Mason Rodriguez Rand and Chris Simotas 9/25/2023 **ME 249 Project 2 Report**

**1. Introduction**

In project one we utilized genetic algorithms to model heat transport. In project two we explore an alternative approach by employing the Keras library which acts as an interface for constructing neural networks in TensorFlow. Keras, an open-source library, offers a suite of user-friendly features facilitating fine-tuning of model performance such as adjustments in network layers, parameters, activation functions, and more. To understand how changing these parameters affects our model, we began with a simple case by comparing a simple neuron model constructed with Keras and then with a First Principles approach. As we will see, the two are very similar and provide a useful reference for comparison. Next, we modeled a gas turbine system using the Keras neural network and evaluated its performance. Through manipulation of weight-updating algorithms, activation functions, weight intervals, and model architecture, we gained a deeper understanding of the tools within Keras and how to leverage them to reduce model error, convergence time, and computational load.

**2. Separation of Tasks**

We worked in lock-step through this project. For each task, we made sure to discuss before and after to ensure we were up to speed. Here is a list of the separation of tasks for Project 2:

● Task 1

○ Tasks 1.1, 1.2(a), and 1.2(b): Chris and Mason each individually completed Task 1.1 separately and then discussed results and trends.

○ Task 1.2(c): Mason created log-log plots and wrote discussion

○ Task 1.2(d): Chris trained network with 20% higher initial guesses and wrote discussion

● Task 2

○ Task 2.1, 2.2, and 2.3: Chris performed data processing and code modifications to the provided code. Chris also created the surface plot and analyzed/plotted results of network’s performance with test data

○ Task 2.4: Chris performed network model training for the various requested

configurations. Mason performed analysis and wrote the discussion.

**3. Task 1**

**3.1. Program Modifications**

While performing Task 1.1, we altered the gamma value shown below to explore the effect on the speed of the algorithm and fitness:

*CodeP2.1-F23*

**We then adjusted the E3 setpoint as instructed in the project file:

*CodeP2.1-F23*

**

To run the neural network, we modified the number of epochs:

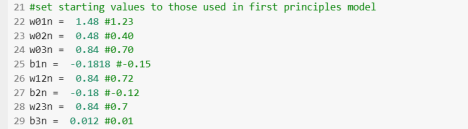
*CodeP2.2-F23*

**We then adjusted the optimizer’s learning rate to aid in training the model:

*CodeP2.2-F23*

**Finally, when exploring the second round of initial guesses, we adjusted them as follows:

*CodeP2.2-F23*

**

**3.2. Discussion of First Principles and Keras Model Performance Using Initial Guesses Near Final Values**

We began our investigation into the Keras neural network by comparing its performance against the First Principles neural network. Using the same initial guesses, we split up and trained each model on our own to see how the performances compared.

The weights and biases for the two model types differed by only a few hundredths. This was expected as the First Principles and Keras neural networks are both models of the same simple neuron architecture. The errors for each of the model types were respectively low.

**Table 1.** Comparison of First Principle and Keras Best Models

| **Weights**  **and Biases** | **First Principles**  **(Chris)** | **Keras**  **First Principles**  **Keras**  **(Chris)**  **(Mason)**  **(Mason)** |
| --- | --- | --- |
| w01 | 1.220772 | 1.2121949 1.2223798 1.2096403 |
| w02 | 0.388512 | 0.35976085 0.390473 0.36096197 |
| w03 | 0.688697 | 0.7124708 0.690621 0.7128528 |
| b1 | -0.161311 | -0.14302076 -0.159386 -0.14252366 |
| w12 | 0.716665 | 0.7057795 0.71723853 0.70485824 |
| b2 | -0.128132 | -0.10898821 -0.1267453 -0.10959321 |
| w23  b3 | 0.696514  0.004317 | 0.681808 0.697113 0.6814442 0.02517959 0.005287745 0.02343724 |

Rms =√(E3) =

√(0.000285) = .0169

Error Loss = 0.0124489

Rms = √(E3) = √(0.0003488) = .0182

Error (loss) = 0.01246227

Examining the predicted output values from the dataset, as shown in **Table 2**, we see a similar relationship; both the First Principles and Keras models produce similar results. For each set of input conditions, each model better predicted a similar amount of y3 values than the First Principles approach.

**Table 2.** First Principle and Keras Best Models Performances

| **x01** | **x02** | **x03** | **Y3**  **data** | **First Principles**  **Keras**  **First**  **Predicted y3**  **Predicted y3**  **Principles**  **(Chris)**  **(Chris)**  **Predicted y3**  **(Mason)** | **Keras**  **Predicted y3 (Mason)** |
| --- | --- | --- | --- | --- | --- |
| 20.0 | 13.0 | 310.8 | 30.97 | 31.06 31.13 31.29 | 30.99 |
| 20.0 | 14.5 | 308.0 | 32.3 | 31.61 31.61 31.85 | 32.20 |
| 20.0 | 15.3 | 306.0 | 31.5 | 31.88 31.85 32.12 | 31.69 |
| 20.2 | 13.0 | 310.8 | 30.91 | 31.26 31.32 31.49 | 30.92 |
| 20.0 | 14.5 | 308.0 | 32.5 | 31.61 31.61 31.85 | 32.40 |

20.0 15.3 306.0 31.4 31.88 32.03 32.13 31.40 24.0 13.0 310.8 35.59 34.97 34.87 35.22 35.53 36.0 14.5 308.0 46.4 47.25 46.58 47.53 46.40

**Figure 1** shows model performance visually with a log-log plot of predicted vs actual *y3* values. Overall, the two models’ outputs were fairly similar. However as the real *y3* value increased, the difference between the predicted *y3* values for each model grew.

|  |
| --- |

**Figure 1.** Log-Log Plot of First-Principles vs. Keras Model Performance

To understand how this difference occurred, we looked at each model’s setup and implementation. Both models were set up very similarly, with three dense layers; the input layer with 4 parameters (3 weights and 1 bias), and the middle and output layers with 2 parameters (1

weight and 1 bias). Additionally, we gave each model the same starting initial weights and biases. Both models operated on the same principle with each epoch consisting of one forward pass and one backward pass utilizing the same simple gradient descent backpropagation algorithm. They each also employ the exponential linear unit function as their activation function and utilize the same data normalized in the same manner. Therefore, we could rule out most model errors as caused by differences in model setup and instead look to implementation.

In both models, we varied the learning rate gamma and the number of epochs. Varying epoch values did not have nearly as large an effect on final ypredicted when training in both models as varying the learning parameter. While varying the number of epochs helped reach the established loss cutoff value in each case, the learning parameter directly influenced the value of each weight and bias backpropagation adjustment. Specifically, the learning parameter affected the path the model took in the solution space towards a minimum, and thus is most likely the reason for the divergence between the two models in **Figure 1**. Adjusting epochs only altered model convergence time. Additionally, as seen in the log-log plot, the two models differ more at higher y3 values. This is largely due to the differences in learning parameters used as the learning parameter is multiplied by the expression [(0− E3 ) / (∂E3 / ∂w01)] for each weight and bias in the model. At larger y values, the contribution of this expression to its respective individual weight or bias has a larger influence on growing or shrinking ypredicted overall.

Furthermore, the two models differed in their execution and speed to convergence. When fine tuned to the goldilocks zone of 0.018 to 0.025, the First Principles model performed with its best fitness at only 200 epochs. When the learning parameter was not in that range, the model struggled to converge towards its best fitness. On the other hand, it seemed that no matter what learning rate we gave the Keras model, it converged close to its best performance. However, it often took around 400 epochs to get near its best fitness and then another training round of 400 epochs to further improve the model to its best fitness.

**3.3. Discussion of First Principle and Keras Model Performance Initial Guesses >20% Than Previous Initial Guesses**

To expand our investigation, we examined model sensitivity to initial guesses, which were very near the final weights and biases of the models in Section 3.2. Thus, we multiplied each initial guess by 20%.

Larger initial guesses only affected the convergence rate of one of the two models. After only needing 200 epochs and a fine tuned learning rate to converge previously, the First Principles model required 400 epochs to converge on a solution with a similar RMS as before. In contrast, the Keras model's convergence rate remained unaffected by the revised initial guesses, achieving convergence within a similar epoch count as before. Moreover, the First Principles model was again sensitive to the inputted learning rate. The model struggled to converge on a solution when the learning rate was not within its goldilocks zone of 0.018 to 0.025. The Keras model again was not greatly impacted by the inputted learning rate. The models’ performances and builds are shown in **Table 3 and Table 4**, respectively.

**Table 3.** Chris’s Task 1 First Principle and Keras Best Model Parameters

| **Weights**  **and Biases** | **Original Guesses - First Principles**  **Model** | **Original Guesses -**  **20% Higher**  **20% Higher**  **Keras Model**  **Guesses -**  **Guesses -**  **First Principles**  **Keras Model**  **Model** |
| --- | --- | --- |
| EPOCHS | 200 | 800 400 800 |
| w01 | 1.220772 | 1.212195 1.422032 1.287018 |
| w02 | 0.388512 | 0.359761 0.296788 0.361990 |
| w03 | 0.688697 | 0.712471 0.696169 0.699549 |
| b1 | -0.161311 | -0.143021 0.033124 0.042590 |
| w12 | 0.716665 | 0.705780 0.591270 0.672685 |
| b2 | -0.128132 | -0.108988 0.035171 0.040003 |
| w23 | 0.696514 | 0.681808 0.736035 0.673418 |
| b3 | 0.004317 | 0.025180 -0.109486 -0.128770 |

RMS =√(E3) =

√(0.000285) = 0.0169

Error = 0.0124489 RMS = √(E3) = √(0.0003304) =

0.0182

Error = 0.012486

**Table 4.** Chris’s Task 1 First Principle and Keras Best Model Parameters

| **x01** | **x02** | **x03** | **Y3**  **data** | **Original**  **Original**  **20% Higher**  **Guesses -**  **Guesses -**  **Guesses -**  **First Principles**  **Keras**  **First Principles**  **Predicted y3**  **Predicted y3**  **Predicted y3** | **20% Higher Guesses -**  **Keras**  **Predicted y3** |
| --- | --- | --- | --- | --- | --- |
| 20.0 | 13.0 | 310.8 | 30.97 | 31.06 31.13 31.27 | 31.15 |
| 20.0 | 14.5 | 308.0 | 32.3 | 31.61 31.61 31.61 | 31.61 |
| 20.0 | 15.3 | 306.0 | 31.5 | 31.88 31.85 31.78 | 31.84 |
| 20.2 | 13.0 | 310.8 | 30.91 | 31.26 31.32 31.47 | 31.34 |
| 20.0 | 14.5 | 308.0 | 32.5 | 31.61 31.61 31.61 | 31.61 |

20.0 15.3 306.0 31.4 31.88 32.03 31.78 32.02 24.0 13.0 310.8 35.59 34.97 34.87 35.24 34.89 36.0 14.5 308.0 46.4 47.25 46.58 47.49 46.57

Surprisingly for this round of initial guesses, both the First Principles and Keras neural networks built models with different weights and biases as before. Per **Table 3**, the magnitudes of *b1*, *b2*, and *b3* flipped between the two initial guess sets for both models. However, since the biases serve as correction factors that get summed into each simple neuron, a change in their magnitudes are not as influential as if a weight’s magnitude changed.

On the other hand, the weights were more consistent between the two sets of guesses. For the Keras model, the percent changes of the weights were less than 5%. The First Principles model experienced more variation with both *w01* and *w12* having a percent change of around 16% and *w02* having a percent change of 24%.

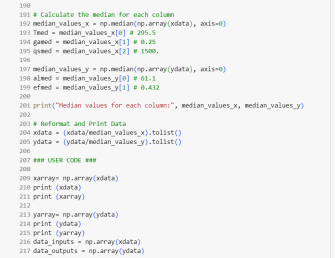
Nevertheless, even with all of the differences between the model’s parameters, all of them converged on solutions with similar fitness. As shown in **Table 4**, given the same input values, they predict *y3* values close to the actual measured *y3* values. While the Keras and (especially) the First Principles models proved to be sensitive to the initial guesses, they were still able to converge on strong solutions. In the wide expansive solution space, there are clearly many local minimums that exist.

**4. Task 2**

**4.1. Program Modifications**

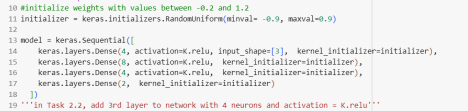
To prep the input data for Task 2, we added in lines of code to calculate the median values of all inputs to normalize the data:

*CodeP2.4-F23*

**

We completed the lines of code to initialize the layers of Keras neural network:

*CodeP2.4-F23*

**Finally, we set the learning parameter for the optimizer and set the epochs to 600:

*CodeP2.4-F23*

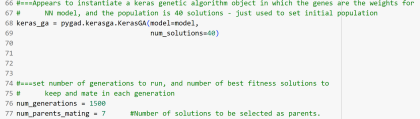
**

When analyzing the various activation functions, layer structures, and weight intervals of the Keras model utilizing the genetic algorithm, we modified the below lines of code:

*CodeP2.5-F23*

**The number of generations and solutions stayed consistent between cases:

*CodeP2.5-F23*

**

**4.2. Discussion of Keras Model With Backpropagation**

We continued our investigations into the performance of the Keras neural network by modeling a gas turbine system. Specifically in this first instance, the model was built using backpropagation to update the weights of the model each epoch.

The gas turbine system involves air entering the compressor at atmospheric pressure and an air inlet temperature, *T1.* After compression, the air undergoes a heating process in the regenerator, transferring waste heat from the turbine exhaust. Solar thermal heat input, *Qs\_dot*, in the exchanger further elevates the temperature. Finally, the air passes through a burner where a fuel of a fuel propane mole fraction, *ɣ,* is injected and burned to achieve the desired temperature or the post-fuel burn temperature, *T4.*

The optimization of a gas turbine system revolves around maximizing the *T4*temperature. As noted prior, *T1*, *Qs\_dot*, and *ɣ* are all operating conditions that play a role in how effective the system can raise *T4*to maximize efficiency. However, if these operating conditions are lower than expected, the air-to-fuel ratio, *ɑ,* can be adjusted per the values of *T1*, *Qs\_dot*, and *ɣ*, to ensure the system system maximizes *T4*. Knowing this relationship, we can build a Keras model off of the inputs of *T1*, *Qs\_dot*, and *ɣ* to output the required *ɑ* to maximize the system’s performance and efficiency, η*sys*.

After training the model several times, slowly lowering the learning parameter from 0.001 to 0.0001 to refine the optimization, we were able to finish with a model that reported a mean absolute error of 3.37% as shown in **Table 5**.

**Table 5.** Gas Turbine Model with Backpropagation Performance

|  |
| --- |
| Lowest Mean  Absolute Error |

**Keras Model with Backpropagation** 0.033864

Fixing *ɣ* to 0.25, we decided to plot the relationship of the inputs *T1* and *Qs\_dot* against *ɑ*, as shown in **Figure 2**, to see if any patterns appear. The striking first observation is that the relationship appears to be linear as the surface plot is flat, resembling a flat piece of paper with a fold in its middle. As *T1* and *Qs\_dot* increase, *ɑ* increases at a linear rate. In fact, considering the scaling of the axes and range of inputs, *T1* appears to have a greater impact on *ɑ* more than *Qs\_dot*. This can be seen as *ɑ* increases at a faster rate in respect to *T1*than *Qs\_dot.* However, in the selected range of *T1, T1* can only increase so greatly. Therefore, it will be more beneficial to maximize *Qs\_dot* if one is hoping to maximize *ɑ*, which checks out with our understanding of the system. The higher solar thermal input rate, the greater the temperature and the less fuel will be needed (a higher *ɑ* value) to obtain the desired *T4*. It is also important to note that the slope of *ɑ* in respect to each of the dependent variables, *T1* or *Qs\_dot*, is affected by the other dependent variable when held constant. For example, according to **Figure 2**, the rate increase of *ɑ* to *Qs\_dot*

is greater at a *T1* of 270K than 320K. On a similar note, even though linear, the rate increase of *ɑ* is greater when *Qs\_dot* is greater than 1750W compared to below that threshold. From all of these observations, there are two key insights we can conclude about the relationships between the selected ranges for *T1*, *Qs\_dot*, and *ɑ*: 1) a lower fuel to air ratio (the inverse of *ɑ*) is required for lower inlet temperatures and solar heat input and 2) the solar heat input rate is the main indicator of the required air to fuel ratio to maintain *T4*.

|  |
| --- |

**Figure 2.** Required Air to Fuel Ratio at Gamma = 0.25

Considering the promising performance of the model, we tested the model against a set of validation data with the main goal to see if the model is overfitting. Overall, the model performed well, however, with a slight bias. The root mean square relative deviation (RMSD) between the predicted and actual *ɑ* values was 6.28% showing strong correlation.

**Table 6.** Gas Turbine Model with Backpropagation Performance With Validation Data

|  |
| --- |
| RMS Relative Deviation |

**Keras Model With Backpropagation**

0.062814

Yet, looking at a plot of the *ɑ* predicted versus measured in **Figure 3**, we noticed that the model produced strong linear fit, but it was biased by an offset of around 2 points. In other words, the model consistently outputted a predicted *ɑ* greater than the measured *ɑ.* The most possible reasons for this bias or discrepancy can be attributed to the model overfitting the original dataset, the model lacking the required structure complexity, or the model requiring more input variables. Considering the low RMSD and the fact that the model’s regression line is only slightly off a perfect fit, training the data more thoroughly with a larger dataset would most likely be the best solution.

|  |
| --- |

**Figure 3.** Log-Log Plot of Alpha Predicted Vs. Measured

**4.3. Discussion of Keras Model With Genetic Algorithm**

With better knowledge of the performance of the Keras model using backpropagation, we wondered how this Keras model compares to the genetic algorithms we explored in our last project. Would updating the model’s weights using a genetic algorithm improve performance? To find out, we ran six different cases of the Keras model utilizing the genetic algorithm and various neural network layers, activation functions, and weight/bias ranges. **Table 7** shows our results. **Figures 4** through **9** show how each case’s predicted *ɑ* compared with the measured *ɑ* of the dataset.

**Note:** The project instructions asked to run each case several times. However, we were only able to run each case at least once as each averaged 3 - 4 hours to process. We also paid for the upgraded Google Colab Pro in order to slightly shorten this training time on one of our machines to around 2.5 hours. Only one of our two machines successfully trained over 1500 generations as the other stopped training usually around 500-900 generations. We attribute this as likely to lack of memory or settings within Google Chrome pausing background processes. We plan to find another solution to this issue in future projects. Other than exploring Google Chrome or Google Colab settings issues, we can explore other ways to increase system memory. One possible idea for example is to play with batch size as mini batches can help reduce RAM usage.

**Table 7: Genetic Algorithm Parameters and Results**

| **Case** | **1** | **2** | **3** | **4 5** | **6** |
| --- | --- | --- | --- | --- | --- |
| NN layers | 4,8,4,2 | 4,8,4,2 | 4,8,4,2 | 4,8,4,2 4,5,4,3,2 | 4,5,4,3,2 |
| Act. Funct. | relu | elu | relu | elu relu | elu |
| w,b min val | -0.9 | -0.9 | -2.9 | -0.4 -2.9 | -0.9 |
| w,b max val | 0.9 | 0.9 | 2.9 | 0.4 2.9 | 0.9 |
| Num\_solutions | 40 | 40 | 40 | 40 40 | 40 |

Number of

generations

(initial)\*

Best Fitness value

1500 1500 1500 1500 1500 1500

at end:26.635 26.84026.54023.0524 17.977 19.817 Corresponding

best combined mean absolute error (mae):

Number of

Generations until Convergence

0.0375440.037257 0.0376790.043379 0.055628 0.050462 1150 1175 1400 625 1500 1100

|  |
| --- |

**Figure 4**. Case 1 Predicted Alpha Against Measured Alpha

|  |
| --- |

**Figure 5**. Case 2 Predicted Alpha Against Measured Alpha

|  |
| --- |

**Figure 6**. Case 3 Predicted Alpha Against Measured Alpha

|  |
| --- |

**Figure 7**. Case 4 Predicted Alpha Against Measured Alpha

|  |
| --- |

**Figure 8**. Case 5 Predicted Alpha Against Measured Alpha

|  |
| --- |

**Figure 9**. Case 6 Predicted Alpha Against Measured Alpha

To compare the cases against each other, there are two main values of interest: 1) epochs to convergence and 2) fitness/mean absolute error (mae). From a birds-eye view of **Table 7,** it is clear that all of the manipulated parameters of the model have trade-offs. They all affect convergence and mae differently.

Examining the relationship between epochs and convergence first, it appears that the allowed ranges for the weights directly affected how quick the model took to converge. For example, when we increase the weights min and max boundary values to -2.9 to 2.9 in Cases 3 and 5, they both result in the most number of generations at 1400 and 1500 to converge, respectively. On the other hand, in Case 4, which had the smallest range for the weights of -0.4 to 0.4, the model converged the quickest needing only 625 epochs. However, this lines up with the logic that when you increase the allowed range of the weights, you are increasing the search area in the solution space for the model, which causes it to take more epochs.

On the note of fitness/mae, it does not appear that the allowed ranges for the weights has any reasonable impact on it above a certain threshold. A clear example is between Case 1 and Case 3 who perform similarly yet only differ in their weight bounds, which are -0.9 to 0.9 and -2.9 to 2.9 respectively. Case 4, using the smallest weight bounds of -0.4 to 0.4, does perform worse by 0.6% in terms of mae, yet we hypothesize these weight bounds are most likely below the required threshold for the model to be successful. In slight similarity, it appears that the activation function, whether its relu or elu, has little to no impact on the performance of the model. The performance of Case 1 and Case 2 and even Case 5 and Case 6 support this. In contrast, the network layers seemed to have the greatest impact on the model’s performance. Cases 4 and 5, imploring a five layer network, performed dramatically worse than their counterparts with a max mae difference of around 2%. This observation is supported by **Figures 4** through **Figure 9** as well. The predicted *ɑ*, especially in Cases 1-4 tend to hug tightly to the perfect fit line. On the other hand, the predicted *ɑ* of Cases 5-6 diverge slightly more, especially towards the max values. This is most likely a result of the five layer network overcomplicating the model for this scenario.

**4.4. Comparison of Keras Model With Backpropagation Versus Genetic Algorithm**

From our testing with the Keras model using a genetic algorithm, Case 2 led to our best result with an mae of 3.73%. In comparison, our Keras Model with backpropagation from Section 4.2 scored an mae value of 3.39%. In addition, all six test cases with the genetic algorithm required between 625-1500 generations to get near its best and final fitness with Case 2 requiring 1175 epochs. On the other hand, the Keras model with backpropagation only required about 1000 epochs. Beside the number of epochs, from the standpoint of total time to process, backpropagation only took a fraction of the amount of time (about 5 minutes) that the genetic algorithm took (over 2 hours). By these metrics, there is strong evidence to suggest that a neural network with a backpropagation algorithm performs slightly better than its genetic algorithm counterpart. Even if the genetic algorithm did perform slightly better in terms of mae, the considerable processing time to train the neural network detracted from any performance benefit.

In summary, we observed that backpropagation is efficient in terms of computational time and memory due to its gradient-based approach. On the other hand, while genetic algorithms are good at exploring a wide range of solutions because of their evolutionary processes, backpropagation excels at exploiting gradient information for rapid convergence. Of all the parameters we explored, we observed that choosing different activation functions, adjusting min and max boundary values on weights and biases, and the

number and distribution of parameters in neural network layers can all affect model convergence in the genetic algorithm case, but almost always experience tradeoffs in resulting model accuracy. Thus, for this problem, we found backpropagation to be a much more effective model.

10/31/23, 11:22 PM Proj 2 Task 1.1 Mason.ipynb - Colaboratory

'''#Intro to Neural Network Modeling

# Python Neural Network Model of Spray Cooling Test System

>>>>> start CodeP2.1F23

V.P. Carey, ME249, Fall 2023'''

# version 3 print function

from \_\_future\_\_ import print\_function

# import math, numpy and other usefuk packages

import math

import numpy

%matplotlib inline

# importing the required module

import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = [10, 8] # for square canvas

#assembling data array

#store array where rows are data vectors [x01, x02, x03, y3] xydata = []

#WHY ARE THESE VALUES DIVIDED BY THINGS? AND WHAT ARE THOSE THINGS?z

Mason Rodriguez Rand 

Oct 21, 2023

(edited Oct 21, 2023)

g = .04 -> E3 = 0.00034880440657169317

Mason Rodriguez Rand 

Oct 21, 2023

30 epochs + g = /04 => E3 = 0.000348804

xydata = [[20./20.2, 13.0/14.5, 310.8/308.0, 30.99/32.4], [20./20.2, 14.5/14.5, 308.0/308.0, 32.2/32.4]] xydata.append([20./20.2, 15.3/14.5, 306.0/308.0, 31.7/32.4])

xydata.append([20.2/20.2, 13.0/14.5, 310.8/308.0, 30.92/32.4])

xydata.append([20./20.2, 14.5/14.5, 308.0/308.0, 32.4/32.4])

xydata.append([20./20.2, 15.3/14.5, 306.0/308.0, 31.4/32.4])

xydata.append([24./20.2, 13.0/14.5, 310.8/308.0, 35.53/32.4])

xydata.append([36./20.2, 14.5/14.5, 308.0/308.0, 46.4/32.4])

print (xydata)

#set starting values

w01n = 1.23

w02n = 0.40

w03n = 0.70

b1n = -0.15

w12n = 0.72

b2n = -0.12

w23n = 0.7

b3n = 0.01

#start of batch loop

for k in range (0,30):

icount = 0

#initialize error and derivative parameters

E3ti = 0.

dE3da3 = 0.

dE3dw01ti = 0.

dE3dw02ti = 0.

dE3dw03ti = 0.

dE3db1ti = 0.

dE3dw12ti = 0.

dE3db2ti = 0.

dE3dw23ti = 0.

dE3db3ti = 0.

w01 = w01n

w02 = w02n

w03 = w03n

b1 = b1n

w12 = w12n

b2 = b2n

w23 = w23n

b3 = b3n

#doing calcuations for each data point

for i in range(0,8):

#compute activation functions and their derivatives

z1 = w01\*xydata[i][0]+w02\*xydata[i][1]+w03\*xydata[i][2]+b1

sig1 = z1

sigp1 = 1.0

if z1 < 0.0:

sig1 = math.exp(z1) - 1.0

https://colab.research.google.com/drive/1o2wDNeqiHzbJ\_ysrgwjTSKzSFiaoAlHh#printMode=true 1/4

10/31/23, 11:22 PM Proj 2 Task 1.1 Mason.ipynb - Colaboratory

sigp1 = math.exp(z1)

a1 = sig1

z2 = w12\*a1+b2

sig2 = z2

sigp2 = 1.0

if z2 < 0.0:

sig2 = math.exp(z2) - 1.0

sigp2 = math.exp(z2)

a2 = sig2

z3 = w23\*a2+b3

sig3 = z3

sigp3 = 1.0

if z3 < 0.0:

sig3 = math.exp(z3) - 1.0

sigp3 = math.exp(z3)

a3 = sig3

#compute derivatives for backpropagation

#add to sum for batch average calculation

E3ti = E3ti +(a3 - xydata[i][3])\*(a3 - xydata[i][3]) #Squared error dE3da3 = 2.\*(a3 - xydata[i][3])

dE3dw01ti = dE3dw01ti + dE3da3\*sigp3\*w23\*sigp2\*w12\*sigp1\*xydata[i][0] dE3dw02ti = dE3dw02ti + dE3da3\*sigp3\*w23\*sigp2\*w12\*sigp1\*xydata[i][1] dE3dw03ti = dE3dw03ti + dE3da3\*sigp3\*w23\*sigp2\*w12\*sigp1\*xydata[i][2] dE3db1ti = dE3db1ti + dE3da3\*sigp3\*w23\*sigp2\*w12\*sigp1

dE3dw12ti = dE3dw12ti + dE3da3\*sigp3\*w23\*sigp2\*a1

dE3db2ti = dE3db2ti + dE3da3\*sigp3\*w23\*sigp2

dE3dw23ti = dE3dw23ti + dE3da3\*sigp3\*a2

dE3db3ti = dE3db3ti + dE3da3\*sigp3

icount = i + 1

# end calculations for each data point in batch

#compute batch averaged values

E3 = E3ti/icount

dE3dw01 = dE3dw01ti/icount

dE3dw02 = dE3dw02ti/icount

dE3dw03 = dE3dw03ti/icount

dE3db1 = dE3db1ti/icount

dE3dw12 = dE3dw12ti/icount

dE3db2 = dE3db2ti/icount

dE3dw23 = dE3dw23ti/icount

dE3db3 = dE3db3ti/icount

#set gam = learning rate

gam = 0.04

if E3 < 0.07:

gam = 0.009

w01n = w01 + gam\*(-E3)/dE3dw01

w02n = w02 + gam\*(-E3)/dE3dw02

w03n = w03 + gam\*(-E3)/dE3dw03

b1n = b1 + gam\*(-E3)/dE3db1

w12n = w12 + gam\*(-E3)/dE3dw12

b2n = b2 + gam\*(-E3)/dE3db2

w23n = w23 + gam\*(-E3)/dE3dw23

b3n = b3 + gam\*(-E3)/dE3db3

#printing for each iteration

print ('last w01, w02, w03, w12, w23:')

print ('last b1, b2, b3:')

print (w01, w02, w03, w12, w23)

print (b1, b2, b3)

print ('E3 = ', E3, 'icount =', icount)

print ('next ws:', w01n, w02n, w03n, w12n, w23n)

print ('next bs:', b1n, b2n, b3n)

#quit if squared error is below target

if E3 < 0.00035:

break

https://colab.research.google.com/drive/1o2wDNeqiHzbJ\_ysrgwjTSKzSFiaoAlHh#printMode=true 2/4

10/31/23, 11:22 PM Proj 2 Task 1.1 Mason.ipynb - Colaboratory

print ('last w01, w02, w03, w12, w23:')

print ('last b1, b2, b3:')

print (w01, w02, w03, w12, w23)

print (b1, b2, b3)

#decomment print statements below if you want to print neuron outputs

#print ('z1 =', z1)

#print ('a1 =', a1)

#print ('z2 =', z2)

#print ('a2 =', a2)

#print ('z3 =', z3)

#print ('a3 =', a3)

#print comparison of data and trained network predictions

# restore raw data values

xydatar = [[20., 13.0, 310.8, 30.97], [20., 14.5, 308.0, 32.3]]

xydatar.append([20., 15.3, 306.0, 31.5])

xydatar.append([20.2, 13.0, 310.8, 30.91])

xydatar.append([20., 14.5, 308.0, 32.5])

xydatar.append([20., 15.3, 306.0, 31.4])

xydatar.append([24., 13.0, 310.8, 35.59])

xydatar.append([36., 14.5, 308.0, 46.4])

print ('Tdbin, Twbin, qdot, Tdbout, ypredicted:')

for i in range(0,8):

z1 = w01\*xydata[i][0]+w02\*xydata[i][1]+w03\*xydata[i][2]+b1

sig1 = z1

sigp1 = 1.0

if z1 < 0.0:

sig1 = math.exp(z1) - 1.0

sigp1 = math.exp(z1)

a1 = sig1

z2 = w12\*a1+b2

sig2 = z2

sigp2 = 1.0

if z2 < 0.0:

sig2 = math.exp(z2) - 1.0

sigp2 = math.exp(z2)

a2 = sig2

z3 = w23\*a2+b3

sig3 = z3

sigp3 = 1.0

if z3 < 0.0:

sig3 = math.exp(z3) - 1.0

sigp3 = math.exp(z3)

a3 = sig3

print (xydatar[i][0], xydatar[i][1], xydatar[i][2], xydatar[i][3], a3\*32.4)

account\_circle [[0.9900990099009901, 0.896551724137931, 1.009090909090909, 0.9564814814814815], [0.99009900 last w01, w02, w03, w12, w23:

last b1, b2, b3:

1.23 0.4 0.7 0.72 0.7

-0.15 -0.12 0.01

E3 = 0.0022821697867831175 icount = 8

next ws: 1.2296083538584661 0.3995281787508194 0.6995380344292121 0.7198609859424181 0.69985 next bs: -0.15046253336682955 -0.12033302402411726 0.009766883183117912

last w01, w02, w03, w12, w23:

last b1, b2, b3:

1.2296083538584661 0.3995281787508194 0.6995380344292121 0.7198609859424181 0.69985474158572 -0.15046253336682955 -0.12033302402411726 0.009766883183117912

E3 = 0.0021212710411942196 icount = 8

next ws: 1.2292290463221385 0.3990704851105931 0.6990898009539908 0.7197261809259404 0.69971 next bs: -0.15091130805468592 -0.12065607941338355 0.009540791337145077

last w01, w02, w03, w12, w23:

last b1, b2, b3:

1.2292290463221385 0.3990704851105931 0.6990898009539908 0.7197261809259404 0.69971385578469 -0.15091130805468592 -0.12065607941338355 0.009540791337145077

E3 = 0.0019717491761602176 icount = 8

next ws: 1.2288615401585181 0.39862627796120775 0.69865467231164 0.7195954022590603 0.699577 next bs: -0.15134695213802693 -0.12096962386572957 0.009321399939434148

last w01, w02, w03, w12, w23:

last b1, b2, b3:

1.2288615401585181 0.39862627796120775 0.69865467231164 0.7195954022590603 0.699577154956064 -0.15134695213802693 -0.12096962386572957 0.009321399939434148

E3 = 0.0018328006637283304 icount = 8

next ws: 1.2285053086523463 0.39819492612585045 0.6982320305729344 0.7194684700369329 0.6994

https://colab.research.google.com/drive/1o2wDNeqiHzbJ\_ysrgwjTSKzSFiaoAlHh#printMode=true 3/4

10/31/23, 11:22 PM Proj 2 Task 1.1 Mason.ipynb - Colaboratory

next bs: -0.1517700843073546 -0.12127410782932567 0.009108389914451839

last w01, w02, w03, w12, w23:

last b1, b2, b3:

1.2285053086523463 0.39819492612585045 0.6982320305729344 0.7194684700369329 0.6994444541279 -0.1517700843073546 -0.12127410782932567 0.009108389914451839

E3 = 0.0017036788052525059 icount = 8

next ws: 1.2281598342207123 0.3977758064041795 0.697821265171252 0.7193452066536696 0.699315 next bs: -0.15218131583787564 -0.12156997594942057 0.008901446598698196

last w01, w02, w03, w12, w23:

last b1, b2, b3:

1.2281598342207123 0.3977758064041795 0.697821265171252 0.7193452066536696 0.699315570509440 -0.15218131583787564 -0.12156997594942057 0.008901446598698196

E3 = 0.001583689726759981 icount = 8

next ws: 1.2278246068847685 0.3973683013666934 0.6974217706850763 0.719225436252248 0.699190 next bs: -0.1525812528040283 -0.12185766868898608 0.008700258586397516

last w01, w02, w03, w12, w23:

last b1, b2, b3:

1.2278246068847685 0.3973683013666934 0.6974217706850763 0.719225436252248 0.699190322936956 -0.1525812528040283 -0.12185766868898608 0.008700258586397516

E3 = 0.0014721886589921227 icount = 8

next ws: 1.22749912255941 0.396971796844089 0.697032944308113 0.7191089840963104 0.699068531 next bs: -0.15297049860542425 -0.12213762417030444 0.00850451642300656

last w01, w02, w03, w12, w23:

last b1, b2, b3:

1.22749912255941 0.396971796844089 0.697032944308113 0.7191089840963104 0.6990685312388447 -0.15297049860542425 -0.12213762417030444 0.00850451642300656

E3 = 0.0013685764825472776 icount = 8

next ws: 1.2271828811111118 0.3965856790282886 0.6966541829216193 0.7189956758434574 0.69895 next bs: -0.15334965688933663 -0.12241028029866037 0.008313911103823506

Best： E3 = 0.00034880440657169317 icount = 8 next ws: 1.2220838452664218 0.39005206614010896 0.6901981761985747 0.7171249658666199 0.6969947347378725 next bs: -0.15980783887878994 -0.127047868574918 0.005076835535459802 last w01, w02, w03, w12, w23: last b1, b2, b3: 1.2223797961460618 0.3904727181655663 0.6906205232675598 0.7172385256446994 0.6971129488550933 -0.15938600842791223 -0.12674531552425844 0.005287749184790198

https://colab.research.google.com/drive/1o2wDNeqiHzbJ\_ysrgwjTSKzSFiaoAlHh#printMode=true 4/4

10/31/23, 11:23 PM Proj 2 Task 1.2 Chris.ipynb - Colaboratory

'''>>>>> start CodeP2.2F23-updated 

V.P. Carey ME249, Fall 2023

Intro to Neural Network Modeling

Keras model for comparison with first principles model'''

#import useful packages

import keras

import pandas as pd

from keras.models import Sequential

import numpy as np

import keras.backend as kb

import tensorflow as tf

#the follwoing 2 lines are only needed for Mac OS machines

import os

os.environ['KMP\_DUPLICATE\_LIB\_OK']='True'

#raw data in dictionary form x01, x02, x03, y3

my\_dict = {

'x01' : [20., 20., 20., 20.2, 20., 20.2, 24.0, 36.],

'x02' : [13., 14.5, 15.3, 13., 14.5, 15.3, 13., 14.5],

'x03' : [310.8, 308.0, 306.0, 310.8, 308.0, 306.0, 310.8, 308.0], 'y3' : [30.99, 32.2, 31.7, 30.92, 32.4, 31.4, 35.53, 46.4]

}

#normalized inputs in array

xdata = []

xdata = [[20./20.2, 13.0/14.5, 310.8/308.0], [20./20.2, 14.5/14.5, 308.0/308.0]] xdata.append([20./20.2, 15.3/14.5, 306.0/308.0])

xdata.append([20.2/20.2, 13.0/14.5, 310.8/308.0])

xdata.append([20./20.2, 14.5/14.5, 308.0/308.0])

xdata.append([20.2/20.2, 15.3/14.5, 306.0/308.0])

xdata.append([24./20.2, 13.0/14.5, 310.8/308.0])

xdata.append([36./20.2, 14.5/14.5, 308.0/308.0])

#data frame

df = pd.DataFrame(my\_dict)

#devide by the median to normalize

df.x01= df.x01/20.2

df.x02= df.x02/14.5

df.x03= df.x03/308.0

#normalize output array

df.y3= df.y3/32.401

df.head

print (df.x01, df.x02, df.x03, df.y3)

xarray= np.array(xdata)

print (xdata)

print (xarray)

output 0 0.990099

1 0.990099

2 0.990099

3 1.000000

4 0.990099

5 1.000000

6 1.188119

7 1.782178

Name: x01, dtype: float64 0 0.896552

1 1.000000

2 1.055172

3 0.896552

4 1.000000

5 1.055172

6 0.896552

7 1.000000

Name: x02, dtype: float64 0 1.009091

1 1.000000

2 0.993506

3 1.009091

4 1.000000

5 0.993506

6 1.009091

7 1.000000

Name: x03, dtype: float64 0 0.956452

1 0.993796

2 0.978365

3 0.954292

https://colab.research.google.com/drive/1HlCKAkSflXPyoWr-kpqiatxoaLiLiqF5#printMode=true 1/6

10/31/23, 11:23 PM Proj 2 Task 1.2 Chris.ipynb - Colaboratory

4 0.999969

5 0.969106

6 1.096571

7 1.432055

Name: y3, dtype: float64

[[0.9900990099009901, 0.896551724137931, 1.009090909090909], [0.9900990099009901, 1.0, 1.0], [ [[0.99009901 0.89655172 1.00909091]

[0.99009901 1. 1. ]

[0.99009901 1.05517241 0.99350649]

[1. 0.89655172 1.00909091]

[0.99009901 1. 1. ]

[1. 1.05517241 0.99350649]

[1.18811881 0.89655172 1.00909091]

[1.78217822 1. 1. ]]

# define model

#As seen below, we have created three dense layers each with just one neuron.

#A dense layer is a layer in neural network that’s fully connected.

#In other words, all the neurons in one layer are connected to all other neurons in the next layer. #In the first layer, we need to provide the input shape, which is 3 in this case.

#The activation function we have chosen is ReLU, which stands for rectified linear unit.

from keras import backend as K

#initialize weights with values between -0.2 and 1.2

initializer = keras.initializers.RandomUniform(minval= -0.2, maxval=1.2)

# define three layer model with one neuron in each layer

model = keras.Sequential([

keras.layers.Dense(1, activation=K.elu, input\_shape=[3], kernel\_initializer=initializer, name="dense\_one"), keras.layers.Dense(1, activation=K.elu, kernel\_initializer=initializer, name="dense\_two"), keras.layers.Dense(1, activation=K.elu, kernel\_initializer=initializer, name="dense\_three") ])

model.summary()

#set starting values to those used in first principles model

w01n = 1.48 #1.23

w02n = 0.48 #0.40

w03n = 0.84 #0.70

b1n = -0.1818 #-0.15

w12n = 0.84 #0.72

b2n = -0.18 #-0.12

w23n = 0.84 #0.7

b3n = 0.012 #0.01

weights0 = [[ w01n], [w02n], [ w03n]]

w0array= np.array(weights0)

print(np.shape(w0array))

bias0 = [b1n]

bias0array= np.array(bias0)

L0=[]

L0.append(w0array)

L0.append(bias0array)

model.layers[0].set\_weights(L0)

weights1 = [[ w12n]]

w1array= np.array(weights1)

print(np.shape(w1array))

bias1 = [b2n]

bias1array= np.array(bias1)

L1=[]

L1.append(w1array)

L1.append(bias1array)

model.layers[1].set\_weights(L1)

weights2 = [[ w23n]]

w2array= np.array(weights2)

print(np.shape(w2array))

bias2 = [b3n]

bias2array= np.array(bias2)

L2=[]

L2.append(w2array)

L2.append(bias2array)

model.layers[2].set\_weights(L2)

https://colab.research.google.com/drive/1HlCKAkSflXPyoWr-kpqiatxoaLiLiqF5#printMode=true 2/6

10/31/23, 11:23 PM Proj 2 Task 1.2 Chris.ipynb - Colaboratory

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense\_one (Dense) (None, 1) 4

dense\_two (Dense) (None, 1) 2

dense\_three (Dense) (None, 1) 2

=================================================================

Total params: 8 (32.00 Byte)

Trainable params: 8 (32.00 Byte)

Non-trainable params: 0 (0.00 Byte)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(3, 1)

(1, 1)

(1, 1)

/usr/local/lib/python3.10/dist-packages/keras/src/initializers/initializers.py:120: UserWarnin warnings.warn(

#We’re using RMSprop as our optimizer here. RMSprop stands for Root Mean Square Propagation. #It’s one of the most popular gradient descent optimization algorithms for deep learning networks. #RMSprop is an optimizer that’s reliable and fast.

#We’re compiling the mode using the model.compile function. The loss function used here #is mean absolute error. After the compilation of the model, we’ll use the fit method with 100 epochs.

#Running model.fit successive times extends the calculation to addtional epochs.

rms = keras.optimizers.RMSprop(0.0008) # USER CODE Original 0.0035 model.compile(loss='mean\_absolute\_error',optimizer=rms)

#After the compilation of the model, we’ll use the fit method with 500 epochs. #I started with epochs value of 100 and then tested the model after training.

Christopher Simotas 

Oct 16, 2023

(edited Oct 16, 2023)

First round, ran Cell 4 multiple times and 0.01249

#The prediction was not that good. Then I modified the number of epochs to 200 and tested the model again. #Accuracy had improved slightly, but figured I’d give it one more try. Finally, at 500 epochs #I found acceptable prediction accuracy. 

Christopher Simotas Oct 16, 2023

#The fit method takes three parameters; namely, x, y, and number of epochs. #During model training, if all the batches of data are seen by the model once, #we say that one epoch has been completed.

# Add an early stopping callback

es = tf.keras.callbacks.EarlyStopping(

monitor='loss',

mode='min',

patience = 80,

restore\_best\_weights = True,

verbose=1)

# Add a checkpoint where loss is minimum, and save that model mc = tf.keras.callbacks.ModelCheckpoint('best\_model.SB', monitor='loss', mode='min', verbose=1, save\_best\_only=True)

Raised gamma from 0.0035 to 0.01, didn't perform better

Christopher Simotas 

Oct 16, 2023

Lowered to 0.002, performed to 0.012528

Christopher Simotas 

Oct 16, 2023

Lowered to 0.0015, performed to 0.0124619

Christopher Simotas 

Oct 16, 2023

Lowered to 0.0008, performed to 0.012448

historyData = model.fit(xarray,df.y3,epochs=400,callbacks=[es]) # USER CODE UPDATED EPOCH FROM 800 loss\_hist = historyData.history['loss'] 

Christopher Simotas Oct 16, 2023

#The above line will return a dictionary, access it's info like this: best\_epoch = np.argmin(historyData.history['loss']) + 1 print ('best epoch = ', best\_epoch)

print('smallest loss =', np.min(loss\_hist))

Epoch 1/400

1/1 [==============================] - 0s 9ms/step - loss: 0.0141 Epoch 2/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0148 Epoch 3/400

1/1 [==============================] - 0s 8ms/step - loss: 0.0143 Epoch 4/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0143 Epoch 5/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0145 Epoch 6/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0141

Same performed 0.0124428

Christopher Simotas 

Oct 22, 2023

(edited Oct 22, 2023)

With higher guesses, first run converges to loss of 0.01295

Christopher Simotas 

Oct 22, 2023

(edited Oct 22, 2023)

4 runs: got to 0.012486

https://colab.research.google.com/drive/1HlCKAkSflXPyoWr-kpqiatxoaLiLiqF5#printMode=true 3/6

10/31/23, 11:23 PM Proj 2 Task 1.2 Chris.ipynb - Colaboratory

Epoch 7/400

1/1 [==============================] - 0s 8ms/step - loss: 0.0140 Epoch 8/400

1/1 [==============================] - 0s 8ms/step - loss: 0.0140 Epoch 9/400

1/1 [==============================] - 0s 8ms/step - loss: 0.0139 Epoch 10/400

1/1 [==============================] - 0s 9ms/step - loss: 0.0139 Epoch 11/400

1/1 [==============================] - 0s 6ms/step - loss: 0.0147 Epoch 12/400

1/1 [==============================] - 0s 6ms/step - loss: 0.0141 Epoch 13/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0142 Epoch 14/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0143 Epoch 15/400

1/1 [==============================] - 0s 8ms/step - loss: 0.0139 Epoch 16/400

1/1 [==============================] - 0s 12ms/step - loss: 0.0139 Epoch 17/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0138 Epoch 18/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0138 Epoch 19/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0137 Epoch 20/400

1/1 [==============================] - 0s 8ms/step - loss: 0.0146 Epoch 21/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0139 Epoch 22/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0141 Epoch 23/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0141 Epoch 24/400

1/1 [==============================] - 0s 8ms/step - loss: 0.0138 Epoch 25/400

1/1 [==============================] - 0s 9ms/step - loss: 0.0137 Epoch 26/400

1/1 [==============================] - 0s 10ms/step - loss: 0.0137 Epoch 27/400

1/1 [==============================] - 0s 8ms/step - loss: 0.0136 Epoch 28/400

1/1 [==============================] - 0s 7ms/step - loss: 0.0136 Epoch 29/400

1/1 [ ] 0 6 / t l 0 0137

from \_\_future\_\_ import print\_function

#For results of training network:

#keras.layer.get\_weights() function retrieves weight values first\_layer\_weights = model.layers[0].get\_weights()[0]

w01 = first\_layer\_weights[0][0]

w02 = first\_layer\_weights[1][0]

w03 = first\_layer\_weights[2][0]

first\_layer\_bias = model.layers[0].get\_weights()[1]

b1 = first\_layer\_bias

second\_layer\_weights = model.layers[1].get\_weights()[0]

w12 = second\_layer\_weights[0][0]

second\_layer\_bias = model.layers[1].get\_weights()[1]

b2 = second\_layer\_bias

third\_layer\_weights = model.layers[2].get\_weights()[0]

w23 = third\_layer\_weights[0][0]

third\_layer\_bias = model.layers[2].get\_weights()[1]

b3 = third\_layer\_bias

#print weights and biases

print (first\_layer\_weights)

print ('w01 = ', w01, 'w02 = ', w02, 'w03 = ', w03)

print (first\_layer\_bias)

print ('b1 = ', b1)

print (second\_layer\_weights)

print ('w12 = ', w12)

print (second\_layer\_bias)

print ('b2 = ', b2)

print (third\_layer\_weights)

print ('w23 = ', w23)

print (third\_layer\_bias)

print ('b3 = ', b3)

#use model.predict() function to print model predictions for data conditions

https://colab.research.google.com/drive/1HlCKAkSflXPyoWr-kpqiatxoaLiLiqF5#printMode=true 4/6

10/31/23, 11:23 PM Proj 2 Task 1.2 Chris.ipynb - Colaboratory

xarray= np.array(xdata)

print ('x01/20.2, x02/14.5, x03/308.0, y3/32.4, a3:')

test = []

for i in range(0,8):

test = [[xarray[i][0], xarray[i][1], xarray[i][2]]]

testarray = np.array(test)

a3 = model.predict(testarray)

print (xarray[i][0], xarray[i][1], xarray[i][2], df.y3[i], a3)

print(' ')

print ('x01, x02, x03, y3, a3\*32.4:')

for i in range(0,8):

test = [[xarray[i][0], xarray[i][1], xarray[i][2]]]

testarray = np.array(test)

a3 = model.predict(testarray)

print (xarray[i][0]\*20.2, xarray[i][1]\*14.5, xarray[i][2]\*308.0, df.y3[i]\*32.4, a3\*32.4)

[[1.2498066 ]

[0.385513 ]

[0.85563534]]

w01 = 1.2498066 w02 = 0.385513 w03 = 0.85563534

[-0.1768089]

b1 = [-0.1768089]

[[0.6940814]]

w12 = 0.6940814

[-0.17479666]

b2 = [-0.17479666]

[[0.67085975]]

w23 = 0.67085975

[0.0174838]

b3 = [0.0174838]

x01/20.2, x02/14.5, x03/308.0, y3/32.4, a3:

1/1 [==============================] - 0s 41ms/step

0.9900990099009901 0.896551724137931 1.009090909090909 0.9564519613592172 [[0.957049]] 1/1 [==============================] - 0s 32ms/step

0.9900990099009901 1.0 1.0 0.9937964877627233 [[0.9719968]]

1/1 [==============================] - 0s 49ms/step

0.9900990099009901 1.0551724137931036 0.9935064935064936 0.9783648652819357 [[0.9793136]] 1/1 [==============================] - 0s 35ms/step

1.0 0.896551724137931 1.009090909090909 0.954291534211907 [[0.9628108]] 1/1 [==============================] - 0s 133ms/step

0.9900990099009901 1.0 1.0 0.9999691367550383 [[0.9719968]]

1/1 [==============================] - 0s 85ms/step

1.0 1.0551724137931036 0.9935064935064936 0.9691058917934631 [[0.9850754]] 1/1 [==============================] - 0s 47ms/step

1.188118811881188 0.896551724137931 1.009090909090909 1.096571093484769 [[1.0722865]] 1/1 [==============================] - 0s 56ms/step

1.7821782178217822 1.0 1.0 1.4320545662170918 [[1.4329467]]

x01, x02, x03, y3, a3\*32.4:

1/1 [==============================] - 0s 56ms/step

20.0 13.0 310.8 30.989043548038634 [[31.008389]]

1/1 [==============================] - 0s 133ms/step

20.0 14.5 308.0 32.19900620351223 [[31.492697]]

1/1 [==============================] - 0s 64ms/step

20.0 15.3 306.0 31.699021635134713 [[31.729763]]

1/1 [==============================] - 0s 84ms/step

20.2 13.0 310.8 30.919045708465788 [[31.195072]]

1/1 [==============================] - 0s 19ms/step

20.0 14.5 308.0 32.39900003086324 [[31.492697]]

1/1 [==============================] - 0s 23ms/step

20.2 15.3 306.0 31.3990308941082 [[31.916445]]

1/1 [==============================] - 0s 19ms/step

23.999999999999996 13.0 310.8 35.52890342890652 [[34.742085]]

1/1 [==============================] - 0s 20ms/step

36.0 14.5 308.0 46.398567945433776 [[46.427475]]

https://colab.research.google.com/drive/1HlCKAkSflXPyoWr-kpqiatxoaLiLiqF5#printMode=true 5/6

10/31/23, 11:24 PM Project 2 Task 1.2C - Colaboratory

import matplotlib.pyplot as plt

import numpy as np

# Data

x1 = [31.06, 31.61, 31.88, 31.26, 31.61, 31.88, 34.97, 47.25] # First-Principles Predicted Y3

x2 = [31.13, 31.61, 31.85, 31.32, 31.61, 32.03, 34.87, 46.58] # Keras Predicted Y3

y = [30.97, 32.3, 31.5, 30.91, 32.5, 31.4, 35.59, 46.4] # Real Y3 Values

# Creating log-log plot

plt.figure(figsize=(8, 6))

plt.loglog(x1, y, 'ro', label='First-Principles Predicted Y3')

plt.loglog(x2, y, 'bo', label='Keras Predicted Y3')

plt.loglog(y, y, color = 'black', label = 'For Perspective: y = x')

# Adding labels and title

plt.xlabel('Predicted y3 Values')

plt.ylabel('Real y3 Values')

plt.title('Log-Log Plot of Predicted vs Real y3 Values 1 - Chris\'s model')

plt.legend()

# Displaying the plot

plt.grid(True)

plt.show()



Comments: Based on your results in the tables, how well do the results of the two models agree? For the two models given the same starting values for the weights and bias values, do they yield exactly the same answer? Does the answer to this question make sense? Briefly explain your answer in your summary report.

Start coding or generate with AI.

https://colab.research.google.com/drive/1ALeS7n8qhxz\_PreeDX8ZR-5lKkD\_RKXV#scrollTo=cmLsDsxpzh3e&printMode=true 1/2

10/31/23, 11:24 PM Project 2 Task 1.2C - Colaboratory

https://colab.research.google.com/drive/1ALeS7n8qhxz\_PreeDX8ZR-5lKkD\_RKXV#scrollTo=cmLsDsxpzh3e&printMode=true 2/2

10/31/23, 11:24 PM Project 2 Task 2.1.ipynb - Colaboratory

ME 249 Project 2 Chris Simotas and Mason Rodriguez Rand Task 2.1

'''>>>>> start CodeP2.3F23

V.P. Carey ME249, Fall 2023

Intro to Neural Network Modeling

Data arrays for hybrid solar/fossil-fuel gas turbine power system'''

### Set White Space for Output Cell

from IPython.display import HTML, display

import numpy

#create input data array, normalizing input temp

#T1(K), gamma, , qsol(kW):

xdata = []

xdata = [[ 318.0 , 0.0 , 500.0 ], [ 318.0 , 0.0 , 1000.0 ]] xdata.append([ 318.0 , 0.0 , 1500.0 ])

xdata.append([ 318.0 , 0.0 , 2000.0 ])

xdata.append([ 318.0 , 0.0 , 2500.0 ])

xdata.append([ 318.0 , 0.25 , 500.0 ])

xdata.append([ 318.0 , 0.25 , 1000.0 ])

xdata.append([ 318.0 , 0.25 , 1500.0 ])

xdata.append([ 318.0 , 0.25 , 2000.0 ])

xdata.append([ 318.0 , 0.25 , 2500.0 ])

xdata.append([ 318.0 , 0.5 , 500.0 ])

xdata.append([ 318.0 , 0.5 , 1000.0 ])

xdata.append([ 318.0 , 0.5 , 1500.0 ])

xdata.append([ 318.0 , 0.5 , 2000.0 ])

xdata.append([ 318.0 , 0.5 , 2500.0 ])

xdata.append([ 303.0 , 0.0 , 500.0 ])

xdata.append([ 303.0 , 0.0 , 1000.0 ])

xdata.append([ 303.0 , 0.0 , 1500.0 ])

xdata.append([ 303.0 , 0.0 , 2000.0 ])

xdata.append([ 303.0 , 0.0 , 2500.0 ])

xdata.append([ 303.0 , 0.25 , 500.0 ])

xdata.append([ 303.0 , 0.25 , 1000.0 ])

xdata.append([ 303.0 , 0.25 , 1500.0 ])

xdata.append([ 303.0 , 0.25 , 2000.0 ])

xdata.append([ 303.0 , 0.25 , 2500.0 ])

xdata.append([ 303.0 , 0.5 , 500.0 ])

xdata.append([ 303.0 , 0.5 , 1000.0 ])

xdata.append([ 303.0 , 0.5 , 1500.0 ])

xdata.append([ 303.0 , 0.5 , 2000.0 ])

xdata.append([ 303.0 , 0.5 , 2500.0 ])

xdata.append([ 288.0 , 0.0 , 500.0 ])

xdata.append([ 288.0 , 0.0 , 1000.0 ])

xdata.append([ 288.0 , 0.0 , 1500.0 ])

xdata.append([ 288.0 , 0.0 , 2000.0 ])

xdata.append([ 288.0 , 0.0 , 2500.0 ])

xdata.append([ 288.0 , 0.25 , 500.0 ])

xdata.append([ 288.0 , 0.25 , 1000.0 ])

xdata.append([ 288.0 , 0.25 , 1500.0 ])

xdata.append([ 288.0 , 0.25 , 2000.0 ])

xdata.append([ 288.0 , 0.25 , 2500.0 ])

xdata.append([ 288.0 , 0.5 , 500.0 ])

xdata.append([ 288.0 , 0.5 , 1000.0 ])

xdata.append([ 288.0 , 0.5 , 1500.0 ])

xdata.append([ 288.0 , 0.5 , 2000.0 ])

xdata.append([ 288.0 , 0.5 , 2500.0 ])

xdata.append([ 268.0 , 0.0 , 500.0 ])

xdata.append([ 268.0 , 0.0 , 1000.0 ])

xdata.append([ 268.0 , 0.0 , 1500.0 ])

xdata.append([ 268.0 , 0.0 , 2000.0 ])

xdata.append([ 268.0 , 0.0 , 2500.0 ])

xdata.append([ 268.0 , 0.25 , 500.0 ])

xdata.append([ 268.0 , 0.25 , 1000.0 ])

xdata.append([ 268.0 , 0.25 , 1500.0 ])

xdata.append([ 268.0 , 0.25 , 2000.0 ])

xdata.append([ 268.0 , 0.25 , 2500.0 ])

xdata.append([ 268.0 , 0.5 , 500.0 ])

d t d([ 268 0 0 5 1000 0 ])

https://colab.research.google.com/drive/14gnI9XlUDCsqwuKKXNUZlijHUW2r3bTH#printMode=true 1/3

10/31/23, 11:24 PM Project 2 Task 2.1.ipynb - Colaboratory

xdata.append([ 268.0 , 0.5 , 1000.0 ])

xdata.append([ 268.0 , 0.5 , 1500.0 ])

xdata.append([ 268.0 , 0.5 , 2000.0 ])

xdata.append([ 268.0 , 0.5 , 2500.0 ])

ydata = [[ 35.1316 , 0.3808 ],[ 40.3764 , 0.38686 ]]

ydata.append([ 47.4620 , 0.3930 ])

ydata.append([ 57.5639 , 0.39949 ])

ydata.append([ 73.1286 , 0.40612 ])

ydata.append([ 49.1110 , 0.4023 ])

ydata.append([ 56.4428 , 0.40605 ])

ydata.append([ 66.3479 , 0.4098 ])

ydata.append([ 80.4695 , 0.413 ])

ydata.append([ 102.2276 , 0.4175 ])

ydata.append([ 63.0904 , 0.41540 ])

ydata.append([ 72.5092 , 0.4175 ])

ydata.append([ 85.2338, 0.4197 ])

ydata.append([ 103.3750 , 0.42192 ])

ydata.append([ 131.3266 , 0.4242 ])

ydata.append([ 34.273 , 0.3952 ])

ydata.append([ 38.99026 , 0.4012 ])

ydata.append([ 45.2133, 0.4073 ])

ydata.append([ 53.8000 , 0.4136 ])

ydata.append([ 66.4130 , 0.4201 ])

ydata.append([ 47.922 , 0.4178 ])

ydata.append([ 54.518 , 0.4215 ])

ydata.append([ 63.220 , 0.4252 ])

ydata.append([ 75.226 , 0.4290 ])

ydata.append([ 92.862 , 0.4329 ])

ydata.append([ 61.572 , 0.4315 ])

ydata.append([ 70.0468 , 0.43373 ])

ydata.append([ 81.226 , 0.43597 ])

ydata.append([ 96.653 , 0.4382 ])

ydata.append([ 119.3124 , 0.44045 ])

ydata.append([ 33.4521 , 0.40913 ])

ydata.append([ 37.6911, 0.4150 ])

ydata.append([ 43.1602 , 0.4209 ])

ydata.append([ 50.4858 , 0.4271 ])

ydata.append([ 60.8067 , 0.4334 ])

ydata.append([ 46.7865 , 0.4328 ])

ydata.append([ 52.7151 , 0.43646 ])

ydata.append([ 60.36425 , 0.44016 ])

ydata.append([ 70.6099 , 0.443926 ])

ydata.append([ 85.0447 , 0.4477 ])

ydata.append([ 60.1208 , 0.44721 ])

ydata.append([ 67.7391 , 0.44940 ])

ydata.append([ 77.56830 , 0.4516 ])

ydata.append([ 90.73410 , 0.4538 ])

ydata.append([ 109.2828 , 0.4560 ])

ydata.append([ 32.4123 , 0.42694 ])

ydata.append([ 36.0807 , 0.4325 ])

ydata.append([ 40.6854 , 0.4383 ])

ydata.append([ 46.6374 , 0.4442 ])

ydata.append([ 54.6293 , 0.4503 ])

ydata.append([ 45.3472 , 0.4519 ])

ydata.append([ 50.4796 , 0.4555 ])

ydata.append([ 56.9219 , 0.4591 ])

ydata.append([ 65.2492 , 0.4628 ])

ydata.append([ 76.4304 , 0.4665 ])

ydata.append([ 58.2822 , 0.4672 ])

ydata.append([ 64.8785 , 0.4693 ])

ydata.append([ 73.1584 , 0.4715 ])

ydata.append([ 83.8610 , 0.4738 ])

ydata.append([ 98.2316 , 0.4760 ])

# Calculate the median for each column

median\_values\_x = numpy.median(numpy.array(xdata), axis=0) median\_values\_y = numpy.median(numpy.array(ydata), axis=0)

print("Median values for each column:", median\_values\_x, median\_values\_y)

xdata = (xdata/median\_values\_x).tolist()

ydata = (ydata/median values y).tolist()

https://colab.research.google.com/drive/14gnI9XlUDCsqwuKKXNUZlijHUW2r3bTH#printMode=true 2/3

10/31/23, 11:24 PM Project 2 Task 2.1.ipynb - Colaboratory

ydata (ydata/ ed a \_ a ues\_y).to st()

print("Normalized xdata: \n", xdata)

print("\n")

print("Normalized ydata: \n",ydata)

account\_circle Median values for each column: [2.955e+02 2.500e-01 1.500e+03] [61.18935 0.432 ]

Normalized xdata:

[[1.0761421319796953, 0.0, 0.3333333333333333], [1.0761421319796953, 0.0, 0.6666666666666666], [1.0761421319796953, 0.0, 1.0], [1.07614

Normalized ydata:

[[0.5741456642373223, 0.8814814814814815], [0.6598599266048748, 0.8955092592592592], [0.7756578554928267, 0.9097222222222223], [0.94075

https://colab.research.google.com/drive/14gnI9XlUDCsqwuKKXNUZlijHUW2r3bTH#printMode=true 3/3

10/31/23, 11:24 PM Project 2 Task 2.2.ipynb - Colaboratory

ME 249 Project 2 Chris Simotas and Mason Rodriguez Rand

Task 2.2

'''>>>>> start CodeP2.4F23

V.P. Carey ME249, Fall 2023

Intro to Neural Network Modeling

Keras model for hybrid solar/fossil-fuel gas turbine power system'''

### Set White Space for Output Cell

from IPython.display import HTML, display

#import useful packages

import keras

import pandas as pd

from keras.models import Sequential

import numpy as np

import keras.backend as kb

import tensorflow as tf

#the follwoing 2 lines are only needed for Mac OS machines

import os

os.environ['KMP\_DUPLICATE\_LIB\_OK']='True'

# Create input data array

'''

xdata = []

xdata = [[ 318.0, 0.0, 500.0], [ 318.0, 0.0, 1000.0]]

xdata.append([ 318.0, 0.0, 1500.0])

xdata.append([ 318.0, 0.0, 2000.0])

xdata.append([ 318.0, 0.0, 2500.0])'''

#convert to:

# meadian values of input variables

'''Tmed = 293.

gamed = 0.25

qsmed = 1250.

#T1(K), gamma, , qsol(kW):

xdata = []

ND = 60

xdata = [[ 318.0/Tmed , 0.0/gamed , 500.0/qsmed ], [ 318.0/Tmed , 0.0/gamed , 1000.0/qsmed ]] xdata.append([ 318.0/Tmed , 0.0/gamed , 1500.0/qsmed ])

xdata.append([ 318.0/Tmed , 0.0/gamed , 2000.0/qsmed ])

xdata.append([ 318.0/Tmed , 0.0/gamed , 2500.0/qsmed ])'''

# add the rest here

### USER CODE ###

#create input data array, normalizing input temp

#T1(K), gamma, , qsol(kW):

xdata = []

xdata = [[ 318.0 , 0.0 , 500.0 ], [ 318.0 , 0.0 , 1000.0 ]]

xdata.append([ 318.0 , 0.0 , 1500.0 ])

xdata.append([ 318.0 , 0.0 , 2000.0 ])

xdata.append([ 318.0 , 0.0 , 2500.0 ])

xdata.append([ 318.0 , 0.25 , 500.0 ])

xdata.append([ 318.0 , 0.25 , 1000.0 ])

xdata.append([ 318.0 , 0.25 , 1500.0 ])

xdata.append([ 318.0 , 0.25 , 2000.0 ])

xdata.append([ 318.0 , 0.25 , 2500.0 ])

xdata.append([ 318.0 , 0.5 , 500.0 ])

xdata.append([ 318.0 , 0.5 , 1000.0 ])

xdata.append([ 318.0 , 0.5 , 1500.0 ])

xdata.append([ 318.0 , 0.5 , 2000.0 ])

xdata.append([ 318.0 , 0.5 , 2500.0 ])

xdata.append([ 303.0 , 0.0 , 500.0 ])

xdata.append([ 303.0 , 0.0 , 1000.0 ])

xdata.append([ 303.0 , 0.0 , 1500.0 ])

xdata.append([ 303.0 , 0.0 , 2000.0 ])

xdata.append([ 303.0 , 0.0 , 2500.0 ])

xdata.append([ 303.0 , 0.25 , 500.0 ])

xdata.append([ 303.0 , 0.25 , 1000.0 ])

xdata.append([ 303.0 , 0.25 , 1500.0 ])

https://colab.research.google.com/drive/1OA\_fd5xWRN34gaGT7IwSoOtaj9cgoZ-e#printMode=true 1/12

10/31/23, 11:24 PM Project 2 Task 2.2.ipynb - Colaboratory

xdata.append([ 303.0 , 0.25 , 2000.0 ])

xdata.append([ 303.0 , 0.25 , 2500.0 ])

xdata.append([ 303.0 , 0.5 , 500.0 ])

xdata.append([ 303.0 , 0.5 , 1000.0 ])

xdata.append([ 303.0 , 0.5 , 1500.0 ])

xdata.append([ 303.0 , 0.5 , 2000.0 ])

xdata.append([ 303.0 , 0.5 , 2500.0 ])

xdata.append([ 288.0 , 0.0 , 500.0 ])

xdata.append([ 288.0 , 0.0 , 1000.0 ])

xdata.append([ 288.0 , 0.0 , 1500.0 ])

xdata.append([ 288.0 , 0.0 , 2000.0 ])

xdata.append([ 288.0 , 0.0 , 2500.0 ])

xdata.append([ 288.0 , 0.25 , 500.0 ])

xdata.append([ 288.0 , 0.25 , 1000.0 ])

xdata.append([ 288.0 , 0.25 , 1500.0 ])

xdata.append([ 288.0 , 0.25 , 2000.0 ])

xdata.append([ 288.0 , 0.25 , 2500.0 ])

xdata.append([ 288.0 , 0.5 , 500.0 ])

xdata.append([ 288.0 , 0.5 , 1000.0 ])

xdata.append([ 288.0 , 0.5 , 1500.0 ])

xdata.append([ 288.0 , 0.5 , 2000.0 ])

xdata.append([ 288.0 , 0.5 , 2500.0 ])

xdata.append([ 268.0 , 0.0 , 500.0 ])

xdata.append([ 268.0 , 0.0 , 1000.0 ])

xdata.append([ 268.0 , 0.0 , 1500.0 ])

xdata.append([ 268.0 , 0.0 , 2000.0 ])

xdata.append([ 268.0 , 0.0 , 2500.0 ])

xdata.append([ 268.0 , 0.25 , 500.0 ])

xdata.append([ 268.0 , 0.25 , 1000.0 ])

xdata.append([ 268.0 , 0.25 , 1500.0 ])

xdata.append([ 268.0 , 0.25 , 2000.0 ])

xdata.append([ 268.0 , 0.25 , 2500.0 ])

xdata.append([ 268.0 , 0.5 , 500.0 ])

xdata.append([ 268.0 , 0.5 , 1000.0 ])

xdata.append([ 268.0 , 0.5 , 1500.0 ])

xdata.append([ 268.0 , 0.5 , 2000.0 ])

xdata.append([ 268.0 , 0.5 , 2500.0 ])

'''ydata = [[ 35.1316, 0.3808], [ 40.3764, 0.38686]]

ydata.append([ 47.4620, 0.3930])

ydata.append([ 57.5639, 0.39949])

ydata.append([ 73.1286, 0.40612])'''

#convert to:

# meadian values of output variables

'''almed = 60.

efmed = 0.4

# alpha, effsys

ydata = [[ 35.1316/almed , 0.3808/efmed ], [ 40.3764/almed , 0.38686/efmed ]] ydata.append([ 47.4620/almed , 0.3930/efmed ])

ydata.append([ 57.5639/almed , 0.39949/efmed ])

ydata.append([ 73.1286/almed , 0.40612/efmed ])'''

# add the rest here

ydata = [[ 35.1316 , 0.3808 ],[ 40.3764 , 0.38686 ]]

ydata.append([ 47.4620 , 0.3930 ])

ydata.append([ 57.5639 , 0.39949 ])

ydata.append([ 73.1286 , 0.40612 ])

ydata.append([ 49.1110 , 0.4023 ])

ydata.append([ 56.4428 , 0.40605 ])

ydata.append([ 66.3479 , 0.4098 ])

ydata.append([ 80.4695 , 0.413 ])

ydata.append([ 102.2276 , 0.4175 ])

ydata.append([ 63.0904 , 0.41540 ])

ydata.append([ 72.5092 , 0.4175 ])

ydata.append([ 85.2338, 0.4197 ])

ydata.append([ 103.3750 , 0.42192 ])

ydata.append([ 131.3266 , 0.4242 ])

ydata.append([ 34.273 , 0.3952 ])

ydata.append([ 38.99026 , 0.4012 ])

ydata.append([ 45.2133, 0.4073 ])

ydata.append([ 53.8000 , 0.4136 ])

ydata.append([ 66.4130 , 0.4201 ])

ydata.append([ 47.922 , 0.4178 ])

https://colab.research.google.com/drive/1OA\_fd5xWRN34gaGT7IwSoOtaj9cgoZ-e#printMode=true 2/12

10/31/23, 11:24 PM Project 2 Task 2.2.ipynb - Colaboratory

ydata.append([ 54.518 , 0.4215 ])

ydata.append([ 63.220 , 0.4252 ])

ydata.append([ 75.226 , 0.4290 ])

ydata.append([ 92.862 , 0.4329 ])

ydata.append([ 61.572 , 0.4315 ])

ydata.append([ 70.0468 , 0.43373 ])

ydata.append([ 81.226 , 0.43597 ])

ydata.append([ 96.653 , 0.4382 ])

ydata.append([ 119.3124 , 0.44045 ])

ydata.append([ 33.4521 , 0.40913 ])

ydata.append([ 37.6911, 0.4150 ])

ydata.append([ 43.1602 , 0.4209 ])

ydata.append([ 50.4858 , 0.4271 ])

ydata.append([ 60.8067 , 0.4334 ])

ydata.append([ 46.7865 , 0.4328 ])

ydata.append([ 52.7151 , 0.43646 ])

ydata.append([ 60.36425 , 0.44016 ])

ydata.append([ 70.6099 , 0.443926 ])

ydata.append([ 85.0447 , 0.4477 ])

ydata.append([ 60.1208 , 0.44721 ])

ydata.append([ 67.7391 , 0.44940 ])

ydata.append([ 77.56830 , 0.4516 ])

ydata.append([ 90.73410 , 0.4538 ])

ydata.append([ 109.2828 , 0.4560 ])

ydata.append([ 32.4123 , 0.42694 ])

ydata.append([ 36.0807 , 0.4325 ])

ydata.append([ 40.6854 , 0.4383 ])

ydata.append([ 46.6374 , 0.4442 ])

ydata.append([ 54.6293 , 0.4503 ])

ydata.append([ 45.3472 , 0.4519 ])

ydata.append([ 50.4796 , 0.4555 ])

ydata.append([ 56.9219 , 0.4591 ])

ydata.append([ 65.2492 , 0.4628 ])

ydata.append([ 76.4304 , 0.4665 ])

ydata.append([ 58.2822 , 0.4672 ])

ydata.append([ 64.8785 , 0.4693 ])

ydata.append([ 73.1584 , 0.4715 ])

ydata.append([ 83.8610 , 0.4738 ])

ydata.append([ 98.2316 , 0.4760 ])

# Calculate the median for each column

median\_values\_x = np.median(np.array(xdata), axis=0)

Tmed = median\_values\_x[0] # 295.5

gamed = median\_values\_x[1] # 0.25

qsmed = median\_values\_x[2] # 1500.

median\_values\_y = np.median(np.array(ydata), axis=0)

almed = median\_values\_y[0] # 61.1

efmed = median\_values\_y[1] # 0.432

print("Median values for each column:", median\_values\_x, median\_values\_y)

# Reformat and Print Data

xdata = (xdata/median\_values\_x).tolist()

ydata = (ydata/median\_values\_y).tolist()

### USER CODE ###

xarray= np.array(xdata)

print (xdata)

print (xarray)

yarray= np.array(ydata)

print (ydata)

print (yarray)

data\_inputs = np.array(xdata)

data\_outputs = np.array(ydata)

Median values for each column: [2.955e+02 2.500e-01 1.500e+03] [61.18935 0.432 ] [[1.0761421319796953, 0.0, 0.3333333333333333], [1.0761421319796953, 0.0, 0.6666666666666666 [[1.07614213 0. 0.33333333]

[1.07614213 0. 0.66666667]

[1.07614213 0. 1. ]

[1.07614213 0. 1.33333333]

[1.07614213 0. 1.66666667]

[1.07614213 1. 0.33333333]

https://colab.research.google.com/drive/1OA\_fd5xWRN34gaGT7IwSoOtaj9cgoZ-e#printMode=true 3/12

10/31/23, 11:24 PM Project 2 Task 2.2.ipynb - Colaboratory

[1.07614213 1. 0.66666667]

[1.07614213 1. 1. ]

[1.07614213 1. 1.33333333]

[1.07614213 1. 1.66666667]

[1.07614213 2. 0.33333333]

[1.07614213 2. 0.66666667]

[1.07614213 2. 1. ]

[1.07614213 2. 1.33333333]

[1.07614213 2. 1.66666667]

[1.02538071 0. 0.33333333]

[1.02538071 0. 0.66666667]

[1.02538071 0. 1. ]

[1.02538071 0. 1.33333333]

[1.02538071 0. 1.66666667]

[1.02538071 1. 0.33333333]

[1.02538071 1. 0.66666667]

[1.02538071 1. 1. ]

[1.02538071 1. 1.33333333]

[1.02538071 1. 1.66666667]

[1.02538071 2. 0.33333333]

[1.02538071 2. 0.66666667]

[1.02538071 2. 1. ]

[1.02538071 2. 1.33333333]

[1.02538071 2. 1.66666667]

[0.97461929 0. 0.33333333]

[0.97461929 0. 0.66666667]

[0.97461929 0. 1. ]

[0.97461929 0. 1.33333333]

[0.97461929 0. 1.66666667]

[0.97461929 1. 0.33333333]

[0.97461929 1. 0.66666667]

[0.97461929 1. 1. ]

[0.97461929 1. 1.33333333]

[0.97461929 1. 1.66666667]

[0.97461929 2. 0.33333333]

[0.97461929 2. 0.66666667]

[0.97461929 2. 1. ]

[0.97461929 2. 1.33333333]

[0.97461929 2. 1.66666667]

[0.90693739 0. 0.33333333]

[0.90693739 0. 0.66666667]

[0.90693739 0. 1. ]

[0.90693739 0. 1.33333333]

[0.90693739 0. 1.66666667]

[0.90693739 1. 0.33333333]

[0.90693739 1. 0.66666667]

[0.90693739 1. 1. ]

[0.90693739 1. 1.33333333]

[0.90693739 1. 1.66666667]

# define neural network model

#As seen below, we have created four dense layers.

#A dense layer is a layer in neural network that’s fully connected.

#In other words, all the neurons in one layer are connected to all other neurons in the next layer. #In the first layer, we need to provide the input shape, which is 1 in our case. #The activation function we have chosen is elu, which stands for exponential linear unit. .

from keras import backend as K

#initialize weights with values between -0.2 and 1.2

initializer = keras.initializers.RandomUniform(minval= -0.9, maxval=0.9)

model = keras.Sequential([

keras.layers.Dense(4, activation=K.relu, input\_shape=[3], kernel\_initializer=initializer), keras.layers.Dense(8, activation=K.relu, kernel\_initializer=initializer),

keras.layers.Dense(4, activation=K.relu, kernel\_initializer=initializer),

keras.layers.Dense(2, kernel\_initializer=initializer)

])

'''in Task 2.2, add 3rd layer to network with 4 neurons and activation = K.relu''' #Print summary of model features

model.summary()

Model: "sequential\_2"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense\_8 (Dense) (None, 4) 16

dense\_9 (Dense) (None, 8) 40

dense\_10 (Dense) (None, 4) 36

https://colab.research.google.com/drive/1OA\_fd5xWRN34gaGT7IwSoOtaj9cgoZ-e#printMode=true 4/12

10/31/23, 11:24 PM Project 2 Task 2.2.ipynb - Colaboratory

dense\_11 (Dense) (None, 2) 10

=================================================================

Total params: 102 (408.00 Byte)

Trainable params: 102 (408.00 Byte)

Non-trainable params: 0 (0.00 Byte)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#We’re using RMSprop as our optimizer here. RMSprop stands for Root Mean Square Propagation. #It’s one of the most popular gradient descent optimization algorithms for deep learning networks. #RMSprop is an optimizer that’s reliable and fast.

#We’re compiling the mode using the model.compile function. The loss function used here #is mean squared error. After the compilation of the model, we’ll use the fit method with ~500 epochs. #Number of epochs can be varied.

#from tf.keras import optimizers. Argument to RMSprop is learning parameter.

rms = keras.optimizers.RMSprop(0.001)

model.compile(loss='mean\_absolute\_error',optimizer=rms)

#After the compilation of the model, we’ll use the fit method with 500 epochs. #I started with epochs value of 100 and then tested the model after training.

Christopher Simotas 

Oct 23, 2023

(edited 12:03 PM Today)

Lowest Loss = 0.03738900646567345

#The prediction was not that good. Then I modified the number of epochs to 200 and tested the model again. #Accuracy had improved slightly, but figured I’d give it one more try. Finally, at 500 epochs #I found acceptable prediction accuracy. 

Christopher Simotas 12:03 PM Today

#The fit method takes three parameters; namely, x, y, and number of epochs. #During model training, if all the batches of data are seen by the model once, #we say that one epoch has been completed.

# Add an early stopping callback

es = keras.callbacks.EarlyStopping(

monitor='loss',

mode='min',

patience = 80,

restore\_best\_weights = True,

verbose=1)

# Add a checkpoint where loss is minimum, and save that model mc = keras.callbacks.ModelCheckpoint('best\_model.SB', monitor='loss', mode='min', verbose=1, save\_best\_only=True)

historyData = model.fit(xarray,yarray,epochs=600,callbacks=[es])

loss\_hist = historyData.history['loss']

#The above line will return a dictionary, access it's info like this: best\_epoch = np.argmin(historyData.history['loss']) + 1

print ('best epoch = ', best\_epoch)

print('smallest loss =', np.min(loss\_hist))

predictions = model.predict(data\_inputs)

Epoch 1/600

2/2 [==============================] - 0s 12ms/step - loss: 0.0394 Epoch 2/600

2/2 [==============================] - 0s 10ms/step - loss: 0.0382 Epoch 3/600

2/2 [==============================] - 0s 12ms/step - loss: 0.0392 Epoch 4/600

2/2 [==============================] - 0s 16ms/step - loss: 0.0393 Epoch 5/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0386 Epoch 6/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0406 Epoch 7/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0399 Epoch 8/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0395 Epoch 9/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0401 Epoch 10/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0390 Epoch 11/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0388 Epoch 12/600

2/2 [==============================] - 0s 6ms/step - loss: 0.0392 Epoch 13/600

2/2 [==============================] - 0s 6ms/step - loss: 0.0397

0.0338638611137867

https://colab.research.google.com/drive/1OA\_fd5xWRN34gaGT7IwSoOtaj9cgoZ-e#printMode=true 5/12

10/31/23, 11:24 PM Project 2 Task 2.2.ipynb - Colaboratory

Epoch 14/600

2/2 [==============================] - 0s 6ms/step - loss: 0.0412 Epoch 15/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0413 Epoch 16/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0402 Epoch 17/600

2/2 [==============================] - 0s 8ms/step - loss: 0.0387 Epoch 18/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0400 Epoch 19/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0394 Epoch 20/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0389 Epoch 21/600

2/2 [==============================] - 0s 6ms/step - loss: 0.0392 Epoch 22/600

2/2 [==============================] - 0s 6ms/step - loss: 0.0399 Epoch 23/600

2/2 [==============================] - 0s 6ms/step - loss: 0.0390 Epoch 24/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0400 Epoch 25/600

2/2 [==============================] - 0s 8ms/step - loss: 0.0395 Epoch 26/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0410 Epoch 27/600

2/2 [==============================] - 0s 7ms/step - loss: 0.0392 Epoch 28/600

2/2 [==============================] - 0s 8ms/step - loss: 0.0396 Epoch 29/600

### PRINT WEIGHTS AND BIAS ###

from \_\_future\_\_ import print\_function

#For results of training network:

#keras.layer.get\_weights() function retrieves weight values first\_layer\_weights = model.layers[0].get\_weights()[0]

w01 = first\_layer\_weights[0][0]

w02 = first\_layer\_weights[1][0]

w03 = first\_layer\_weights[2][0]

first\_layer\_bias = model.layers[0].get\_weights()[1]

b1 = first\_layer\_bias

second\_layer\_weights = model.layers[1].get\_weights()[0]

w12 = second\_layer\_weights[0][0]

second\_layer\_bias = model.layers[1].get\_weights()[1]

b2 = second\_layer\_bias

third\_layer\_weights = model.layers[2].get\_weights()[0]

w23 = third\_layer\_weights[0][0]

third\_layer\_bias = model.layers[2].get\_weights()[1]

b3 = third\_layer\_bias

#print weights and biases

print (first\_layer\_weights)

print ('w01 = ', w01, 'w02 = ', w02, 'w03 = ', w03)

print (first\_layer\_bias)

print ('b1 = ', b1)

print (second\_layer\_weights)

print ('w12 = ', w12)

print (second\_layer\_bias)

print ('b2 = ', b2)

print (third\_layer\_weights)

print ('w23 = ', w23)

print (third\_layer\_bias)

print ('b3 = ', b3)

[[-0.07818508 -0.8592682 0.68138176 0.55355525]

[-0.7264853 0.07107359 -0.40343848 0.8241015 ]

[-0.62351745 -0.28795207 0.76705 0.14476855]]

w01 = -0.07818508 w02 = -0.7264853 w03 = -0.62351745

[ 0. 0. -0.116648 -0.01164606]

b1 = [ 0. 0. -0.116648 -0.01164606]

[[-0.07818508 -0.8592682 0.66105974 0.5724138 -0.7264853 0.07107359 -0.67831886 0.32231402]

[-0.62351745 -0.28795207 0.5613973 -0.33145857 0.08687824 -0.76026464 0.50479794 -0.82465243]

Christopher Simotas 

Oct 23, 2023

Did this for fun, but no idea how to read it

https://colab.research.google.com/drive/1OA\_fd5xWRN34gaGT7IwSoOtaj9cgoZ-e#printMode=true 6/12

10/31/23, 11:24 PM Project 2 Task 2.2.ipynb - Colaboratory

[ 0.4051201 0.10365887 -0.37055156 0.64738166 0.40841305 0.7338768 0.17370227 0.48080054]

[-0.00648928 0.45852435 0.20108356 -0.6645981 -0.45105177 -0.01228085 1.0435193 -0.49823236]]

w12 = -0.07818508

[ 0.03630962 -0.04948874 -0.13093425 -0.12930813 0.297065 -0.07355705 0.0116918 -0.34688905]

b2 = [ 0.03630962 -0.04948874 -0.13093425 -0.12930813 0.297065 -0.07355705 0.0116918 -0.34688905]

[[-0.16024345 -0.68838304 0.52671957 0.44823143]

[-0.80897963 0.2519825 -0.8194737 0.18075241]

[-0.62351745 -0.66833395 0.5329366 -0.33338946]

[ 0.01315431 -0.6308349 0.37373525 -0.7934497 ]

[ 0.47132698 0.4837944 -0.249353 0.7172749 ]

[ 0.10691621 0.76628065 -0.08885805 0.49732572]

[ 0.12811598 0.51628065 0.04404363 -0.97995555]

[-0.51939243 0.7037906 0.64412713 -0.12350308]]

w23 = -0.16024345

[-0.08265618 -0.03927265 -0.13177943 0.00193784]

b3 = [-0.08265618 -0.03927265 -0.13177943 0.00193784]

# EXAMPLE SHOWING HOW TO USE MODEL.PREDICT

test = []

outpt=[]

predictions = model.predict(data\_inputs)

#first point (row [0])comparison of data and prediction

# put in a loop to print comparion for all data points

testarray = np.array([[ xarray[0][0] , xarray[0][1] , xarray[0][2] ]])

outpt = model.predict(testarray)

print ('row [0] data: T1= ', xarray[0][0]\*Tmed, ', gam= ', xarray[0][1]\*gamed, \ ', qsol= ', xarray[0][2]\*qsmed,', alpha= ', yarray[0][0]\*almed,\ ', predicted alpha = ', outpt[0][0]\*almed)

# SETTING UP PLOT

%matplotlib inline

# importing the required module

import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = [8, 8] # for square canvas

#========

'''CALCULATE PREDICTED VALUES AND RETRIEVE DATA VALUES TO PLOT'''

plt.scatter(predictions[:, 0], data\_outputs[:, 0])

plt.title('Genetic Algorithm training of Neural Network ==> output = alpha') plt.xlabel('predicted output for NN (units)')

plt.ylabel('data output (units)')

plt.loglog()

plt.xlim(xmax = 10, xmin = 0.1)

plt.ylim(ymax = 10, ymin = 0.1)

# Generate red y=x line

x\_data = np.linspace(0.1, 10.0, num=3)

y\_data = x\_data

plt.plot(x\_data, y\_data, color='red')

plt.show()

https://colab.research.google.com/drive/1OA\_fd5xWRN34gaGT7IwSoOtaj9cgoZ-e#printMode=true 7/12

10/31/23, 11:24 PM Project 2 Task 2.2.ipynb - Colaboratory

2/2 [==============================] - 0s 6ms/step

1/1 [==============================] - 0s 16ms/step

row [0] data: T1= 317.99999999999994 , gam= 0.0 , qsol= 500.0 , alpha= 35.1316 , predicted alpha = 35.32870097308159

### BUILD SURFACE PLOT ### 

import matplotlib.cm as cm

# Build Test Arrays

test\_T1 = np.linspace(268/Tmed, 318/Tmed, 100)

test\_Qs = np.linspace(500/qsmed, 2500/qsmed, 100)

test\_gamma = 0.25/gamed

# Create Meshgrid

TEST\_T1, TEST\_QS = np.meshgrid(test\_T1, test\_Qs)

# Create a 3D array of shape (num\_rows, num\_cols, 3) for the input data

input\_data = np.stack((TEST\_T1, np.full\_like(TEST\_T1, test\_gamma), TEST\_QS), axis=-1)

# Reshape the input data to have (num\_rows \* num\_cols, 3)

input\_data = input\_data.reshape(-1, 3)

# Use model.predict to get all outputs at once

outputs = model.predict(input\_data)

# Reshape the outputs to match the original shape

ALPHA = (outputs[:, 0].reshape(TEST\_T1.shape))

# Create a 3D plot

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

# Plot the surface

surf = ax.plot\_surface(TEST\_T1\*Tmed, TEST\_QS\*qsmed, ALPHA\*almed, cmap=cm.coolwarm, linewidth=0, antialiased=False) #ax.view\_init(elev=10, azim=270) # Adjust 'elev' for elevation and 'azim' for azimuth angles

fig.colorbar(surf, ax=ax)

ax.set\_xlabel('T1 Temperature (K)', labelpad = 15, fontsize=12)

ax.set\_ylabel('Solar Thermal Heat Input Rate (W)', labelpad = 15, fontsize=12)

ax.set\_zlabel('Required Air to Fuel Ratio', labelpad = 1, fontsize=12)

ax.tick\_params(axis='y', labelsize=12)

ax.tick\_params(axis='x', labelsize=12)

ax.tick\_params(axis='z', labelsize=12)

plt.title('Required Air to Fuel Ratio at Gamma = 0.25')

#ax.set\_xlim(200, 1000 )

#ax.set\_ylim(200, 3000)

plt.tight\_layout()

plt.show()

https://colab.research.google.com/drive/1OA\_fd5xWRN34gaGT7IwSoOtaj9cgoZ-e#printMode=true 8/12

10/31/23, 11:24 PM Project 2 Task 2.2.ipynb - Colaboratory

313/313 [==============================] - 1s 2ms/step

# Another way to create surface plot with Looping 

'''

import matplotlib.pyplot as plt

from matplotlib import cm

# Build Test Arrays

test\_T1 = np.linspace(268/Tmed, 318/Tmed, 20)

test\_Qs = np.linspace(500/qsmed, 2500/qsmed, 20)

test\_gamma = 0.25/gamed

# Create Meshgrid

TEST\_T1, TEST\_QS = np.meshgrid(test\_T1, test\_Qs)

# Compute Z (height) values using the function

num\_rows, num\_cols = TEST\_T1.shape

ALPHA = np.full\_like(TEST\_T1, 0)

for i in range(num\_rows):

for j in range(num\_cols):

testarray = np.array([[TEST\_T1[i][j], test\_gamma, TEST\_QS[i][j]]])

outputs = model.predict(testarray)

ALPHA[i][j] = outputs[0][0]\*almed

# Create a 3D plot

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

# Plot the surface

surf = ax.plot\_surface(TEST\_T1\*Tmed, TEST\_QS\*qsmed, ALPHA, cmap=cm.coolwarm, linewidth=0, antialiased=False) #ax.view\_init(elev=0, azim=110) # Adjust 'elev' for elevation and 'azim' for azimuth angles fig.colorbar(surf, ax=ax)

ax.set\_xlabel('T1 Temperature (K)', labelpad = 15, fontsize=12)

ax.set\_ylabel('Rate of Solar Thermal Heat Input (C/s)', labelpad = 15, fontsize=12) ax.set\_zlabel('Required Air to Fuel Ratio', labelpad = -300, fontsize=12)

ax.tick\_params(axis='y', labelsize=12)

ax.tick\_params(axis='x', labelsize=12)

ax.tick\_params(axis='z', labelsize=12)

https://colab.research.google.com/drive/1OA\_fd5xWRN34gaGT7IwSoOtaj9cgoZ-e#printMode=true 9/12

10/31/23, 11:24 PM Project 2 Task 2.2.ipynb - Colaboratory

plt.title('Required Air to Fuel Ratio at Gamma = 0.25')

#ax.set\_xlim(14, 62)

#ax.set\_ylim(14, 62)

plt.tight\_layout()

plt.show()

'''

'

\nimport matplotlib.pyplot as plt\nfrom matplotlib import cm\n\n# Build Test Arrays\ntest\_T1 = np.linspace(268/Tmed, 318/Tmed, 20)\ntest ace(500/qsmed, 2500/qsmed, 20)\ntest\_gamma = 0.25/gamed\n\n# Create Meshgrid\nTEST\_T1, TEST\_QS = np.meshgrid(test\_T1, test\_Qs)\n\n# Compu values using the function\nnum\_rows, num\_cols = TEST\_T1.shape\nALPHA = np.full\_like(TEST\_T1, 0)\nfor i in range(num\_rows):\n for j in ra s):\n testarray = np.array([[TEST\_T1[i][j], test\_gamma, TEST\_QS[i][j]]])\n outputs = model.predict(testarray)\n ALPHA[i][j] = ou med\n\n# Create a 3D plot\nfig = plt.figure()\nax = fig.add\_subplot(111, projection='3d')\n\n# Plot the surface\nsurf = ax.plot\_surface(T EST\_QS\*qsmed, ALPHA, cmap=cm.coolwarm, linewidth=0, antialiased=False)\n#ax.view\_init(elev=0, azim=110) # Adjust 'elev' for elevation an zimuth angles\nfig.colorbar(surf, ax=ax)\n\nax.set\_xlabel('T1 Temperature (K)', labelpad = 15, fon

…'

### TEST VALIDATION DATA

# Build Test Arrays

datax = [

[318.0, 0.0, 500.0],

[318.0, 0.0, 1500.0],

[318.0, 0.0, 2500.0],

[318.0, 0.25, 1500.0],

[318.0, 0.5, 500.0],

[318.0, 0.5, 1500.0],

[318.0, 0.5, 2500.0],

[303.0, 0.0, 1000.0],

[303.0, 0.0, 2000.0],

[303.0, 0.25, 1000.0],

[303.0, 0.25, 2000.0],

[303.0, 0.5, 1000.0],

[303.0, 0.5, 2000.0],

[288.0, 0.0, 500.0],

[288.0, 0.0, 2500.0],

[288.0, 0.25, 2500.0],

[288.0, 0.5, 1500.0],

[268.0, 0.0, 1500.0],

[268.0, 0.25, 2000.0],

[268.0, 0.5, 2500.0]

]

test\_xdata = np.array(datax) # convert to np array

datay = [

[ 35.13 , 0.3808 ],

[ 47.46 , 0.3930 ],

[ 73.12 , 0.4061 ],

[ 66.34 , 0.4098 ],

[ 63.09, 0.4154 ],

[ 85.23 , 0.4197 ],

[131.32 , 0.4242 ],

[ 38.99 , 0.4012 ],

[ 53.80 , 0.4136 ],

[ 54.51 , 0.4215 ],

[ 75.22 , 0.4290 ],

[ 70.04, 0.4337 ],

[ 96.65, 0.4382 ],

[ 33.45 , 0.4091 ],

[ 60.80 , 0.4334 ],

[ 85.044, 0.4477],

[ 77.56 , 0.4516 ],

[ 40.68 , 0.4383 ],

[ 65.24 , 0.4628 ],

[ 98.23 , 0.4760 ],

]

test\_ydata = np.array(datay) # convert to np array

# Calculate Median For Each Column

median\_values\_x = np.median(test\_xdata, axis=0)

Tmed = median\_values\_x[0]

gamed = median\_values\_x[1]

qsmed = median\_values\_x[2]

https://colab.research.google.com/drive/1OA\_fd5xWRN34gaGT7IwSoOtaj9cgoZ-e#printMode=true 10/12

10/31/23, 11:24 PM Project 2 Task 2.2.ipynb - Colaboratory

median\_values\_y = np.median(test\_ydata, axis=0)

almed = median\_values\_y[0]

efmed = median\_values\_y[1]

print("Median values for each column:", median\_values\_x, median\_values\_y)

print("\n")

# Normalize Data

test\_xdata = (test\_xdata/median\_values\_x)

test\_ydata = (test\_ydata/median\_values\_y)

# Use model.predict to get all outputs at once

outputs = model.predict(test\_xdata)

# Regression Line

m, b = np.polyfit(test\_ydata[:, 0]\*almed, outputs[:, 0]\*almed, 1)

regress\_input = np.linspace(np.min(test\_ydata[:, 0])\*almed, np.max(test\_ydata[:, 0])\*almed, 50)

# Create a Log-Log plot

plt.loglog(test\_ydata[:, 0]\*almed, outputs[:, 0]\*almed, marker = 'o', linestyle = '') plt.loglog(regress\_input, m\*(regress\_input) + b, label=f'Regression Line: y = {m:.2f}x + {b:.2f}') plt.loglog(regress\_input, regress\_input, label=f'Perfect Fit Line: y = x')

plt.title('Log-Log Plot of Alpha Predicted Vs. Measured')

plt.xlabel('Measured Alpha ', labelpad = 10)

plt.ylabel('Predicted Alpha', labelpad = 10)

plt.legend()

plt.show()

### Output RMSD Value ###

diff = ((test\_ydata[:, 0]\*almed) - (outputs[:, 0]\*almed))/(test\_ydata[:, 0]\*almed) rms\_dev = np.sqrt(np.mean(diff\*\*2))

print("RMS Relative Deviation between Predicted and Measured Alpha: " + str(rms\_dev) + "\n") account\_circle

https://colab.research.google.com/drive/1OA\_fd5xWRN34gaGT7IwSoOtaj9cgoZ-e#printMode=true 11/12

10/31/23, 11:24 PM Project 2 Task 2.2.ipynb - Colaboratory

Median values for each column: [3.03e+02 2.50e-01 1.50e+03] [65.79 0.42285]

1/1 [==============================] - 0s 19ms/step



https://colab.research.google.com/drive/1OA\_fd5xWRN34gaGT7IwSoOtaj9cgoZ-e#printMode=true 12/12

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

ME 249 Project 2 Chris Simotas and Mason Rodriguez Rand

Task 2.4

!pip install pygad

'''To use pygad, first import it.'''

import tensorflow.keras

import numpy

import pygad

print('pygad installed correctly if no error messages')

output Collecting pygad

Downloading pygad-3.2.0-py3-none-any.whl (80 kB)

━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 80.8/80.8 kB 2.1 MB/s eta 0:00:00

Requirement already satisfied: cloudpickle in /usr/local/lib/python3.10/dist-packages (from pygad) (2.2.1) Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (from pygad) (3.7.1)

Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from pygad) (1.23.5)

Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (1.1.1) Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (0.12.1) Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (4.43.1) Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (1.4.5) Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (23.2) Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (9.4.0) Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (3.1.1) Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (2.8.2) Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib->pygad) (1.16. Installing collected packages: pygad

Successfully installed pygad-3.2.0

pygad installed correctly if no error messages

#CodeP2.5F23 ME249 V.P. Carey

#===import relevant packages

import tensorflow

import tensorflow.keras

import pygad.kerasga

import numpy as np

import pygad

#the following 2 lines are only needed for Mac OS machines

import os

os.environ['KMP\_DUPLICATE\_LIB\_OK']='True'

#========================

'''

#create input data array

# meadian values of input variables

Tmed = 293.

gamed = 0.25

qsmed = 1250.

#T1(K), gamma, , qsol(kW):

xdata = []

ND = 60

xdata = [[ 318.0/Tmed , 0.0/gamed , 500.0/qsmed ], [ 318.0/Tmed , 0.0/gamed , 1000.0/qsmed ]]

xdata.append([ 318.0/Tmed , 0.0/gamed , 1500.0/qsmed ])

xdata.append([ 318.0/Tmed , 0.0/gamed , 2000.0/qsmed ])

xdata.append([ 318.0/Tmed , 0.0/gamed , 2500.0/qsmed ])

# meadian values of output variables

almed = 60.

efmed = 0.4

# alpha, effsys

ydata = []

ydata = [[ 35.1316/almed , 0.3808/efmed ], [ 40.3764/almed , 0.38686/efmed ]]

ydata.append([ 47.4620/almed , 0.3930/efmed ])

ydata.append([ 57.5639/almed , 0.39949/efmed ])

ydata.append([ 73.1286/almed , 0.40612/efmed ])

'''

### USER CODE ###

#create input data array, normalizing input temp

#T1(K), gamma, , qsol(kW):

xdata = []

xdata = [[ 318.0 , 0.0 , 500.0 ], [ 318.0 , 0.0 , 1000.0 ]]

xdata.append([ 318.0 , 0.0 , 1500.0 ])

xdata.append([ 318.0 , 0.0 , 2000.0 ])

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 1/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

xdata.append([ 318.0 , 0.0 , 2500.0 ])

xdata.append([ 318.0 , 0.25 , 500.0 ])

xdata.append([ 318.0 , 0.25 , 1000.0 ])

xdata.append([ 318.0 , 0.25 , 1500.0 ])

xdata.append([ 318.0 , 0.25 , 2000.0 ])

xdata.append([ 318.0 , 0.25 , 2500.0 ])

xdata.append([ 318.0 , 0.5 , 500.0 ])

xdata.append([ 318.0 , 0.5 , 1000.0 ])

xdata.append([ 318.0 , 0.5 , 1500.0 ])

xdata.append([ 318.0 , 0.5 , 2000.0 ])

xdata.append([ 318.0 , 0.5 , 2500.0 ])

xdata.append([ 303.0 , 0.0 , 500.0 ])

xdata.append([ 303.0 , 0.0 , 1000.0 ])

xdata.append([ 303.0 , 0.0 , 1500.0 ])

xdata.append([ 303.0 , 0.0 , 2000.0 ])

xdata.append([ 303.0 , 0.0 , 2500.0 ])

xdata.append([ 303.0 , 0.25 , 500.0 ])

xdata.append([ 303.0 , 0.25 , 1000.0 ])

xdata.append([ 303.0 , 0.25 , 1500.0 ])

xdata.append([ 303.0 , 0.25 , 2000.0 ])

xdata.append([ 303.0 , 0.25 , 2500.0 ])

xdata.append([ 303.0 , 0.5 , 500.0 ])

xdata.append([ 303.0 , 0.5 , 1000.0 ])

xdata.append([ 303.0 , 0.5 , 1500.0 ])

xdata.append([ 303.0 , 0.5 , 2000.0 ])

xdata.append([ 303.0 , 0.5 , 2500.0 ])

xdata.append([ 288.0 , 0.0 , 500.0 ])

xdata.append([ 288.0 , 0.0 , 1000.0 ])

xdata.append([ 288.0 , 0.0 , 1500.0 ])

xdata.append([ 288.0 , 0.0 , 2000.0 ])

xdata.append([ 288.0 , 0.0 , 2500.0 ])

xdata.append([ 288.0 , 0.25 , 500.0 ])

xdata.append([ 288.0 , 0.25 , 1000.0 ])

xdata.append([ 288.0 , 0.25 , 1500.0 ])

xdata.append([ 288.0 , 0.25 , 2000.0 ])

xdata.append([ 288.0 , 0.25 , 2500.0 ])

xdata.append([ 288.0 , 0.5 , 500.0 ])

xdata.append([ 288.0 , 0.5 , 1000.0 ])

xdata.append([ 288.0 , 0.5 , 1500.0 ])

xdata.append([ 288.0 , 0.5 , 2000.0 ])

xdata.append([ 288.0 , 0.5 , 2500.0 ])

xdata.append([ 268.0 , 0.0 , 500.0 ])

xdata.append([ 268.0 , 0.0 , 1000.0 ])

xdata.append([ 268.0 , 0.0 , 1500.0 ])

xdata.append([ 268.0 , 0.0 , 2000.0 ])

xdata.append([ 268.0 , 0.0 , 2500.0 ])

xdata.append([ 268.0 , 0.25 , 500.0 ])

xdata.append([ 268.0 , 0.25 , 1000.0 ])

xdata.append([ 268.0 , 0.25 , 1500.0 ])

xdata.append([ 268.0 , 0.25 , 2000.0 ])

xdata.append([ 268.0 , 0.25 , 2500.0 ])

xdata.append([ 268.0 , 0.5 , 500.0 ])

xdata.append([ 268.0 , 0.5 , 1000.0 ])

xdata.append([ 268.0 , 0.5 , 1500.0 ])

xdata.append([ 268.0 , 0.5 , 2000.0 ])

xdata.append([ 268.0 , 0.5 , 2500.0 ])

ydata = [[ 35.1316 , 0.3808 ],[ 40.3764 , 0.38686 ]] ydata.append([ 47.4620 , 0.3930 ])

ydata.append([ 57.5639 , 0.39949 ])

ydata.append([ 73.1286 , 0.40612 ])

ydata.append([ 49.1110 , 0.4023 ])

ydata.append([ 56.4428 , 0.40605 ])

ydata.append([ 66.3479 , 0.4098 ])

ydata.append([ 80.4695 , 0.413 ])

ydata.append([ 102.2276 , 0.4175 ])

ydata.append([ 63.0904 , 0.41540 ])

ydata.append([ 72.5092 , 0.4175 ])

ydata.append([ 85.2338, 0.4197 ])

ydata.append([ 103.3750 , 0.42192 ])

ydata.append([ 131.3266 , 0.4242 ])

ydata.append([ 34.273 , 0.3952 ])

ydata append([ 38 99026 0 4012 ])

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 2/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

ydata.append([ 38.99026 , 0.4012 ])

ydata.append([ 45.2133, 0.4073 ])

ydata.append([ 53.8000 , 0.4136 ])

ydata.append([ 66.4130 , 0.4201 ])

ydata.append([ 47.922 , 0.4178 ])

ydata.append([ 54.518 , 0.4215 ])

ydata.append([ 63.220 , 0.4252 ])

ydata.append([ 75.226 , 0.4290 ])

ydata.append([ 92.862 , 0.4329 ])

ydata.append([ 61.572 , 0.4315 ])

ydata.append([ 70.0468 , 0.43373 ])

ydata.append([ 81.226 , 0.43597 ])

ydata.append([ 96.653 , 0.4382 ])

ydata.append([ 119.3124 , 0.44045 ])

ydata.append([ 33.4521 , 0.40913 ])

ydata.append([ 37.6911, 0.4150 ])

ydata.append([ 43.1602 , 0.4209 ])

ydata.append([ 50.4858 , 0.4271 ])

ydata.append([ 60.8067 , 0.4334 ])

ydata.append([ 46.7865 , 0.4328 ])

ydata.append([ 52.7151 , 0.43646 ])

ydata.append([ 60.36425 , 0.44016 ])

ydata.append([ 70.6099 , 0.443926 ])

ydata.append([ 85.0447 , 0.4477 ])

ydata.append([ 60.1208 , 0.44721 ])

ydata.append([ 67.7391 , 0.44940 ])

ydata.append([ 77.56830 , 0.4516 ])

ydata.append([ 90.73410 , 0.4538 ])

ydata.append([ 109.2828 , 0.4560 ])

ydata.append([ 32.4123 , 0.42694 ])

ydata.append([ 36.0807 , 0.4325 ])

ydata.append([ 40.6854 , 0.4383 ])

ydata.append([ 46.6374 , 0.4442 ])

ydata.append([ 54.6293 , 0.4503 ])

ydata.append([ 45.3472 , 0.4519 ])

ydata.append([ 50.4796 , 0.4555 ])

ydata.append([ 56.9219 , 0.4591 ])

ydata.append([ 65.2492 , 0.4628 ])

ydata.append([ 76.4304 , 0.4665 ])

ydata.append([ 58.2822 , 0.4672 ])

ydata.append([ 64.8785 , 0.4693 ])

ydata.append([ 73.1584 , 0.4715 ])

ydata.append([ 83.8610 , 0.4738 ])

ydata.append([ 98.2316 , 0.4760 ])

# Calculate the median for each column

median\_values\_x = np.median(np.array(xdata), axis=0)

Tmed = median\_values\_x[0] # 295.5

gamed = median\_values\_x[1] # 0.25

qsmed = median\_values\_x[2] # 1500.

median\_values\_y = np.median(np.array(ydata), axis=0)

almed = median\_values\_y[0] # 61.1

efmed = median\_values\_y[1] # 0.432

print("Median values for each column:", median\_values\_x, median\_values\_y)

# Reformat and Print Data

xdata = (xdata/median\_values\_x).tolist()

ydata = (ydata/median\_values\_y).tolist()

### USER CODE ###

data\_inputs = numpy.array(xdata)

print (data\_inputs)

data\_outputs = numpy.array(ydata)

print (data\_outputs)

Median values for each column: [2.955e+02 2.500e-01 1.500e+03] [61.18935 0.432 ] [[1.07614213 0. 0.33333333]

[1.07614213 0. 0.66666667]

[1.07614213 0. 1. ]

[1.07614213 0. 1.33333333]

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 3/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

[1.07614213 0. 1.66666667]

[1.07614213 1. 0.33333333]

[1.07614213 1. 0.66666667]

[1.07614213 1. 1. ]

[1.07614213 1. 1.33333333]

[1.07614213 1. 1.66666667]

[1.07614213 2. 0.33333333]

[1.07614213 2. 0.66666667]

[1.07614213 2. 1. ]

[1.07614213 2. 1.33333333]

[1.07614213 2. 1.66666667]

[1.02538071 0. 0.33333333]

[1.02538071 0. 0.66666667]

[1.02538071 0. 1. ]

[1.02538071 0. 1.33333333]

[1.02538071 0. 1.66666667]

[1.02538071 1. 0.33333333]

[1.02538071 1. 0.66666667]

[1.02538071 1. 1. ]

[1.02538071 1. 1.33333333]

[1.02538071 1. 1.66666667]

[1.02538071 2. 0.33333333]

[1.02538071 2. 0.66666667]

[1.02538071 2. 1. ]

[1.02538071 2. 1.33333333]

[1.02538071 2. 1.66666667]

[0.97461929 0. 0.33333333]

[0.97461929 0. 0.66666667]

[0.97461929 0. 1. ]

[0.97461929 0. 1.33333333]

[0.97461929 0. 1.66666667]

[0.97461929 1. 0.33333333]

[0.97461929 1. 0.66666667]

[0.97461929 1. 1. ]

[0.97461929 1. 1.33333333]

[0.97461929 1. 1.66666667]

[0.97461929 2. 0.33333333]

[0.97461929 2. 0.66666667]

[0.97461929 2. 1. ]

[0.97461929 2. 1.33333333]

[0.97461929 2. 1.66666667]

[0.90693739 0. 0.33333333]

[0.90693739 0. 0.66666667]

[0.90693739 0. 1. ]

[0.90693739 0. 1.33333333]

[0.90693739 0. 1.66666667]

[0.90693739 1. 0.33333333]

[0.90693739 1. 0.66666667]

[0.90693739 1. 1. ]

[0.90693739 1. 1.33333333]

[0.90693739 1. 1.66666667]

[0.90693739 2. 0.33333333]

[0.90693739 2. 0.66666667]

#===define fitness function used in Genetic Algorithm - added ga\_instance input as needed for # In PyGAD 2.20.0, the fitness function must accept 3 parameters: # 1) The instance of the 'pygad.GA' class.

# 2) A solution to calculate its fitness value.

# 3) The solution's index within the population.

def fitness\_func(ga\_instance, solution, sol\_idx):

global data\_inputs, data\_outputs, keras\_ga, model

model\_weights\_matrix = pygad.kerasga.model\_weights\_as\_matrix(model=model, weights\_vector=solution)

model.set\_weights(weights=model\_weights\_matrix)

predictions = model.predict(data\_inputs)

mae = tensorflow.keras.losses.MeanAbsoluteError()

#data output array and predictions array are provided

#-->error for each data point is calcuated and mean abs error is computed for them #thus a mean fitness is determined for each solution using all data points meanabs\_error = mae(data\_outputs, predictions).numpy() + 0.00000001

solution\_fitness = 1.0/meanabs\_error

return solution\_fitness

#========================END

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 4/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

#===define callback that keeps track of best solution for each generation and saves the best of # of all of them over the generations analyzed

def callback\_generation(ga\_instance):

print("Generation = {generation}".format(generation=ga\_instance.generations\_completed)) print("Fitness = {fitness}".format(fitness=ga\_instance.best\_solution()[1]))

#========================END

#===defining a sequential Neural Network

#initialize weights with values between minval and maxval

initializer = tensorflow.keras.initializers.RandomUniform(minval= -0.9, maxval=0.9)

model = tensorflow.keras.Sequential([

tensorflow.keras.layers.Dense(4, activation='relu', input\_shape=[3], kernel\_initializer=initializer), tensorflow.keras.layers.Dense(8, activation='relu', kernel\_initializer=initializer), tensorflow.keras.layers.Dense(4, activation='relu', kernel\_initializer=initializer), tensorflow.keras.layers.Dense(2, kernel\_initializer=initializer)

])

#Print summary of model features

model.summary()

#===define a vector that contains all the weights for Neural Network model

weights\_vector = pygad.kerasga.model\_weights\_as\_vector(model=model)

#===KerasGA Info below

'''KerasGA is part of the PyGAD library for training Keras models using the genetic algorithm (GA). The KerasGA project has a single module named kerasga.py which has a class named KerasGA for preparing an initial population of Keras model parameters.

PyGAD is an open-source Python library for building the genetic algorithm

and training machine learning algorithms.

Check the library's documentation at Read The Docs: https://pygad.readthedocs.io'''

#===Appears to instantiate a keras genetic algorithm object in which the genes are the weights for # NN model, and the population is 40 solutions - just used to set initial population keras\_ga = pygad.kerasga.KerasGA(model=model,

num\_solutions=40)

#===set number of generations to run, and number of best fitness solutions to

# keep and mate in each generation

num\_generations = 1500

num\_parents\_mating = 7 #Number of solutions to be selected as parents.

'''sol\_per\_pop = number of best solutions kept??'''

sol\_per\_pop = 37 # if fitness\_batch\_size is supported to calculate the fitness function in batches, # then the solutions are grouped into batches of size

parent\_selection\_type = "sss"

'''Steady State Selection

In every generation few chromosomes are selected

(good - with high fitness) for creating a new offspring.

Then some (bad - with low fitness) chromosomes are removed

and the new offspring is placed in their place.

The rest of population survives to new generation.'''

keep\_parents = 5

''' keep\_parents=-1: Number of parents to keep in the current population.

-1 (default) means to keep all parents in the next population.

0 means keep no parents in the next population. A value greater than 0

means keeps the specified number of parents in the next population. '''

crossover\_type = "single\_point"

''' crossover\_type="single\_point": Type of the crossover operation. Supported types are single\_point (for single-point crossover), two\_points (for two points crossover), uniform (for uniform crossover), and scattered (for scattered crossover). Scattered crossover is supported from PyGAD 2.9.0 and higher. It defaults to single\_point.'''

mutation\_type = "random"

''' mutation\_type="random": Type of the mutation operation. Supported types are random (for random mutation), swap (for swap mutation), inversion (for inversion mutation), scramble (for scramble mutation), and

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 5/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

adaptive (for adaptive mutation). It defaults to random.'''

mutation\_percent\_genes = "default"

''' mutation\_percent\_genes="default": Percentage of genes to mutate. It defaults to the string "default" which is later translated into the integer 10 which means 10% of the genes will be mutated.

It must be >0 and <=100. Out of this percentage, the number of genes to mutate is deduced

which is assigned to the mutation\_num\_genes parameter.'''

#===set initial population as the neural network weights

initial\_population = keras\_ga.population\_weights

#===instantiate a pygad genetic algorithm object with desired attributes

ga\_instance = pygad.GA(num\_generations=num\_generations,

num\_parents\_mating=num\_parents\_mating,

initial\_population=initial\_population,

fitness\_func=fitness\_func,

sol\_per\_pop=sol\_per\_pop,

parent\_selection\_type = parent\_selection\_type,

keep\_parents = keep\_parents,

crossover\_type = crossover\_type,

mutation\_type = mutation\_type,

mutation\_percent\_genes = mutation\_percent\_genes,

on\_generation=callback\_generation)

#===run this instance of GA object that trains NN

ga\_instance.run()

#===the following just plots and prints results after run for specified number of generations

# After the generations complete, some plots are showed that summarize how the outputs/fitness values evolve over generations. ga\_instance.plot\_fitness(title="PyGAD & Keras - Iteration vs. Fitness", linewidth=4)

# Returning the details of the best solution.

solution, solution\_fitness, solution\_idx = ga\_instance.best\_solution()

print("Fitness value of the best solution = {solution\_fitness}".format(solution\_fitness=solution\_fitness)) print("Index of the best solution : {solution\_idx}".format(solution\_idx=solution\_idx))

# Fetch the parameters of the best solution.

best\_solution\_weights = pygad.kerasga.model\_weights\_as\_matrix(model=model,

weights\_vector=solution)

model.set\_weights(best\_solution\_weights)

predictions = model.predict(data\_inputs)

print("Predictions : \n", predictions)

print("data\_outputs : \n", data\_outputs)

mae = tensorflow.keras.losses.MeanAbsoluteError()

abs\_error = mae(data\_outputs, predictions).numpy()

print("Absolute Error : ", abs\_error)

#========================END

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 6/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense (Dense) (None, 4) 16

dense\_1 (Dense) (None, 8) 40

dense\_2 (Dense) (None, 4) 36

dense\_3 (Dense) (None, 2) 10

=================================================================

Total params: 102 (408.00 Byte)

Trainable params: 102 (408.00 Byte)

Non-trainable params: 0 (0.00 Byte)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/usr/local/lib/python3.10/dist-packages/keras/src/initializers/initializers.py:120: UserWarning: The initializer RandomUniform is unseede warnings.warn(

**Streaming output truncated to the last 5000 lines.**

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 5ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 5ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 3ms/step

2/2 [==============================] - 0s 6ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 5ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

#SETTING UP PLOT

2/2 [==============================] - 0s 4ms/step

%matplotlib inline

2/2 [==============================] - 0s 4ms/step

# importing the required module

2/2 [==============================] - 0s 4ms/step

import matplotlib.pyplot as plt

2/2 [==============================] - 0s 3ms/step

plt.rcParams['figure.figsize'] = [8, 8] # for square canvas

2/2 [==============================] - 0s 4ms/step

#========

2/2 [==============================] - 0s 6ms/step

2/2 [==============================] - 0s 4ms/step

'''CALCULATE PREDICTED VALUES, RETRIEVE DATA VALUES AND PLOT'''

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 3ms/step

plt.scatter(predictions[:, 0], data\_outputs[:, 0])

2/2 [==============================] - 0s 4ms/step

plt.title('Case 1 Genetic Algorithm Training of Neural Network ==> Output = Alpha')

2/2 [==============================] - 0s 4ms/step

plt.xlabel('predicted output for NN (units)')

2/2 [==============================] - 0s 4ms/step

plt.ylabel('data output (units)')

2/2 [==============================] - 0s 4ms/step

plt.loglog()

2/2 [==============================] - 0s 4ms/step

plt.xlim(xmax = 10, xmin = 0.1)

2/2 [==============================] - 0s 4ms/step

plt.ylim(ymax = 10, ymin = 0.1)

2/2 [==============================] - 0s 5ms/step

# Generate red y=x line

2/2 [==============================] - 0s 4ms/step

x\_data = numpy.linspace(0.1, 10.0, num=3)

2/2 [==============================] - 0s 5ms/step

2/2 [==============================] - 0s 5ms/step

y\_data = x\_data

2/2 [==============================] - 0s 4ms/step

plt.plot(x\_data, y\_data, color='red')

2/2 [==============================] - 0s 5ms/step

plt.show()

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 5ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

Fitness = 26.635100146974565

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 5ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 6ms/step

2/2 [==============================] - 0s 8ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [ ] 0s 5ms/step

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 7/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

Fitness value of the best solution = 26.635100146974565 Index of the best solution : 0 2/2 [==============================] - 0s 4ms/step Predictions : [[0.56829846 0.9462137 ] [0.6203693 0.97661936] [0.8336084 0.93810165] [1.0213232 0.9107852 ] [1.1432731 0.9123293 ] [0.7549653 1.0074414 ] [0.92246556 1.015717 ] [1.1150432 1.0119892 ] [1.3282826 0.9734715 ] [1.6031833 1.0700694 ] [1.053571 1.0221943 ] [1.2385082 1.0313312 ] [1.4234459 1.0404683 ] [1.6248109 1.0804329 ] [1.9202292 1.2219908 ] [0.5685359 0.94643164] [0.60767317 0.98236823] [0.7547207 0.97215724] [0.9283874 0.9510057 ] [1.0503371 0.9525498 ] [0.7553216 1.007459 ] [0.8654754 1.0129013 ] [1.0504127 1.0220382 ] [1.2493947 1.0075272 ] [1.4823246 1.0121566 ] [0.99658084 1.0193787 ] [1.1815183 1.0285157 ] [1.3664556 1.0376526 ] [1.551393 1.0467896 ] [1.7993708 1.1640782 ] [0.56877327 0.94664955] [0.60791063 0.98258615] [0.6773796 1.0036082 ] [0.83545125 0.9912263 ] [0.9574008 0.99277043] [0.7556778 1.0074766 ] [0.8144337 1.0103796 ] [0.9934225 1.0192225 ] [1.1783597 1.0283597 ] [1.3837459 1.0030651 ] [0.9723499 1.0181816 ] [1.1245282 1.0257 ] [1.3094654 1.034837 ] [1.4944026 1.0439739 ] [1.7105603 1.1215222 ] [0.56908965 0.9469402 ] [0.608227 0.98287666] [0.6569927 1.002601 ] [0.73435986 1.0064234 ] [0.85372424 1.0123208 ] [0.756153 1.0075002 ] [0.81490874 1.0104029 ] [0.91743565 1.0154684 ] [1.1023726 1.0246054 ] [1.2873104 1.0337424 ] [0.9728248 1.0182049 ] [1.0485411 1.0219458 ] [1.2334783 1.0310827 ] [1.418416 1.0402198 ] [1.6154985 1.0759706 ]] data\_outputs : [[0.57414566 0.88148148] [0.65985993 0.89550926] [0.77565786 0.90972222] [0.94075031 0.92474537] [1.19511974 0.94009259] [0.80260699 0.93125 ] [0.92242849 0.93993056] [1.0843047 0.94861111] [1.31508996 0.95601852] [1.67067635 0.96643519] [1.03106831 0.96157407] [1.18499706 0.96643519] [1.39295155 0.97152778] [1.68942798 0.97666667] [2.14623296 0.98194444] [0.56011381 0.91481481] [0.63720664 0.9287037 ] [0.738908 0.94282407] [0.87923797 0.95740741] [1.08536861 0.9724537 ] [0.7831755 0.96712963] [0.89097204 0.97569444] [1.03318633 0.98425926] [1.22939695 0.99305556] [1.51761704 1.00208333] [1.00625354 0.99884259] [1.14475477 1.00400463] [1.32745323 1.00918981] [1.57957226 1.01435185] [1.94988834 1.01956019] [0.54669808 0.94706019] [0.61597484 0.96064815] [0.70535477 0.97430556] [0.82507495 0.98865741] [0.99374646 1.00324074] [0.76461835 1.00185185] [0.86150776 1.01032407] [0.98651563 1.01888889] [1.15395735 1.02760648] [1.38986114 1.03634259] [0.98253699 1.03520833] [1.10704069 1.04027778] [1.26767648 1.04537037] [1.48284138 1.05046296] [1.78597746 1.05555556] [0.52970492 0.98828704] [0.58965653 1.00115741] [0.66490982 1.01458333] [0.76218165 1.02824074] [0.89279098 1.04236111] [0.74109629 1.04606481] [0.82497363 1.05439815] [0.93025829 1.06273148] [1.06634896 1.0712963 ] [1.24908011 1.07986111] [0.95248928 1.08148148] [1.06029072 1.08634259] [1.19560675 1.09143519] [1.37051627 1.09675926] [1.60537087 1.10185185]] Absolute Error : 0.037544433

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 8/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

ME 249 Project 2 Chris Simotas and Mason Rodriguez Rand

Task 2.4

!pip install pygad

'''To use pygad, first import it.'''

import tensorflow.keras

import numpy

import pygad

print('pygad installed correctly if no error messages')

output Collecting pygad

Downloading pygad-3.2.0-py3-none-any.whl (80 kB)

━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 80.8/80.8 kB 2.1 MB/s eta 0:00:00

Requirement already satisfied: cloudpickle in /usr/local/lib/python3.10/dist-packages (from pygad) (2.2.1) Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (from pygad) (3.7.1)

Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from pygad) (1.23.5)

Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (1.1.1) Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (0.12.1) Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (4.43.1) Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (1.4.5) Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (23.2) Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (9.4.0) Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (3.1.1) Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib->pygad) (2.8.2) Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib->pygad) (1.16. Installing collected packages: pygad

Successfully installed pygad-3.2.0

pygad installed correctly if no error messages

#CodeP2.5F23 ME249 V.P. Carey

#===import relevant packages

import tensorflow

import tensorflow.keras

import pygad.kerasga

import numpy as np

import pygad

#the following 2 lines are only needed for Mac OS machines

import os

os.environ['KMP\_DUPLICATE\_LIB\_OK']='True'

#========================

'''

#create input data array

# meadian values of input variables

Tmed = 293.

gamed = 0.25

qsmed = 1250.

#T1(K), gamma, , qsol(kW):

xdata = []

ND = 60

xdata = [[ 318.0/Tmed , 0.0/gamed , 500.0/qsmed ], [ 318.0/Tmed , 0.0/gamed , 1000.0/qsmed ]]

xdata.append([ 318.0/Tmed , 0.0/gamed , 1500.0/qsmed ])

xdata.append([ 318.0/Tmed , 0.0/gamed , 2000.0/qsmed ])

xdata.append([ 318.0/Tmed , 0.0/gamed , 2500.0/qsmed ])

# meadian values of output variables

almed = 60.

efmed = 0.4

# alpha, effsys

ydata = []

ydata = [[ 35.1316/almed , 0.3808/efmed ], [ 40.3764/almed , 0.38686/efmed ]]

ydata.append([ 47.4620/almed , 0.3930/efmed ])

ydata.append([ 57.5639/almed , 0.39949/efmed ])

ydata.append([ 73.1286/almed , 0.40612/efmed ])

'''

### USER CODE ###

#create input data array, normalizing input temp

#T1(K), gamma, , qsol(kW):

xdata = []

xdata = [[ 318.0 , 0.0 , 500.0 ], [ 318.0 , 0.0 , 1000.0 ]]

xdata.append([ 318.0 , 0.0 , 1500.0 ])

xdata.append([ 318.0 , 0.0 , 2000.0 ])

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 1/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

xdata.append([ 318.0 , 0.0 , 2500.0 ])

xdata.append([ 318.0 , 0.25 , 500.0 ])

xdata.append([ 318.0 , 0.25 , 1000.0 ])

xdata.append([ 318.0 , 0.25 , 1500.0 ])

xdata.append([ 318.0 , 0.25 , 2000.0 ])

xdata.append([ 318.0 , 0.25 , 2500.0 ])

xdata.append([ 318.0 , 0.5 , 500.0 ])

xdata.append([ 318.0 , 0.5 , 1000.0 ])

xdata.append([ 318.0 , 0.5 , 1500.0 ])

xdata.append([ 318.0 , 0.5 , 2000.0 ])

xdata.append([ 318.0 , 0.5 , 2500.0 ])

xdata.append([ 303.0 , 0.0 , 500.0 ])

xdata.append([ 303.0 , 0.0 , 1000.0 ])

xdata.append([ 303.0 , 0.0 , 1500.0 ])

xdata.append([ 303.0 , 0.0 , 2000.0 ])

xdata.append([ 303.0 , 0.0 , 2500.0 ])

xdata.append([ 303.0 , 0.25 , 500.0 ])

xdata.append([ 303.0 , 0.25 , 1000.0 ])

xdata.append([ 303.0 , 0.25 , 1500.0 ])

xdata.append([ 303.0 , 0.25 , 2000.0 ])

xdata.append([ 303.0 , 0.25 , 2500.0 ])

xdata.append([ 303.0 , 0.5 , 500.0 ])

xdata.append([ 303.0 , 0.5 , 1000.0 ])

xdata.append([ 303.0 , 0.5 , 1500.0 ])

xdata.append([ 303.0 , 0.5 , 2000.0 ])

xdata.append([ 303.0 , 0.5 , 2500.0 ])

xdata.append([ 288.0 , 0.0 , 500.0 ])

xdata.append([ 288.0 , 0.0 , 1000.0 ])

xdata.append([ 288.0 , 0.0 , 1500.0 ])

xdata.append([ 288.0 , 0.0 , 2000.0 ])

xdata.append([ 288.0 , 0.0 , 2500.0 ])

xdata.append([ 288.0 , 0.25 , 500.0 ])

xdata.append([ 288.0 , 0.25 , 1000.0 ])

xdata.append([ 288.0 , 0.25 , 1500.0 ])

xdata.append([ 288.0 , 0.25 , 2000.0 ])

xdata.append([ 288.0 , 0.25 , 2500.0 ])

xdata.append([ 288.0 , 0.5 , 500.0 ])

xdata.append([ 288.0 , 0.5 , 1000.0 ])

xdata.append([ 288.0 , 0.5 , 1500.0 ])

xdata.append([ 288.0 , 0.5 , 2000.0 ])

xdata.append([ 288.0 , 0.5 , 2500.0 ])

xdata.append([ 268.0 , 0.0 , 500.0 ])

xdata.append([ 268.0 , 0.0 , 1000.0 ])

xdata.append([ 268.0 , 0.0 , 1500.0 ])

xdata.append([ 268.0 , 0.0 , 2000.0 ])

xdata.append([ 268.0 , 0.0 , 2500.0 ])

xdata.append([ 268.0 , 0.25 , 500.0 ])

xdata.append([ 268.0 , 0.25 , 1000.0 ])

xdata.append([ 268.0 , 0.25 , 1500.0 ])

xdata.append([ 268.0 , 0.25 , 2000.0 ])

xdata.append([ 268.0 , 0.25 , 2500.0 ])

xdata.append([ 268.0 , 0.5 , 500.0 ])

xdata.append([ 268.0 , 0.5 , 1000.0 ])

xdata.append([ 268.0 , 0.5 , 1500.0 ])

xdata.append([ 268.0 , 0.5 , 2000.0 ])

xdata.append([ 268.0 , 0.5 , 2500.0 ])

ydata = [[ 35.1316 , 0.3808 ],[ 40.3764 , 0.38686 ]] ydata.append([ 47.4620 , 0.3930 ])

ydata.append([ 57.5639 , 0.39949 ])

ydata.append([ 73.1286 , 0.40612 ])

ydata.append([ 49.1110 , 0.4023 ])

ydata.append([ 56.4428 , 0.40605 ])

ydata.append([ 66.3479 , 0.4098 ])

ydata.append([ 80.4695 , 0.413 ])

ydata.append([ 102.2276 , 0.4175 ])

ydata.append([ 63.0904 , 0.41540 ])

ydata.append([ 72.5092 , 0.4175 ])

ydata.append([ 85.2338, 0.4197 ])

ydata.append([ 103.3750 , 0.42192 ])

ydata.append([ 131.3266 , 0.4242 ])

ydata.append([ 34.273 , 0.3952 ])

ydata append([ 38 99026 0 4012 ])

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 2/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

ydata.append([ 38.99026 , 0.4012 ])

ydata.append([ 45.2133, 0.4073 ])

ydata.append([ 53.8000 , 0.4136 ])

ydata.append([ 66.4130 , 0.4201 ])

ydata.append([ 47.922 , 0.4178 ])

ydata.append([ 54.518 , 0.4215 ])

ydata.append([ 63.220 , 0.4252 ])

ydata.append([ 75.226 , 0.4290 ])

ydata.append([ 92.862 , 0.4329 ])

ydata.append([ 61.572 , 0.4315 ])

ydata.append([ 70.0468 , 0.43373 ])

ydata.append([ 81.226 , 0.43597 ])

ydata.append([ 96.653 , 0.4382 ])

ydata.append([ 119.3124 , 0.44045 ])

ydata.append([ 33.4521 , 0.40913 ])

ydata.append([ 37.6911, 0.4150 ])

ydata.append([ 43.1602 , 0.4209 ])

ydata.append([ 50.4858 , 0.4271 ])

ydata.append([ 60.8067 , 0.4334 ])

ydata.append([ 46.7865 , 0.4328 ])

ydata.append([ 52.7151 , 0.43646 ])

ydata.append([ 60.36425 , 0.44016 ])

ydata.append([ 70.6099 , 0.443926 ])

ydata.append([ 85.0447 , 0.4477 ])

ydata.append([ 60.1208 , 0.44721 ])

ydata.append([ 67.7391 , 0.44940 ])

ydata.append([ 77.56830 , 0.4516 ])

ydata.append([ 90.73410 , 0.4538 ])

ydata.append([ 109.2828 , 0.4560 ])

ydata.append([ 32.4123 , 0.42694 ])

ydata.append([ 36.0807 , 0.4325 ])

ydata.append([ 40.6854 , 0.4383 ])

ydata.append([ 46.6374 , 0.4442 ])

ydata.append([ 54.6293 , 0.4503 ])

ydata.append([ 45.3472 , 0.4519 ])

ydata.append([ 50.4796 , 0.4555 ])

ydata.append([ 56.9219 , 0.4591 ])

ydata.append([ 65.2492 , 0.4628 ])

ydata.append([ 76.4304 , 0.4665 ])

ydata.append([ 58.2822 , 0.4672 ])

ydata.append([ 64.8785 , 0.4693 ])

ydata.append([ 73.1584 , 0.4715 ])

ydata.append([ 83.8610 , 0.4738 ])

ydata.append([ 98.2316 , 0.4760 ])

# Calculate the median for each column

median\_values\_x = np.median(np.array(xdata), axis=0)

Tmed = median\_values\_x[0] # 295.5

gamed = median\_values\_x[1] # 0.25

qsmed = median\_values\_x[2] # 1500.

median\_values\_y = np.median(np.array(ydata), axis=0)

almed = median\_values\_y[0] # 61.1

efmed = median\_values\_y[1] # 0.432

print("Median values for each column:", median\_values\_x, median\_values\_y)

# Reformat and Print Data

xdata = (xdata/median\_values\_x).tolist()

ydata = (ydata/median\_values\_y).tolist()

### USER CODE ###

data\_inputs = numpy.array(xdata)

print (data\_inputs)

data\_outputs = numpy.array(ydata)

print (data\_outputs)

Median values for each column: [2.955e+02 2.500e-01 1.500e+03] [61.18935 0.432 ] [[1.07614213 0. 0.33333333]

[1.07614213 0. 0.66666667]

[1.07614213 0. 1. ]

[1.07614213 0. 1.33333333]

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 3/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

[1.07614213 0. 1.66666667]

[1.07614213 1. 0.33333333]

[1.07614213 1. 0.66666667]

[1.07614213 1. 1. ]

[1.07614213 1. 1.33333333]

[1.07614213 1. 1.66666667]

[1.07614213 2. 0.33333333]

[1.07614213 2. 0.66666667]

[1.07614213 2. 1. ]

[1.07614213 2. 1.33333333]

[1.07614213 2. 1.66666667]

[1.02538071 0. 0.33333333]

[1.02538071 0. 0.66666667]

[1.02538071 0. 1. ]

[1.02538071 0. 1.33333333]

[1.02538071 0. 1.66666667]

[1.02538071 1. 0.33333333]

[1.02538071 1. 0.66666667]

[1.02538071 1. 1. ]

[1.02538071 1. 1.33333333]

[1.02538071 1. 1.66666667]

[1.02538071 2. 0.33333333]

[1.02538071 2. 0.66666667]

[1.02538071 2. 1. ]

[1.02538071 2. 1.33333333]

[1.02538071 2. 1.66666667]

[0.97461929 0. 0.33333333]

[0.97461929 0. 0.66666667]

[0.97461929 0. 1. ]

[0.97461929 0. 1.33333333]

[0.97461929 0. 1.66666667]

[0.97461929 1. 0.33333333]

[0.97461929 1. 0.66666667]

[0.97461929 1. 1. ]

[0.97461929 1. 1.33333333]

[0.97461929 1. 1.66666667]

[0.97461929 2. 0.33333333]

[0.97461929 2. 0.66666667]

[0.97461929 2. 1. ]

[0.97461929 2. 1.33333333]

[0.97461929 2. 1.66666667]

[0.90693739 0. 0.33333333]

[0.90693739 0. 0.66666667]

[0.90693739 0. 1. ]

[0.90693739 0. 1.33333333]

[0.90693739 0. 1.66666667]

[0.90693739 1. 0.33333333]

[0.90693739 1. 0.66666667]

[0.90693739 1. 1. ]

[0.90693739 1. 1.33333333]

[0.90693739 1. 1.66666667]

[0.90693739 2. 0.33333333]

[0.90693739 2. 0.66666667]

#===define fitness function used in Genetic Algorithm - added ga\_instance input as needed for # In PyGAD 2.20.0, the fitness function must accept 3 parameters: # 1) The instance of the 'pygad.GA' class.

# 2) A solution to calculate its fitness value.

# 3) The solution's index within the population.

def fitness\_func(ga\_instance, solution, sol\_idx):

global data\_inputs, data\_outputs, keras\_ga, model

model\_weights\_matrix = pygad.kerasga.model\_weights\_as\_matrix(model=model, weights\_vector=solution)

model.set\_weights(weights=model\_weights\_matrix)

predictions = model.predict(data\_inputs)

mae = tensorflow.keras.losses.MeanAbsoluteError()

#data output array and predictions array are provided

#-->error for each data point is calcuated and mean abs error is computed for them #thus a mean fitness is determined for each solution using all data points meanabs\_error = mae(data\_outputs, predictions).numpy() + 0.00000001

solution\_fitness = 1.0/meanabs\_error

return solution\_fitness

#========================END

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 4/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

#===define callback that keeps track of best solution for each generation and saves the best of # of all of them over the generations analyzed

def callback\_generation(ga\_instance):

print("Generation = {generation}".format(generation=ga\_instance.generations\_completed)) print("Fitness = {fitness}".format(fitness=ga\_instance.best\_solution()[1]))

#========================END

#===defining a sequential Neural Network

#initialize weights with values between minval and maxval

initializer = tensorflow.keras.initializers.RandomUniform(minval= -0.9, maxval=0.9)

model = tensorflow.keras.Sequential([

tensorflow.keras.layers.Dense(4, activation='relu', input\_shape=[3], kernel\_initializer=initializer), tensorflow.keras.layers.Dense(8, activation='relu', kernel\_initializer=initializer), tensorflow.keras.layers.Dense(4, activation='relu', kernel\_initializer=initializer), tensorflow.keras.layers.Dense(2, kernel\_initializer=initializer)

])

#Print summary of model features

model.summary()

#===define a vector that contains all the weights for Neural Network model

weights\_vector = pygad.kerasga.model\_weights\_as\_vector(model=model)

#===KerasGA Info below

'''KerasGA is part of the PyGAD library for training Keras models using the genetic algorithm (GA). The KerasGA project has a single module named kerasga.py which has a class named KerasGA for preparing an initial population of Keras model parameters.

PyGAD is an open-source Python library for building the genetic algorithm

and training machine learning algorithms.

Check the library's documentation at Read The Docs: https://pygad.readthedocs.io'''

#===Appears to instantiate a keras genetic algorithm object in which the genes are the weights for # NN model, and the population is 40 solutions - just used to set initial population keras\_ga = pygad.kerasga.KerasGA(model=model,

num\_solutions=40)

#===set number of generations to run, and number of best fitness solutions to

# keep and mate in each generation

num\_generations = 1500

num\_parents\_mating = 7 #Number of solutions to be selected as parents.

'''sol\_per\_pop = number of best solutions kept??'''

sol\_per\_pop = 37 # if fitness\_batch\_size is supported to calculate the fitness function in batches, # then the solutions are grouped into batches of size

parent\_selection\_type = "sss"

'''Steady State Selection

In every generation few chromosomes are selected

(good - with high fitness) for creating a new offspring.

Then some (bad - with low fitness) chromosomes are removed

and the new offspring is placed in their place.

The rest of population survives to new generation.'''

keep\_parents = 5

''' keep\_parents=-1: Number of parents to keep in the current population.

-1 (default) means to keep all parents in the next population.

0 means keep no parents in the next population. A value greater than 0

means keeps the specified number of parents in the next population. '''

crossover\_type = "single\_point"

''' crossover\_type="single\_point": Type of the crossover operation. Supported types are single\_point (for single-point crossover), two\_points (for two points crossover), uniform (for uniform crossover), and scattered (for scattered crossover). Scattered crossover is supported from PyGAD 2.9.0 and higher. It defaults to single\_point.'''

mutation\_type = "random"

''' mutation\_type="random": Type of the mutation operation. Supported types are random (for random mutation), swap (for swap mutation), inversion (for inversion mutation), scramble (for scramble mutation), and

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 5/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

adaptive (for adaptive mutation). It defaults to random.'''

mutation\_percent\_genes = "default"

''' mutation\_percent\_genes="default": Percentage of genes to mutate. It defaults to the string "default" which is later translated into the integer 10 which means 10% of the genes will be mutated.

It must be >0 and <=100. Out of this percentage, the number of genes to mutate is deduced

which is assigned to the mutation\_num\_genes parameter.'''

#===set initial population as the neural network weights

initial\_population = keras\_ga.population\_weights

#===instantiate a pygad genetic algorithm object with desired attributes

ga\_instance = pygad.GA(num\_generations=num\_generations,

num\_parents\_mating=num\_parents\_mating,

initial\_population=initial\_population,

fitness\_func=fitness\_func,

sol\_per\_pop=sol\_per\_pop,

parent\_selection\_type = parent\_selection\_type,

keep\_parents = keep\_parents,

crossover\_type = crossover\_type,

mutation\_type = mutation\_type,

mutation\_percent\_genes = mutation\_percent\_genes,

on\_generation=callback\_generation)

#===run this instance of GA object that trains NN

ga\_instance.run()

#===the following just plots and prints results after run for specified number of generations

# After the generations complete, some plots are showed that summarize how the outputs/fitness values evolve over generations. ga\_instance.plot\_fitness(title="PyGAD & Keras - Iteration vs. Fitness", linewidth=4)

# Returning the details of the best solution.

solution, solution\_fitness, solution\_idx = ga\_instance.best\_solution()

print("Fitness value of the best solution = {solution\_fitness}".format(solution\_fitness=solution\_fitness)) print("Index of the best solution : {solution\_idx}".format(solution\_idx=solution\_idx))

# Fetch the parameters of the best solution.

best\_solution\_weights = pygad.kerasga.model\_weights\_as\_matrix(model=model,

weights\_vector=solution)

model.set\_weights(best\_solution\_weights)

predictions = model.predict(data\_inputs)

print("Predictions : \n", predictions)

print("data\_outputs : \n", data\_outputs)

mae = tensorflow.keras.losses.MeanAbsoluteError()

abs\_error = mae(data\_outputs, predictions).numpy()

print("Absolute Error : ", abs\_error)

#========================END

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 6/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense (Dense) (None, 4) 16

dense\_1 (Dense) (None, 8) 40

dense\_2 (Dense) (None, 4) 36

dense\_3 (Dense) (None, 2) 10

=================================================================

Total params: 102 (408.00 Byte)

Trainable params: 102 (408.00 Byte)

Non-trainable params: 0 (0.00 Byte)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/usr/local/lib/python3.10/dist-packages/keras/src/initializers/initializers.py:120: UserWarning: The initializer RandomUniform is unseede warnings.warn(

**Streaming output truncated to the last 5000 lines.**

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 5ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 5ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 3ms/step

2/2 [==============================] - 0s 6ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 5ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

#SETTING UP PLOT

2/2 [==============================] - 0s 4ms/step

%matplotlib inline

2/2 [==============================] - 0s 4ms/step

# importing the required module

2/2 [==============================] - 0s 4ms/step

import matplotlib.pyplot as plt

2/2 [==============================] - 0s 3ms/step

plt.rcParams['figure.figsize'] = [8, 8] # for square canvas

2/2 [==============================] - 0s 4ms/step

#========

2/2 [==============================] - 0s 6ms/step

2/2 [==============================] - 0s 4ms/step

'''CALCULATE PREDICTED VALUES, RETRIEVE DATA VALUES AND PLOT'''

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 3ms/step

plt.scatter(predictions[:, 0], data\_outputs[:, 0])

2/2 [==============================] - 0s 4ms/step

plt.title('Case 1 Genetic Algorithm Training of Neural Network ==> Output = Alpha')

2/2 [==============================] - 0s 4ms/step

plt.xlabel('predicted output for NN (units)')

2/2 [==============================] - 0s 4ms/step

plt.ylabel('data output (units)')

2/2 [==============================] - 0s 4ms/step

plt.loglog()

2/2 [==============================] - 0s 4ms/step

plt.xlim(xmax = 10, xmin = 0.1)

2/2 [==============================] - 0s 4ms/step

plt.ylim(ymax = 10, ymin = 0.1)

2/2 [==============================] - 0s 5ms/step

# Generate red y=x line

2/2 [==============================] - 0s 4ms/step

x\_data = numpy.linspace(0.1, 10.0, num=3)

2/2 [==============================] - 0s 5ms/step

2/2 [==============================] - 0s 5ms/step

y\_data = x\_data

2/2 [==============================] - 0s 4ms/step

plt.plot(x\_data, y\_data, color='red')

2/2 [==============================] - 0s 5ms/step

plt.show()

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 5ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

Fitness = 26.635100146974565

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 5ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [==============================] - 0s 6ms/step

2/2 [==============================] - 0s 8ms/step

2/2 [==============================] - 0s 4ms/step

2/2 [ ] 0s 5ms/step

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 7/8

10/31/23, 11:25 PM Project 2 Task 2.4 Base Case (Case 1).ipynb - Colaboratory

Fitness value of the best solution = 26.635100146974565 Index of the best solution : 0 2/2 [==============================] - 0s 4ms/step Predictions : [[0.56829846 0.9462137 ] [0.6203693 0.97661936] [0.8336084 0.93810165] [1.0213232 0.9107852 ] [1.1432731 0.9123293 ] [0.7549653 1.0074414 ] [0.92246556 1.015717 ] [1.1150432 1.0119892 ] [1.3282826 0.9734715 ] [1.6031833 1.0700694 ] [1.053571 1.0221943 ] [1.2385082 1.0313312 ] [1.4234459 1.0404683 ] [1.6248109 1.0804329 ] [1.9202292 1.2219908 ] [0.5685359 0.94643164] [0.60767317 0.98236823] [0.7547207 0.97215724] [0.9283874 0.9510057 ] [1.0503371 0.9525498 ] [0.7553216 1.007459 ] [0.8654754 1.0129013 ] [1.0504127 1.0220382 ] [1.2493947 1.0075272 ] [1.4823246 1.0121566 ] [0.99658084 1.0193787 ] [1.1815183 1.0285157 ] [1.3664556 1.0376526 ] [1.551393 1.0467896 ] [1.7993708 1.1640782 ] [0.56877327 0.94664955] [0.60791063 0.98258615] [0.6773796 1.0036082 ] [0.83545125 0.9912263 ] [0.9574008 0.99277043] [0.7556778 1.0074766 ] [0.8144337 1.0103796 ] [0.9934225 1.0192225 ] [1.1783597 1.0283597 ] [1.3837459 1.0030651 ] [0.9723499 1.0181816 ] [1.1245282 1.0257 ] [1.3094654 1.034837 ] [1.4944026 1.0439739 ] [1.7105603 1.1215222 ] [0.56908965 0.9469402 ] [0.608227 0.98287666] [0.6569927 1.002601 ] [0.73435986 1.0064234 ] [0.85372424 1.0123208 ] [0.756153 1.0075002 ] [0.81490874 1.0104029 ] [0.91743565 1.0154684 ] [1.1023726 1.0246054 ] [1.2873104 1.0337424 ] [0.9728248 1.0182049 ] [1.0485411 1.0219458 ] [1.2334783 1.0310827 ] [1.418416 1.0402198 ] [1.6154985 1.0759706 ]] data\_outputs : [[0.57414566 0.88148148] [0.65985993 0.89550926] [0.77565786 0.90972222] [0.94075031 0.92474537] [1.19511974 0.94009259] [0.80260699 0.93125 ] [0.92242849 0.93993056] [1.0843047 0.94861111] [1.31508996 0.95601852] [1.67067635 0.96643519] [1.03106831 0.96157407] [1.18499706 0.96643519] [1.39295155 0.97152778] [1.68942798 0.97666667] [2.14623296 0.98194444] [0.56011381 0.91481481] [0.63720664 0.9287037 ] [0.738908 0.94282407] [0.87923797 0.95740741] [1.08536861 0.9724537 ] [0.7831755 0.96712963] [0.89097204 0.97569444] [1.03318633 0.98425926] [1.22939695 0.99305556] [1.51761704 1.00208333] [1.00625354 0.99884259] [1.14475477 1.00400463] [1.32745323 1.00918981] [1.57957226 1.01435185] [1.94988834 1.01956019] [0.54669808 0.94706019] [0.61597484 0.96064815] [0.70535477 0.97430556] [0.82507495 0.98865741] [0.99374646 1.00324074] [0.76461835 1.00185185] [0.86150776 1.01032407] [0.98651563 1.01888889] [1.15395735 1.02760648] [1.38986114 1.03634259] [0.98253699 1.03520833] [1.10704069 1.04027778] [1.26767648 1.04537037] [1.48284138 1.05046296] [1.78597746 1.05555556] [0.52970492 0.98828704] [0.58965653 1.00115741] [0.66490982 1.01458333] [0.76218165 1.02824074] [0.89279098 1.04236111] [0.74109629 1.04606481] [0.82497363 1.05439815] [0.93025829 1.06273148] [1.06634896 1.0712963 ] [1.24908011 1.07986111] [0.95248928 1.08148148] [1.06029072 1.08634259] [1.19560675 1.09143519] [1.37051627 1.09675926] [1.60537087 1.10185185]] Absolute Error : 0.037544433

https://colab.research.google.com/drive/1zF3MxDSNexQFgGKlekCaUduRgrLGkvax#printMode=true 8/8