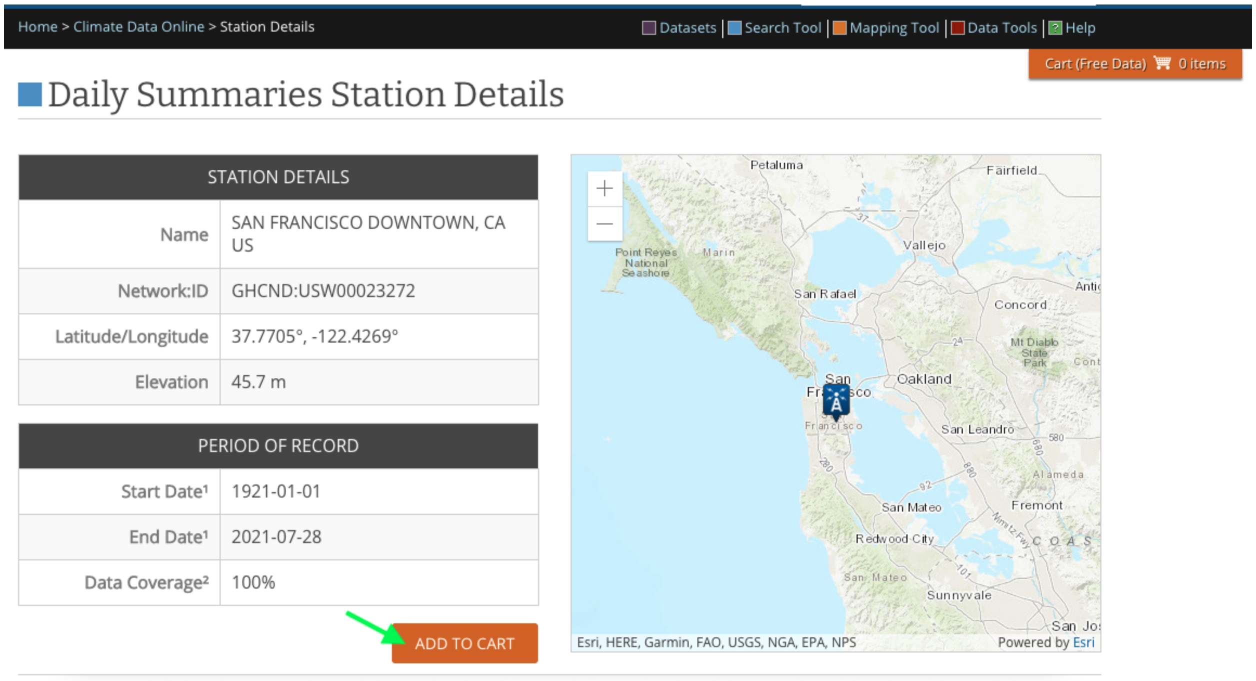
Chapter 07| Linear Regression Application

~~~~~~

Dataset Download

The first step to building a model is to first get the data to train the model on. To keep things simple we are going to predict the temperature (minimum and maximum) of San Francisco in 2020 given the day of the year. For example, January 1st would be an input of 0 (python indexing starts at 0), January 2nd would be an input of 1, and since 2020 is a leap year December 31st would be an input of 365. As a result, we are going to have one input feature (day of year) and 2 output features (minimum and maximum of temperature of that day. To download this dataset follows these steps or download it from github:

1. Go To: *https://www.ncdc.noaa.gov/cdo-web/datasets/GHCND/stations/GHCND:USW00023272/detail* Click "ADD TO CART" (no payment needed)

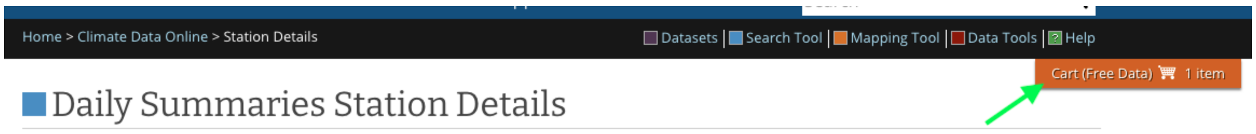

Home > Climate Data Online > Station Details
Datasets | Search Tool | Mapping Tool | Data Tools | Help
Cart (Free Data) 0 items
ADD TO CART
Esri, HERE, Garmin, FAO, USGS, NGA, EPA, NPS
Powered by Esri

Daily Summaries Station Details

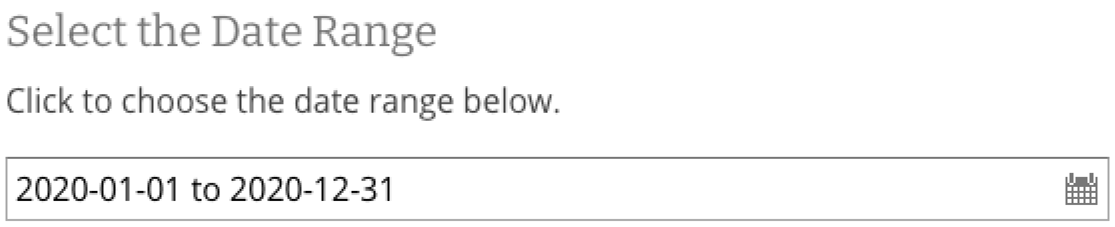
STATION DETAILS	
Name	SAN FRANCISCO DOWNTOWN, CA US
Network:ID	GHCND:USW00023272
Latitude/Longitude	37.7705°, -122.4269°
Elevation	45.7 m

PERIOD OF RECORD	
Start Date[1]	1921-01-01
End Date[1]	2021-07-28
Data Coverage[2]	100%

Click "Cart" (top right)


Home > Climate Data Online > Station Details
Datasets | Search Tool | Mapping Tool | Data Tools | Help
Cart (Free Data) 1 item

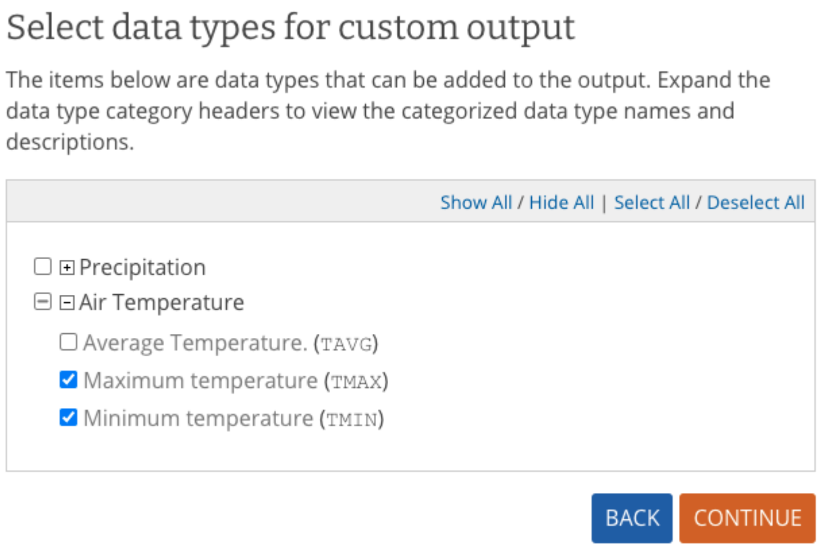
Daily Summaries Station Details
~~~~~~

2. *Step 1: Choose Options*
    a. Select the Output Format -- GHCND (CSV)
    b. 
    Select the Date Range

    Click to choose the date range below.

    2020-01-01 to 2020-12-31

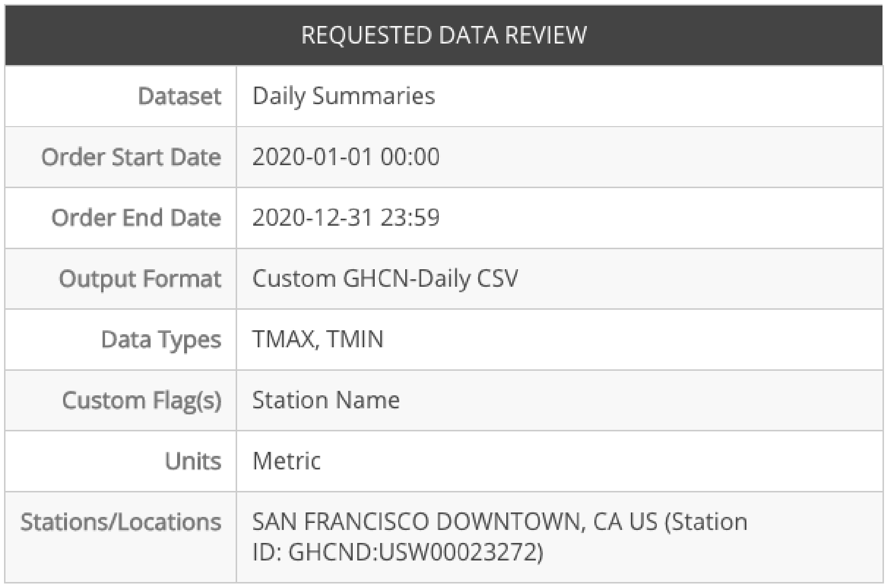
    Change the Date Range to the above.
    c. Press Continue
    d. Station Detail & Data Flag Options -- Check Station Name, Change Units to Metric.
    e. Select data types for custom output

    
    Select data types for custom output

    The items below are data types that can be added to the output. Expand the data type category headers to view the categorized data type names and descriptions.

    ☐ ⊞ Precipitation
    ⊟ ⊟ Air Temperature
    ☐ Average Temperature. (TAVG)
    ✓ Maximum temperature (TMAX)
    ✓ Minimum temperature (TMIN)

    Air Temperature -- TMAX, TMIN
    f. Press Continue

3. *Step 2: Review Order*
    a. 

REQUESTED DATA REVIEW	
Dataset	Daily Summaries
Order Start Date	2020-01-01 00:00
Order End Date	2020-12-31 23:59
Output Format	Custom GHCN-Daily CSV
Data Types	TMAX, TMIN
Custom Flag(s)	Station Name
Units	Metric
Stations/Locations	SAN FRANCISCO DOWNTOWN, CA US (Station ID: GHCND:USW00023272)

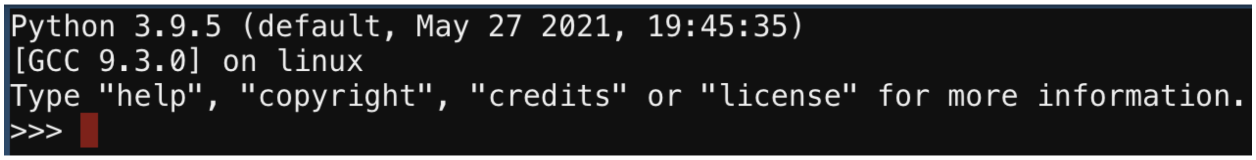
    Check if the table matches this.
    b. Enter Email Address -- Enter and Verify
    c. Press "Submit Order"

4. *Step 3: Order Complete*
    a. Check your email, the sender will be noreply@noaa.gov Should arrive in less than five minutes. (I've sent about five orders already and all came in that interval).
    b. The first email you receive notifies you that the order has been received. You can also check the status by going to: *https://www.ncdc.noaa.gov/cdo-web/orders*, enter the email address and order id (can be found in the first email). Press "CHECK NOW".
    c. If the status is not Complete, you can reload the page and continue checking until it is. If the status never becomes Complete, you can contact them.
    d. Lastly, when it's Complete, or you receive the second email, you can download from the status checker or from the email (it doesn't matter).

Now the file you downloaded is a **csv** (**c**omma **s**eparated **v**alues), in the next section we will learn to read this file in code and create a plot of the temperature vs. date graph to visualize the dataset.

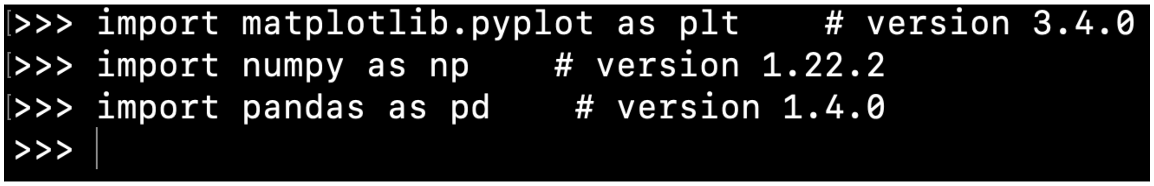
Data Setup

To read a **csv** file, we can use Pandas, a data analysis library.
Here is the pip install link for Pandas: https://pypi.org/project/pandas/
As for reference if major changes in Pandas occur in later versions, this book uses **Pandas 1.4.0**. Before writing a script to use pandas to read and use the data, it is best to see one operation at a time. Thus, launch the command prompt/line or terminal on your computer, navigate to the directory the **csv** is located in and then use the 'python' command to launch the shell. The shell should look something like this:



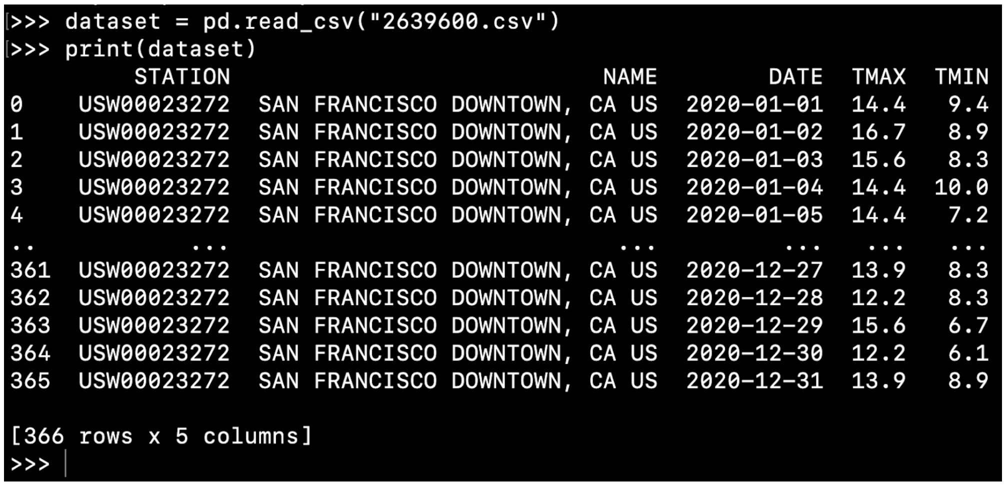
```
Python 3.9.5 (default, May 27 2021, 19:45:35)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Reading the **csv** file through pandas allows us to do two things. One is what we introduced in the last section, creating a plot of the temperature vs. date graph for visualizing the dataset using matplotlib. Another is to convert the data into a numpy array which is a format we can use for training. Thus, let's first import all three libraries.



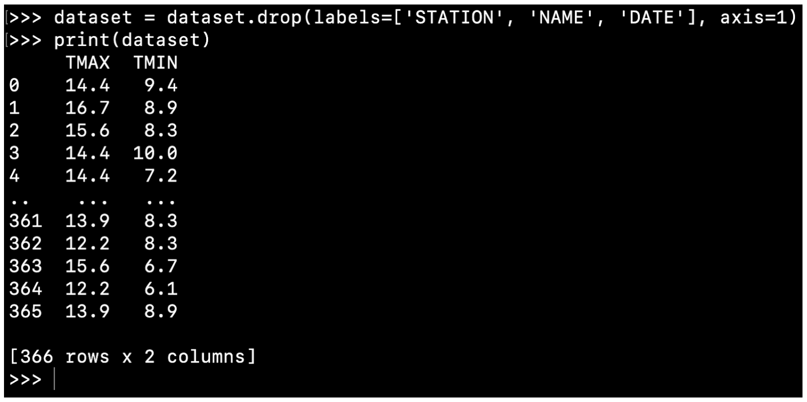
```
>>> import matplotlib.pyplot as plt    # version 3.4.0
>>> import numpy as np    # version 1.22.2
>>> import pandas as pd    # version 1.4.0
>>>
```

The comments indicate the version of the libraries I am using, for reference.
pandas.read_csv(), is a function we can use to read the csv file and the first argument is a string representing the name of the csv or file path to the csv.



```
>>> dataset = pd.read_csv("2639600.csv")
>>> print(dataset)
         STATION                            NAME        DATE  TMAX  TMIN
0    USW00023272  SAN FRANCISCO DOWNTOWN, CA US  2020-01-01  14.4   9.4
1    USW00023272  SAN FRANCISCO DOWNTOWN, CA US  2020-01-02  16.7   8.9
2    USW00023272  SAN FRANCISCO DOWNTOWN, CA US  2020-01-03  15.6   8.3
3    USW00023272  SAN FRANCISCO DOWNTOWN, CA US  2020-01-04  14.4  10.0
4    USW00023272  SAN FRANCISCO DOWNTOWN, CA US  2020-01-05  14.4   7.2
..           ...                            ...         ...   ...   ...
361  USW00023272  SAN FRANCISCO DOWNTOWN, CA US  2020-12-27  13.9   8.3
362  USW00023272  SAN FRANCISCO DOWNTOWN, CA US  2020-12-28  12.2   8.3
363  USW00023272  SAN FRANCISCO DOWNTOWN, CA US  2020-12-29  15.6   6.7
364  USW00023272  SAN FRANCISCO DOWNTOWN, CA US  2020-12-30  12.2   6.1
365  USW00023272  SAN FRANCISCO DOWNTOWN, CA US  2020-12-31  13.9   8.9

[366 rows x 5 columns]
>>>
```

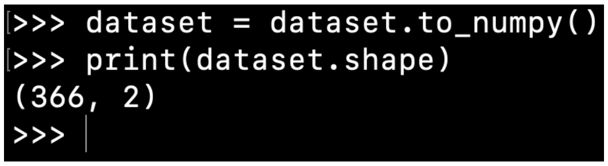
In my case, the file for the dataset is "2639600.csv". I also print the contents of the variable *'dataset'* which shows the first two columns for the whole file are the same so we should remove those. Also the dataset is in chronological order already (first row is the first day of the year, last row is the last day of the year). Hence, the DATE column can also be removed as long as we do not change the order of the rows.



```
>>> dataset = dataset.drop(labels=['STATION', 'NAME', 'DATE'], axis=1)
>>> print(dataset)
     TMAX  TMIN
0    14.4   9.4
1    16.7   8.9
2    15.6   8.3
3    14.4  10.0
4    14.4   7.2
..    ...   ...
361  13.9   8.3
362  12.2   8.3
363  15.6   6.7
364  12.2   6.1
365  13.9   8.9

[366 rows x 2 columns]
>>>
```

Using the method *drop*, we specify the names of the columns we want to remove using a list for the first argument. The second argument tells pandas where to look for these names that we would like to remove, in the row (axis 0) or column (axis 1)?

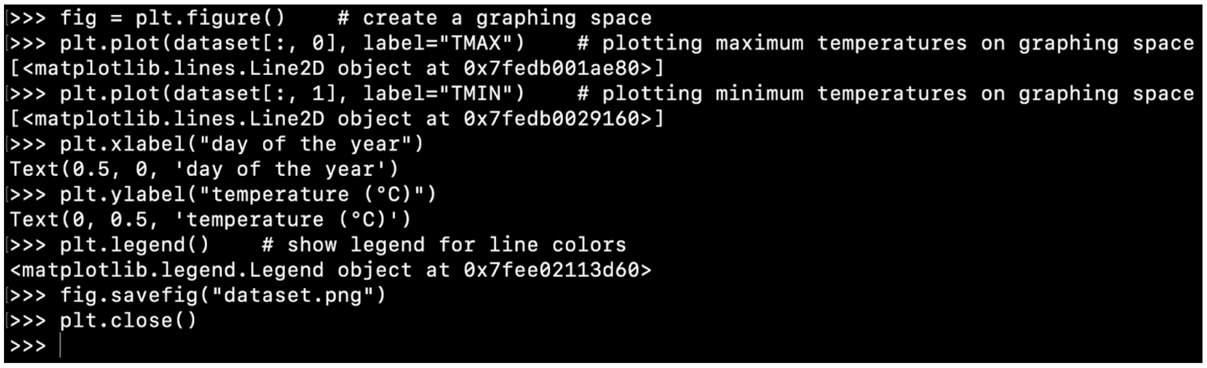
Printing the new contents, we see that we are left with the maximum and minimum temperatures of each day. Since matplotlib cannot work with this pandas data frame for plotting, let's convert this to a numpy array. Luckily, pandas dataframes have a method *to_numpy()* for achieving this.



```
>>> dataset = dataset.to_numpy()
>>> print(dataset.shape)
(366, 2)
>>>
```

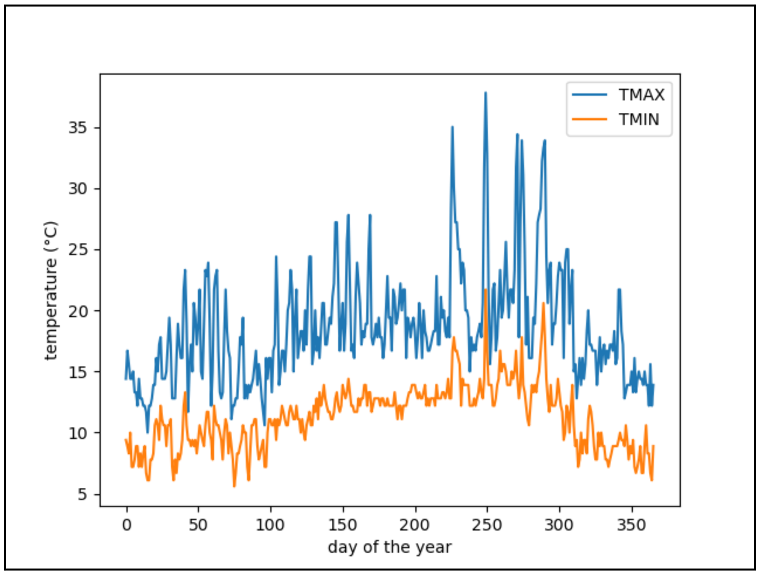
Checking the shape, we still have 366 rows (366 days in 2020) and 2 columns (one for the maximum temperature and another for the minimum temperature).

Now that the data is in numpy we can start plotting.



```
>>> fig = plt.figure()    # create a graphing space
>>> plt.plot(dataset[:, 0], label="TMAX")    # plotting maximum temperatures on graphing space
[<matplotlib.lines.Line2D object at 0x7fedb001ae80>]
>>> plt.plot(dataset[:, 1], label="TMIN")    # plotting minimum temperatures on graphing space
[<matplotlib.lines.Line2D object at 0x7fedb0029160>]
>>> plt.xlabel("day of the year")
Text(0.5, 0, 'day of the year')
>>> plt.ylabel("temperature (°C)")
Text(0, 0.5, 'temperature (°C)')
>>> plt.legend()    # show legend for line colors
<matplotlib.legend.Legend object at 0x7fee02113d60>
>>> fig.savefig("dataset.png")
>>> plt.close()
>>>
```

When plotting, we must use all the rows of the dataset but one column of data at a time. The outputs of matplotlib can be ignored. We use a keyword argument, label to give each differently colored line a label in which *plt.legend()* adds a legend to specify which line color corresponds to what type of data. Lastly, we save the plot to 'dataset.png' and close. To exit the python shell use 'quit()'. Here is our visualization:



Seems very messy, ups and downs everywhere. Although us humans can find the upward trend of the temperature as we get to the middle of the year and the downward trend of the temperature as we get to the end of the year, the trend is really hard for computers to find out. Using a very **"noisy"**, lots of inconsistent, with only a few data points indicating a trend, for weather prediction requires a different type of model, a **time-series model**. However, this type of model is out of the scope for an introduction. To convert this as data for us to be able to use easily, where the trend is more obvious, we are going to use a 125-day moving average.

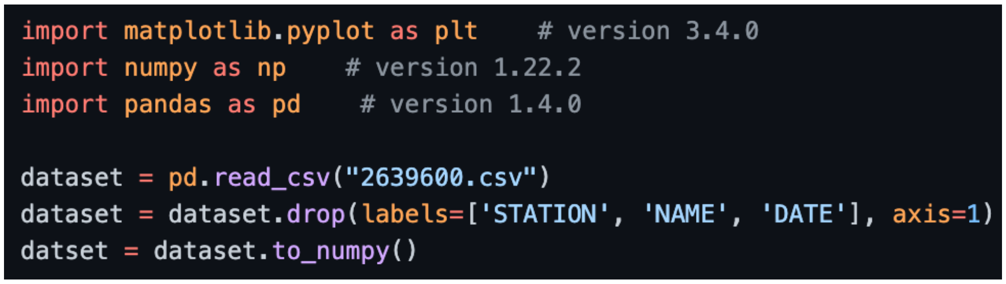
This means that we are going to calculate the average value of the first 125 days and that will be our first datapoint. Our second datapoint will be the average of the first 126 days except the first day. Our third datapoint will be the average of the first 127 days except the first and second days, and so on until we used every day of the original dataset at least once (or you can say until we reach the end of the original dataset).

If we have 7 original data points and we do a 3-day moving average:

New Data Point #	1	2	3	4	5
Data Points Used to Calculate Average	1 2 3	2 3 4	3 4 5	4 5 6	5 6 7

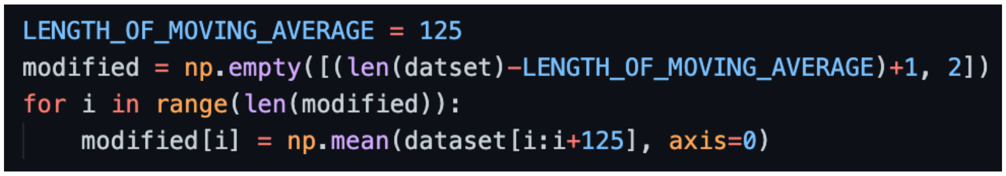
There would be 5 new data points: (7 - 3) + 1 so the number of new data points can be determined by subtracting the length of the moving average from the number of original data points and adding one.

Instead of using the python shell, we can go back to scripts since you understand how to read a csv file and convert the data into numpy using pandas now. Here is the code for a 125-day moving average:



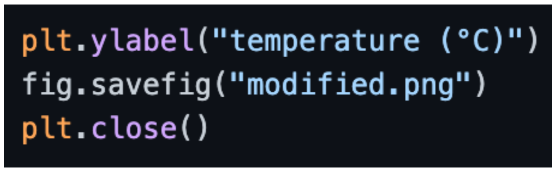
```
import matplotlib.pyplot as plt    # version 3.4.0
import numpy as np    # version 1.22.2
import pandas as pd    # version 1.4.0

dataset = pd.read_csv("2639600.csv")
dataset = dataset.drop(labels=['STATION', 'NAME', 'DATE'], axis=1)
datset = dataset.to_numpy()
```

Since the process of reading the csv file is explained, the print statements are removed. Now we need to create new data points.



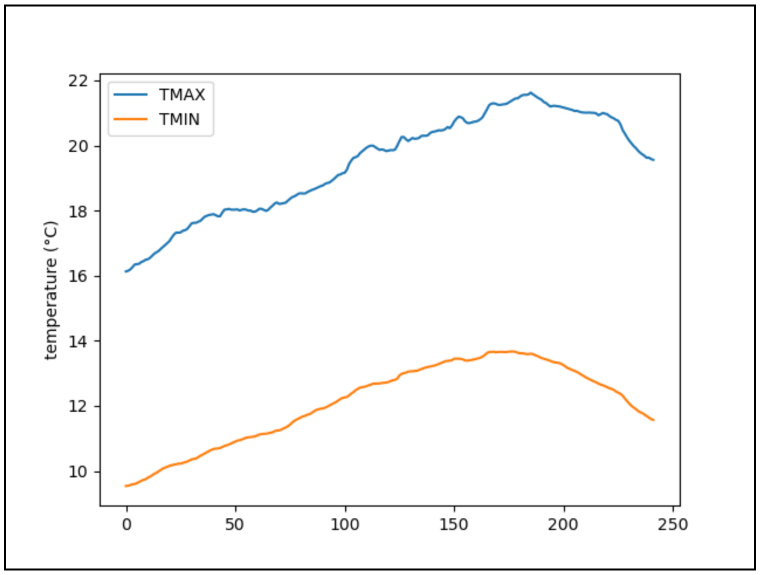
```
LENGTH_OF_MOVING_AVERAGE = 125
modified = np.empty([(len(datset)-LENGTH_OF_MOVING_AVERAGE)+1, 2])
for i in range(len(modified)):
    modified[i] = np.mean(dataset[i:i+125], axis=0)
```

We first create an empty array as a placeholder for our new dataset. Using the formula from, the number of rows for the new dataset would be equal to the number of original data points (len(dataset)) minus the length of the moving average and then plus 1. The number of columns would still be 2 (one for the maximum temperature and another for the minimum temperature). We loop only all the rows of the empty array and set the new row to be equal to the mean of the original dataset starting from the current row number and stopping 125 days after. Using the rows (axis 0) of the indexed data points we calculate the average values by using *np.mean*.



```
plt.ylabel("temperature (°C)")
fig.savefig("modified.png")
plt.close()
```

To graph/visualize our new data, we create a graphing space and we plot all rows but one column at a time. To differentiate the colored lines and understand what the line represents we add a legend that uses the labels we created in the previous two lines. We label the y-axis, save the figure to “modified.png” and close the figure. We do not label the x-axis because we have modified the data points so it no longer makes sense to call the x-values the day of the year.

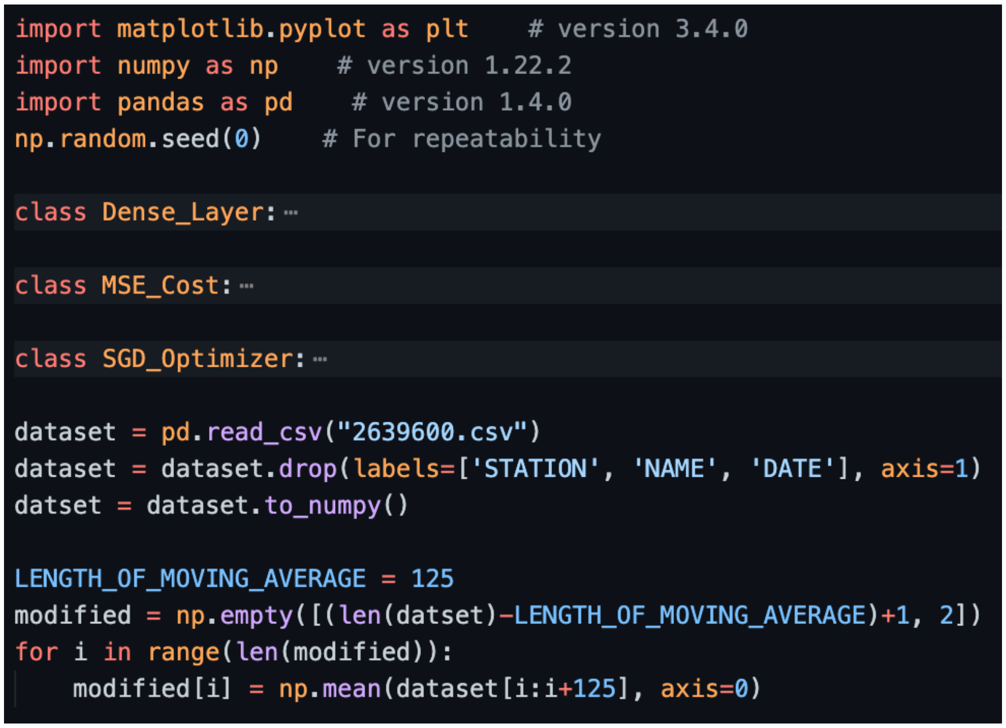


As you can see, the graph is much smoother and we can clearly see the general trend of an increase in temperatures towards the middle of the year and a decrease in temperatures towards the end of the year.

Now that we have our actual answers (y), we can start to train a model to predict the minimum and maximum temperatures.

Training

This section is going to focus on the code that trains the model so here is the code:



```
import matplotlib.pyplot as plt    # version 3.4.0
import numpy as np    # version 1.22.2
import pandas as pd    # version 1.4.0
np.random.seed(0)    # For repeatability

class Dense_Layer:⋯

class MSE_Cost:⋯

class SGD_Optimizer:⋯

dataset = pd.read_csv("2639600.csv")
dataset = dataset.drop(labels=['STATION', 'NAME', 'DATE'], axis=1)
datset = dataset.to_numpy()

LENGTH_OF_MOVING_AVERAGE = 125
modified = np.empty([(len(datset)-LENGTH_OF_MOVING_AVERAGE)+1, 2])
for i in range(len(modified)):
    modified[i] = np.mean(dataset[i:i+125], axis=0)
```

First we import the libraries and seed the Numpy random generator. Copy and paste the dense layer class, MSE cost class, and the SGD optimizer class. After that, we add the data setup part without the plotting.

Now we have the correct answer in the variable *modified.* We can use this to compare the model's outputs to the cost function. To get the outputs of the model, we need to input something in the model in which we do not have yet. Before coding, we need to understand what our inputs should be.

Well, given the day of the year (input feature) the model should predict what the maximum and minimum temperature of that day is. Since we have one input feature, our input would contain one column, but how many rows? Recall that each row of a dataset should be an example, so we would have the same number of rows as the correct answers, *modified*. In other words, each day of the year would be an example.

```
X = np.expand_dims(np.array(range(len(modified))), axis=1)
y = modified.copy()
```

We create a numpy array using *np.array()* and the data would be the output of the *range()* function starting at 0 and stopping until we reach the same number of rows as our correct answers, *modified*. The numpy array created is a row-vector since there is only one column. However, the model only accepts a matrix so to create a matrix with one column from a row-vector we just use *np.expand_dims()* where the column axis would be axis 1. We make a copy of *modified* to use the variable *"y"* instead.

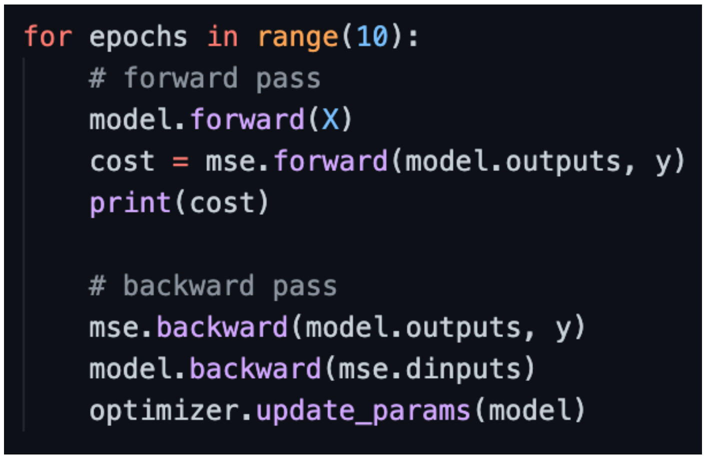
```
model = Dense_Layer(1, 2)    # one input feature, 2 neurons (output features)
mse = MSE_Cost()    # define cost function
optimizer = SGD_Optimizer(0.001)    # learning rate of 0.001
```

Now we can initialize our model with one input feature and 2 neurons (output features) from the maximum and minimum temperatures and we initialize the optimizer with a learning rate of 0.001.

```
model.forward(X)
print("Initial Cost:", mse.forward(model.outputs, y))
```

Also we get the initial cost of the model before training by getting the model's outputs. Now we can code the training loop.

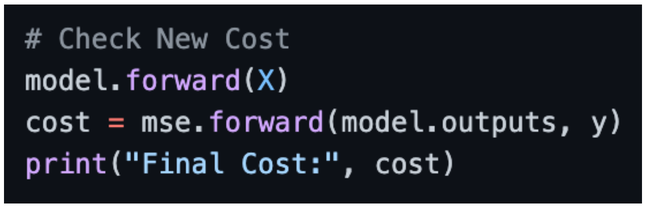
We keep things simple by only training for 10 epochs where we get the cost of the model after getting the model's outputs and print the cost out. Then, we call the backward method of the cost function so that we can call the backward method of the cost function, allowing us to update the parameters using the optimizer.



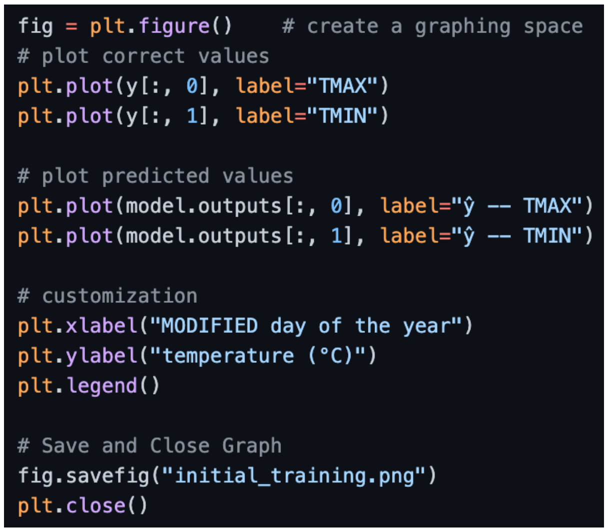
```python
for epochs in range(10):
    # forward pass
    model.forward(X)
    cost = mse.forward(model.outputs, y)
    print(cost)

    # backward pass
    mse.backward(model.outputs, y)
    model.backward(mse.dinputs)
    optimizer.update_params(model)
```

After the training loop, we print the cost after training by getting the model's outputs to input into the cost function.

Lastly, we plot the correct answer with the model's predictions on the same plot to understand the model's performance.



```python
# Check New Cost
model.forward(X)
cost = mse.forward(model.outputs, y)
print("Final Cost:", cost)
```



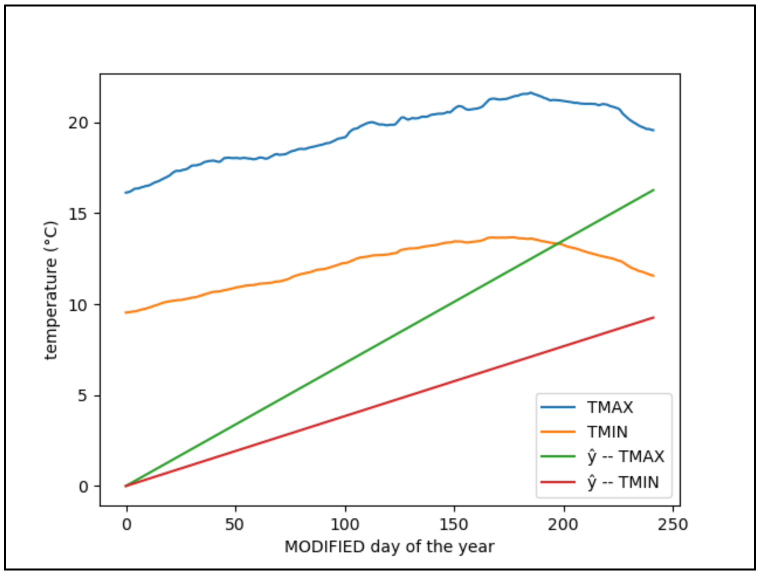
```python
fig = plt.figure()    # create a graphing space
# plot correct values
plt.plot(y[:, 0], label="TMAX")
plt.plot(y[:, 1], label="TMIN")

# plot predicted values
plt.plot(model.outputs[:, 0], label="ŷ -- TMAX")
plt.plot(model.outputs[:, 1], label="ŷ -- TMIN")

# customization
plt.xlabel("MODIFIED day of the year")
plt.ylabel("temperature (°C)")
plt.legend()

# Save and Close Graph
fig.savefig("initial_training.png")
plt.close()
```

We label the model's predictions with a ŷ. The x-axis label has "MODIFIED" because the day of the year no longer correlates with actual days of the year since we modified the dataset to make it easier to train the model. After labeling, we add a legend for all the different lines on the graph. In the end, we save the figure to "initial_training.png" and close the plot.

<u>Terminal Output:</u>

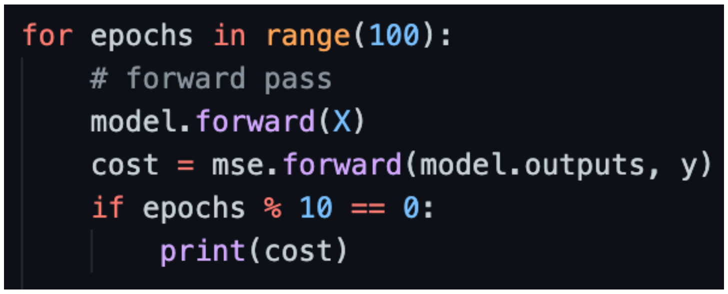
```
Initial Cost: 218.40823409136365
218.40823409136365
56843.43768474931
19230423.55509895
6511518642.409367
2204838904734.089
746571560392471.5
```

```
2.5279356854725942e+17
8.559740511045948e+19
2.8983790227536445e+22
9.81408367309474e+24
Final Cost: 3.32310707420861e+27
```



We see that the cost has actually increased rather than decreased during training and that is a result of? Our learning is too high. Putting the learning down from 0.001 to 0.0001, our final cost is now less than 100.

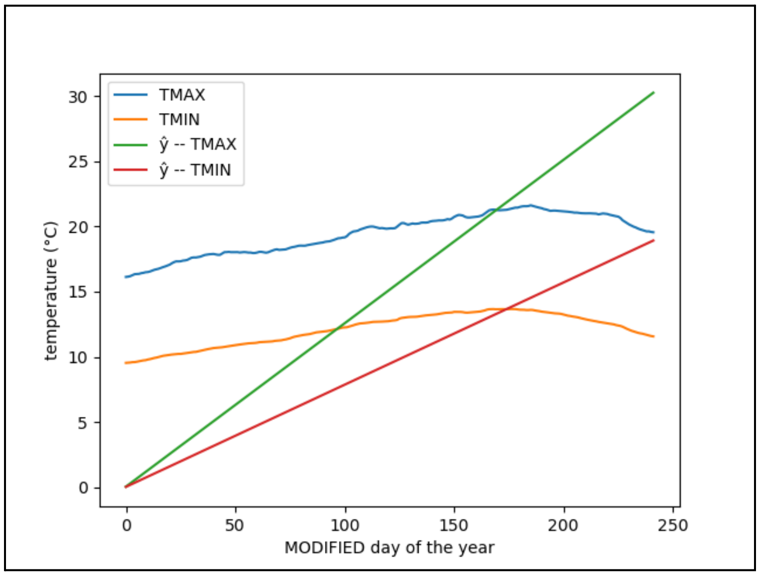
Since we also see that the model barely learned anything so let's push the number of epochs from 10 to 100. Also since we are printing the cost of the model during training, let's modify the training loop so the terminal is not flooded with values.
Now every 10 epochs, we print the cost out to the terminal.



```
for epochs in range(100):
    # forward pass
    model.forward(X)
    cost = mse.forward(model.outputs, y)
    if epochs % 10 == 0:
        print(cost)
```

Here is the terminal output:

```
Initial Cost: 218.40823409136365
218.40823409136365
99.44527060681818
64.82369678143556
54.73517158745953
51.78277511092108
50.9061291924535
50.63328205563995
50.53607557077715
50.48996802958329
50.458733006829505
Final Cost: 50.4318330343753
```



The graph we created is on the right. Clearly we see we have hit a problem, the model is stuck at around the cost of 50, unable to decrease anymore and the model's predictions are way off the correct answer. It seems as if the model has decided to neglect the bias parameter where for TMIN it should be around 10 and for TMAX it should be around 15. We see that instead, it seems like the model did not update the bias parameter at all, leaving it at its initialized value of 0. To compensate for the bias parameter being useless, the model decides to have a huge weight parameter for both TMAX and TMIN. The weight parameter is also experiencing problems since the correct slopes of TMIN and TMAX should be small.

These training problems with the weight and bias can be fixed by modifying the dataset once again through **normalization** and **standardization**, discussed in the next two sections.

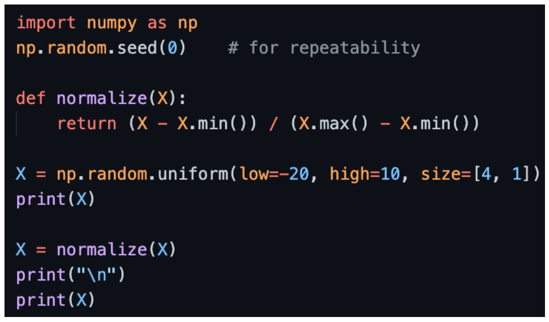
Normalization

Normalization is the process of bringing each input feature of the dataset into a range with a minimum of 0 and a maximum of 1 or a minimum of -1 and a maximum of 1. Since the weight and bias is initialized around the value of 0, in theory this should help the model learn. In case we want to model to have smaller weights through effective use of the bias.

First let's normalize every input feature of the dataset to a range of [0, 1]. Concentrating on only one input feature, we would find the difference between every data point and the input feature's minimum value. After that we divide by the difference between the input feature's maximum and minimum value. Here is a mathematical representation:
[4, 5, 6] →minimum: 0, maximum: 1

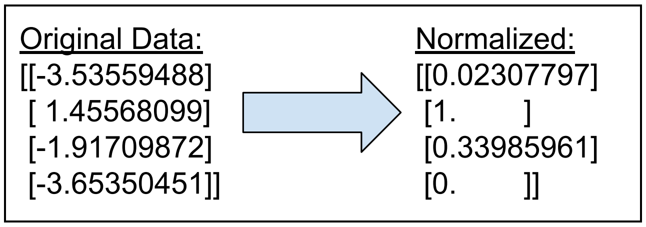
$$newX_i = \frac{(X_i - X_{min})}{(X_{max} - X_{min})}$$

Note that this is for only one input feature of the dataset where the new datapoint is $newX_i$, the original datapoint is X_i, the minimum value is X_{min}, and the maximum value is X_{max}. We can better understand what is happening with an example. Here is the code:



```
import numpy as np
np.random.seed(0)    # for repeatability

def normalize(X):
    return (X - X.min()) / (X.max() - X.min())

X = np.random.uniform(low=-20, high=10, size=[4, 1])
print(X)

X = normalize(X)
print("\n")
print(X)
```

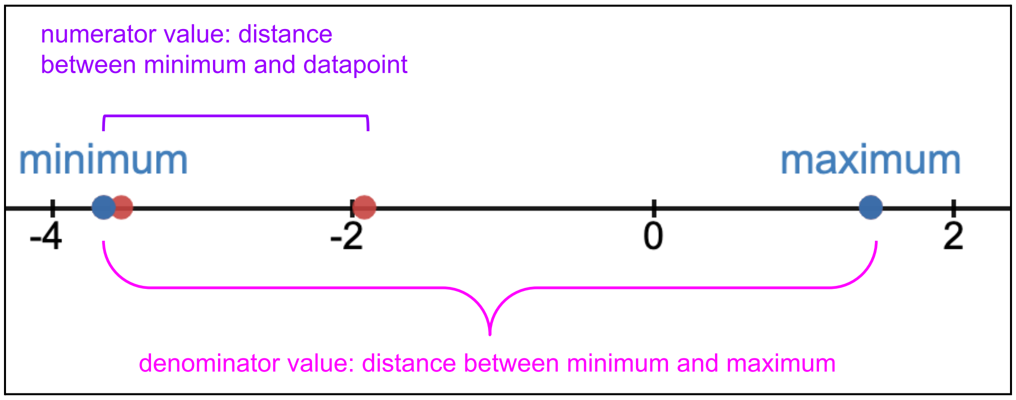
We create a function to normalize the data and then we ask numpy to randomly generate a matrix with 4 examples and one input feature using *np.random.uniform()* where the lowest value can be -20 and the highest cannot include 10. The 'uniform' part of the function means that all values within those ranges have the same probability of being chosen.



```
Original Data:                      Normalized:
[[-3.53559488]                      [[0.02307797]
 [ 1.45568099]                       [1.        ]
 [-1.91709872]                       [0.33985961]
 [-3.65350451]]                      [0.        ]]
```

It's pretty obvious here that the minimum and maximum values of our original data is -3.65350451 and 1.45568099 respectively. The minimum and maximum values have been 0 and 1 respectively.

Here is a graphic to understand the calculation (note that the denominator value for each datapoint is the same):


numerator value: distance between minimum and datapoint
minimum
maximum
-4
-2
0
2
denominator value: distance between minimum and maximum

Looking at the extremes first, since the distance between the minimum value and minimum value for the numerator we get 0, if the numerator is zero and the denominator is nonzero, we get a result of 0. The distance between the minimum value and the maximum value for the numerator is the same as the denominator value so the result is 1.

For the other data points, we are technically calculating the part-to-whole ratio where the part is the numerator value. Since parts cannot be greater than the whole (denominator) the rest of the data points will be less than 1.

Now that we have a solid understanding of normalizing one input feature to the range of [0, 1], let's move on to normalizing one input feature to the range of [-1, 1]. To do this, we can first normalize the input feature to a range of [0, 1] and go from there.

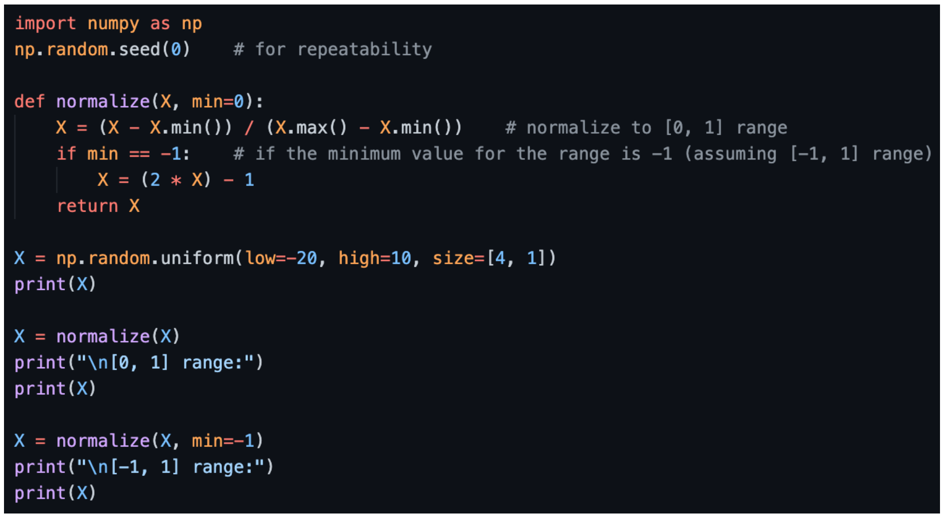
Comparing the **ranges** (max - min) of the **normalization ranges** ([0, 1] or [-1, 1]), we see that the range of [0, 1] is 1 (1 - 0) while the range of [-1, 1] is 2 (2 - 0). As a result, starting from the [0, 1] range we need to first increase the **range** of the **normalized range** to 2. To do this, all we need to do is to multiply each data point by 2. Since the minimum (0) multiplied by 2 is still 0 and the maximum (1) multiplied by 2 is now 2, the range of our normalized range of [0, 2] is 2. Now we see that all we have to do is to subtract 1 from each datapoint. The minimum (0) subtracted by 1 is -1 and the maximum (2) subtracted by 1 is 1, creating the normalized range of [-1, 1].

Here is a formula to sum up the calculations we do for a normalized range of [-1, 1]:

$$newX_i = 2\frac{(X_i - X_{min})}{(X_{max} - X_{min})} - 1$$

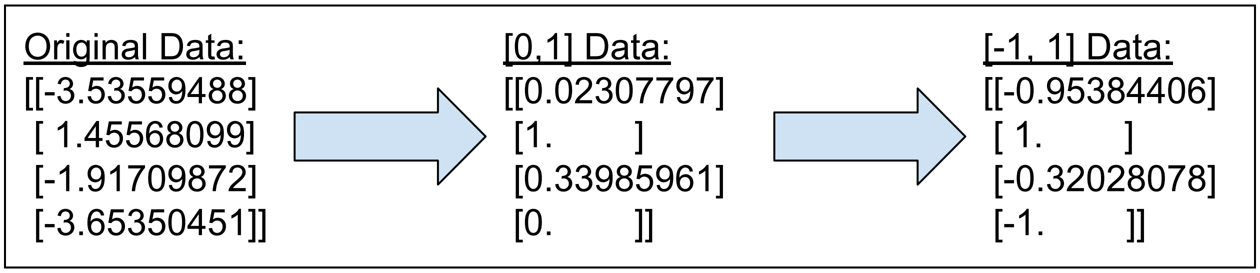
Note that this is still only for one input feature of the dataset where the new datapoint is newX$_i$, the original datapoint is X$_i$, the minimum value is X$_{min}$, and the maximum value is X$_{max}$. In addition to the normalization to a range of [0, 1] we multiply each datapoint by 2 and then subtract 1.

Here is the code for an example of normalizing to a range of [-1, 1]:



```
import numpy as np
np.random.seed(0)    # for repeatability

def normalize(X, min=0):
    X = (X - X.min()) / (X.max() - X.min())    # normalize to [0, 1] range
    if min == -1:    # if the minimum value for the range is -1 (assuming [-1, 1] range)
        X = (2 * X) - 1
    return X

X = np.random.uniform(low=-20, high=10, size=[4, 1])
print(X)

X = normalize(X)
print("\n[0, 1] range:")
print(X)

X = normalize(X, min=-1)
print("\n[-1, 1] range:")
print(X)
```

In the normalize function we add a keyword argument *min* with a default value of 0. We would normalize X to the [0, 1] range but if the keyword argument wants the minimum value to be -1 then we multiply by 2 and minus 1 to normalize X to the [-1, 1] range. We print out the results of normalizing the data to [0, 1] and [-1, 1] to better understand the formula that brings the [0, 1] data to a [-1, 1] range.



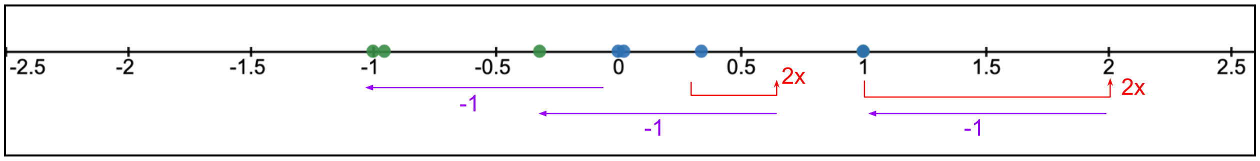
<u>Original Data:</u>
```
[[-3.53559488]
 [ 1.45568099]
 [-1.91709872]
 [-3.65350451]]
```

<u>[0,1] Data:</u>
```
[[0.02307797]
 [1.        ]
 [0.33985961]
 [0.        ]]
```

<u>[-1, 1] Data:</u>
```
[[-0.95384406]
 [ 1.        ]
 [-0.32028078]
 [-1.        ]]
```

As you can see the maximum of the [0, 1] data is 1 and after normalizing to [-1, 1] we are still at 1. The minimum of the [0, 1] data is 0 and after normalizing to [-1, 1] we are now at -1. The other data points are still in between the two extreme values.

Here is a graphic to understand the calculation from [0, 1] data to [-1, 1] data:

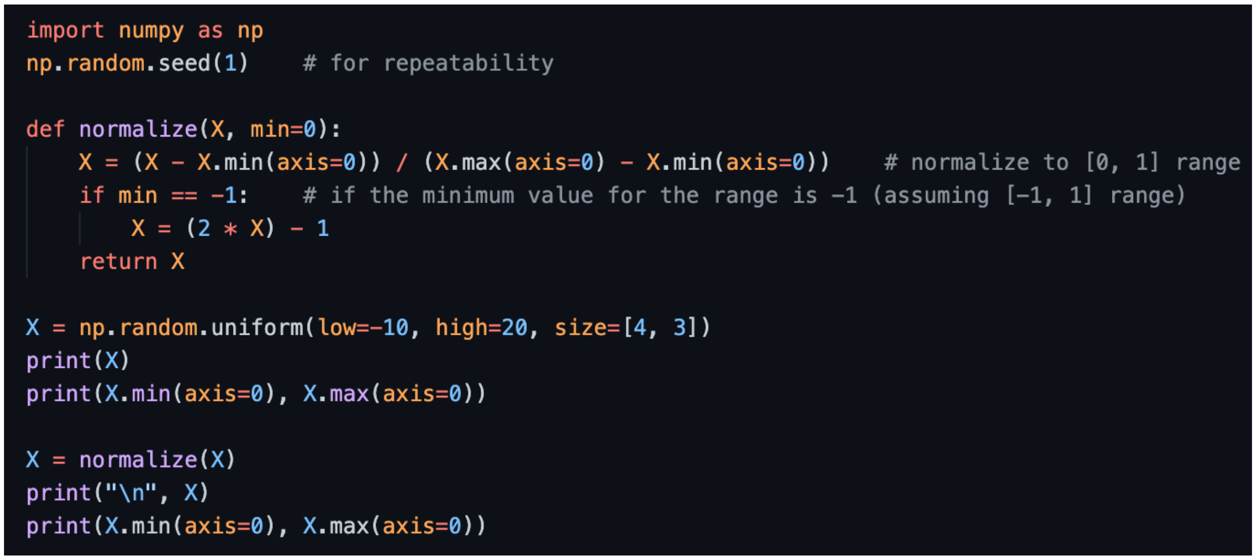


The blue points represent the [0, 1] data and the green points represent the [-1, 1] data. The maximum of the [0, 1] data is still 1 after normalizing to [-1, 1] because doubling its value and then subtracting by 1 results in the same value. The minimum of the [0, 1] is now -1 since 0 multiplied by 2 is still 0 so we only subtract 1. The datapoint right next to the minimum value would be doubled by such a small amount, there is no need for the doubling arrow.

Now that we can normalize one input feature to [0, 1] or [-1, 1] lets learn how to normalize multiple input features, which we would typically see in most datasets.

To do this, we need to decide on whether the input features would be normalized to [0, 1] or [-1, 1] and then we would normalize one input feature of the dataset to the desired normalization range and then another and another until every input feature of the dataset has been normalized to the desired normalization range. Luckily we are working with numpy so we do not need a loop.

Here is the code:



```
import numpy as np
np.random.seed(1)    # for repeatability

def normalize(X, min=0):
    X = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))    # normalize to [0, 1] range
    if min == -1:    # if the minimum value for the range is -1 (assuming [-1, 1] range)
        X = (2 * X) - 1
    return X

X = np.random.uniform(low=-10, high=20, size=[4, 3])
print(X)
print(X.min(axis=0), X.max(axis=0))

X = normalize(X)
print("\n", X)
print(X.min(axis=0), X.max(axis=0))
```

Instead of using X.min() without any arguments, which would return the minimum of the whole dataset, we add the keyword argument to compute the minimum value when compared to the other examples/rows (axis 0) of the dataset. The same reasoning applies to X.max() where we want the maximum value compared to other rows.

Now we ask numpy to randomly generate a [4, 3] matrix from a uniform distribution, where all values in between *low* (includes -10) and *high* (does not include 20) have equal probability of being drawn. The 4 by 3 matrix means that we have 4 rows/examples and 3 columns/input features. We can get the initial minimum and maximum values of each input feature by printing *X.min(axis=0)* and *X.max(axis=0)*. After calling the normalize function with no keyword arguments to normalize to the [0, 1] range, we print out the result and the minimum and maximum values of each input feature to check that they are all zeros and ones, respectively.

Please run this yourself to verify to yourself that this code makes sense and successfully normalizes each input feature to the [0, 1] range, with each input feature having a minimum of 0 and maximum of 1.

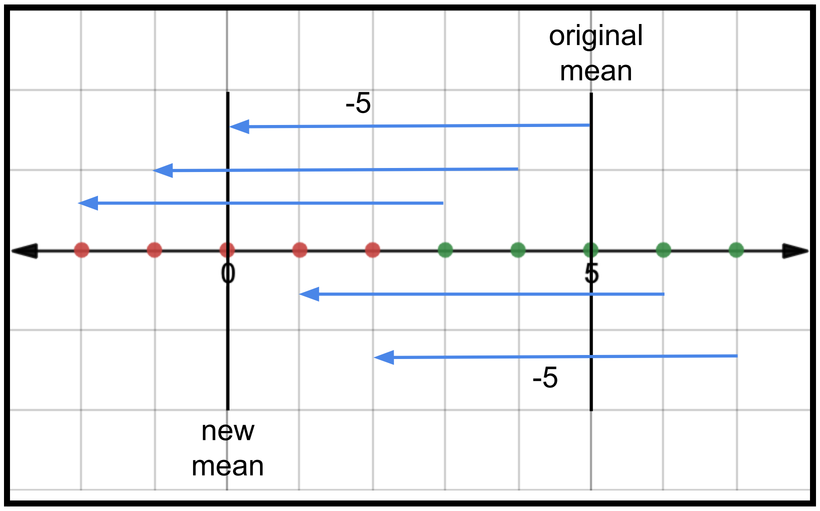
Normalizing each input feature to the [0, 1] range or [-1, 1] is one way to help with the training progress of the model, another is to **standardize** each input feature.

Standardization

Standardization is the process of bringing each input feature to have a mean (average value) of 0 and a standard deviation (average distance of each datapoint to the mean) of 1. The calculations for standardization are very simple. Suppose, we have one input feature:

$newX_i = \frac{X_i - \bar{x}}{\sigma}$ where the new datapoint is $newX_i$, the original datapoint is X_i, the mean of the input feature is $\bar{x}$, and the standard deviation of the input feature is σ.

For example, if our we have 5 examples and this is the one of input features of each example: [3, 4, 5, 6, 7], the mean is 5 ($\frac{3+4+5+6+7}{5\ examples}$) so to bring the dataset to have a mean of 0, we subtract 5 from each datapoint to get: [-2, -1, 0, 1, 2].



The green dots are the part of the original data while the red dots are the new data which now have a mean of 0 ($\frac{-2+(-1)+0+1+2}{5\ examples}$). The graphic on the right gives a visual understanding of the calculation in the numerator which brings the data to have a mean of 0.

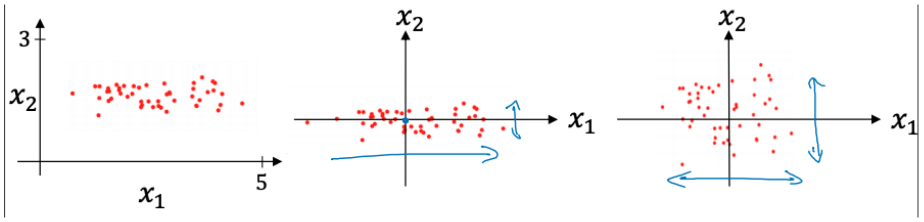
To understand how the denominator brings the data to have a standard deviation of 1, let's consider another example, this time we have brought the mean to 0 already: [-4.5, -2.75, 0.25, 3, 4]. Here is the calculation of the current standard deviation:

$$\sigma = \sqrt{\frac{(-4.5-0)^2+(-2.75-0)^2+(0.25-0)^2+(3-0)^2+(4-0)^2}{5\ examples}}$$

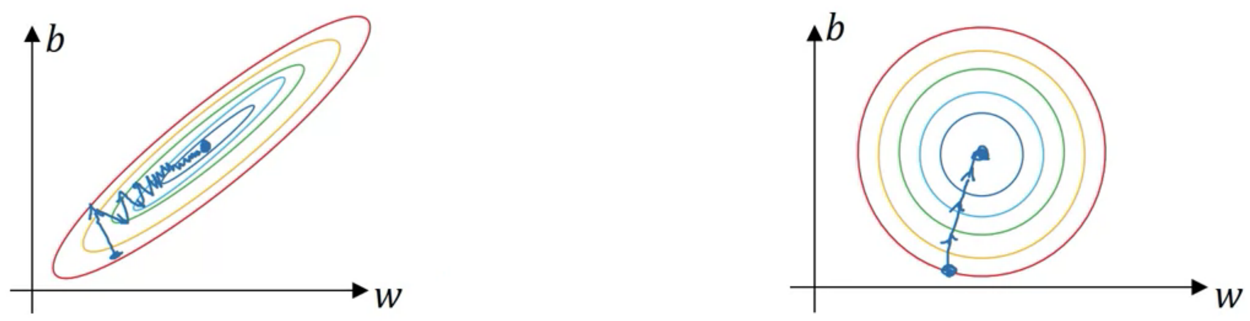
$$= \sqrt{\frac{(-4.5)^2+(-2.75)^2+(0.25)^2+(3)^2+(4)^2}{5\ examples}} = \sqrt{\frac{20.25+7.5625+0.0625+9+16}{5\ examples}}$$

$\sqrt{\frac{52.875}{5\ examples}} = \sqrt{10.575} \approx 3.252$ If you think of 3.252 as **how much the data was scaled to differ from the mean**, dividing by 3.252 would **“undo”** this scaling to make the average difference from the mean equal to 1. So we would have:
[-4.5/3.252, -2.75/3.252, 0.25/3.252, 3/3.252, 4/3.252]
= [~ -1.384, ~ -0.846, ~0.077, ~0.923, ~1.23]
Calculating the mean would still result in a mean of 0 but now the standard deviation is 0.99995234~1.000386, which is close enough and really close to exactly 1. A common positive side-effect of using standardization is that the minimum and maximum value ranges are pretty close to [-1, 1], so you can think of standardization both getting the mean to 0, standard deviation to 1 and partially normalizing the data to [-1, 1]. This is not a coincidence since [-1, 1] normalizations also have a mean close 0 and a standard deviation close 1 due to data points confined to at least having a value of -1 and at max a value of 1.

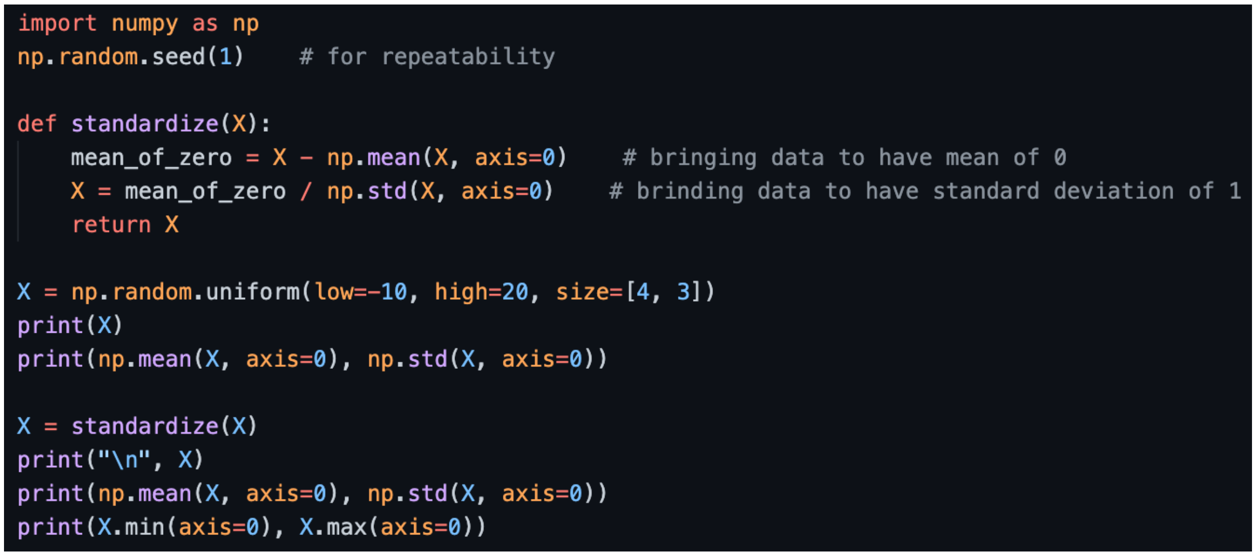
Before the code implementation, here are images that will help you understand where standardization will be useful. (Borrowed from Coursera’s Improving Deep Neural Networks course by DeepLearning.AI)



The first graph represents the original data. The second graph represents data with a mean of 0 but the data in the x_1 variable (think of one of the input features) is much more spread out (higher standard deviation) than the x_2 variable (think of another one of the input features). The third graph represents the data with both a mean of 0 and standard deviation of 1, making the dataset more **“standard”/symmetric**.



Here, the cost function contour plot using the original data is on the left contour plot and the cost function contour plot using the standardized data is on the right contour plot. As you can see, standardized data would make the contour plot more **"standard"/symmetric**, and take out the need for RMSProp to help the model navigate the cost function. Now here is the code for multiple input features:



```
import numpy as np
np.random.seed(1)    # for repeatability

def standardize(X):
    mean_of_zero = X - np.mean(X, axis=0)    # bringing data to have mean of 0
    X = mean_of_zero / np.std(X, axis=0)    # brinding data to have standard deviation of 1
    return X

X = np.random.uniform(low=-10, high=20, size=[4, 3])
print(X)
print(np.mean(X, axis=0), np.std(X, axis=0))

X = standardize(X)
print("\n", X)
print(np.mean(X, axis=0), np.std(X, axis=0))
print(X.min(axis=0), X.max(axis=0))
```

np.mean() and *np.std()* gives the mean and standard deviation respectively. We add the keyword argument in addition to the argument of the dataset to compute the mean and standard deviation when compared to the other examples/rows (axis 0) of the dataset. Before we call the *standardize()* function, we print the mean and standard deviation of each input feature. After calling the *standardize()* function, we check each input feature has a new mean is 0 and standard deviation is 1. At the same time, we print the minimum and maximum value of each input feature just for fun. Here is the output of means, standard deviations, minimums, and maximums:

```
[-5.55111512e-17 -6.07153217e-17  5.55111512e-17] [1. 1. 1.]
[-1.33384993 -1.2676956  -1.08879708] [1.35567688 1.51600271 1.45277571]
```

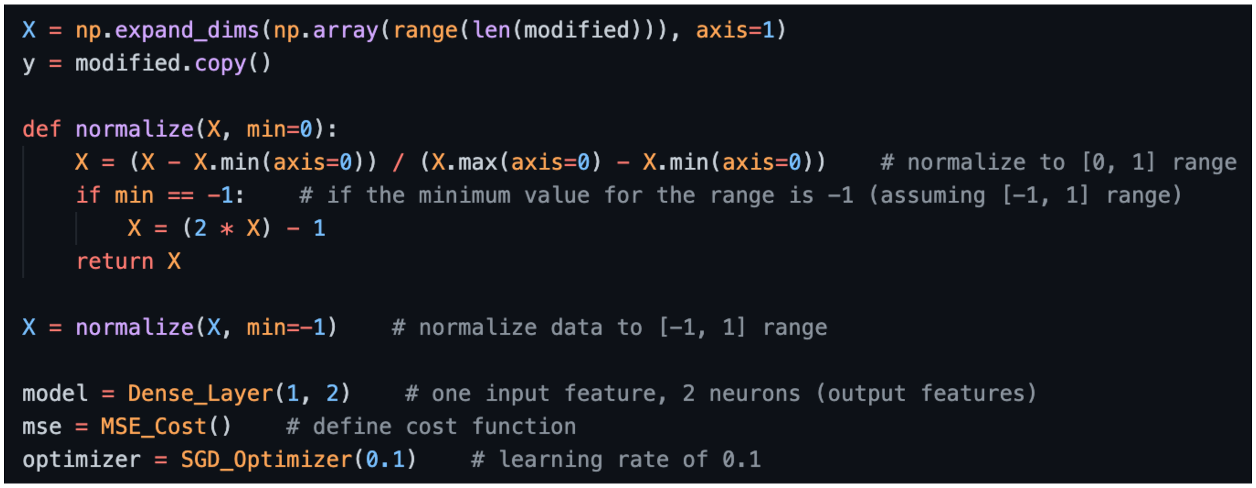
We see that the means are really, really close to 0 and the standard deviations are 1. At the same time, like we noted, standardization partially normalizes each input feature to the range of [-1, 1] .

Now we know how to normalize and standardize the input features to fix the training problems. In the next section, we will use these new techniques and hope that the modifications to the dataset would help improve the model and allow the model to learn a little more.

Linear Regression Is Not Good Enough!

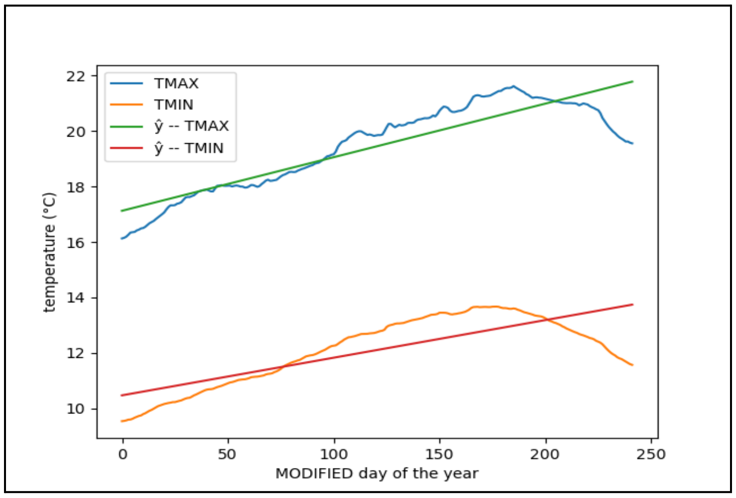
Since we are trying to predict values, maximum temperature and minimum temperature and our model only does not have a hidden layer, only an input layer and output layer, this type of model is called "**linear regression**". The regression comes from the type of supervised learning we are conducting (predicting values) while linear comes from the fact that we have no hidden layers.

Now from the title of the section, get ready to see the problems of having no hidden layers in these last two sections of the unit. Now here is the code that incorporates our normalization to the range of [-1, 1]:



```
X = np.expand_dims(np.array(range(len(modified))), axis=1)
y = modified.copy()

def normalize(X, min=0):
    X = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))    # normalize to [0, 1] range
    if min == -1:    # if the minimum value for the range is -1 (assuming [-1, 1] range)
        X = (2 * X) - 1
    return X

X = normalize(X, min=-1)    # normalize data to [-1, 1] range

model = Dense_Layer(1, 2)    # one input feature, 2 neurons (output features)
mse = MSE_Cost()    # define cost function
optimizer = SGD_Optimizer(0.1)    # learning rate of 0.1
```

Right after using a 125-day moving average to create a new dataset and right before defining the model, we add in the *normalize()* function and call it. Now we start with a learning rate of 0.1 and everything else stays the same. Here is the graph we created and the terminal output:



```
Initial Cost: 264.3984666835997
264.3984666835997
33.133261362043456
4.744147275248193
1.155075238416936
0.6492693031265316
0.5527181120036495
0.5232858383336073
0.5107771271163584
```

```
0.5047490078377698
0.5017411869091924
Final Cost: 0.5002273014160387
```

We see that the model is **stuck** at a cost of around 0.5 and we see see that ŷ (the model's prediction) is a **straight line!** The model does not even adjust to the downward trend towards the end of the year. To understand why the model outputs a straight line, remember that the calculation of the model's neurons originates from the equation **y = mx + b**, which produces a **linear line**. This is also where the "linear" in "linear regression" comes from. A model without any hidden layers creates a linear graph which is not very sophisticated since most real-world datasets do not have linear trends.

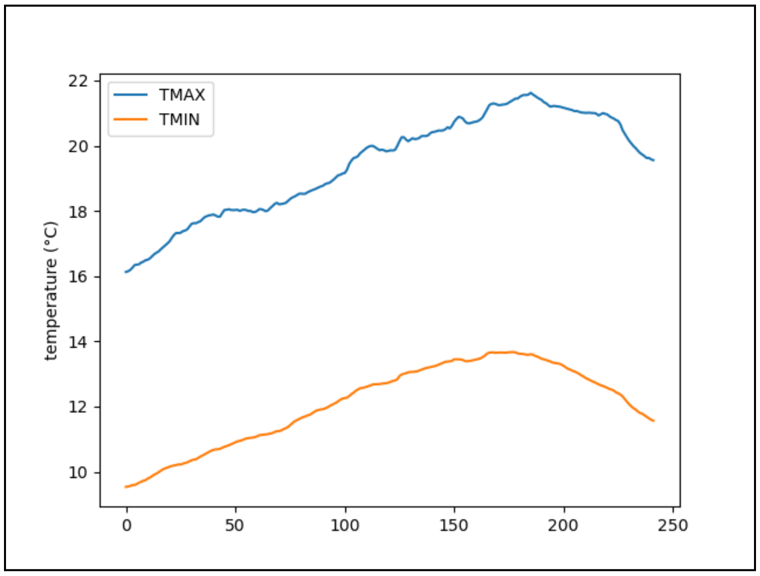
Clearly, our dataset follows the shape of a curve rather than a line, so one way we can get the model to output curves is to change the calculation in the neuron. We can also think of regression as **fitting/creating** a **function** that produces a graph of the dataset.

As a result, the AI is technically learning to find a mathematical function that we can use to **represent** the dataset using its parameters (weights and biases). For the AI to predict on examples (data) that it has not seen in training which would occur when deploying the model, the AI would plug in all the input features into the function it has created.

In the next section, we will focus on **Polynomial Regression** since the curves for our dataset can be created using **polynomials**.

Higher-Order Polynomial Regression Is Better than Linear Regression!

If we were to use a simple polynomial without many degrees to **fit** our data, what type of polynomial would be used? In other words, what is the minimum value for d in the leading term of the polynomial: x^d? Our dataset is on the right as a reminder of the shape of our data while you hypothesize an answer to these questions.



Let's analyze this graph again. For TMAX and TMIN, these temperatures are small but increase as days pass. Until most of the days of our modified year have passed, these temperatures begin to decrease. What polynomial increases for some time and then begins to decrease? The answer is an up-side-down parabola, more specially a quadratic polynomial that has a leading degree of 2.

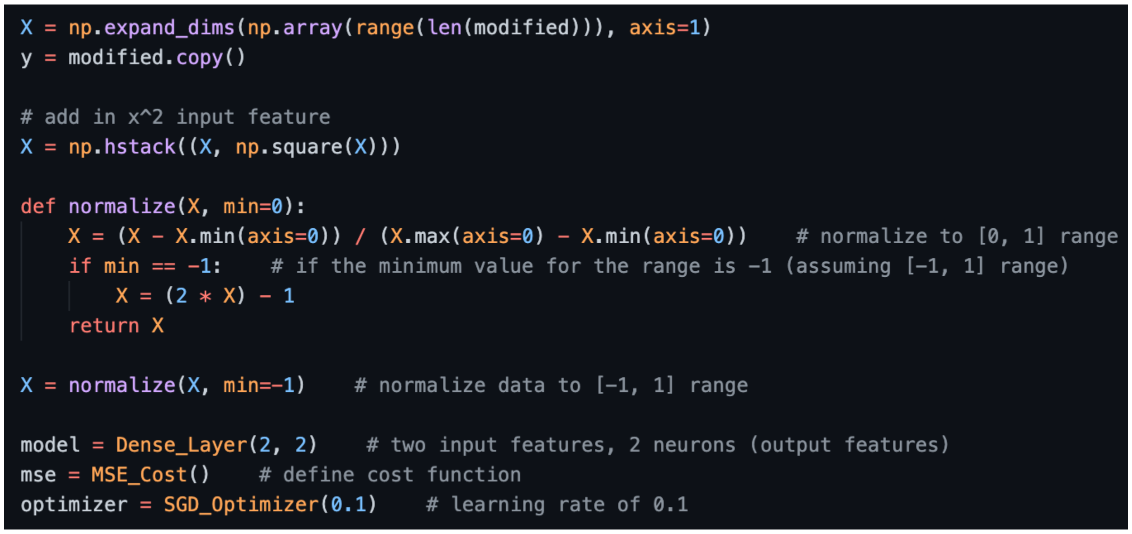
The equation for this polynomial would be $-ax^2 + bx + c$. The negative in front of the variable *a* is responsible for flipping the parabola up-side-down. Now how do we implement this equation into our calculation for the output of the neuron? Do we have to change both the *forward()* and *backward()* ?

Recall that the computation of the current neuron is: $w_1X_1 + w_2X_2 + \ldots + b$
The equation for the quadratic polynomial is $-ax^2 + bx + c$.
In reality the computation of the current neuron is similar to the equation for a quadratic polynomial.

The bias parameter in the neuron is similar to the variable *c* in the quadratic polynomial. The coefficient, *-a* in front of x^2 is similar to the first weight parameter while the coefficient, *b* in front of x is similar to the second weight parameter. Since the quadratic polynomial equation only has 2 terms that depend on the input, we would only need two weight parameters and one bias parameter. Wait, but what about the input features? We need 2 input features for the 2 weight parameters. Well the first input feature for the first weight parameter would be our original data squared (x^2) and the second input feature for the second weight parameter would be our original data (x). To summarize our computation for the current neuron this situation would be:

$\hat{y} = w_1X_1 + w_2X_2 + b$ where $X_1 = x^2$ and $X_2 = x$.

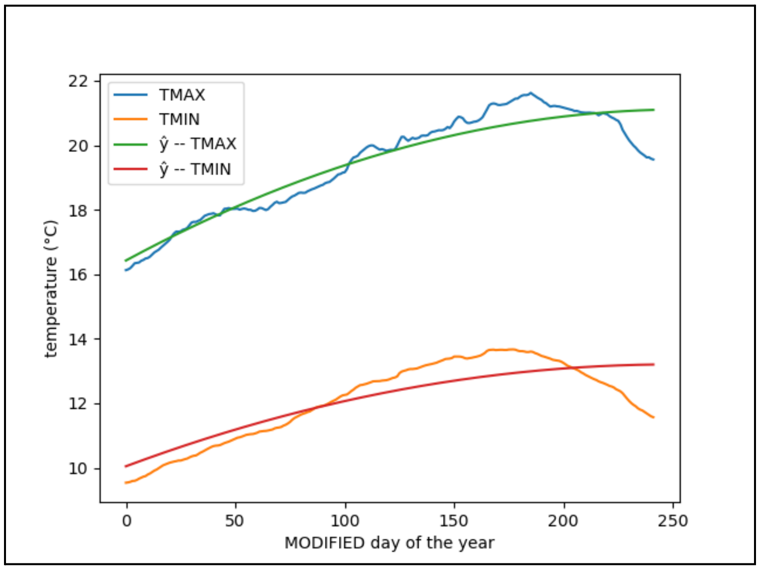
So in addition to our input feature x, we need a new input feature x^2. Here is the code:



```
X = np.expand_dims(np.array(range(len(modified))), axis=1)
y = modified.copy()

# add in x^2 input feature
X = np.hstack((X, np.square(X)))

def normalize(X, min=0):
    X = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))    # normalize to [0, 1] range
    if min == -1:    # if the minimum value for the range is -1 (assuming [-1, 1] range)
        X = (2 * X) - 1
    return X

X = normalize(X, min=-1)    # normalize data to [-1, 1] range

model = Dense_Layer(2, 2)    # two input features, 2 neurons (output features)
mse = MSE_Cost()    # define cost function
optimizer = SGD_Optimizer(0.1)    # learning rate of 0.1
```

Right after using a 125-day moving average to create a new dataset and right before we define the *normalize()*, we add in the x^2 input feature. To get x^2 we use *np.square(X)* where each element in the dataset is squared. We combine X and the result of *np.square(X)* in a tuple as an argument to *np.**hstack**()* which **stacks** X, and the result of *np.square(X)* **horizontally**. We stack X and the result of *np.square(X)* horizontally, because the horizontal axis is the column axis which is the input feature axis.

Note that we normalize our data **after** we have all our input features rather than normalize our data and then create new input features. Now when we define the model, the first argument would be 2 instead of 1. Also, we save the figure of our graph we created to "*QR.png*".

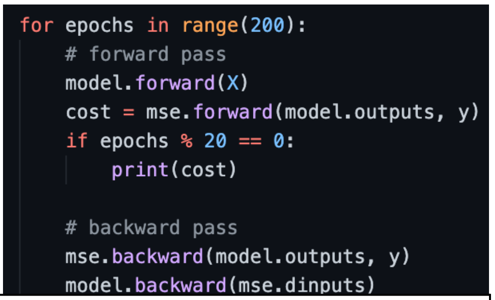


Initial Cost: 264.5343255867199 → Final Cost: 0.2613055794166701

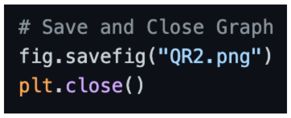
We see that the model has learned to create some sort of a curve. The model still has to learn to curve downwards so let's increase the number of epochs to 200 and use a momentum of 0.9 for the model to learn faster.

Here is the code:

In the training loop, we print out the cost every 20 epochs instead of every 10 epochs to compensate for the 100 more epochs. Also save the figure of our graph we created to "*QR2.png*".

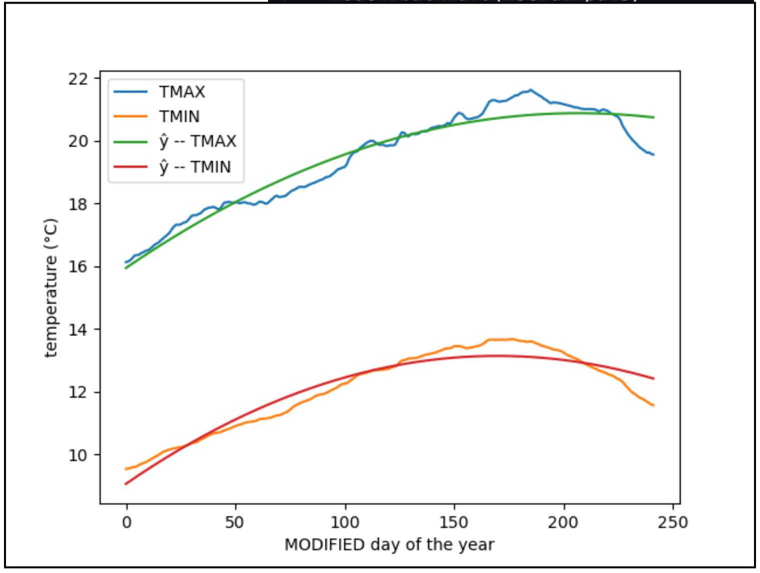


```
for epochs in range(200):
    # forward pass
    model.forward(X)
    cost = mse.forward(model.outputs, y)
    if epochs % 20 == 0:
        print(cost)

    # backward pass
    mse.backward(model.outputs, y)
    model.backward(mse.dinputs)
```



```
# Save and Close Graph
fig.savefig("QR2.png")
plt.close()
```

```
Initial Cost: 264.5343255867199
264.5343255867199
13.69513229135725
0.8204689785440524
0.38067196867474884
0.21476034091574095
0.15716407186349216
0.14525492486409258
0.1408018078193619
0.1379875032101439
0.13625002615971687
Final Cost: 0.13517400091726112
```

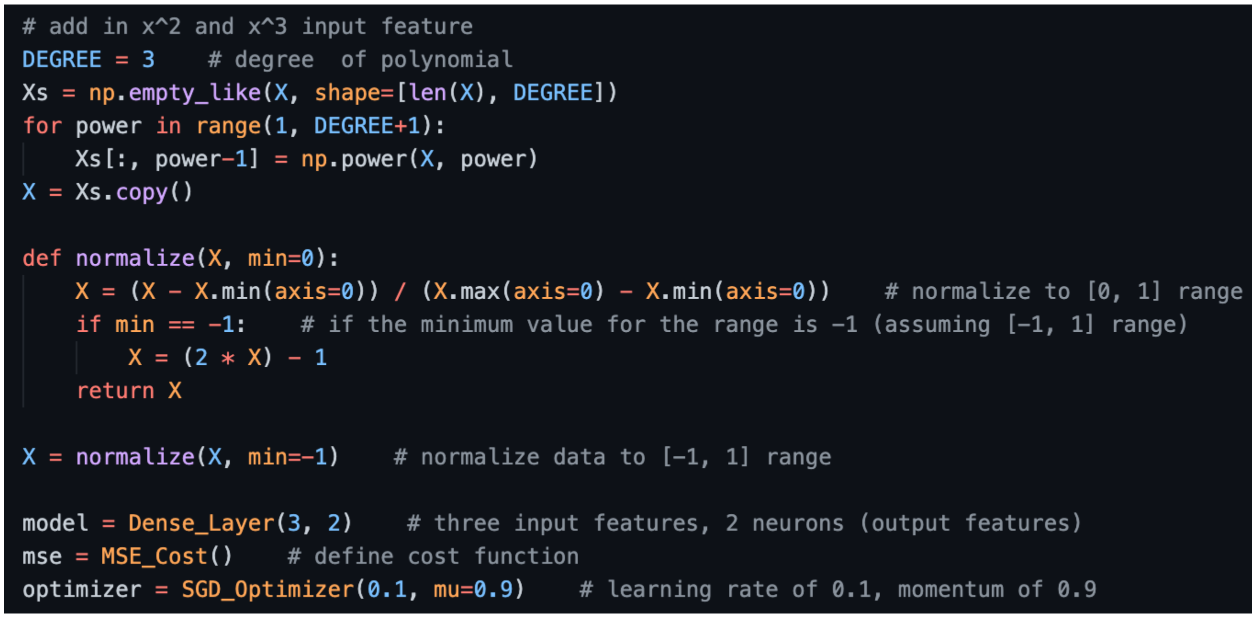


We now the model has found at least the general trend of increase for most of the days of our modified year and then some decrease. Now seen from the training progress, more training would not make much of a difference since the model is stuck at a cost around 0.13. Since our model's (quadratic polynomial) predictions still do not really fit our data very well, a polynomial with a leading degree of 2 is not enough. To improve we can use **higher-order polynomials** which have higher leading degrees.

Just to recap: we have started from **Linear Regression (LR)** to **Quadratic Regression (QR)** and now we are going to use **Higher-Order Polynomial Regression (PR)**.

Let’s start with a 3rd degree polynomial: $ax^3 + bx^2 + cx + d$. If you have guessed, we have three input features x^3, x^2, and x^2 where the three corresponding weight parameters are the variables *a*, *b*, *c*, respectively. The bias parameter would represent the variable *d*.

Here is the code for creating the x^2 and x^3 input features:



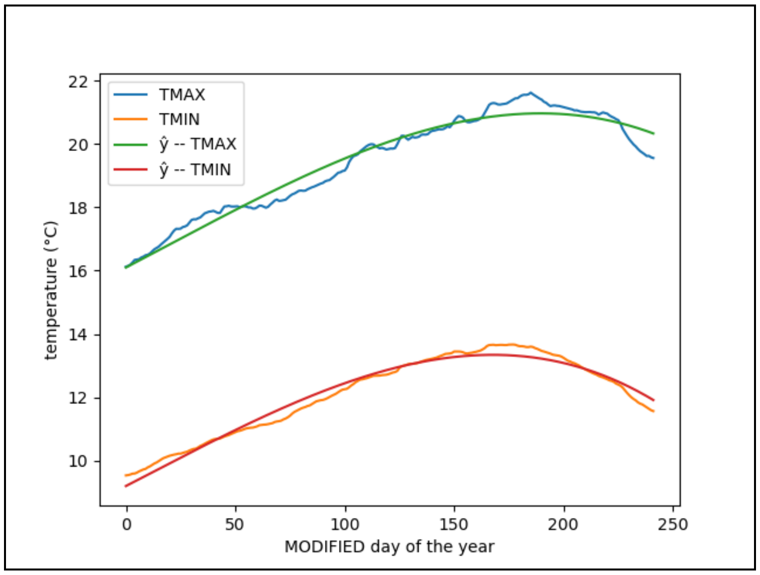
```python
# add in x^2 and x^3 input feature
DEGREE = 3    # degree  of polynomial
Xs = np.empty_like(X, shape=[len(X), DEGREE])
for power in range(1, DEGREE+1):
    Xs[:, power-1] = np.power(X, power)
X = Xs.copy()

def normalize(X, min=0):
    X = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))    # normalize to [0, 1] range
    if min == -1:    # if the minimum value for the range is -1 (assuming [-1, 1] range)
        X = (2 * X) - 1
    return X

X = normalize(X, min=-1)    # normalize data to [-1, 1] range

model = Dense_Layer(3, 2)    # three input features, 2 neurons (output features)
mse = MSE_Cost()    # define cost function
optimizer = SGD_Optimizer(0.1, mu=0.9)    # learning rate of 0.1, momentum of 0.9
```

We delete the function to make X a matrix right after we use a 125-day moving average to create a new dataset. This is because now the X matrix will be created by our multiple input features. To start, we define a constant *DEGREE* which represents the degree of the polynomial we are creating. After that we create a new variable *Xs* to represent all the input features which is an empty numpy array with properties of the variable *X* but has the number of columns equal to *DEGREE* (3, input features).

Next we use a for loop that starts at 1 and stops at *DEGREE+1* which does not include *DEGREE+1* so the first value the variable *power* would be is 1 and the last would be equal to *DEGREE* (3). For every iteration, we index all rows of Xs but the column is *power-1* (*power* starts at 1, so column 0; *power* ends at 3, so column 2) and set them equal to the variable *X* raised to the variable *power* using *np.power()* where the first argument is the base and the second argument is the **power** to raise the base to. After we finish filling up the matrix *Xs* we copy it to the variable *X*.

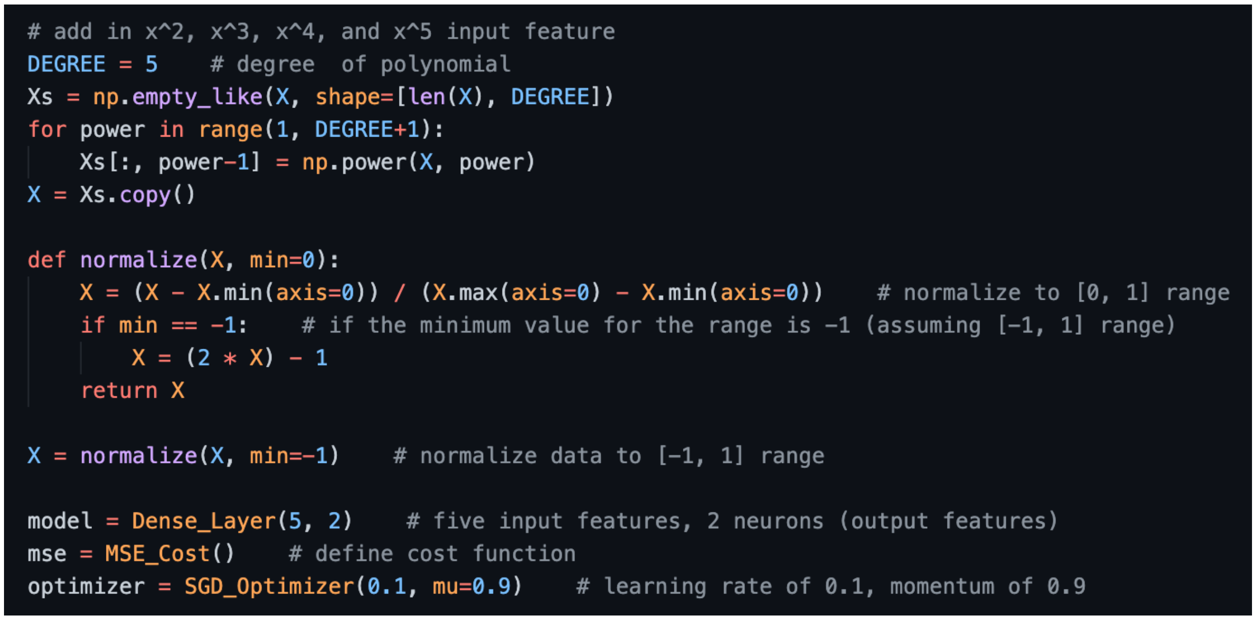
Don't forget that now since we have 3 input features, the first argument to define the model should be 3. Lastly, save the figure of our graph we created to “*PR_deg3.png*”.

```
Initial Cost: 264.64876535301676
264.64876535301676
9.85647356351811
0.8902061016978299
0.5338898067708867
0.09803493422264978
0.07641928609905825
0.07506788967784367
0.07368725640837713
0.07299030375672243
0.07234961989740749
Final Cost: 0.07171993904431259
```



We see that the cost has halved from about 0.13 to 0.07 and the model's prediction for TMIN is almost good enough however the model's prediction for TMAX still needs more improvement so let's try a polynomial degree of 5: $ax^5 + bx^4 + cx^3 + dx^2 + fx + g$ (skipped letter **e** to not confuse with **Euler's Number**).

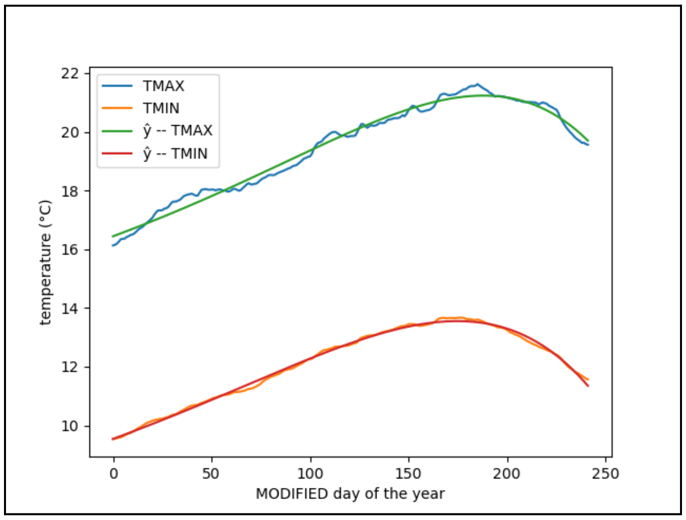
Same pattern, 5 input features so 5 weight parameters. The bias parameter represents the variable *g*. The 5 input features are x^5, x^4, x^3, x^2, x and their corresponding weight parameters are the variables *a*, *b*, *c*, *d*, *f*, respectively. Here is the code:



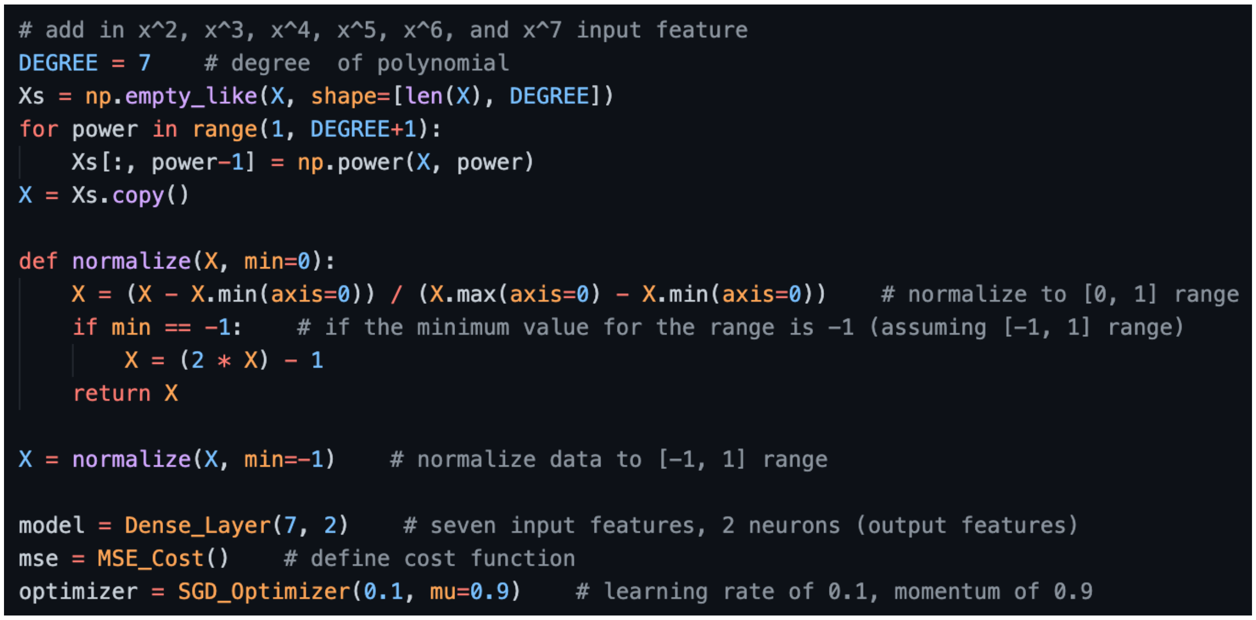
```python
# add in x^2, x^3, x^4, and x^5 input feature
DEGREE = 5    # degree  of polynomial
Xs = np.empty_like(X, shape=[len(X), DEGREE])
for power in range(1, DEGREE+1):
    Xs[:, power-1] = np.power(X, power)
X = Xs.copy()

def normalize(X, min=0):
    X = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))    # normalize to [0, 1] range
    if min == -1:    # if the minimum value for the range is -1 (assuming [-1, 1] range)
        X = (2 * X) - 1
    return X

X = normalize(X, min=-1)    # normalize data to [-1, 1] range

model = Dense_Layer(5, 2)    # five input features, 2 neurons (output features)
mse = MSE_Cost()    # define cost function
optimizer = SGD_Optimizer(0.1, mu=0.9)    # learning rate of 0.1, momentum of 0.9
```

The only changes are the variable *DEGREE* where the value assigned is now 5 and the first argument to define the model is now 5, for our 5 input features. Also we save the figure of the graph we created to "*deg5.png*".

```
Initial Cost: 264.76352497733455
264.76352497733455
7.9325676576110835
1.5065613827493884
0.47983246512583383
0.028791285543918525
0.029251762510347907
0.02472655007719962
0.023924067953832444
0.023685638099032905
0.023470662462838885
Final Cost: 0.023283506246733695
```



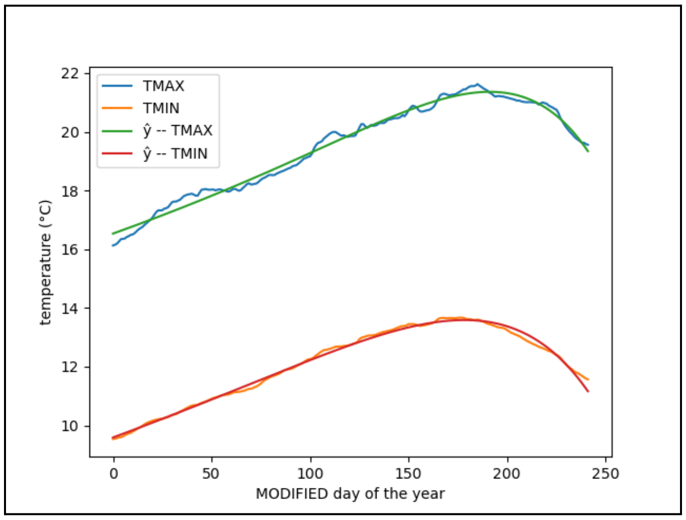
Now the model's prediction of TMIN is good enough but although the model's prediction of TMAX has improved, we need the model to do better. So let's try a polynomial degree of 7: $ax^7 + bx^6 + cx^5 + dx^4 + fx^3 + gx^2 + hx + j$ (skipped letter **e** to not confuse with **Euler's Number** and skipped letter ***i*** to not confuse with **imaginary numbers**). For the sake of preventing repetition, you should get the idea of how we can use linear neurons and compute polynomials. Here is the code:



```
# add in x^2, x^3, x^4, x^5, x^6, and x^7 input feature
DEGREE = 7    # degree  of polynomial
Xs = np.empty_like(X, shape=[len(X), DEGREE])
for power in range(1, DEGREE+1):
    Xs[:, power-1] = np.power(X, power)
X = Xs.copy()

def normalize(X, min=0):
    X = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))    # normalize to [0, 1] range
    if min == -1:    # if the minimum value for the range is -1 (assuming [-1, 1] range)
        X = (2 * X) - 1
    return X

X = normalize(X, min=-1)    # normalize data to [-1, 1] range

model = Dense_Layer(7, 2)    # seven input features, 2 neurons (output features)
mse = MSE_Cost()    # define cost function
optimizer = SGD_Optimizer(0.1, mu=0.9)    # learning rate of 0.1, momentum of 0.9
```

The only changes are the variable *DEGREE* where the value assigned is now 7 and the first argument to define the model is now 7, for our 7 input features. Also we save the figure of the graph we created to "*deg7.png*".

```
Initial Cost: 265.0258507803095
265.0258507803095
3.3306326927131997
3.0170200559492177
0.22383130209194257
0.06577942190016044
0.026171641415278357
0.023976040937241122
0.022794099201022777
0.022216891786003605
0.021816118228230798
Final Cost: 0.021500716979133368
```



We see that the improvement in the model's prediction of TMAX is very small and the final cost of about ~0.0215 for a 7th-degree polynomial is really close to the final cost of about ~0.0232835 for a 5th-degree polynomial. As a result, using more and more higher-degree polynomials has limitations where increasing the degree in low-order polynomials returns a better and better cost but increasing the degree in high-order polynomials returns a better cost but at a smaller extent. Despite the fact that we simplified the temperature dataset by using a 125-day moving average, a 7th degree polynomial is still not good enough for TMAX. What else can we add or modify to make our model smarter?

Remember that this unit is to show the limitations of a model without hidden layers, only an input and output layer. As a result, we should add in hidden layers to our model so that we will have more neurons. These hidden layers would allow us to remove the need for the extra input features we have created in this unit. In the next unit, we will introduce the new concepts that come with hidden layers in the model and the model's prediction for TMAX will be much better and closer to the actual answer. At the same time, the final cost of the model after training will also be lowered as a result of hidden layers.