Mason Rodriguez Rand and Chris Simotas 11/15/2023

ME 249 Project 3 Report

1. Introduction

As the world embarks on the journey towards a sustainable future, solar panels have and will furthermore become a vital part of our energy future. With advancements in solar cell technology slowly plateauing, engineers have been looking towards new methods to maximize their performance. One method includes utilizing machine learning to model and predict their performance. Through this modeling practice, we can better tune solar panel systems to maximize their power output.

Specifically, this project explores two possible applications: 1) using machine learning models trained on low solar intensity data to see if they can predict accurate power outputs when provided high solar intensity data and 2) using machine learning models to predict the optimal solar panel mode configuration to drive performance under varying conditions. To experiment with these applications, we trained machine learning models on a few key input and output parameters. The input parameters consisted of the outside air temperature (T_{air}), incident direct normal solar radiation intensity (I_D), load resistance (R_L), and the categorical parameter of Mode (M). The outputs were voltage to load (V_L) and power output (W).

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 3:

- Part 1
    - Task 1.1: Chris completed task and wrote discussion
    - Task 1.2: Mason completed task and wrote discussion
    - Task 1.3: Chris completed task and Mason wrote discussion
- Part 2
    - Task 2.1: Mason completed task and wrote discussion
    - Task 2.2: Chris and Mason built model and both wrote discussion

3. Part 1: Feasibility of "Extrapolating" Machine Learning Model Outside Its Trained Scope

For our first machine learning application, we explored the effectiveness of a machine learning model trained on solar radiation (I_D) below 1300 W/m^2 correctly predicting the power output of the system when given solar radiation values outside its scope, or specifically high flux solar radiation values greater than 1300 W/m^2. This can be a useful application for solar system operators if there is a lack of intense solar days to gather enough data for those conditions. Our data for this task consisted of T_{air}, I_D, and R_L as our inputs and V_L and W as our outputs.

3.1. Program Modifications for Task 1.1

While performing Task 1.1, we standardized the dataset to better evaluate its covariance:

DS3.1.1LowfluxF23

```
xarray= np.array(xdata)
yarray= np.array(ydata)

# Calculate the median for each column
mean_values_x = np.mean(xarray, axis=0)
std_values_x = np.std(xarray, axis=0)
mean_values_y = np.mean(yarray, axis=0)
std_values_y = np.std(yarray, axis=0)

xstand = (xarray-mean_values_x)/std_values_x
ystand = (yarray-mean_values_y)/std_values_y
```

We then modified the program to find the matrix's eigenvalues:

P3pcaExampleF23

```
# Find Covariance

C = np.cov(xstand.T)  #transpose is matrix we want to work with - compute covariance matrix
print (C)
```

```
# Find Eigenvalues and Eigenvectors

w, v = LA.eig(C)  # get the eigenvalues w and the eigenvectors
```

3.2. Discussion of Covariance of Low Flux Solar Panel Data

Before diving into training the model, we first examined the covariance or independence of the input data parameters to each other to evaluate the strength of our data. Ideally when building a model, you would like to use parameters that are not dependent on each other. For instance, if you have two variables which increase and decrease with each other, then the second variable is essentially a duplicate of the first variable. It does not provide new insight into the relationship for the model to capitalize on. Thus, the independence of variables is a key modeling best-practice.

To examine the covariance of the input dataset, we computed the eigenvalues of our covariance matrix of our dataset standardized with the help of some python tools as shown in **Table 1**. We also plotted all of the standardized input data points against each other as shown in **Figure 1**.

Table 1. Eigenvalues of Low Flux Solar Panel Data

Eigenvalue	Value
1	1.02857143
2	1.02857143
3	1.02857143

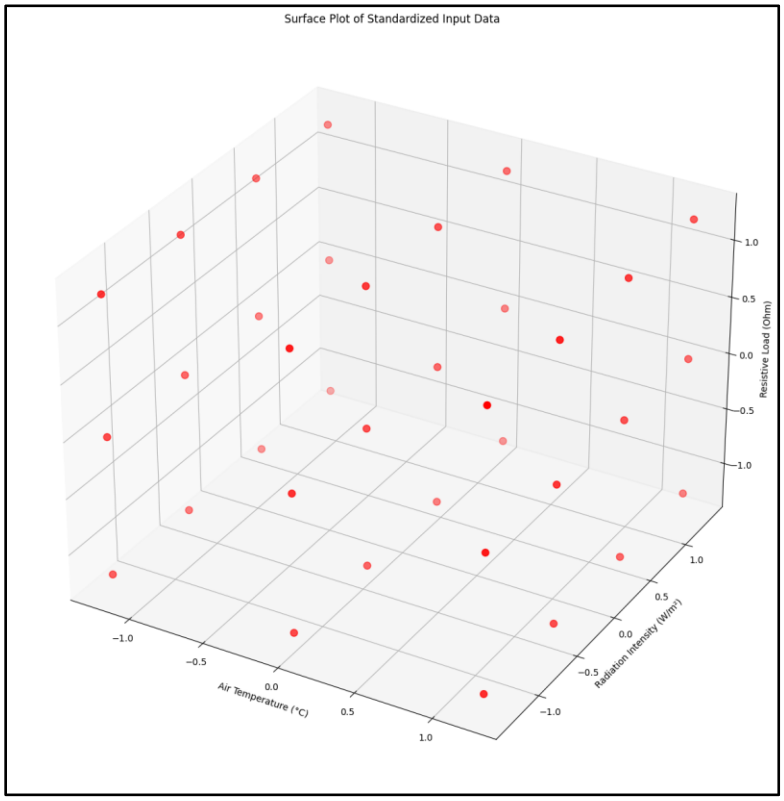


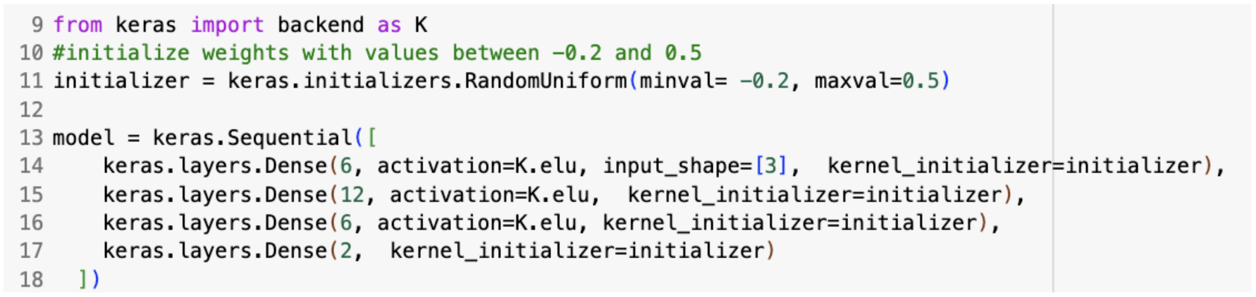
Figure 1. Surface Plot of Standardized Air Temperature, Radiation Intensity, and Resistive Load Data

At a high-level, eigenvalues describe key behaviors of matrices. In this case, the eigenvalues of our covariance matrix provided information about the spread or variability of the data along different directions in the feature space. All three of our eigenvalues were equivalent, suggesting that the spread of our data was roughly equal in all directions. In other words, there were no underlying dependencies between the input parameters. The variables were not correlated. They all provide important descriptors for the relationship being modeled. Visually, this pattern is seen in **Figure 1**. There are no data points clustering in certain regions of the data space. They are all evenly spaced from each other. Thus, the dataset appears to have a low covariance and they all can be considered of equal importance.

3.3. Program Modifications for Task 1.2

We modified the below neural network structure to begin modeling the solar panel's performance.

CodeP3.1.2F23



```
from keras import backend as K
#initialize weights with values between -0.2 and 0.5
initializer = keras.initializers.RandomUniform(minval= -0.2, maxval=0.5)

model = keras.Sequential([
    keras.layers.Dense(6, activation=K.elu, input_shape=[3],  kernel_initializer=initializer),
    keras.layers.Dense(12, activation=K.elu,  kernel_initializer=initializer),
    keras.layers.Dense(6, activation=K.elu, kernel_initializer=initializer),
    keras.layers.Dense(2,  kernel_initializer=initializer)
  ])
```

3.4. Discussion of Low Flux Model with 6-12-6 Layer Structure (Task 1.2)

To begin training a machine learning model for solar panel application, we started with a four-layer dense neural network. We then trained it on a normalized dataset for low and high solar radiation intensity conditions and analyzed the results. The low flux solar radiation data was split into training and validation sections.

Table 2. Mean Absolute Relative Errors of Datasets

Dataset	Mean Absolute Error	Mean Absolute Relative Error (%)
Training Data	1.68	1.90
Validation Data	9.92	11.6
High Flux Data	12.85	9.32

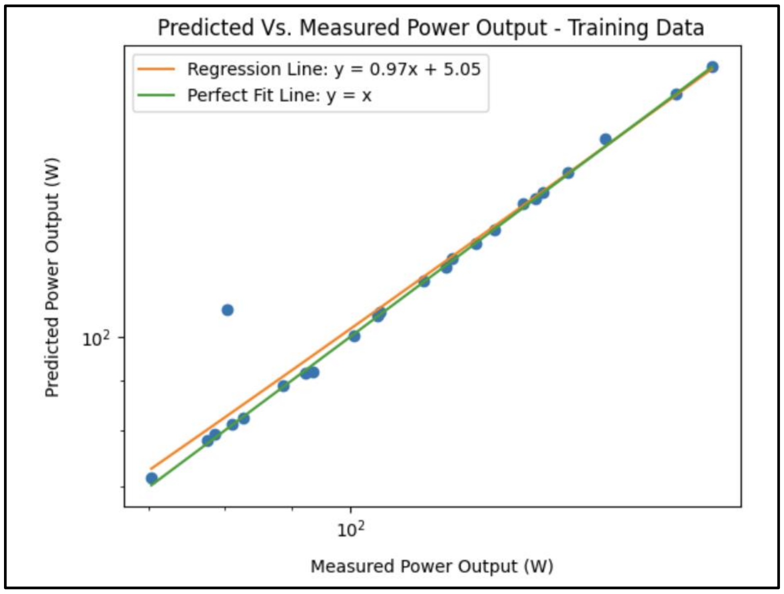


Figure 2. Predicted vs. Measured Power Output Using Training Data

First examining the model's performance against the training dataset, the predicted power output is proportional to the measured power output, with a slight adjustment (the slope is slightly less than 1, and there is a positive y-intercept). The green line is the line of perfect prediction, where the predicted power output equals the measured power output (y = x). This line is the benchmark for an ideal predictive model. The closeness of the regression line to the perfect fit line indicates that the predictive model is performing well. However, since the regression line's slope is less than 1, the model tends to slightly underpredict the actual power output. The positive y-intercept (5.05) also suggests that for very low values close to zero, the model predicts a power output higher than what is measured. It is also important to point out how there is one distinct outlier in this graph (more on this in Section 3.6).

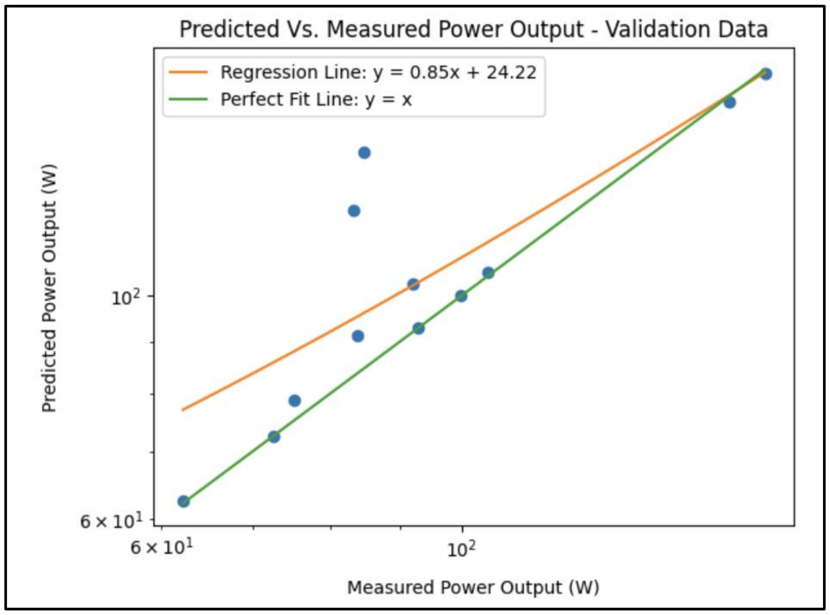


Figure 3. Predicted vs. Measured Power Output Using Validation Data

The regression line on the validation graph has the equation y = 0.85x + 24.22. This indicates a flatter slope than the training data's regression line (which had a slope of 0.97). The model predicts a lower value for a given measured power output compared to the training data. The positive y-intercept is significantly higher in the validation data (24.22 vs. 5.05 in the training data). This suggests that when the measured power output is very low (approaching zero), the model predicts a much higher power output in the validation set than it did in the training set. The data points in the validation graph seem to be more spread out from the line of perfect fit compared to the training data. This spread suggests more variance in the model's predictions when applied to the validation data. The differences between the regression line and the perfect fit line, as well as the increased spread of data points, suggest that the model may not generalize as well to the validation data as it did to the training data. This could be a sign of overfitting to the training data, or it could indicate that the validation data has different characteristics that the model did not learn to predict as accurately.

This observation is supported by the performance metrics in **Table 2**, which show a mean absolute error (MAE) of 9.92 for the validation data, markedly higher than the MAE of 1.68 for the training data. The mean absolute relative error (MARE) presents a similar trend, with the validation data showing a MARE of 11.6% compared to just 1.90% for the training data. These metrics suggest that the model, while reasonably accurate on the training data, fails to maintain this performance on the validation data, indicating overfitting issues or a lack of generalizability. The significant increase in both absolute and relative errors points to the necessity for model refinement or the development of a new model that can better capture the underlying patterns in the data and thus perform more consistently across different datasets.

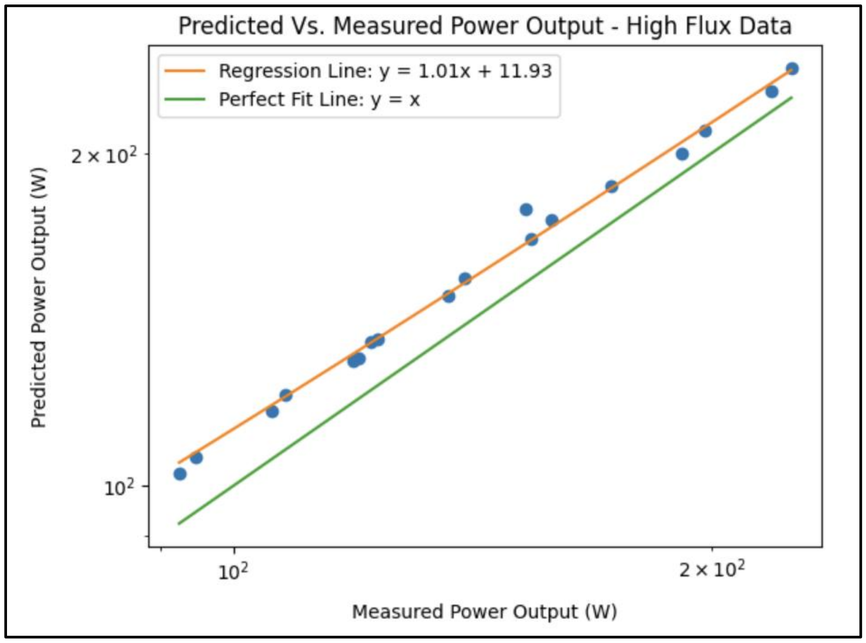


Figure 4. Predicted vs. Measured Power Output Using High Flux Data

The predictive model closely approximates the measured power output for the high flux data, as indicated by a regression line with a slope of 1.01 and a y-intercept of 11.93. This near unity slope suggests a strong correlation between predicted and measured values, which is further supported by the data points' proximity to the perfect fit line (y = x). However, the mean absolute error of 12.85 and the mean absolute relative error of 9.32% suggest that there is still some discrepancy between predictions and actual measurements. The positive y-intercept indicates a systematic overestimation for lower values, which could imply a bias that needs addressing. This could involve tuning the initial weight range, as it is adjusting hyperparameters, such as the number of neurons in each layer, the activation functions, the learning rate, and implementing regularization techniques. We experimented with some such adjustments in Section 3.6 (Task, 1.3).

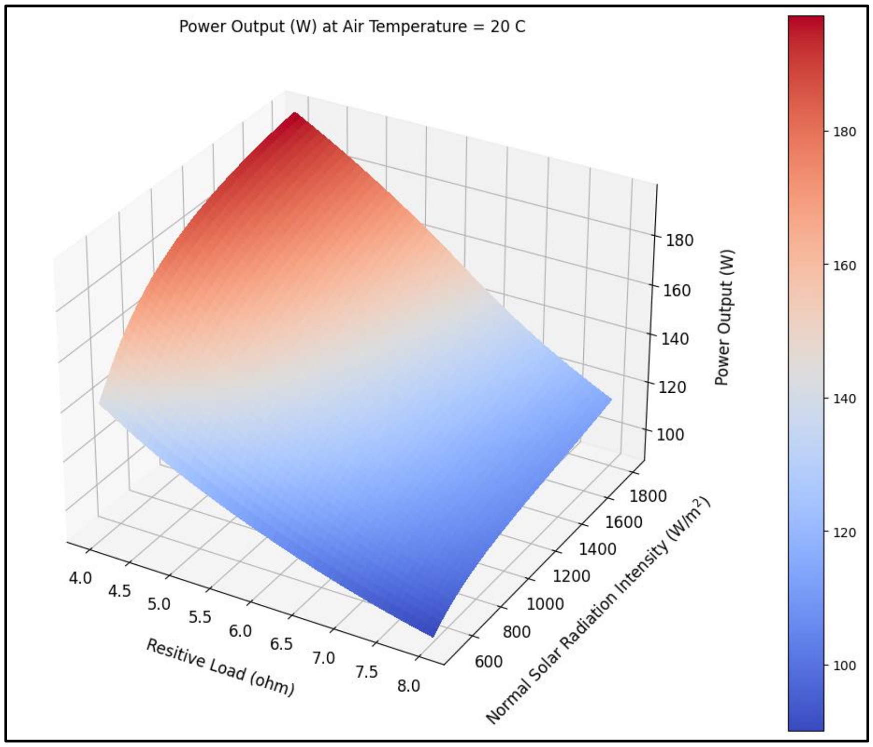


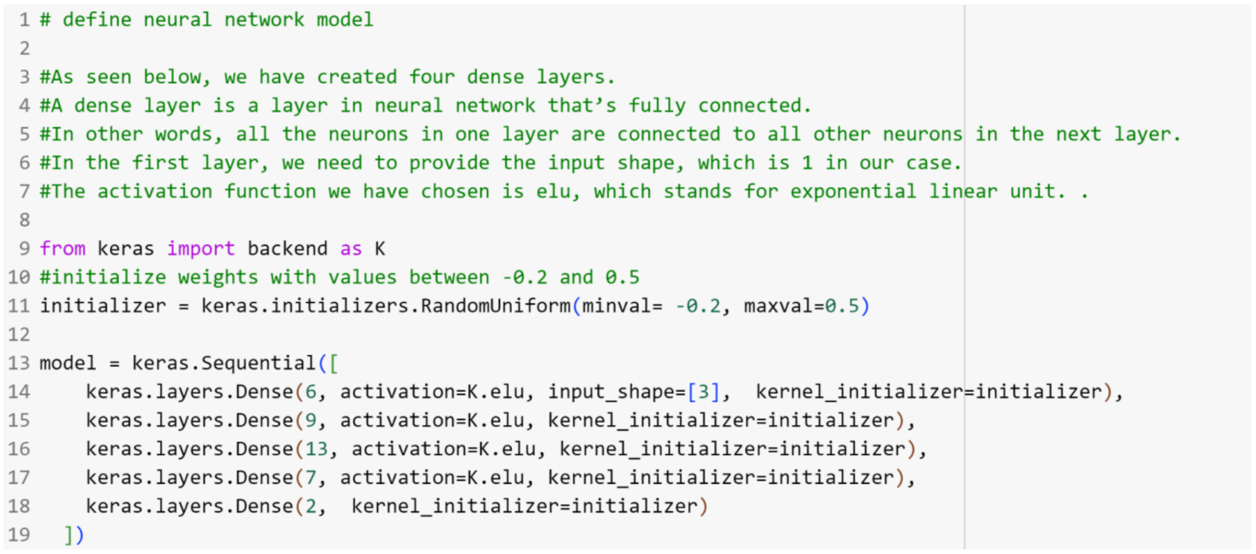
Figure 5. Power Output vs Load & Radiation Intensity

Figure 5. presents a three-dimensional surface that models the relationship between resistive load, solar radiation intensity, and power output at a constant air temperature of 20°C. The surface's slight upward curvature towards the higher end of solar radiation intensity suggests a non-linear increase in power output, indicating a more pronounced effect on power generation as solar intensity reaches the upper end of the measured spectrum. This non-linearity, although subtle, could be indicative of the photovoltaic system's increasing efficiency or a threshold beyond which the power output rises more steeply. The plot also hints that changes in resistive load have a less pronounced, yet still observable, impact on power output across the observed range, aligning with Ohm’s law which states that power should increase with decreasing resistance if voltage remains consistent. Consequently, while the trends largely appear linear, the graph's subtle non-linearities along both axes indicate that it might be useful to include non-linear dynamics in real use of this model, such as the integration of a polynomial or higher-order terms to capture the increased rate of power output growth with rising solar radiation intensity and decreasing resistive load.

3.5. Program Modifications for Task 1.3

We modified the model to incorporate a 6-9-13-7 neural network structure to explore its performance.

CodeP3.1.2F23



```python
# 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 0.5
initializer = keras.initializers.RandomUniform(minval= -0.2, maxval=0.5)

model = keras.Sequential([
    keras.layers.Dense(6, activation=K.elu, input_shape=[3],  kernel_initializer=initializer),
    keras.layers.Dense(9, activation=K.elu, kernel_initializer=initializer),
    keras.layers.Dense(13, activation=K.elu, kernel_initializer=initializer),
    keras.layers.Dense(7, activation=K.elu, kernel_initializer=initializer),
    keras.layers.Dense(2,  kernel_initializer=initializer)
  ])
```

3.6. Discussion of Low Flux Model with 6-9-13-7 Layer Structure

After examining the performance of the previous machine learning model, we decided to attempt to improve it. Specifically, we wondered if an increase in the number of layers and neurons would improve the model’s performance in predicting power output for high flux solar data. Thus, we added an additional layer to create a 6-9-13-7 neural network model to train against the low flux solar data. To aid in comparing the two models, we trained this model until it approached a similar mean absolute relative loss value of the prior one for the training data. Once again, the model was validated against the training, validation, and high flux data. **Table 3** shows the mean absolute relative errors (MARE) of each of the datasets. **Figures 6, 7,** and **8,** show the predicted power outputs against the measured power outputs for each of the datasets as well.

Table 3. Mean Absolute Relative Errors of Datasets

Dataset	Mean Absolute Relative Error (%)
Training Data	2.83
Validation Data	11.8
High Flux Data	8.38

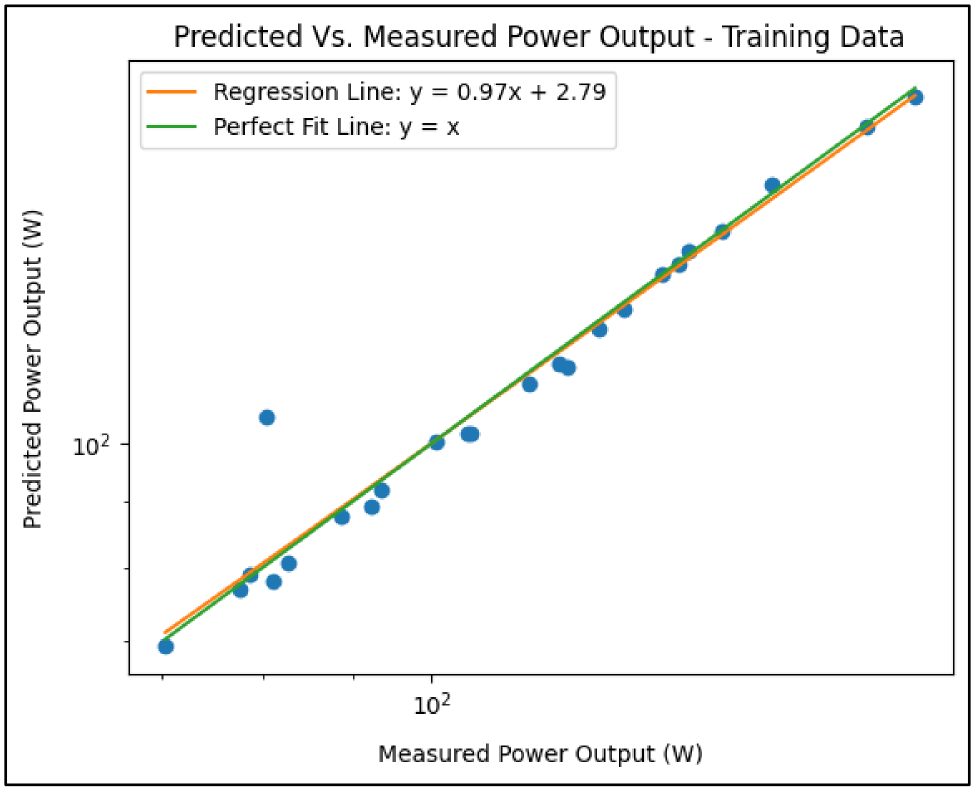


Figure 6. Predicted vs. Measured Power Output Using Training Data

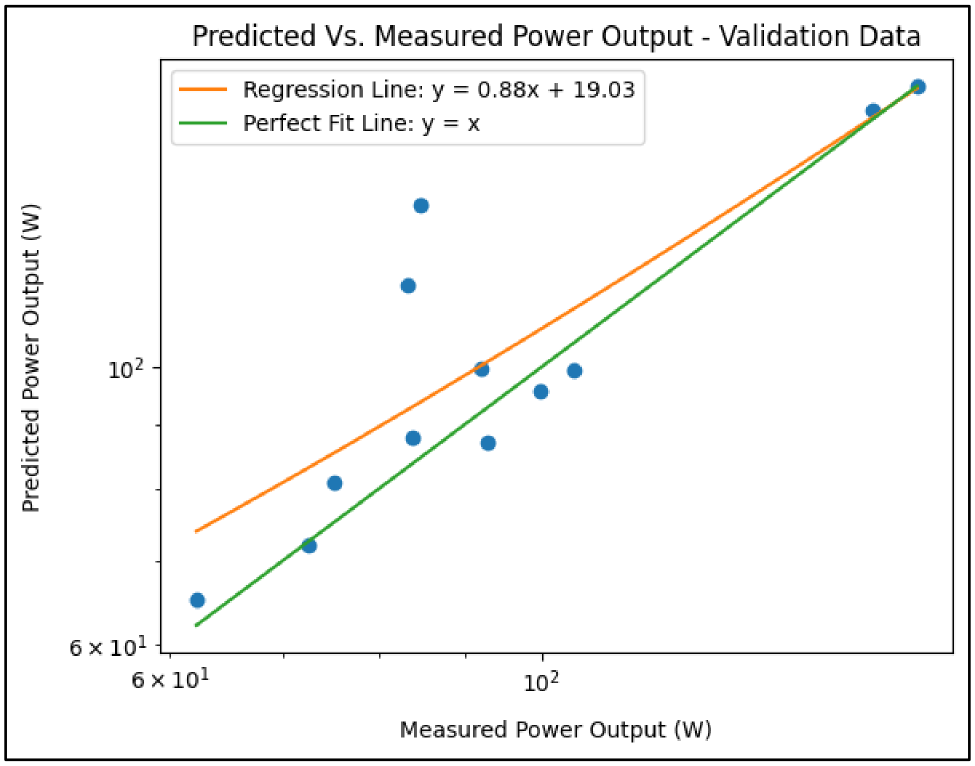


Figure 7. Predicted vs. Measured Power Output Using Validation Data

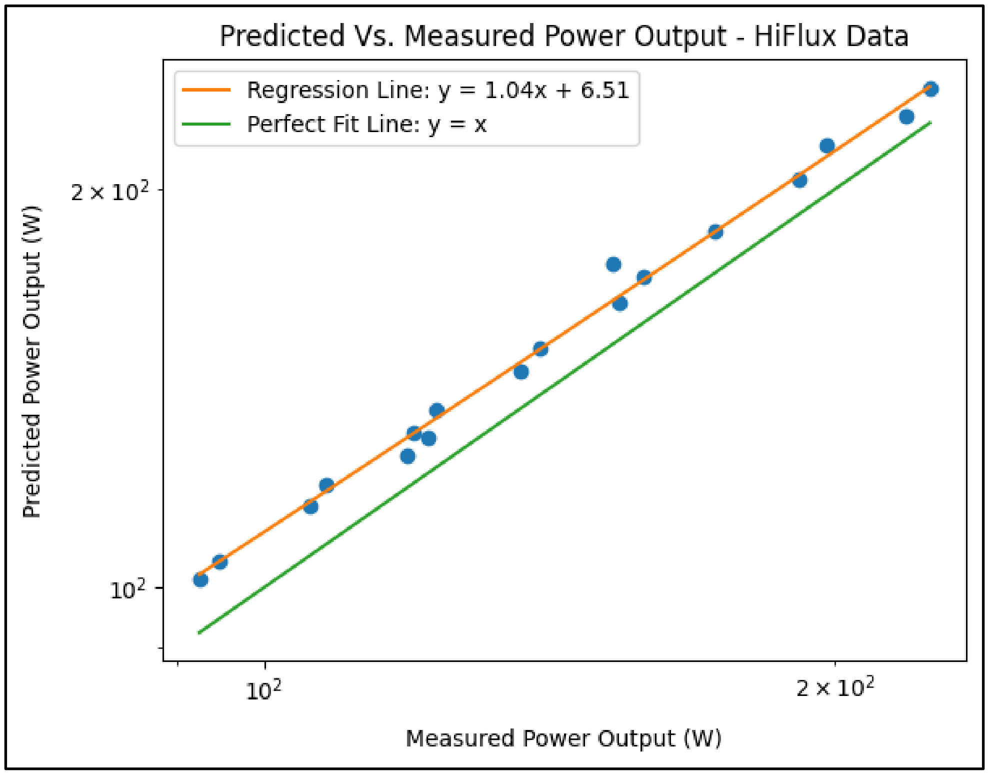


Figure 8. Predicted vs. Measured Power Output Using HiFlux Data

Not surprising, the model against the training data performed very well with a MARE of 2.83% and a regression line slope close to 1.0. The model's predicted power outputs lined up well with their measured counterpart, except for one outlier as seen in **Figure 6**, similar to what happened with the prior model. Since this same outlier has appeared over two different model configurations, it may in fact be a true outlier that the model is struggling to integrate without overfitting.

The model performed decently well for the validation data with a MARE of 11.8%. Majority of the data points shown in **Figure 7** hug the perfect line of predicted power output equaling measure power output, minus the two outliers. Similar to the suggestion before, additional training most likely would have improved the model's performance against this dataset.

Finally, again similar to the prior model, this model performed well with the High Flux data with a MARE of 8.38% and again surprisingly better than the validation data. The regression line in **Figure 8** shows that the model's predictions are consistently off the true values by the same offset. The model is predicting the power output to be greater for the high flux data than measured. Thus, possibly the solar panel's power output performance may taper off at higher flux values; the higher solar intensity may be saturating the solar panels a bit, reducing efficiency.

To gain a clearer picture of the solar panel's power output, we created a surface plot of the resistive load and solar intensity against power output while the air temperature was held constant in **Figure 9.**

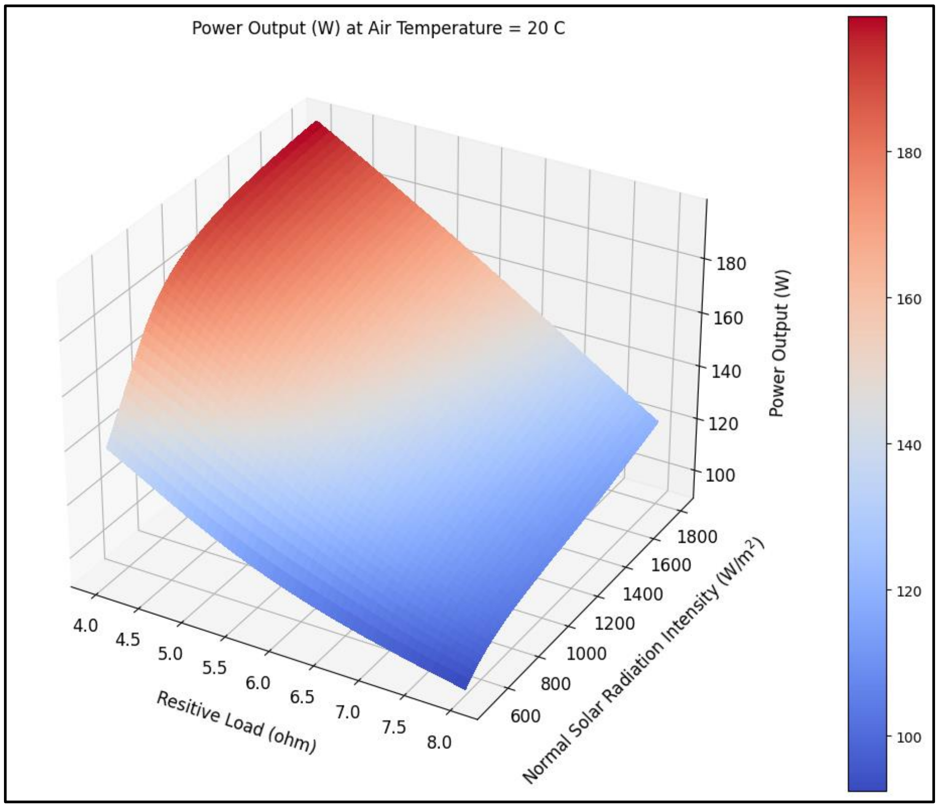


Figure 9. Surface Plot of Resistive Load and Solar Intensity at Constant Outside Temperature of 20 C

Overall, the general flat shape of the surface plot shows a linear relationship among the variables. As either the resistive load or solar intensity increase, the power output increases at a constant rate. Yet, it is important to point out the slight folds or curves in the surface plot at the minimum and maximum values of the input variables. It appears the power output increases at a slower rate at R_L greater than 7.5 ohms and I_D less than 600 W/m^2 than at the other end of the range for both variables. In addition, considering the scaling of R_L and I_D axes, it appears that the R_L has more of an impact on the power output of the solar panel than the I_D . Power output increases at a faster rate per ohm than per W/m^2. Remembering ohm's law, this makes sense. Assuming voltage is mostly held steady, lowering the resistance allows for more current to run through the solar panel and in turn increase power output.

3.7. Comparison of 6-12-6 and 6-9-13-7 Neural Network Performance

Although we changed the model to include more layers with different numbers of neurons in part 1.3, performance was fairly similar, if not only slightly better than our model in 1.2. This is seen in the training, validation, and high-flux datasets exhibiting MAREs of 1.9, 11.6, 9.32 and 2.83, 11.8, and 8.38 for 1.2 and 2.3, respectively. This may be due to training 1.2 over many more epochs (≥3200 total) to get the loss values to similar points as compared with the model in 1.3. In contrast, the model in 1.3 was trained in ≤2000 epochs, indicating its superior efficiency in training time needed to obtain satisfactory loss values. Nevertheless, both models performed nearly identical to each other.

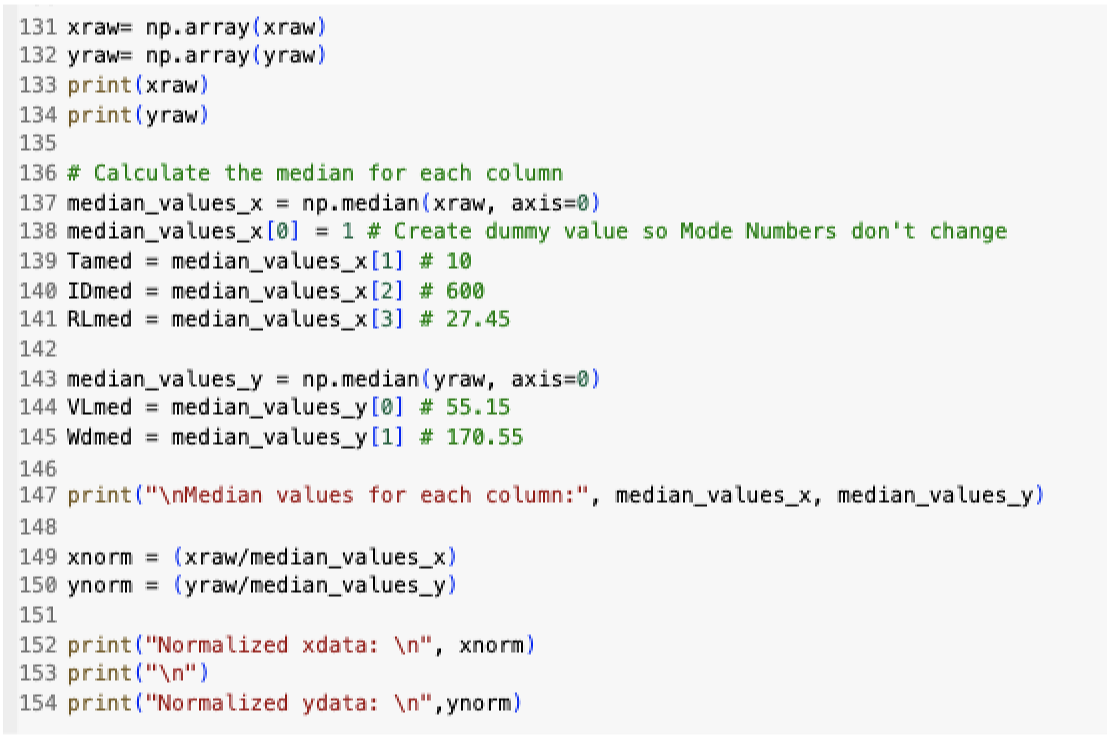
Strong covariance also likely helped with the two models' accuracy as independent variable relationships can help the model to generalize better from the training data to unseen data. In this case, resistive load, and solar radiation intensity had distinct and independent effects on the power output, allowing for better model training performance. We also observed likely overfitting on both validation data graphs, indicating the lower performance in the model's predictions when applied to unseen data. Yet, in the case of the high flux data, we strangely observed a linear graph along the log-log plot indicating strong prediction performance of the model. However, this possibly may be a coincidence or may suggest that the model may be more useful predicting power outputs in the range of greater than 100 W.

4. Part 2

For our second machine learning application, we added another input variable, Mode, to the dataset to seek its influence on the machine learning model and performance on a solar panel system. Mode, a categorical parameter, indicates the three different configurations of the solar panel system. The solar panels can either be in series, parallel, or a hybrid mix of the two. Since each configuration performs optimally in varying operating conditions, we aim to build a machine learning model that can predict this behavior in the hope that it could be incorporated in a control system that sets the configuration of a solar panel system during operation.

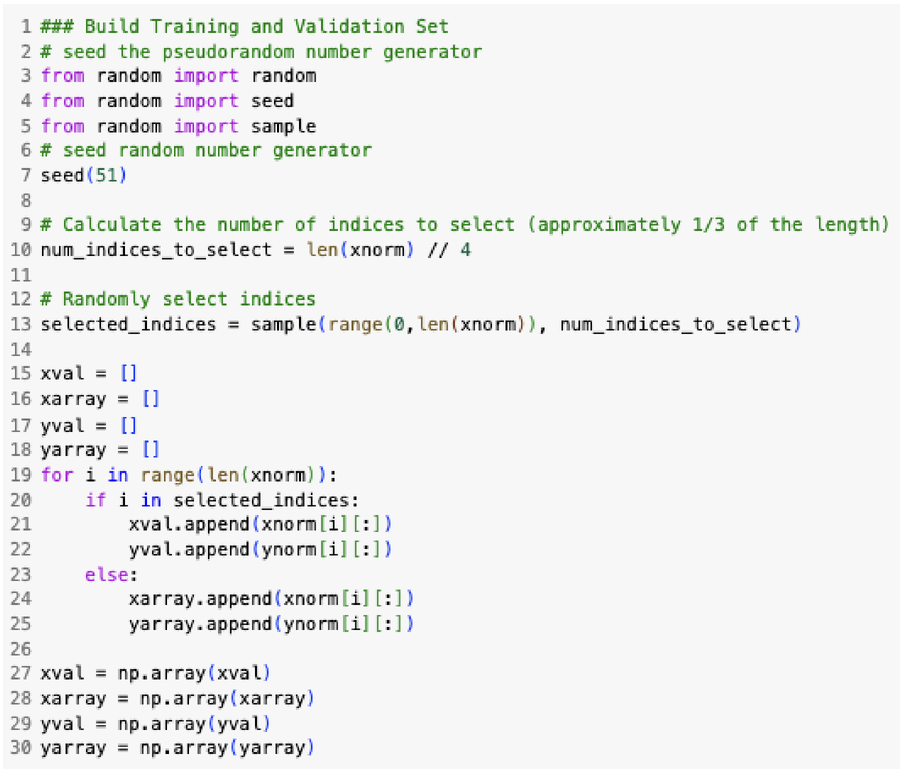
4.1. Program Modifications and Discussion of Neural Network Model with Mode Input (Task 2.1)

We started by normalizing a set of data made up of mode numbers, T_{air}, I_D, and R_L as well as outputs V_L and $\dot{W}$. We normalized everything except the mode numbers as follows:



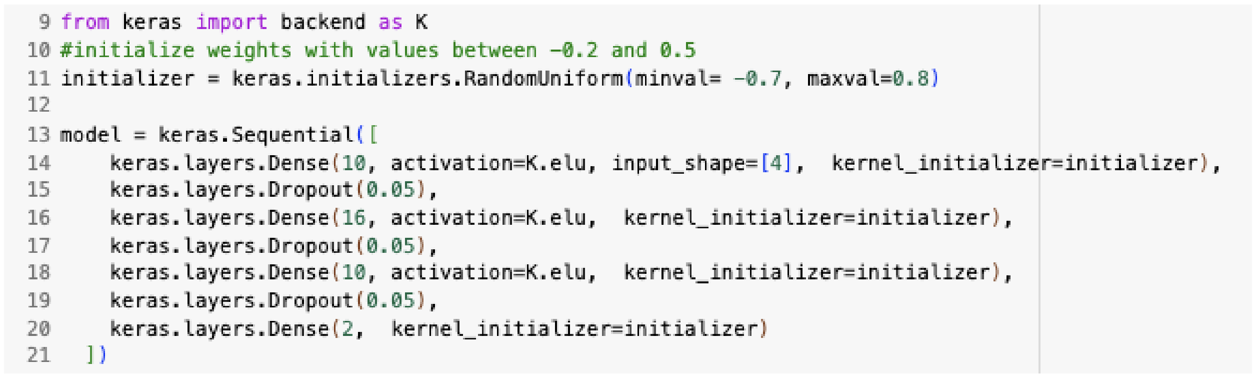
```python
xraw= np.array(xraw)
yraw= np.array(yraw)
print(xraw)
print(yraw)

# Calculate the median for each column
median_values_x = np.median(xraw, axis=0)
median_values_x[0] = 1 # Create dummy value so Mode Numbers don't change
Tamed = median_values_x[1] # 10
IDmed = median_values_x[2] # 600
RLmed = median_values_x[3] # 27.45

median_values_y = np.median(yraw, axis=0)
VLmed = median_values_y[0] # 55.15
Wdmed = median_values_y[1] # 170.55

print("\nMedian values for each column:", median_values_x, median_values_y)

xnorm = (xraw/median_values_x)
ynorm = (yraw/median_values_y)

print("Normalized xdata: \n", xnorm)
print("\n")
print("Normalized ydata: \n",ynorm)
```

We then split the data into a training and testing set with ¾ and ¼ of the initial data selected at randomly going into each respectively:



```python
### Build Training and Validation Set
# seed the pseudorandom number generator
from random import random
from random import seed
from random import sample
# seed random number generator
seed(51)

# Calculate the number of indices to select (approximately 1/3 of the length)
num_indices_to_select = len(xnorm) // 4

# Randomly select indices
selected_indices = sample(range(0,len(xnorm)), num_indices_to_select)

xval = []
xarray = []
yval = []
yarray = []
for i in range(len(xnorm)):
    if i in selected_indices:
        xval.append(xnorm[i][:])
        yval.append(ynorm[i][:])
    else:
        xarray.append(xnorm[i][:])
        yarray.append(ynorm[i][:])

xval = np.array(xval)
xarray = np.array(xarray)
yval = np.array(yval)
yarray = np.array(yarray)
```

Finally we designed our model and included dropout layers. We tried multiple models which are listed below in **Table 5.**



```
from keras import backend as K
#initialize weights with values between -0.2 and 0.5
initializer = keras.initializers.RandomUniform(minval= -0.7, maxval=0.8)

model = keras.Sequential([
    keras.layers.Dense(10, activation=K.elu, input_shape=[4],  kernel_initializer=initializer),
    keras.layers.Dropout(0.05),
    keras.layers.Dense(16, activation=K.elu,  kernel_initializer=initializer),
    keras.layers.Dropout(0.05),
    keras.layers.Dense(10, activation=K.elu,  kernel_initializer=initializer),
    keras.layers.Dropout(0.05),
    keras.layers.Dense(2,  kernel_initializer=initializer)
  ])
```

For our model that performed the best on our validation data, the performances were as listed in **Table 4** below and in **Figures 10 and 11**.

Table 4. Mean Absolute Relative Errors of Datasets

Dataset	Mean Absolute Error	Mean Absolute Relative Error (%)
Training Data	14.2	12.5
Validation Data	105	45.1

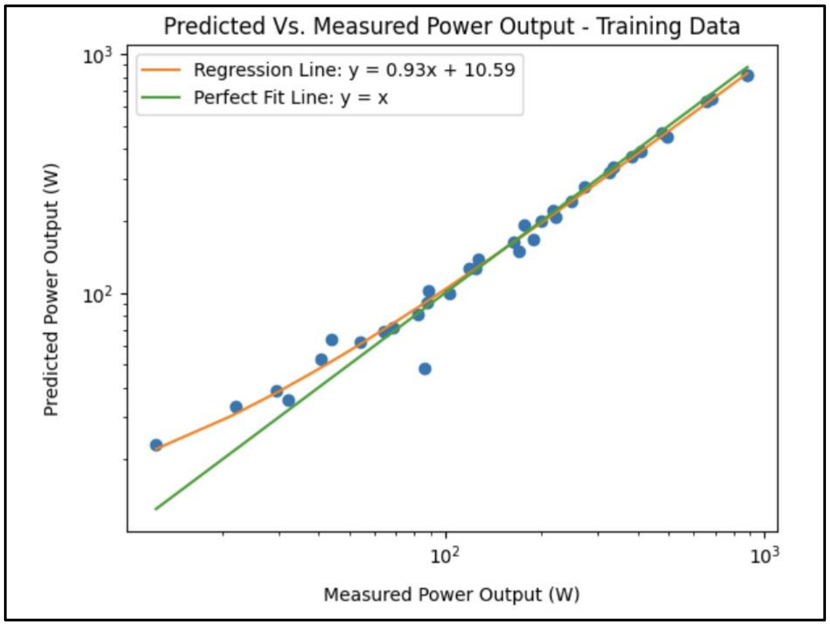


Figure 10. Predicted vs. Measured Power Output Using Training Data

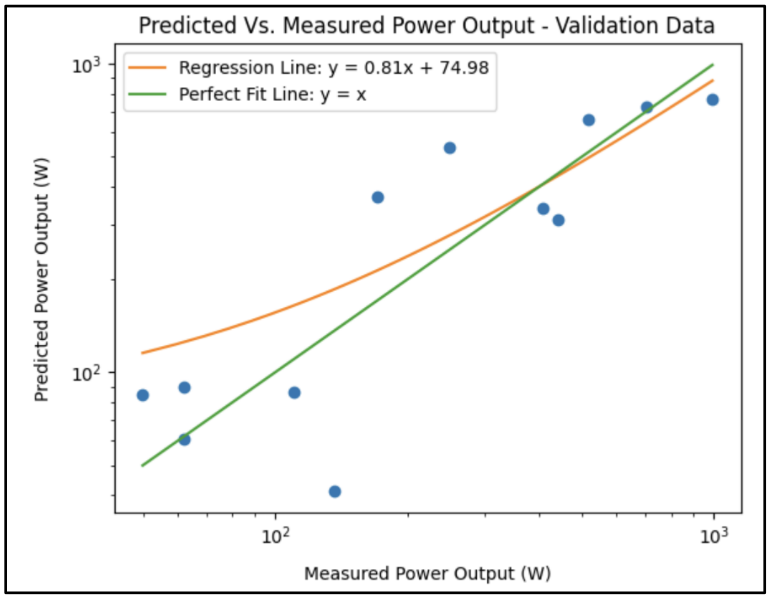


Figure 11. Predicted vs. Measured Power Output Using Validation Data

Both graphs above represent the results from the best performing model at 45.1% mean absolute relative error for the validation data graph. In both graphs, the slopes of the regression lines are less than 1, meaning the model tends to underpredict higher values. The large y-intercept on the validation graph also suggests a significant overestimation of lower values. This discrepancy in performance between the training and validation sets, especially with better performance on the latter, is counterintuitive and warrants a closer examination of the data distribution, feature selection, and model architecture. Far superior performance on the training data compared to the validation dataset also points to potential overfitting, wherein our model happens to be training for specific enough in the training dataset, leading it to perform worse on the generalized validation. We found this same trend of superior training data performance across multiple models as seen in **Table 5** below.

Table 5. Task 2.1 Model Performances Across Designs

Model	**Loss**	**Training Data MARE**	**Validation Data MARE**
10 x 16 x 10 x 2 Neural Network Weight Bounds (-.5, .8) (Dropout Rate of 0.05)	0.097	0.125	0.451
10 x 16 x 10 x 2 Neural Network Weight Bounds (-.5, .7) Dropout Rate of .05	.066	.124	.453
20 x 32 x 32 x 24 x 2 Neural Network Weight Bounds (-.5, .7)	0.037	.062	.67
13 x 26 x 13 x 2 Neural Network Weight Bounds (-.5, .8)	.027	.036	.668

As seen in the table, we found that expanding the min and max bounds on the weights from the initial (-.2, .5) setting to values closer to (-.5, .7) lead to longer model training time (3-4x the number of epochs) but better performance on the validation data overall. When adding dropout layers, we also found setting a larger percentage of neurons to be deactivated lead to far longer training times though greater robustness in our model. We found a balance between training time constraints and model robustness by creating more dropout layers with lower neuron dropout percentages. For example, we initially created models with dropout percentages of 10% or higher. Once these took far too long to train to a somewhat acceptable loss value, we added a third dropout layer and lowered the dropout value to 5%. This converged in a more reasonable timeframe while increasing model accuracy and decreasing mean average error.

We found above in the explanation for **Figures 10** and **11** that our training data test far outperformed our validation data test. This trend held across different model testing. This was likely due to overfitting.

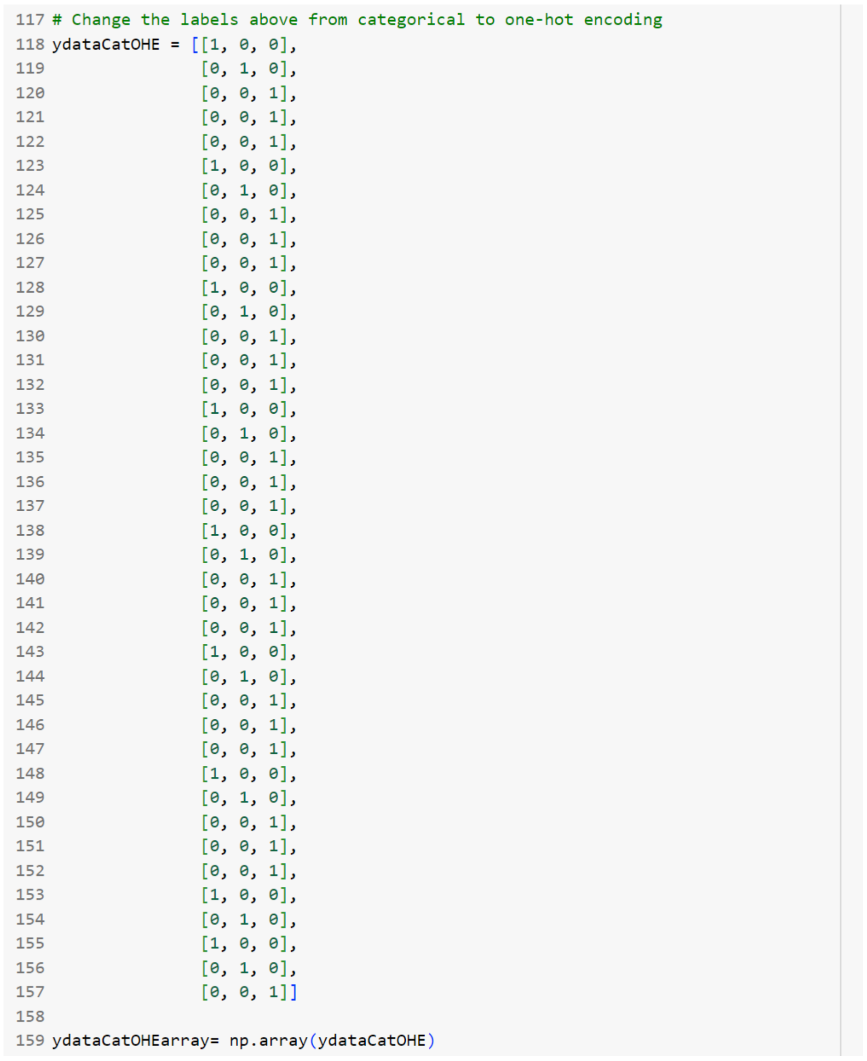
Many different types of models were tried and led to a similar result, further demonstrating this phenomenon. In fact, we found in most cases, especially for our higher order neural network models, that after reducing our MARE for the training data below 10%, we noticed our MARE for the validation data was actually increasing; a clear sign that the model was overfitting

We also found that adding dropout layers caused better performance in the validation data. The training process often takes longer when dropout layers are included but makes the model more robust to diverse test sets. This is exactly what we found in our validation data. As we increased the dropout layers and percentages, our model grew in performance on validation data testing. In the end, we still experienced MARE values above 40% even in the best model scenarios however. While different models and larger training times or epochs may help, we found the models' performance often plateaued. This occurred even when we increased the "search area" midway by adjusting the loss cutoff values, patience, and learning rate. This suggests we may want to try very different models altogether, and may also be limited by the amount of data at hand.

4.2. Program Modifications for Task 2.2

To run the model, we converted the categorical Mode parameter into the One-Hot Encoding format as follows:

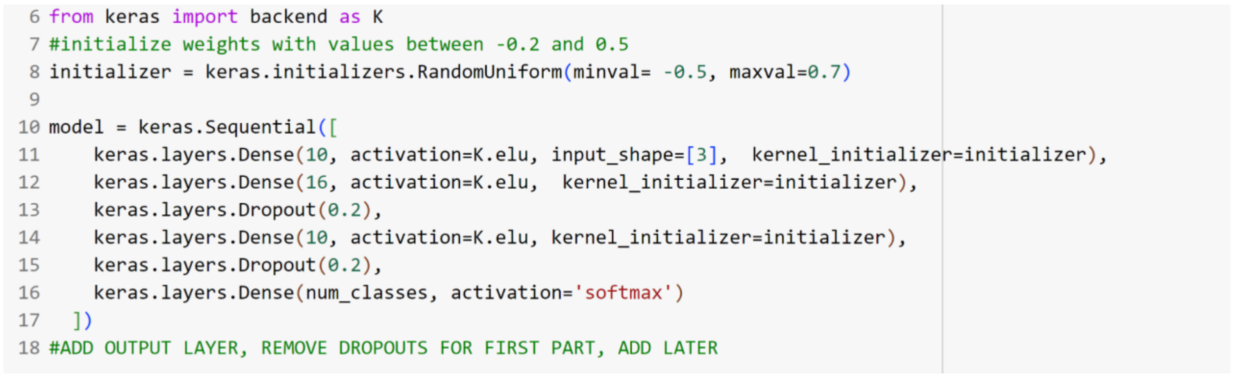
CodeP3.2.2F23



```
# Change the labels above from categorical to one-hot encoding
ydataCatOHE = [[1, 0, 0],
               [0, 1, 0],
               [0, 0, 1],
               [0, 0, 1],
               [0, 0, 1],
               [1, 0, 0],
               [0, 1, 0],
               [0, 0, 1],
               [0, 0, 1],
               [0, 0, 1],
               [1, 0, 0],
               [0, 1, 0],
               [0, 0, 1],
               [0, 0, 1],
               [0, 0, 1],
               [1, 0, 0],
               [0, 1, 0],
               [0, 0, 1],
               [0, 0, 1],
               [0, 0, 1],
               [1, 0, 0],
               [0, 1, 0],
               [0, 0, 1],
               [0, 0, 1],
               [0, 0, 1],
               [1, 0, 0],
               [0, 1, 0],
               [0, 0, 1],
               [0, 0, 1],
               [0, 0, 1],
               [1, 0, 0],
               [0, 1, 0],
               [0, 0, 1],
               [0, 0, 1],
               [0, 0, 1],
               [1, 0, 0],
               [0, 1, 0],
               [1, 0, 0],
               [0, 1, 0],
               [0, 0, 1]]

ydataCatOHEarray= np.array(ydataCatOHE)
```

To optimize our machine learning model using one-hot-encoding, expanded our range for the model's weights and went with a 10-16-10-2 neural network structure. We also incorporated a 20% dropout rate for the two middle layers:

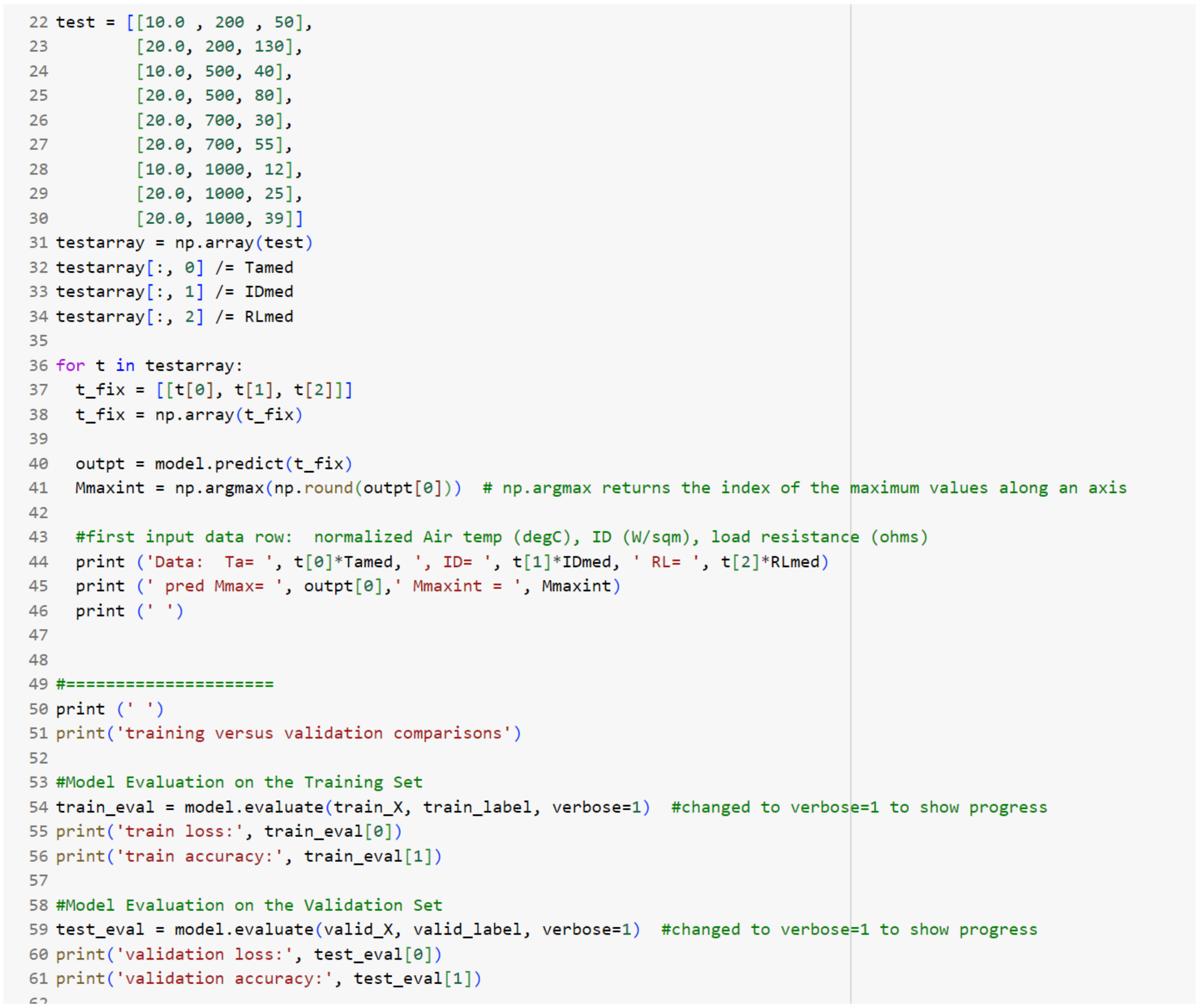
CodeP3.2.2F23



```
from keras import backend as K
#initialize weights with values between -0.2 and 0.5
initializer = keras.initializers.RandomUniform(minval= -0.5, maxval=0.7)

model = keras.Sequential([
    keras.layers.Dense(10, activation=K.elu, input_shape=[3],  kernel_initializer=initializer),
    keras.layers.Dense(16, activation=K.elu,  kernel_initializer=initializer),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10, activation=K.elu, kernel_initializer=initializer),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(num_classes, activation='softmax')
  ])
#ADD OUTPUT LAYER, REMOVE DROPOUTS FOR FIRST PART, ADD LATER
```

The loss and accuracy of the test data as well as the training and validation data was scripted as follows:

CodeP3.2.2F23



```
test = [[10.0 , 200 , 50],
        [20.0, 200, 130],
        [10.0, 500, 40],
        [20.0, 500, 80],
        [20.0, 700, 30],
        [20.0, 700, 55],
        [10.0, 1000, 12],
        [20.0, 1000, 25],
        [20.0, 1000, 39]]
testarray = np.array(test)
testarray[:, 0] /= Tamed
testarray[:, 1] /= IDmed
testarray[:, 2] /= RLmed

for t in testarray:
  t_fix = [[t[0], t[1], t[2]]]
  t_fix = np.array(t_fix)

  outpt = model.predict(t_fix)
  Mmaxint = np.argmax(np.round(outpt[0]))  # np.argmax returns the index of the maximum values along an axis

  #first input data row:  normalized Air temp (degC), ID (W/sqm), load resistance (ohms)
  print ('Data:  Ta= ', t[0]*Tamed, ', ID= ', t[1]*IDmed, ' RL= ', t[2]*RLmed)
  print (' pred Mmax= ', outpt[0],' Mmaxint = ', Mmaxint)
  print (' ')


#=====================
print (' ')
print('training versus validation comparisons')

#Model Evaluation on the Training Set
train_eval = model.evaluate(train_X, train_label, verbose=1)  #changed to verbose=1 to show progress
print('train loss:', train_eval[0])
print('train accuracy:', train_eval[1])

#Model Evaluation on the Validation Set
test_eval = model.evaluate(valid_X, valid_label, verbose=1)  #changed to verbose=1 to show progress
print('validation loss:', test_eval[0])
print('validation accuracy:', test_eval[1])
```

4.3. Discussion of Machine Learning Model to Predict Optimal Mode

With a model developed to predict the power output given a set of operating conditions and Mode configuration, we next built a machine learning model to predict which Mode would produce the highest power output under different sets of conditions. For this model, our input data consisted of T_{air}, I_D, and R_L and our output data consisted solely of M. Since our Mode is categorical, we decided to format it using the One-Hot Encoding method instead of 0, 1, or 2. One-Hot Encoding is a technique used to represent categorical variables with a binary matrix since categorical variables don't have a natural order like numbers. Using the latter in a model may unintentionally imply an order or magnitude that doesn't exist. In addition, this model uses a softmax activation function, which is best used when categorizing data into different output categories.

As we have learned, it can be tricky to select the right weight range and network layer distribution to best optimize a model, especially for both training and validation data. Often as models become more complex, they are more likely to overfit the training data, as seen in Section 4.2. Incorporating dropout layers which randomly drop neurons from the neural network during training help build the robustness of the model and prevent it from overfitting. Over two network configurations, we explored the effects of incorporating the dropout layers, shown in **Table 6.** For background, loss is a measure of the error between the predicted and actual Mode values. Accuracy is a metric that measures the proportion of correctly classifying each set of operating conditions with the correct best performing Mode.

Model	Training Data Loss	Training Data Accuracy	Validation Data Loss	Validation Data Accuracy
10 - 16 - 10 - 3 Neural Network (No Dropouts)	0.000967	1.0	0.125	0.900
10 - 16 - 10 - 3 Neural Network (Dropout Rate of 0.20)	0.0123	1.0	0.0703	1.0
10 - 16 - 10 - 3 Neural Network (Dropout Rate of 0.30)	0.0517	1.0	0.180	0.900
13 - 26 -13 - 3 Neural Network (No Dropouts)	0.000232	1.0	0.148	0.900
13 - 26 -13 - 3 (Dropout Rate of 0.20)	0.0145	1.0	0.0368	1.0
13 - 26 -13 - 3 (Dropout Rate of 0.30)	0.00702	1.0	0.0834	1.0

Table 6. Comparison One-Hot Encoding Neural Network With/Without Dropout Layers

We see signs of overfitting among the two neural networks without dropout layers. The 10-16-10-3 and 13-26-13-3 neural networks performed 128% and 636% worse in terms of loss for the validation data versus the training data. For both networks, the training data achieved a loss of below 1%, while neither of them achieved below a 12% loss with the validation data.

On the other hand, we saw a noticeable improvement in terms of loss and accuracy with the incorporation of dropout layers. For the 10-16-10-3 neural network, the model performed 43% better on the validation data. Similarly, the 13-26-13-3 network's performance with the validation data improved 75%. The network saw minimal decreases in loss performance with the training data of around 1%. It also displayed a significant improvement in loss from 15% to 4% for the validation data. Furthermore, the accuracy improved from 90% to 100% when dropout layers were included for both neural networks. We found these improvements to be most significant as increased performance across validation sets is more useful in real-life settings.

It is also important to note the increased improvement of incorporating dropout layers for the more complex 13-26-13-3 network versus the 10-16-10-3 network. Here, we experienced a near 30% improvement in validation data loss. Thus, we reason that dropout layers more effectively counteract overfitting for networks with more neurons as compared with fewer.

We also found that there is a goldilocks zone when selecting the right dropout rate for our networks. When experimenting with a dropout rate of 30%, the 10-16-10-3 model performed the worst of all the trials. The neural network likely either changed too drastically to form a high degree of reliability in the model, or would require much more data and training epochs to become reliable. On the other hand, the 13-26-13 model performed decently well with a validation data loss of 8.34% and a dropout rate of 30%. Yet, with a dropout rate of 20%, it still experienced a validation data loss as high as 3.68%. The larger neural network was likely able to maintain accuracy while undergoing a higher dropout rate as it had more neurons to compensate for any network changes.

4.4. How Do the Two Machine Learning Models Compare in Predicting the Optimal Mode

Now with two working machine learning models that can predict the optimal Mode configuration for a solar panel system, we next wanted to analyze how they compared to each other. Specifically, with a set of new data for operating conditions, we wanted to see if both the Task 2.1 and Task 2.2 models converged on the same optimal Mode configuration. While the Task 2.2 model can predict the optimal model with a set of inputs, the Task 2.1 model cannot directly. Thus, we entered the test data along with each Mode configuration into the latter model and recorded the predicted power outputs. The Mode configuration that outputs the most power is the winner. **Table 7.** shows the overall results.

Table 7. Mode Predictions for both Task 2.1 and Task 2.2 Models

#	T_{air} (°C)	I_D (W/m^2)	R_L (Ohms)	$M_{max, int}$ [Task 2.2 Model]	*W_dot* (W) [Task 2.1 Model for M = 0]	*W_dot* (W) [Task 2.1 Model for M = 1]	*W_dot* (W) [Task 2.1 Model for M = 2]	***Do Models Match? (Y/N)***
1	10.0	200	50.	1	58.847	160.12	75.676	Y
2	20.0	200	130	2	20.638	54.418	116.426	Y
3	10.0	500	40	2	69.383	220.91	346.72	Y
4	20.0	500	80	2	38.937	85.237	104.10	Y
5	20.0	700	30	2	77.553	275.841	80.750	N
6	20.0	700	55	2	54.876	116.99	149.54	Y
7	10.0	1000	12	1	207.26	802.61	294.54	Y
8	20.0	1000	25	2	97.072	346.31	446.56	Y
9	20.0	1000	39	2	72.865	237.46	529.80	Y

We observed that the two models match almost every time in the optimal predicted Mode for each set of T_{air}, I_D, and R_L conditions. That said, it is important to remember that the best MARE being 45% for the Task 2.1 model is not great. Thus, this proves to be one example of how one must be thoughtful of what form of machine learning model is selected for the desired application. For instance, for the Task 2.1 model, despite its predictive power in selecting the best performing Mode in **Table 7**, we would advise caution on deploying this model in the real-world knowing its poor MARE performance with our validation data. On the other hand, the Task 2.2 model performed extremely well in predicting the correct Mode and shows great promise in providing benefit in a solar panel system application.

However, there are always cautions one must be aware of when selecting a model for a specific application. For example, specifically for our Task 2.1 model, it failed 1/9 times. Without more data, we have no way of knowing if it would actually fail 11% of the time (or more) or if this case was an outlier. In a real-life setting, if we expanded our dataset across 1000 different conditions, failing 11% or more of the time could lead to significant energy and cost ramifications. Further, both models were only required to distinguish between three different modes of operation. This lowered the chances any given guess alone would be wrong to 67%. If instead our model needed to choose between ten or even one hundred different settings or control inputs as it may need to in a real life setting, we may see far more failures. Thus, while our models seemed to perform decently well in this analysis for this specific purpose, it is incredibly important to train a model to the accuracy needed for its specific application. Expanding the complexity of the data in any way, whether that be in the number of test cases, or in the number of predictions needed for each run, would require far more data and a much lower error rate.

5. Conclusion

In conclusion, the integration of machine learning into the optimization of solar panel systems represents a promising avenue for enhancing their performance. The exploration of two specific applications, predicting power outputs under high solar intensity conditions and optimizing solar panel configurations, revealed valuable insights. However, as with any machine learning model, utilizing the right neural network structure, dropout layers, and good training data are critical in limiting overfitting and functional performance.