EN.535.742 – Applied Machine Learning for Mechanical Engineers
Final Report - Zhang, Kanashiro, Stickney

a. A description of each nine steps, their results, figures, and tables.

1. For step 1, all data points with the property 'Age (day)' equal to 28 were isolated. As expected, this resulted in a total of 425 data points, with each individual data point consisting of 8 values, where 7 values are inputs and 1 value is the output.

2. For step 2, the data was preprocessed to make it ready for the machine learning model. Because all available data will be used as training data, no train-test split was required. Preprocessing only consisted of getting data keys, separating the data into inputs (the first 7 columns) and outputs (the last column), and then scaling those inputs and outputs between 0 and 1.

3. In step 3, the first deep neural network was created. This network has 7 inputs, four hidden layers with 64, 32, 16, and 8 hidden neurons (all with ReLU activations), and one output (Linear activation). The network was trained with the full set of 425 data points that were isolated in step 1, for 1000 epochs. The epoch vs. loss plot is shown below (Figure 1). From this figure, we can see that the training and validation loss values remain similar, so the network is not overfitted. A plot of the training data vs. estimated values is also provided for data in the calibrated domain (Figure 2). We can see that the data is clustered around the bisector line, showing that the estimated values align well with the actual values.

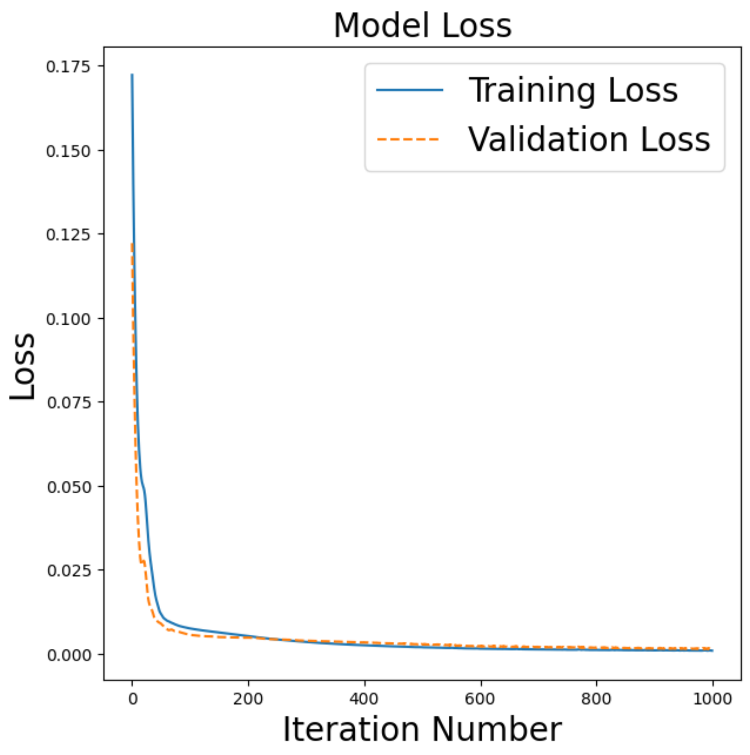

Figure 1: Epoch vs. Loss Plot for Training and Validation

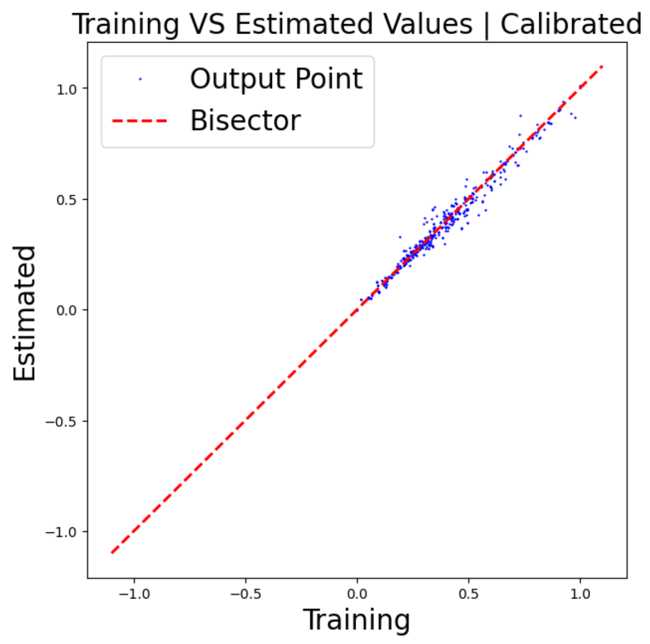

Figure 2: Training vs. Estimated Values on the Calibrated domain

4. In step 4, a GAN model was developed and trained using the 425 input data points from step 1. The discriminator neural network was created with 7 inputs, one for each of the variables being used as input data for the overall problem. The hidden layers had 32, 16, and 8 neurons, with one output. The hidden layers used Relu activation and the output used a sigmoid activation function, as specified.

```
$$$$$$$$$$$$$$$$$$$$$$$$$$
Discriminator
$$$$$$$$$$$$$$$$$$$$$$$$$$
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_10 (Dense)            (None, 32)                256

 dense_11 (Dense)            (None, 16)                528

 dense_12 (Dense)            (None, 8)                 136

 dense_13 (Dense)            (None, 1)                 9

=================================================================
Total params: 929
Trainable params: 0
Non-trainable params: 929
```

The generator was also created with 3 hidden layers. The number of neurons in each layer was chosen to be 8, 16, and 32 neurons, corresponding with the number of neurons in the discriminator. Relu activation functions were also used for the hidden layers because it ensures all values will remain positive, and a sigmoid activation function was used for the output layer to ensure all output values were between 0 and 1, like the calibrated data that was used as an input to the GAN. The generator was defined with 7 outputs, corresponding to the number of inputs in this problem.

```
$$$$$$$$$$$$$$$$$$$$$$$$$
Generator
$$$$$$$$$$$$$$$$$$$$$$$$$
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_14 (Dense)            (None, 8)                 264

 dense_15 (Dense)            (None, 16)                144

 dense_16 (Dense)            (None, 32)                544

 dense_17 (Dense)            (None, 7)                 231

=================================================================
Total params: 1,183
Trainable params: 1,183
Non-trainable params: 0
```

Once the discriminator and generator had been developed, they were combined into a single GAN model and trained using the 425 input data points. Initially, the number of training epochs was set to 500, and the batch size was set to 5. However, when the full 500 epochs were attempted, the model couldn't complete training due to the RAM limitations of Google Colab. To try to address the problem, the number of parameters in the generator was reduced, but this did not affect the RAM use enough to complete a full 500 epochs. Ultimately, to solve the problem the model was trained for 100 epochs with the originally selected number of parameters. The printed summary of the full model is provided below.

```
$$$$$$$$$$$$$$$$$$$$$$$$$
GAN
$$$$$$$$$$$$$$$$$$$$$$$$$
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_3 (Sequential)   (None, 7)                 1183

 sequential_2 (Sequential)   (None, 1)                 929

=================================================================
Total params: 2,112
Trainable params: 1,183
Non-trainable params: 929
```

5. In step 5, the trained model from step 4 was used to generate an additional 575 sets of input data (where each set has 7 values). This was achieved by calling the generator portion of the trained model to generate 575 points. These 575 points were then used as inputs to the neural network from step 3, which estimated what the output values would be for each point. Five sample points of the provided data and five sample points of the generated inputs and predicted outputs are provided below. All values are between 0 and 1, as expected since the calibrated input data was used.

Table I: Sample data points, Provided and Generated/Estimated data

	Inputs							**Output**
	1	**2**	**3**	**4**	**5**	**6**	**7**	
Provided Data	1	0	0	0.32108626	0.07763975	0.69476744	0.20572002	0.97595957
	1	0	0	0.32108626	0.07763975	0.73837209	0.20572002	0.72872558
	0.6347032	0.26432944	0	0.84824281	0	0.38081395	0	0.38123207
	0.37442922	0.31719533	0	0.84824281	0	0.38081395	0.19066734	0.50962983
	0.85159817	0	0	0.84824281	0	0.38081395	0	0.42002459
Generated /Estimated Data	2.36E-01	1.48E-02	9.45E-01	3.75E-01	4.34E-02	5.06E-03	7.55E-01	0.32103917
	3.35E-01	1.54E-01	7.64E-01	4.68E-01	3.25E-01	2.03E-01	5.96E-01	0.42710203
	2.89E-01	1.08E-01	8.48E-01	4.43E-01	3.38E-01	1.54E-01	6.33E-01	0.419897
	3.43E-01	1.31E-01	8.29E-01	4.74E-01	3.26E-01	1.70E-01	6.41E-01	0.46370926
	9.04E-01	9.09E-04	3.83E-05	9.38E-01	2.70E-05	9.31E-01	1.05E-01	0.46370926

6. In step 6 we defined the appropriate RTT and RRS, and we used the data created in step 5 to train the model. The function "fun_data" breaks the data that will go into the deep neural network into the appropriate length as defined by the RRS. A "fun_deep function" was created were a deep neural network with an epoch of 1000, and a validation split of 0.1. A dropout layer of 0.4 was added to prevent overfitting.

RTT 0.1 | RRS 001
mse_tr - mse_te
0.015393142622993506-0.014453879200111553

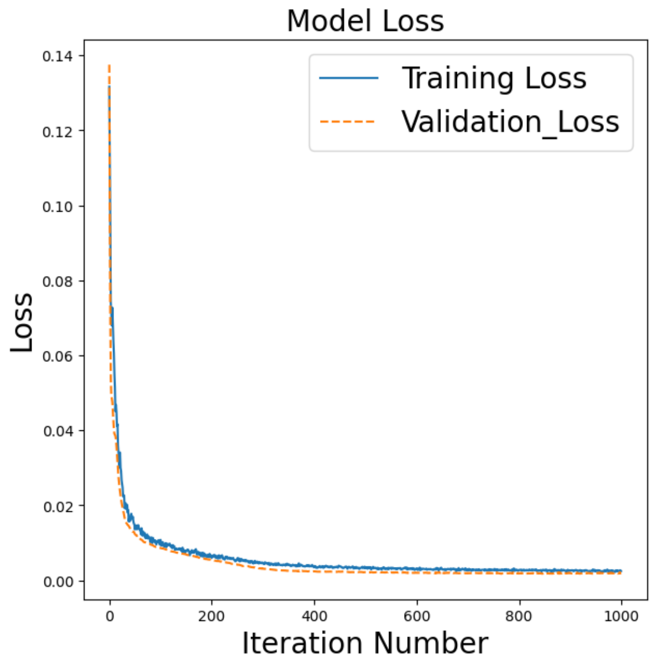

Figure 3: Epoch vs. Loss Plot for both validation and training loss

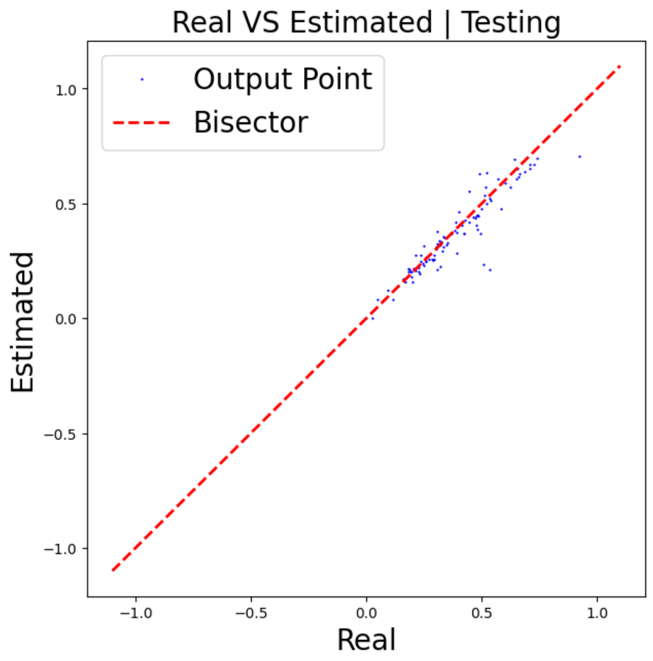

Figure 4: Training vs. Estimated Values on the Calibrated Domain

RTT 0.1 | RRS 002
mse_tr - mse_te
0.002151354846225678-0.004144591401371258

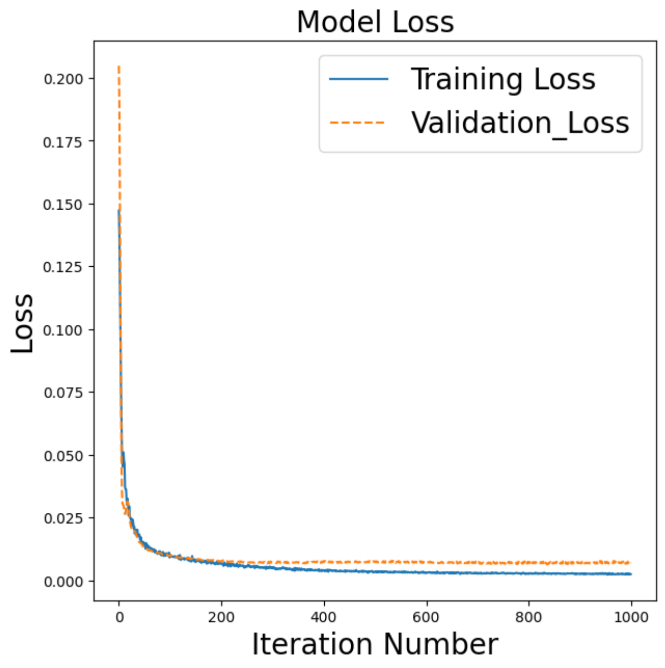

Figure 5: Epoch vs. Loss Plot for both validation and training loss

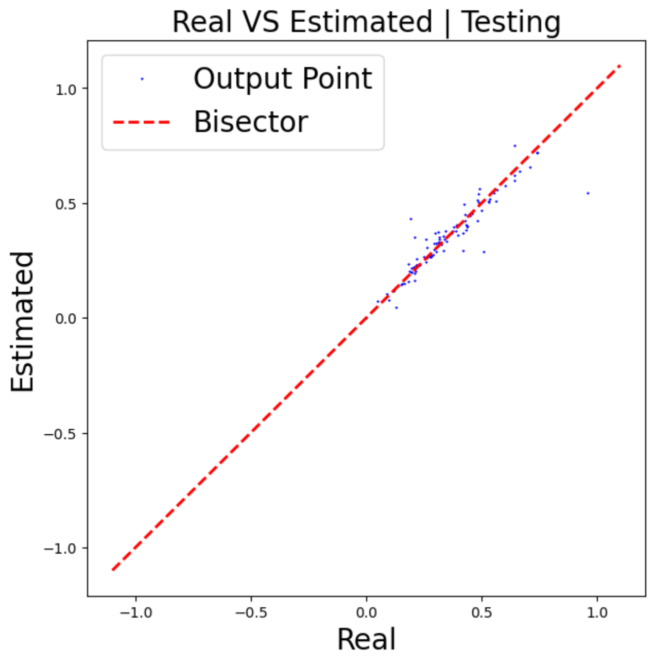

Figure 6: Training vs. Estimated Values on the Calibrated Domain

RTT 0.1 | RRS 003
mse_tr - mse_te
0.0018456865443952047-0.003970006911129221

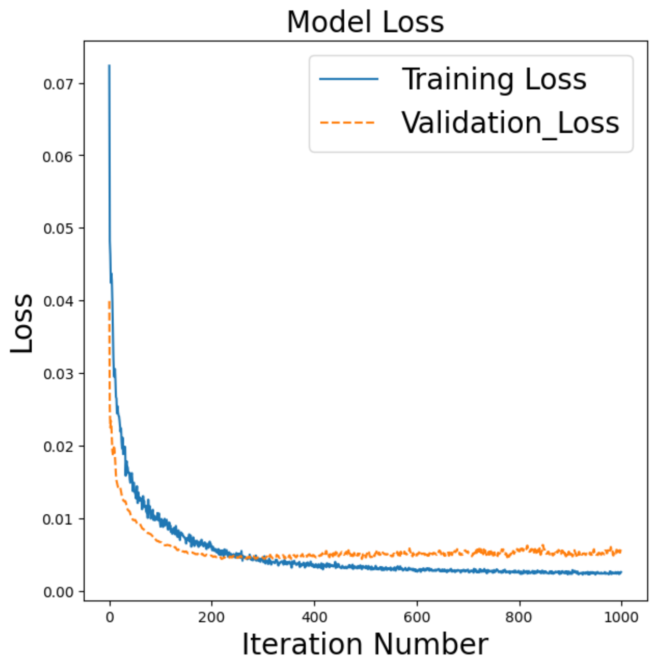

Figure 7: Epoch vs. Loss Plot for both validation and training loss

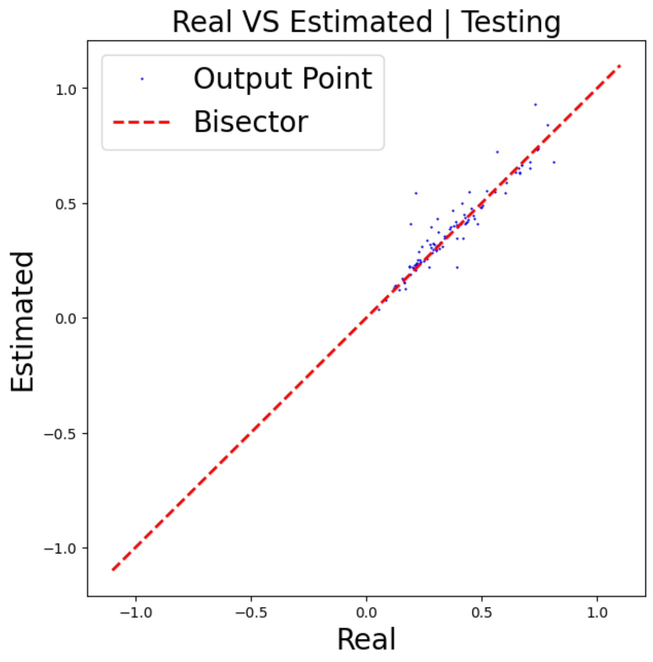

Figure 8: Training vs. Estimated Values on the Calibrated Domain

Most of the Epoch vs. Loss Plot appears to be well fitted, nevertheless, Figure 7 shows a gap between the validation and training loss and it appears that those two lines will not converge. This suggests that the model is under fitted for this specific RTT 0.1 and RRS 003. For the Training vs. Estimated Value graphs, all showed that the estimated data fits nicely with the real data. The output points are around the bisector, and while some deviate more than others, in general they maintain the same trend.

7. In step 7 we created a combinatory run that looks for the best combination of inputs while making sure that water, cement, and one of the three additives are part of the best combination.
This was the report and top 10% selection rates that this process yielded:

```
Best Combinations and Their Corresponding Accuracies
----------------------------------------------
              Cement   Blast Furnace Slag   Fly Ash   Water    \
DNN Training       1                    0         1        1
DNN Testing        1                    0         1        1

              Superplasticizer   Coarse Aggregate    Fine Aggregate   \
DNN Training                 0                  1                 1
DNN Testing                  0                  1                 1

              Accuracy Training  Accuracy Testing
DNN Training           0.057742          0.062579
DNN Testing            0.062579          0.057742
```

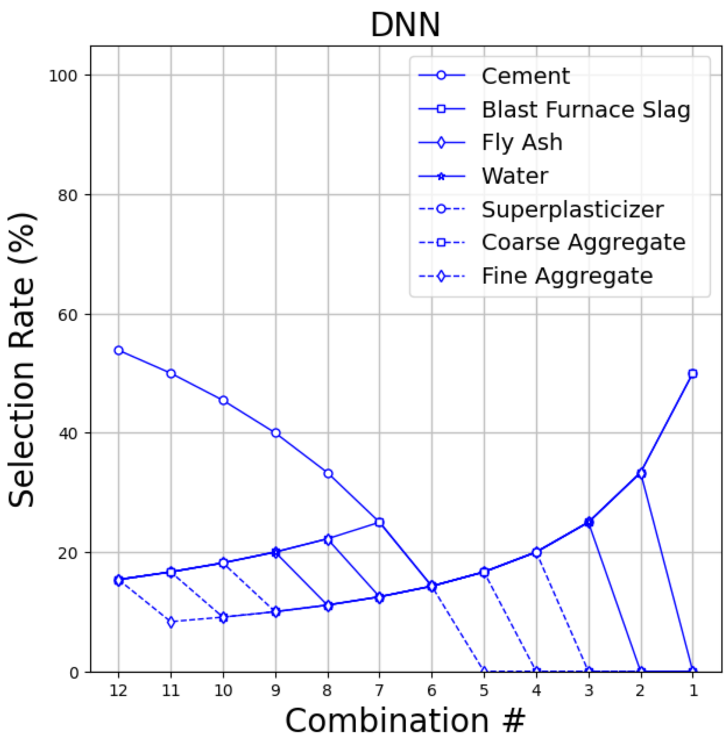

Figure 9. Combination vs Selection rate % based on the DNN model.

The best combination display is defined as follows: `[ True, False,  True,  True, False, True,  True])` where water and cement are present as well as at least one of the three additives. The accuracy was measured by finding the lowest MSE for training and testing. We choose to use MSE since this is a regression problem.

8. In step 8, we took the best combination and used that to pick the proper columns from our initial data (1000 points), and we used the same dataou that was previously used in step 6.
RTT 0.1 | RRS 001
mse_tr - mse_te
0.0026168432970569803-0.005039213383453822

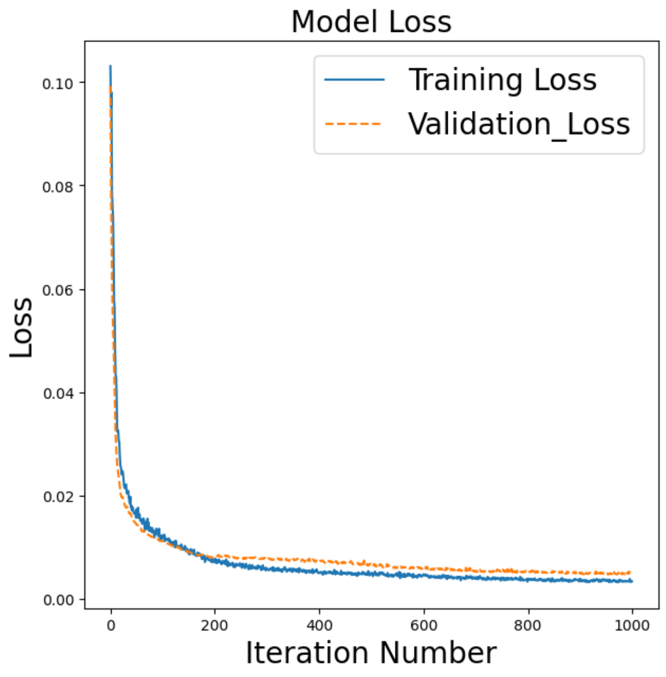

Figure 10: Epoch vs. Loss Plot for both validation and training loss

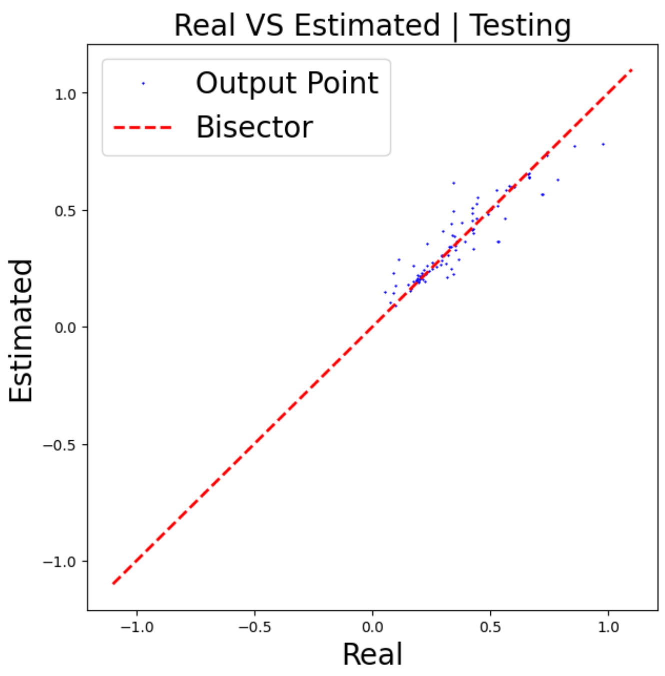

Figure 11: Training vs. Estimated Values on the Calibrated Domain

RTT 0.1 | RRS 002
mse_tr - mse_te
0.0028300481906798-0.006768298572833726

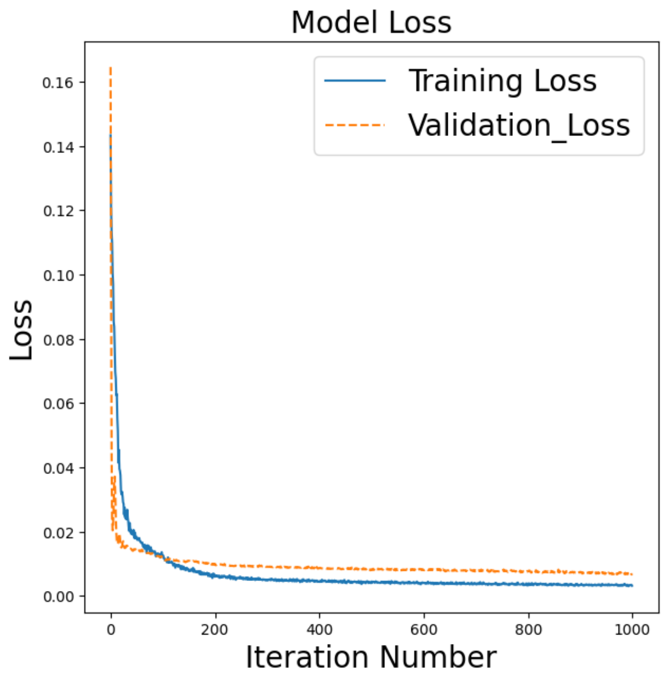

Figure 12: Epoch vs. Loss Plot for both validation and training loss

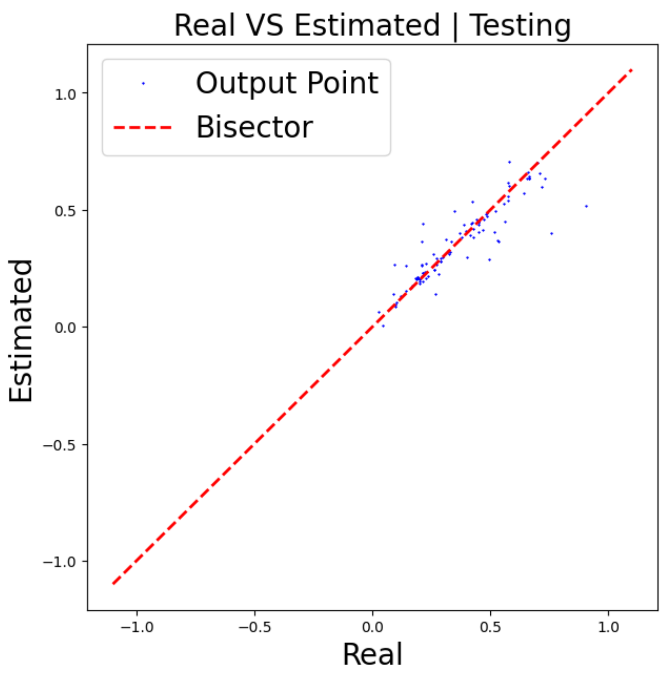

Figure 13: Training vs. Estimated Values on the Calibrated Domain

RTT 0.1 | RRS 003
mse_tr - mse_te
0.002771484002118917-0.006183610843704744

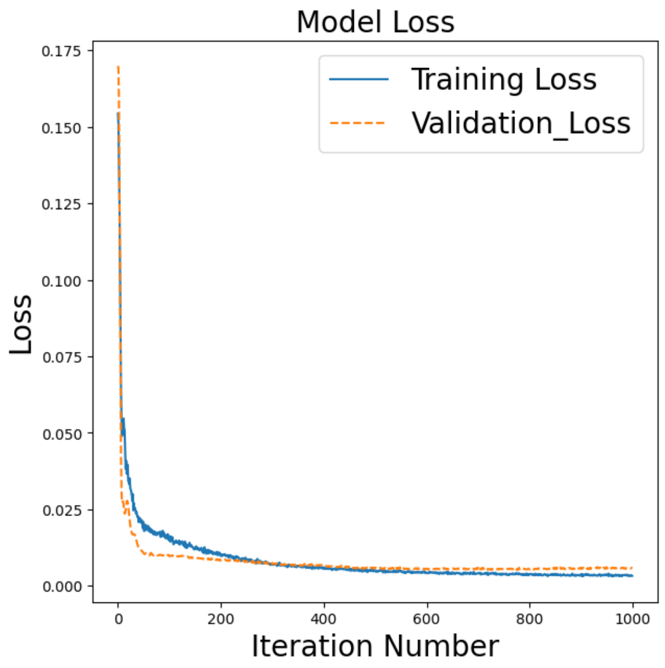

Figure 14: Epoch vs. Loss Plot for both validation and training loss

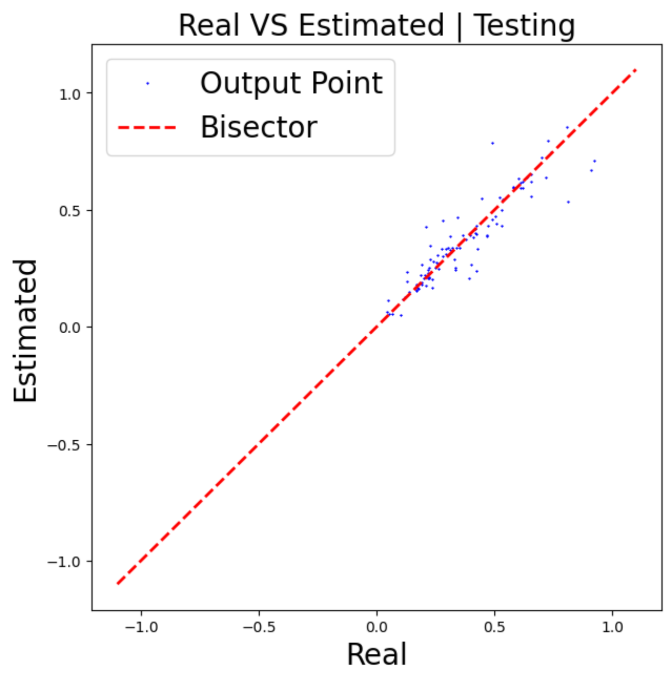

Figure 15: Training vs. Estimated Values on the Calibrated Domain

Most of the Epoch vs. Loss Plots appear to be well-fitted, nevertheless, Figure 12 shows a gap between the validation and training loss and it appears that those two lines will not converge. This suggests that the model is under fitted for this specific RTT 0.1 and RRS 002. Something similar happened in step 6 with Figure 7. Something interesting is that this new model shows a better fit for RTT 0.1 RRS 003 (Figure 7 vs Figure 14). Finding the best combination of inputs in step 7 seemed to improve the model when it was re-run. The Training vs. Estimated Value graphs all showed that the estimated data follow the general trend of the bisector.

9. In step 9 we use the minimization function from the Scipy library to minimize the ratio of mix weight to 28-day compressive strength of the concrete mixture. In the objective function, we bring in a NumPy array of 7 values which represent calibrated input data for the trained model in step 8. The trained model from step 8 only accepts 5 inputs as it has been created and trained on only the best combination of data. This set of data does not include blast furnace slag or coarse aggregate. The first step in the objective function is to create a new vector with only the 5 inputs that we care about using in the trained model. We then pass this input vector to the trained model from step 8 and get back a calibrated value for compressive strength. We then uncalibrated both the compressive strength and the original input vector with all 7 values. With the uncalibrated input vector, we take the sum of the values which gives us the weight of the concrete mixture. We then calculate the ratio of weight to compressive strength. The constraints for this are that weight must be equal to or between 2300 and 2500 kg/m^3 and the ratio of water to cement must be equal to or between 0.48 and 0.59. These constraints were added to the minimization function call as a list of 4 constraint functions. Each of these represents a single inequality of the four above. In each, the input vector is uncalibrated and then compared against the respective inequality or for the ratio, calculated and then compared against the inequality. The bounds are that all elements in the input vector must be between 0 and 1 and the initial input vector is generated randomly. The solution vector X in the calibrated and uncalibrated domain as well as the final ratio are shown below:

	Cement (component 1)(kg in a m^3 mixture)	Blast Furnace Slag (component 2)(kg in a m^3 mixture)	Fly Ash (component 3)(kg in a m^3 mixture)	Water (component 4)(kg in a m^3 mixture)	Superplasticizer (component 5)(kg in a m^3 mixture)	Coarse Aggregate (component 6)(kg in a m^3 mixture)	Fine Aggregate (component 7)(kg in a m^3 mixture)	Weight/Compressive Strength
Calibrated	0.74579000	0.14646000	0.00000053	0.67121660	0.88598000	0.11316000	0.37735600	21.5015
Uncalibrated	428.66	52.637000	0.000107	205.83	28.528000	839.93	744.41	

__Discussions:__

b. A paragraph (3-4 lines) or more to answer this question: our problem is a "maximization" problem. Hence, we proposed our objective function to compute *mix weight* **28−***day compressive strength* **and by minimizing it, we are maximizing 28−***day compressive strength mix weight***. Was that a good strategy? What other objective functions could we define?**

This strategy does work, however, it is not the only objective function we could have used. This also pushes us to minimize the weight of the materials and also maximize the compressive strength. It could push us to a result that is both light in weight and also strong, but perhaps not the strongest value. Other objective functions could just limit the weight of particular materials and maximize the compressive strength. Or we could cap the total weight and maximize the compressive strength of the final product.

c. A paragraph (minimum four lines) or more to answer this question: In steps 4 and 5, instead of GAN, would that be OK to generate our 575 new calibrated input data points randomly, where each input has a value between 0 and 1? What if our data were images?

It wouldn't be good practice to generate the 575 new points randomly. This could potentially skew the training of the neural network to predict incorrect or impossible compressive strength values for the given input data. The same is true for data for images. If data were generated randomly for pixels, the pixels would distort the image and it is unlikely that the data would represent what the user is trying to train their model to recognize. This could work if the user only wants data for the model to test against and prove that the image is discarded, but not to train against.

d. A paragraph (minimum four lines) or more to answer this question: Would our optimization answer be more reliable if we had only used the initial 425 data points to train a model in step 8 rather than 1000 data points? Why?

The optimization answer would not be more reliable if we only used 425 data points rather than 1000. Using the 1000 data points is superior to the 425 data points by themselves as long as the GAN is properly trained and produces realistic data. Having more data is helpful for training and testing neural networks as long as the network has the depth and the users have the time required to train a higher-fidelity model. If we have less data, we have less variety to train the model against. We can train the model on the same set of data for more RRSs, but if we do that for too many iterations we will run into overfitting. More data allows us to train the model over more iterations without the risk of overfitting.

e. A paragraph (minimum four lines) or more to answer this question: In the combinatory model, we used a DNN with a simple architecture to be trained for a few epochs to manage our computer intensity on Google Colab. In our combinatory model, if we had to use a neural network configuration similar to the one in step 6, and with 5 RTTS and RRS = 50, how would you distribute your computation load on multiple nodes and cores on MARCC?

If we could run this on MARCC we could use enough nodes to compute each RTT-RRS independently. There are many ways to do this, but one way could be to use 25 nodes, each running 10 cores. This would allow each RTT and RRS to run on a separate core and could potentially decrease the time to run by 250 times. If fewer nodes are available, then we could distribute our RTTs and RRSs across 20 nodes using 12 cores each with one additional node using only 10 cores. If we do not have enough access to run each RTT and RRSs at once, we could distribute batches of RTTs and RRSs across the nodes and cores we have available.

f. A paragraph (minimum four lines) or more to answer this question: How can we validate our optimization solution's estimated objective value?

The best way to validate the prediction would be to build a test block of this concrete mixture, however, that could be timely and costly. The next best way to validate the optimization problem is to investigate the values that the optimizer is trying to solve. One good indicator is that the total weight of the value our optimizer found is almost exactly 2300 kg/m^3, which is right up against the lower limit. From that, we can already tell that the solution is trying to minimize the ratio of weight to strength. The next step is to check if the solution is maximizing the compressive strength value. For that, we could take all of the data across our 1000 data points and sift for data points that fall within the constraints of our minimization problem. Next, we can compare the compressive strength of those data points to the data point found by the optimization function. Next would be to check the ratio of weight to compressive strength for all data points within our constraints that are in our data set. We want to make sure that the optimizer isn't just maximizing the strength or minimizing the weight but looking for the best of both. A good way to see if that is the case is to plot the iterations of the optimization function as it seeks a solution. It should be clear to see that the ratio of weight to strength is what is targeted, not just one or the other. Finally, I would plot all data points that exist within our constraints and their respective ratios and confirm that our optimization value falls to near the minimum.