Deep learning project – Weather prediction

Submitters:

Hakeem Abushqara

Basel Mousa

Introduction:

Weather forecasting is essential because it affect many aspects of society, including agriculture, transportation, and disaster management. The ability to accurately predict weather conditions, such as temperature and sunshine, helps in planning and decision-making processes that can minimize risks and improve how well different areas of work run.

In this project our goal is to predict London's weather based on historical weather observations. Our dataset is provided by the European Climate Assessment (ECA), which includes a wide array of weather attributes recorded by a weather station near Heathrow airport in London, UK. Our approach is creating a many-to-one bidirectional LSTM model (RNN) as these types of models are suited for time-series forecasting tasks such as weather prediction, where given a sequence of days, we predict the mean_temp and sunshine values for the next day.

Data pre-processing and preparing:

Data preprocessing includes:

- filling in the missing values of the data using "backward fill" and "forward fill" method.

```
df.interpolate(method='bfill', inplace=True)#fill in the missing values
df.interpolate(method='ffill', inplace=True
```

- We selected the features we want to work with for the learning phase, we chose to work with all the features since sunshine is very challenging to predict, and because that has led to the best performance of the model.

- We split the data to train (63.75%), validation (21.25%), and test set (15%), and we scale the data (standard scaling) as it is one the steps to prepare it to be the input of the model.

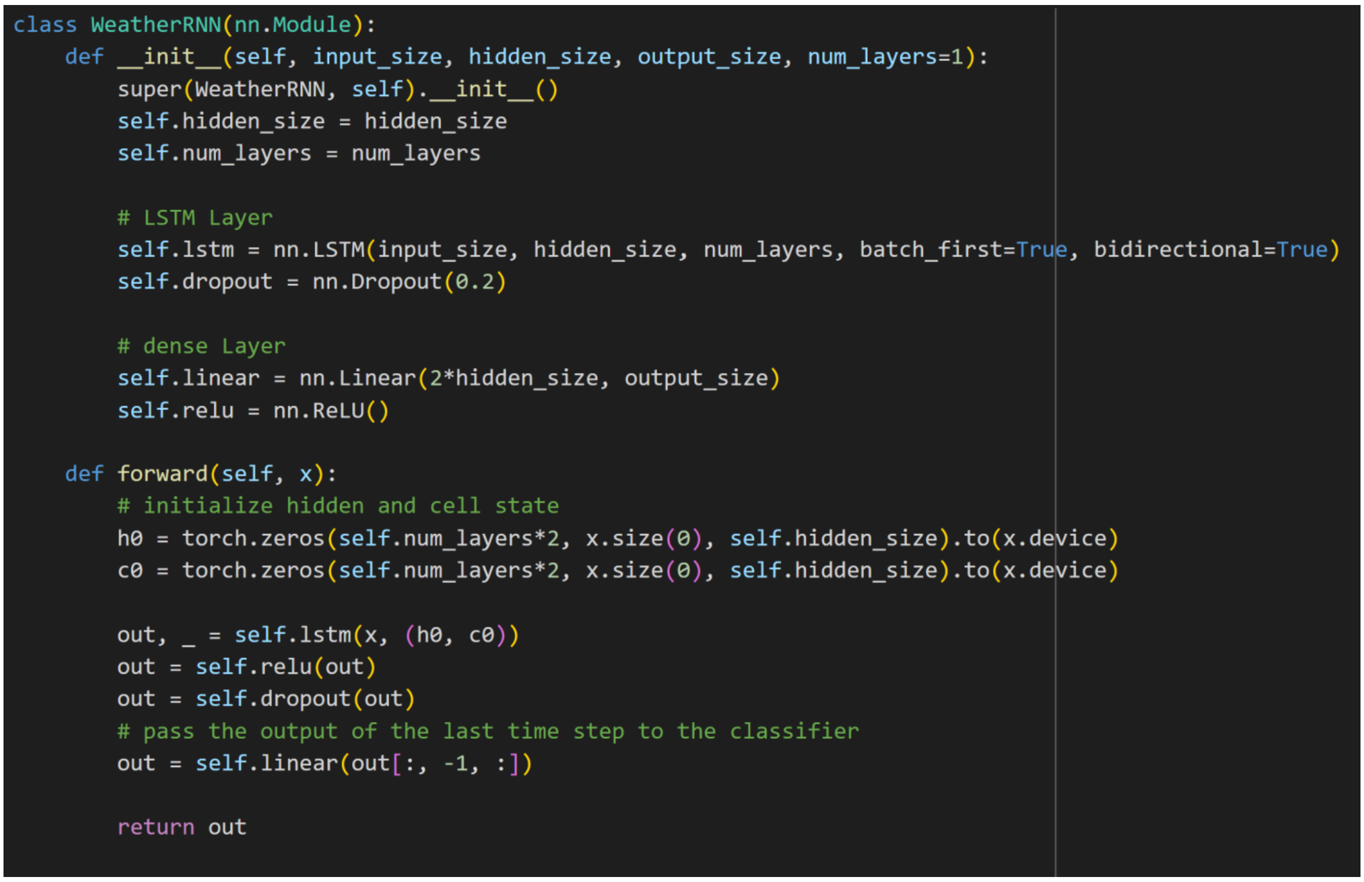
- We have created a function which receives the features, the labels, and a window size and creates sequences of size 'window' because as we mentioned earlier the model's input would be a sequence of days. We apply this function on the data to create sequences and then we convert the sequences into tensors and then create DataLoaders for train, validation, and test data.

Model’s Architecture:

- **BI-LSTM layer**: we used a bidirectional LSTM with two layers with a hidden size of 256, this enables the model to process the sequences in forward and backward directions, ensuring a comprehensive understanding of the sequence.
- **Activation function**: Relu activation function is applied after the LSTM layer.
- **Dropout**: a dropout layer with a 0.2 rate to decrease the risk of overfitting and increase the model’s capability to generalize.
- **Linear layer**: a fully connected layer with hidden size of 512 and output size of 2 (the number of the features to predict)

The optimizer used is Adam, batch size selected in our work is 32, loss function used in training is customized weighted MSEloss, window size for creating sequences is 7.

The model’s code:



```python
class WeatherRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(WeatherRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # LSTM Layer
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True, bidirectional=True)
        self.dropout = nn.Dropout(0.2)

        # dense Layer
        self.linear = nn.Linear(2*hidden_size, output_size)
        self.relu = nn.ReLU()

    def forward(self, x):
        # initialize hidden and cell state
        h0 = torch.zeros(self.num_layers*2, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers*2, x.size(0), self.hidden_size).to(x.device)

        out, _ = self.lstm(x, (h0, c0))
        out = self.relu(out)
        out = self.dropout(out)
        # pass the output of the last time step to the classifier
        out = self.linear(out[:, -1, :])

        return out
```

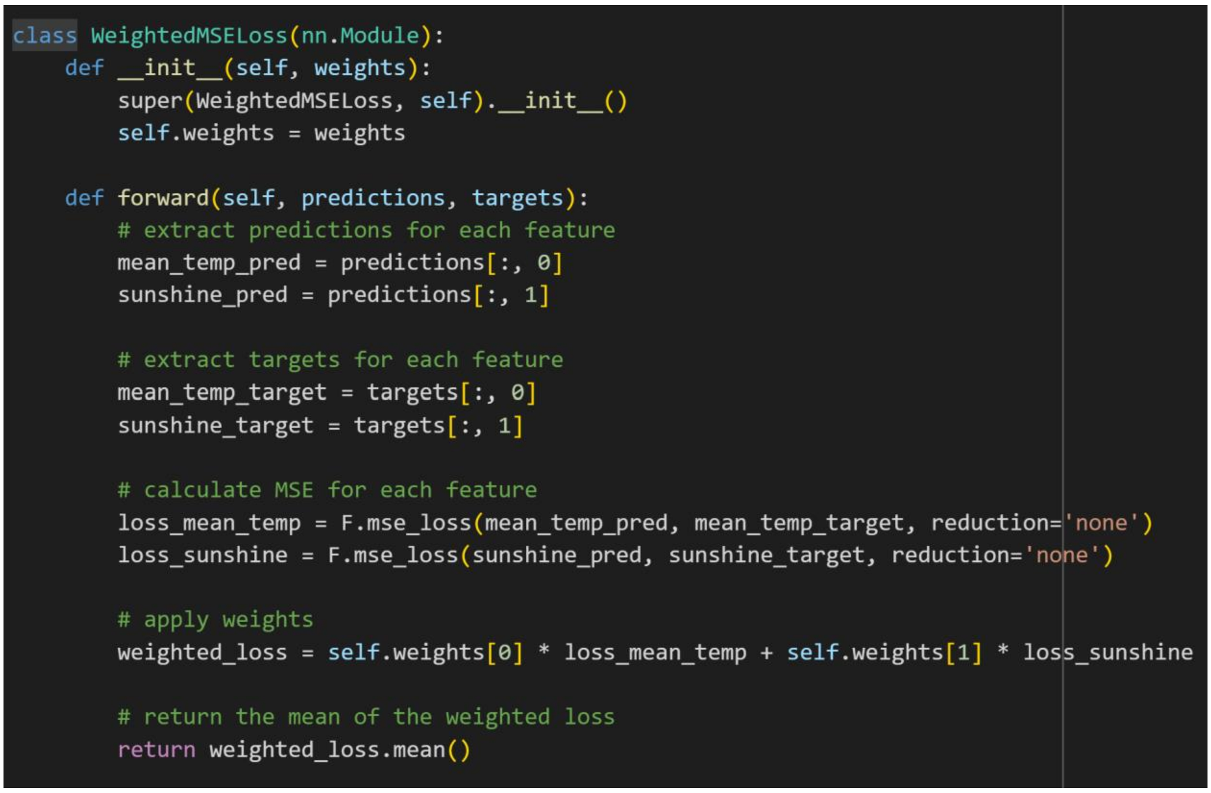
we tried two more different and more complex models but did not get any better results, so we stuck with this simple one.

Training phase:

As we mentioned earlier, for our training phase we used customized weighted MSE loss function so we can put more weight for the sunshine loss since it was much more challenging to learn and predict than the mean_temp feature.

The loss function receives the weights values as an input, calculates and returns the weighted MSE loss. The weights given for mean_temp and sunshine were 0.2, 0.8 respectively.

The function's code:


```
class WeightedMSELoss(nn.Module):
    def __init__(self, weights):
        super(WeightedMSELoss, self).__init__()
        self.weights = weights

    def forward(self, predictions, targets):
        # extract predictions for each feature
        mean_temp_pred = predictions[:, 0]
        sunshine_pred = predictions[:, 1]

        # extract targets for each feature
        mean_temp_target = targets[:, 0]
        sunshine_target = targets[:, 1]

        # calculate MSE for each feature
        loss_mean_temp = F.mse_loss(mean_temp_pred, mean_temp_target, reduction='none')
        loss_sunshine = F.mse_loss(sunshine_pred, sunshine_target, reduction='none')

        # apply weights
        weighted_loss = self.weights[0] * loss_mean_temp + self.weights[1] * loss_sunshine

        # return the mean of the weighted loss
        return weighted_loss.mean()
```

Initially, our number of epochs is 40, but we used early stoppage for the training phase, after every epoch in the training, we evaluate the model's performance on the validation set, we keep track of the best (lowest) validation error, and once we notice that there has been 5 epochs with no improvement (validation error did not get lower than the lowest value we got so far) we stop the training phase due to no improvement and to prevent the risk of overfitting.

Testing method and results:

For evaluation phase, we calculated the average loss, and we saved the model's predictions and compared them to the actual values using different metrics like MSE, RMSE, MAE, and R_squared.

The results:

Average test loss: 7.1566

MSE: 5.1902

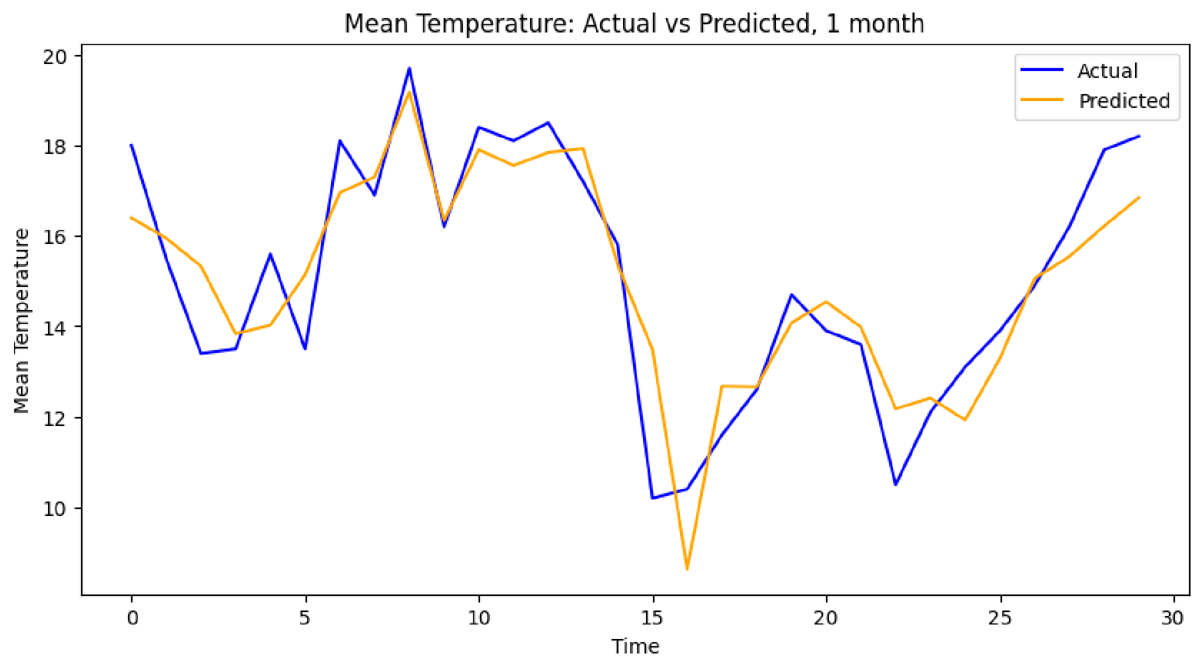
RMSE: 2.2782

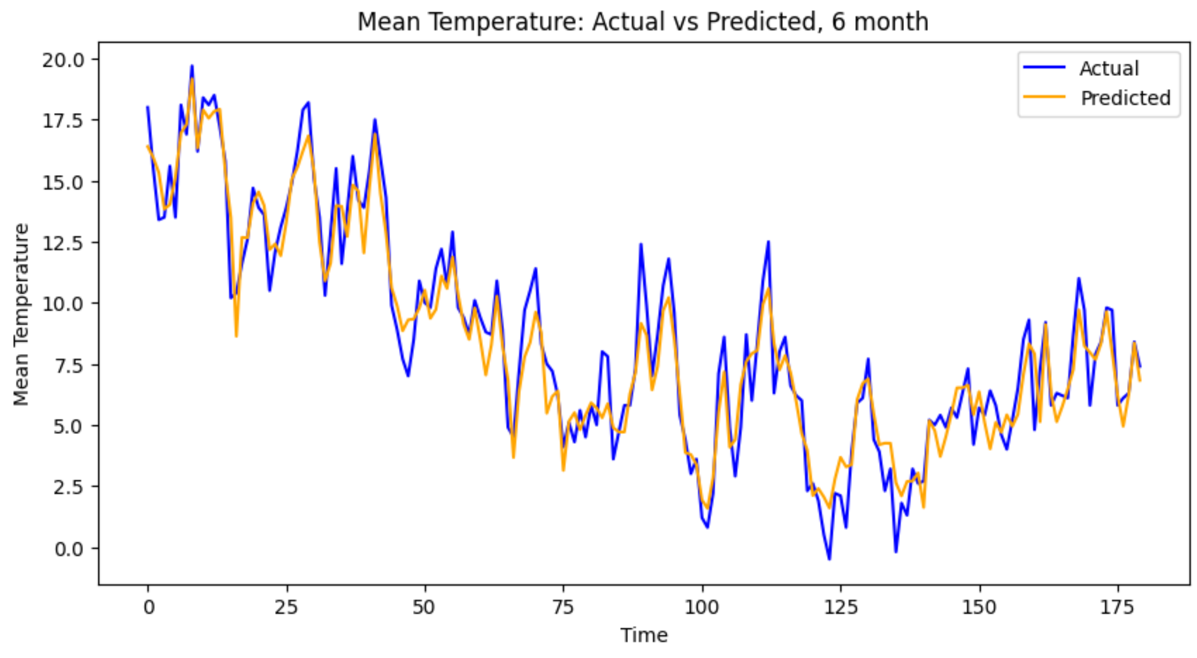
MAE: 1.7263

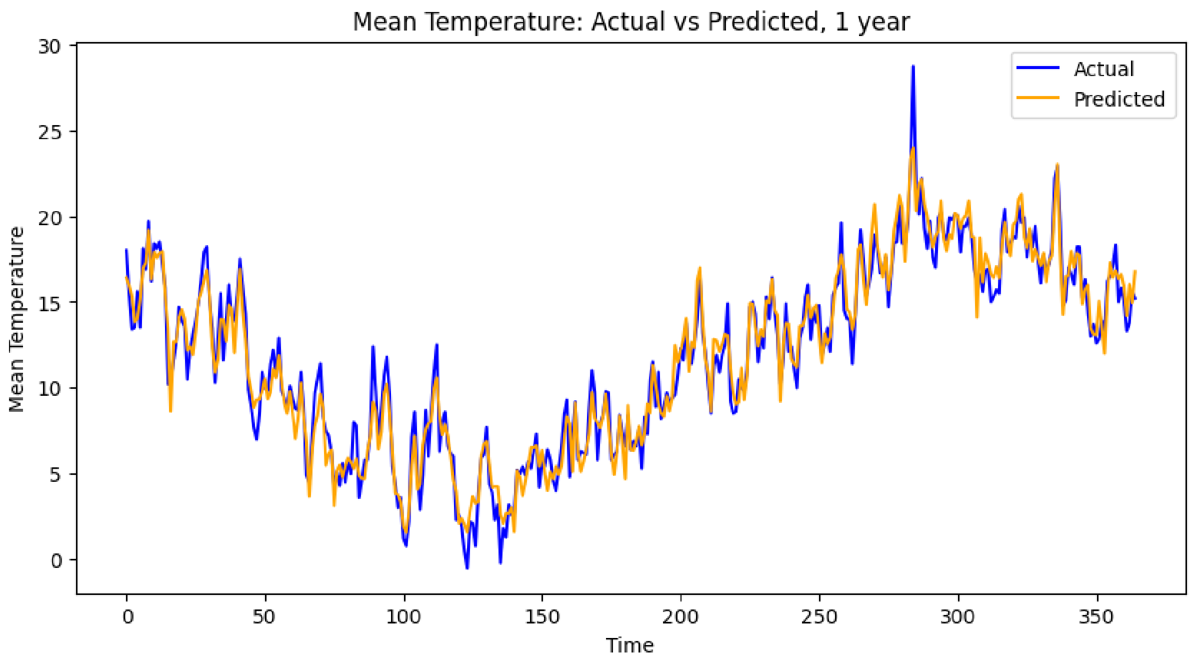
R_squared: 0.7106.

In addition, we plotted a graph comparing between the predictions and the results including results for one month, 6 months, 1 year, and all the predictions. In addition, we plot a graph for each feature that shows the distribution of the differences between the predictions and actuals, and a graph that shows the train and validation loss.

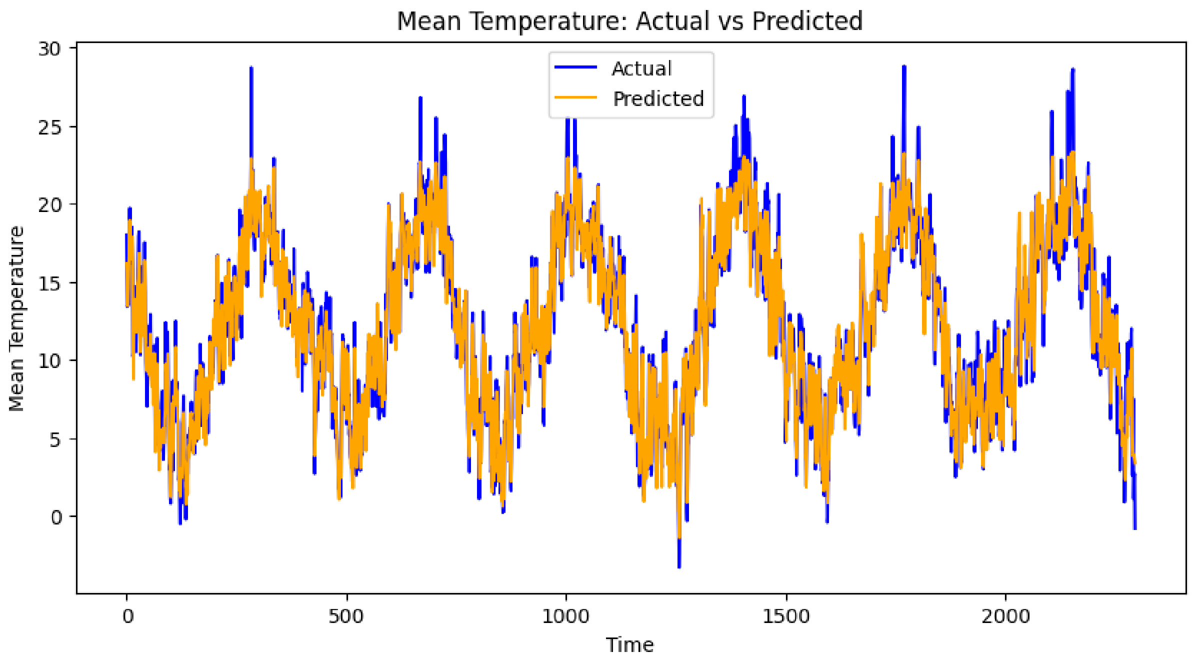
The plots:



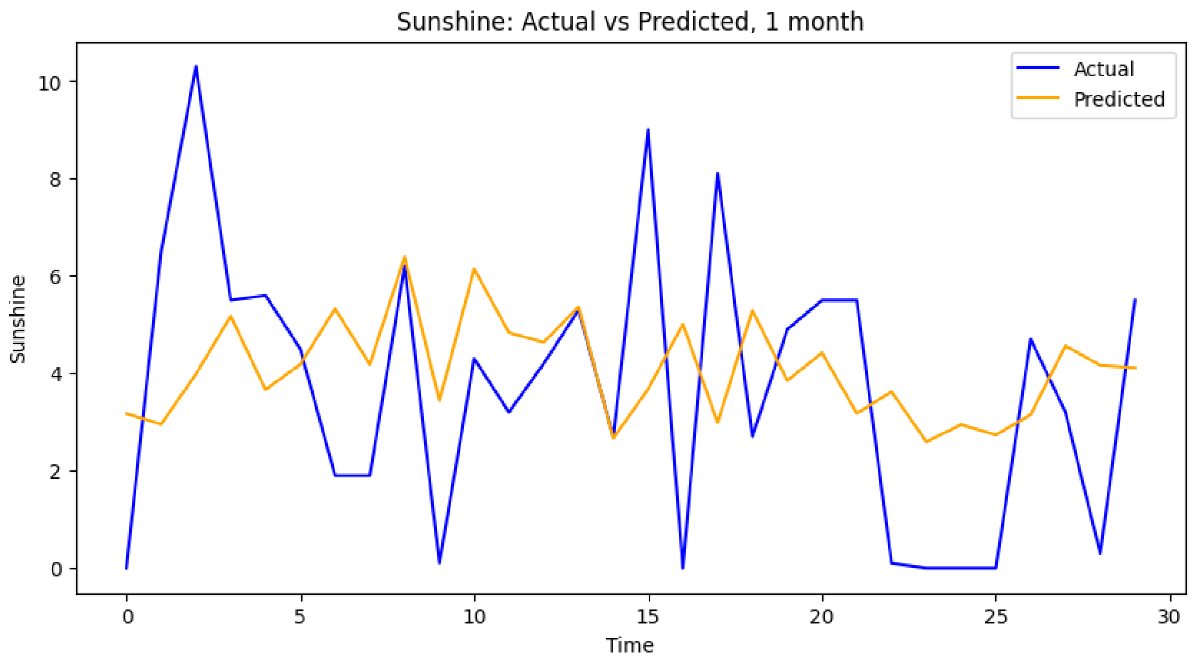


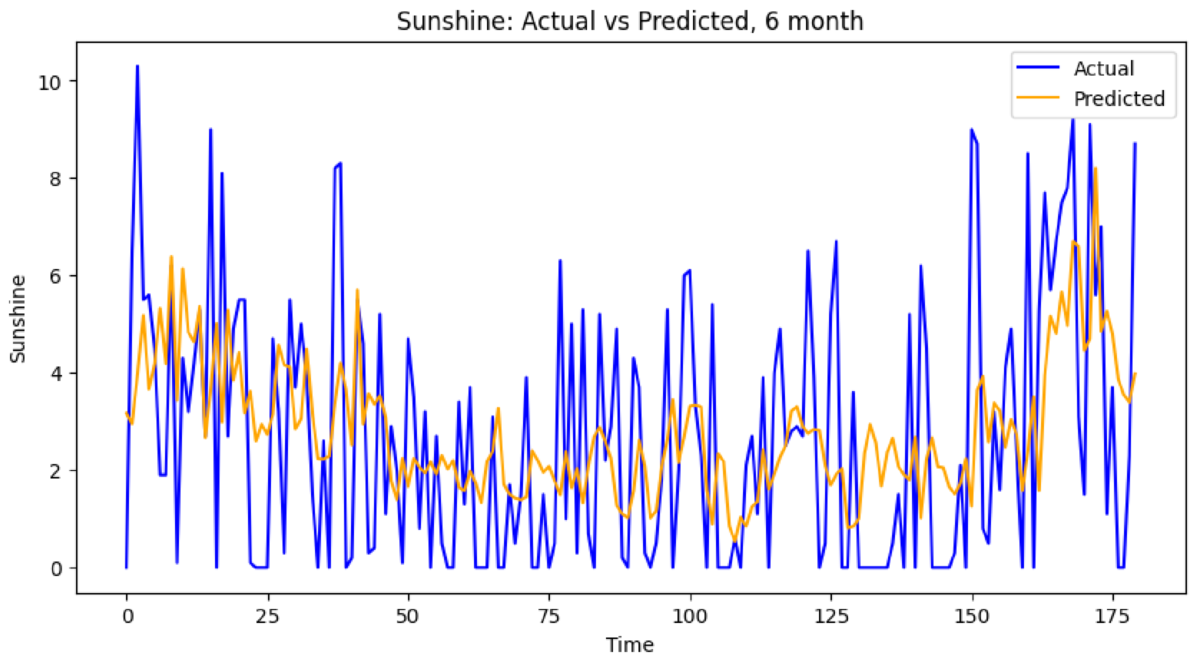


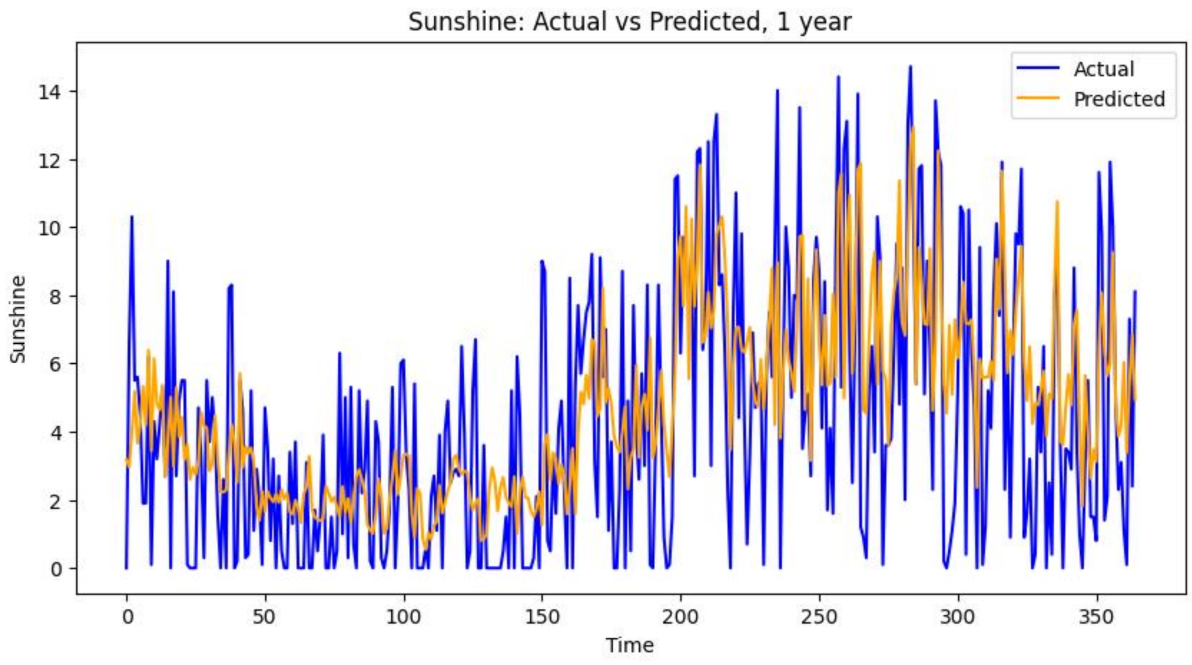
Mean Temperature: Actual vs Predicted, 1 year
Actual
Predicted
Mean Temperature
Time
0
5
10
15
20
25
30
0
50
100
150
200
250
300
350

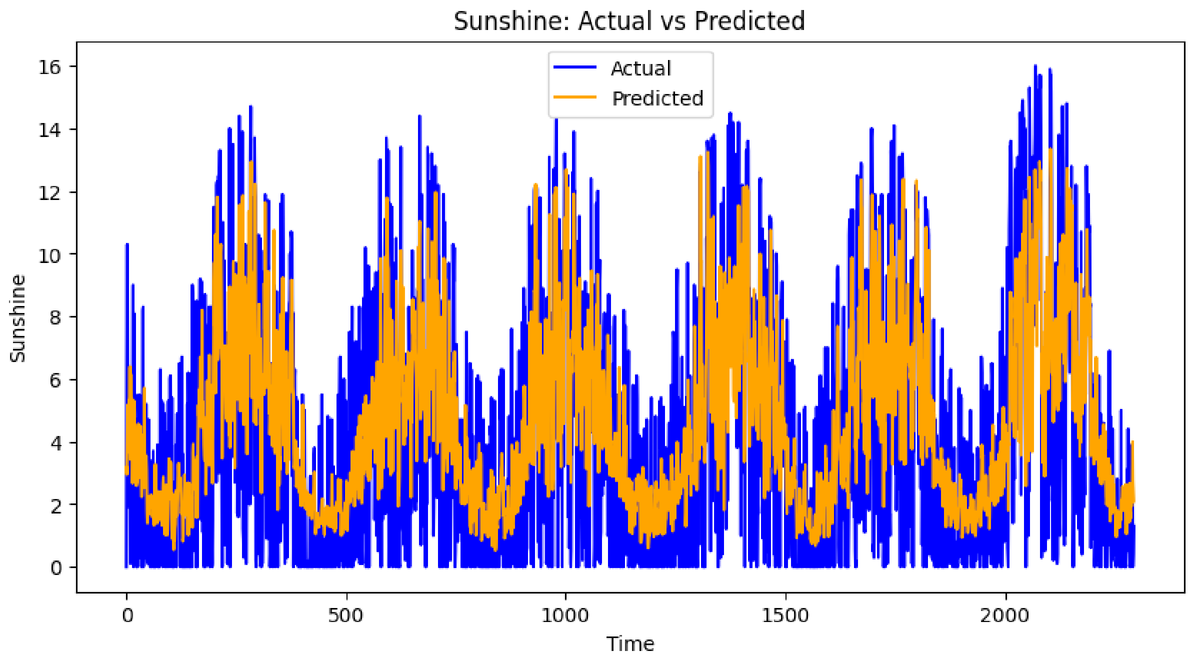


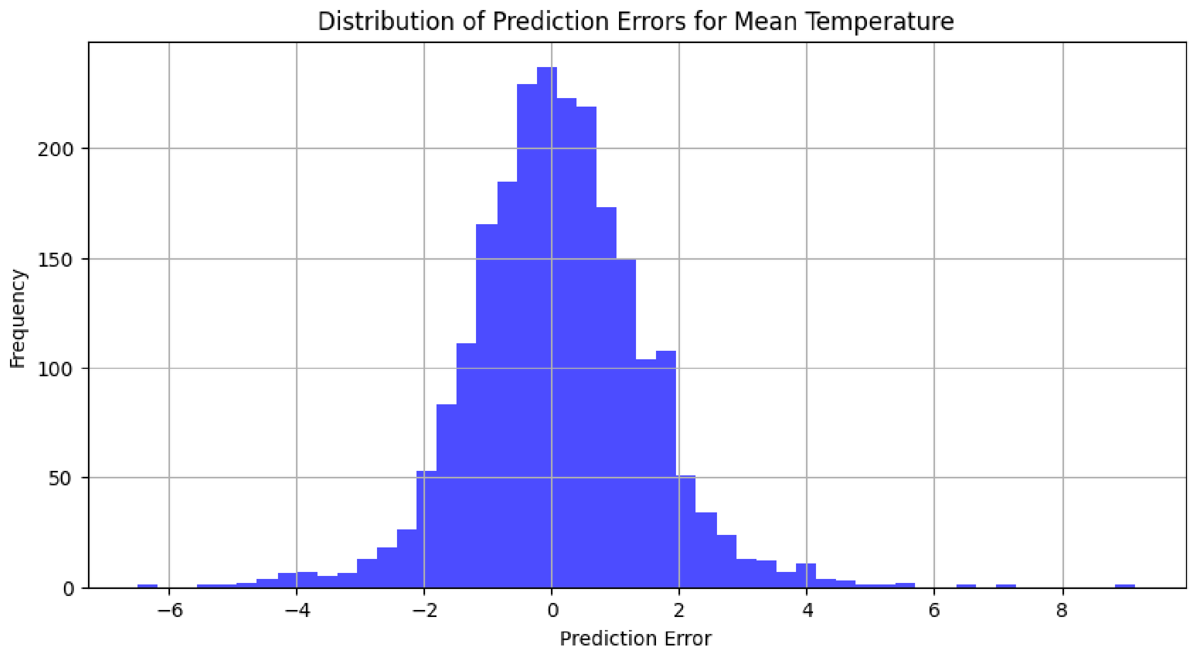
Mean Temperature: Actual vs Predicted
Actual
Predicted
Mean Temperature
Time
0
5
10
15
20
25
30
0
500
1000
1500
2000


Sunshine: Actual vs Predicted, 1 month
Actual
Predicted
Sunshine
Time
0
2
4
6
8
10
0
5
10
15
20
25
30

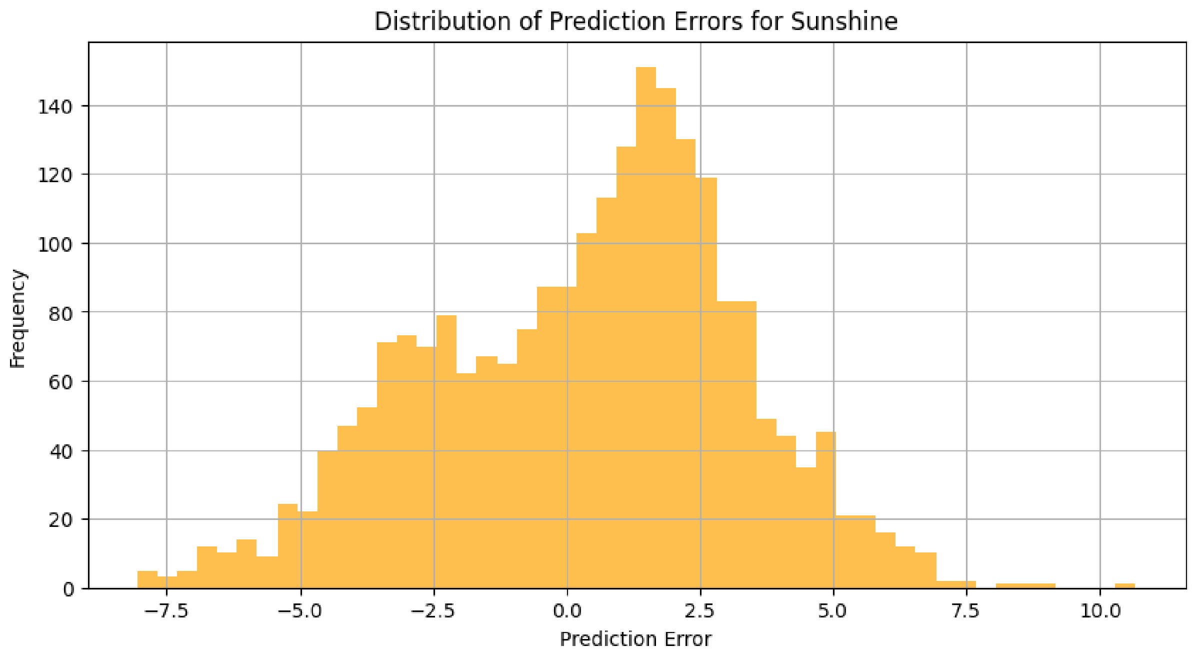

Sunshine: Actual vs Predicted, 6 month
Actual
Predicted
Sunshine
Time
0
2
4
6
8
10
0
25
50
75
100
125
150
175


Sunshine: Actual vs Predicted, 1 year
Actual
Predicted
Sunshine
Time
0
2
4
6
8
10
12
14
0
50
100
150
200
250
300
350

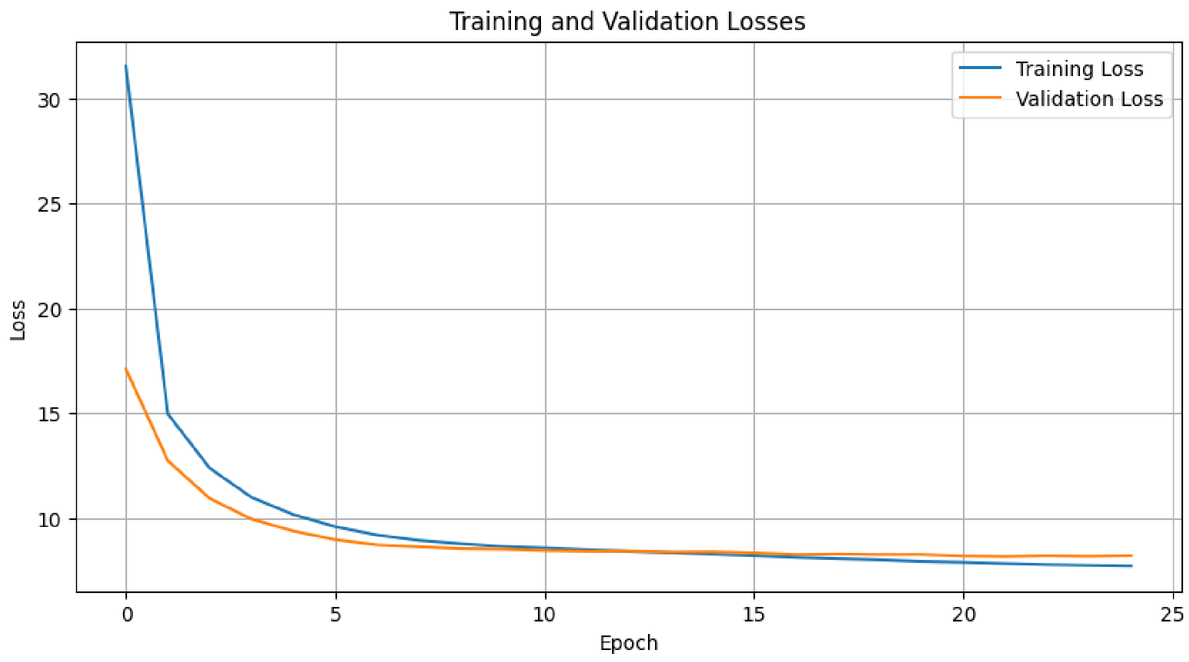

Sunshine: Actual vs Predicted
Actual
Predicted
Sunshine
Time
0
2
4
6
8
10
12
14
16
0
500
1000
1500
2000



Distribution of Prediction Errors for Mean Temperature
Frequency
Prediction Error
0
50
100
150
200
−6
−4
−2
0
2
4
6
8



Distribution of Prediction Errors for Sunshine
Frequency
Prediction Error
0
20
40
60
80
100
120
140
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0



How to run the code:

Simply visit this Link and download the dataset “london_weather.csv”, attach it to your project environment and run the code, it will automatically read the data and do all the preprocessing and then it will print the training phase steps which is the train and validation loss for every epoch, then the results of the test set, and finally the plots.