notion of neural network has been widely integrated in control system. I concern about the false combination of these tool as some may deviate the original purpose and model such as the paper in question 5. It is not an exception and is presented in numerous journals. The misuse seems legitimate, but actually complexifies the problem. Rebuttals on these issues are needed for further application of neural network on control theory.

Bibliography

[1] Adam Chehouri et al. “A constraint-handling technique for genetic algorithms using a violation factor”. In: *arXiv preprint arXiv:1610.00976* (2016).

[2] Longlong Cheng et al. “Radial basis function neural network-based PID model for functional electrical stimulation system control”. In: *2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. IEEE. 2009, pp. 3481–3484.

[3] François Chollet. *keras*. `https://github.com/fchollet/keras`. 2015.

[4] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[5] S Narendra Kumpati, Parthasarathy Kannan, et al. “Identification and control of dynamical systems using neural networks”. In: *IEEE Transactions on neural networks* 1.1 (1990), pp. 4–27.

[6] Zbigniew Michalewicz. “Genetic algorithms, numerical optimization, and constraints”. In: *Proceedings of the sixth international conference on genetic algorithms*. Vol. 195. Citeseer. 1995, pp. 151–158.

[7] Kumpati S Narendra and Snehasis Mukhopadhyay. “Adaptive control using neural networks and approximate models”. In: *IEEE Transactions on neural networks* 8.3 (1997), pp. 475–485.

[8] D Soloway and PJ Haley. “Neural Generalized Predictive Control Proceedings of the 1996”. In: *IEEE International Symposium on Intelligent Control*. 1996, pp. 277–281.

[9] Zhao Xu, Qing Song, and Danwei Wang. “Recurrent neural network based tracking control”. In: *2010 11th International Conference on Control Automation Robotics & Vision*. IEEE. 2010, pp. 2454–2459.

[10] Yan Zhang et al. “Application of RBF neural network PID controller in the rectification column temperature control system”. In: *2013 Sixth International Symposium on Computational Intelligence and Design*. Vol. 2. IEEE. 2013, pp. 72–75.