Michael Robinson
Professor Tomuro
CSC 578 Section 901
08 June 2023

Final Project: Weather Forecasting Using RNN

**Introduction:**

Although meteorology had been discussed as early as the Greeks, it was not until the 17th and 18th centuries that it was done in more scientific manner [1]. Now in the modern age, machine learning techniques are used in weather prediction; global machine learning contests have been established in predicting various weather features [2] [3] [4]. Various different architectures, such as CNN, RNN and RNN-LSTM have been used; a more detailed literature review is presented in the survey paper. This project will use four hours of radar images to forecast future weather conditions four hours ahead.

**Project description:**

The sole dataset that was used was from Kaggle and uploaded by SkillSmuggler [5]. The dataset consisted of approximately 22,000 images taken 10 minutes apart of weather radar centered around King City, Canada. One sample image is shown below in figure 1.

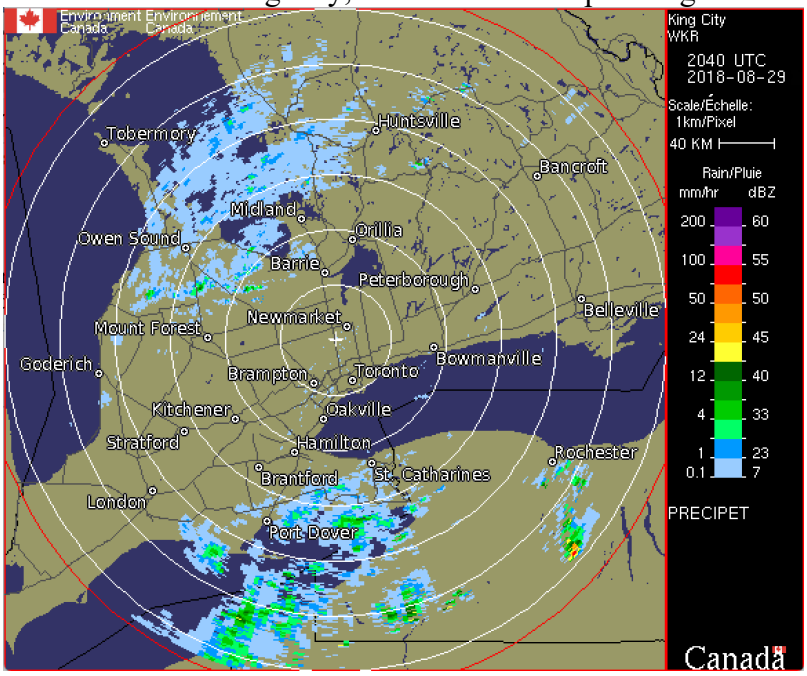


Figure 1: A sample image from the dataset.

First every image was labeled. The innermost circle was examined for pixels with colors corresponding to the legend. If the sum of the pixels exceeded a net 4000 dBZ, then the image was labeled as wet (1), if not, as dry (0). The dBZ measures reflectivity of a radar signal, more reflectivity corresponds to more precipitation and moisture in the air [6]. My choice of 4000 dBZ-km$^2$ was a ballpark estimate of what I would consider some kind of wet weather. Light rain is detectable around 20 dBZ so this occurring in 200 km$^2$ would meet the 4000 dBZ threshold [6]. I also was aware that highly imbalanced classes would make prediction of the positive class

even tougher. My initial threshold of 4000 dBZ-km$^2$ yielded dry conditions 86% of the time and wet conditions 14% of the time.

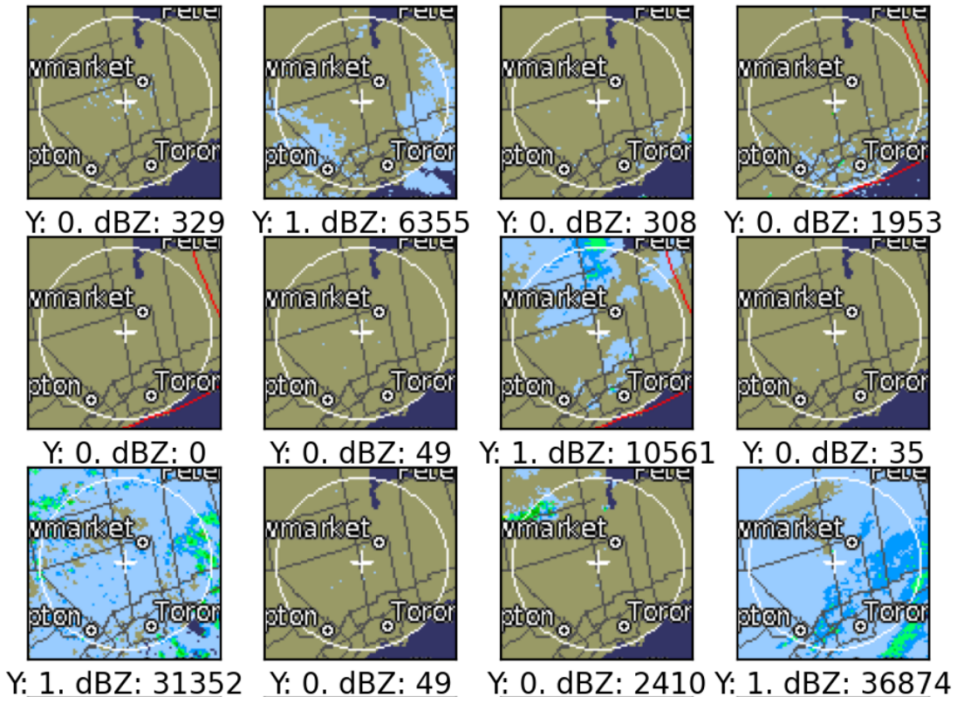


Figure 2: Twelve randomly chosen images with their labels and dBZ sums. Note that the images have been cropped here to about 50 x 50 for display purposes.

These single-channel 580 x 480 pixel images were cropped down first to 480 x 480 to remove the legend on the right side. Then all pixels that did not represent precipitation were changed to 0 (black) and the remaining pixels were scaled by dividing by 60 (the maximum dBZ value of a single pixel). Given the images were not normalized (no standard deviation or mean were involved) I scaled all images including the test inputs in this way.

The data size of the samples was problematic at times. To work with this, I further cropped my images to 400 x 400. Storms travel around 20km / hour on average and thus over the 8 hour time period of prediction, they would move on average 160 km [7]. Given the 40 km radius in the center of the map that I was making predictions for this was 160 km from the very outside of the images (at 400 x 400 pixels). This cropping reduced the number of features to 160,000 but this was far too many. Both Google Colab and my local machine kernel crashed due to lack of RAM. Given my initial intentions with the project was to use a CNN to find patterns in the images before using the RNN, I had CNN layers in mind when I encountered this data shortage problem. I applied a 4 x 4 filter with stride 4 over all the images to reduce them down to 100x100. The 10,000 features I was left with now were significantly more manageable. After windowing the images into 24 element arrays (24 10 minute intervals totaling 4 hours) and windowing the outputs into 24 element arrays consisting of 1's and 0's then I dealt with the corrupted files. Over 100 (123) files were missing, this divided the series into six subseries. Due to the importance of the sequential data in the prediction process, I did not want to mix subseries. I kept only the two longest ones; I split each subsequence into three consecutive sections (80% train, 10% validation and 10% test). Finally the data was ready for modeling.

A simple no-change baseline as described here [8] was implemented. The baseline model looked at the label of the last timestep of the input and predicted that this label would be the same for all predictions. On subseries 1 it scored 93% accuracy for the validation and 98% on the test, on subseries 2 it scored 84% on validation and 80% for the test.

```
Baseline Model Validation results:
              precision    recall  f1-score   support

           0       0.96      0.96      0.96     16152
           1       0.56      0.56      0.56      1416

    accuracy                           0.93     17568
   macro avg       0.76      0.76      0.76     17568
weighted avg       0.93      0.93      0.93     17568

              precision    recall  f1-score   support

           0       0.90      0.92      0.91      9247
           1       0.29      0.26      0.28      1217

    accuracy                           0.84     10464
   macro avg       0.60      0.59      0.59     10464
weighted avg       0.83      0.84      0.84     10464

Accuracy 1: 92.8%
Accuracy 2: 83.9%
```

Figure 3: Validation results from the baseline model

```
Baseline Model Test results:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99     17376
           1       0.07      0.07      0.07       216

    accuracy                           0.98     17592
   macro avg       0.53      0.53      0.53     17592
weighted avg       0.98      0.98      0.98     17592

              precision    recall  f1-score   support

           0       0.88      0.86      0.87      7889
           1       0.60      0.63      0.61      2599

    accuracy                           0.80     10488
   macro avg       0.74      0.74      0.74     10488
weighted avg       0.81      0.80      0.81     10488

Accuracy 1: 97.7%
Accuracy 2: 80.4%
```

Figure 4: Test results from the baseline model

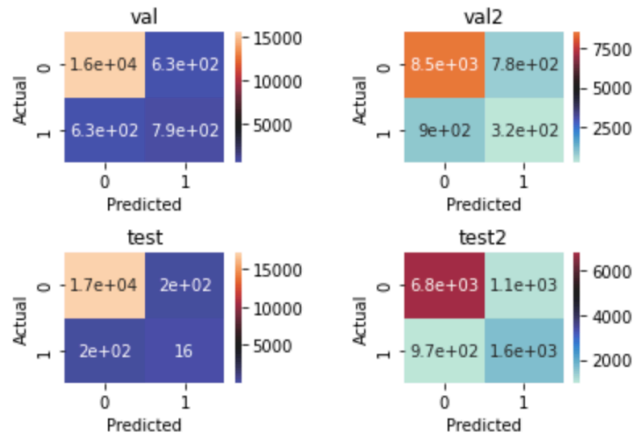



Figure 5: Confusion matrices for the baseline model for both validation and test subseries.

Many different model architectures were experimented with. Two promising models that were not able to be explored were models using ConvLSTM2d layers and ConvLSTM1d layers. These are what I initially had in mind with this project, as they could find patterns in each image and then pass the images in sequentially. The issue with using a CNN after an RNN is that the CNN would be reading the data in of all timesteps at once. The computational power needed to run the ConvLSTM2d and ConvLSTM1d layers were far more than I had available. I experimented with SimpleRNN layers and LSTM layers and eventually settled on a relatively simple model that used a high dropout rate between layers and some recurrent dropout as well. The model initially struggled to predict the positive class as the classes were highly imbalanced. Up-sampling the majority class or down-sampling the minority class would cause interruptions in the time series. I explored ways to penalize false negatives and encourage more positive class predictions. The Poisson loss function (y_pred – y_true * log(y_pred)) accomplishes this by not punishing false negatives as harshly.

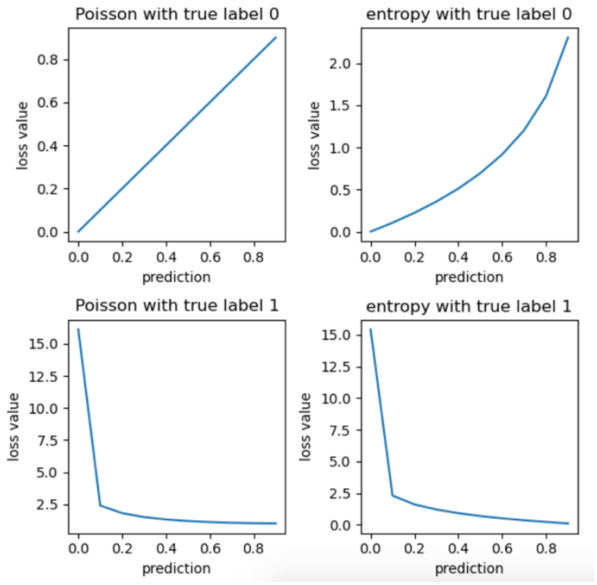



Figure 6: Poisson loss function versus Binary Crossentropy. Notice how the false positives are punished less with Poisson. A prediction of 1 for the negative class yields a loss of 1 for Poisson while a near infinite loss for binary cross entropy.

Sample weighting I discovered using this resource [9]. It was vital to my success. I hypothesized that it would be easier for the model to learn to predict wet weather when there was a lot of it in a given 4 hour window, rather than when there was just an occasional sparse wet condition. For this reason, when I applied sample weighting, I looked at the training labels – if more than two thirds of them were wet I applied a 4 fold increase in that sample's weighting. My goal for my model during validation was to simply get comparable accuracies, recall and f1 score to the baseline model. The model summary is shown below, the exact build code is in Appendix B.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| simple_rnn_133 (SimpleRNN) | (None, 24, 512) | 5382656 |
| dropout_278 (Dropout) | (None, 24, 512) | 0 |
| lstm_137 (LSTM) | (None, 24) | 51552 |
| dropout_279 (Dropout) | (None, 24) | 0 |
| flatten_60 (Flatten) | (None, 24) | 0 |
| dense_118 (Dense) | (None, 48) | 1200 |
| dropout_280 (Dropout) | (None, 48) | 0 |
| dense_119 (Dense) | (None, 24) | 1176 |

Total params: 5,436,584
Trainable params: 5,436,584
Non-trainable params: 0

Figure 7: Final model architecture

**Results:**

Training for the model was up to 16 epochs, but Keras's Early Stopping callback did monitor the sensitivity of the model (for a given specificity = 0.5) and stop training early if it did not improve after 5 epochs. Note again that models for subseries 1 and 2 are different models, but have the same architecture.

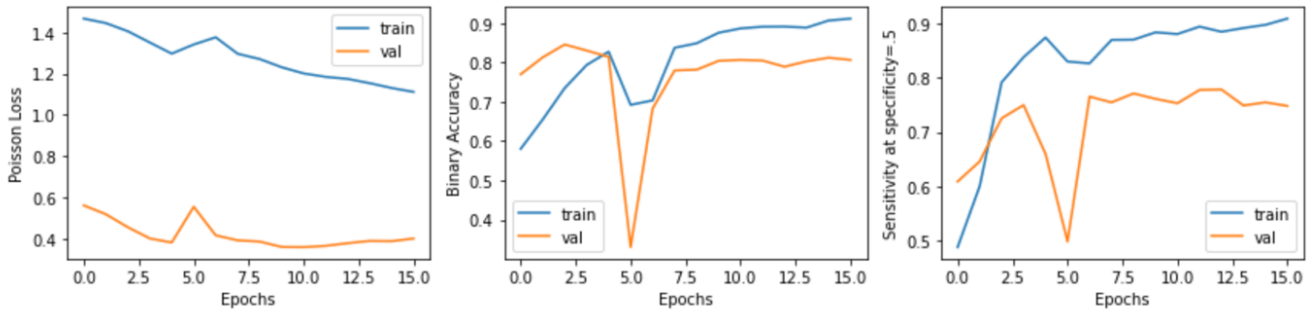



Figure 8: Training curve for subseries 1

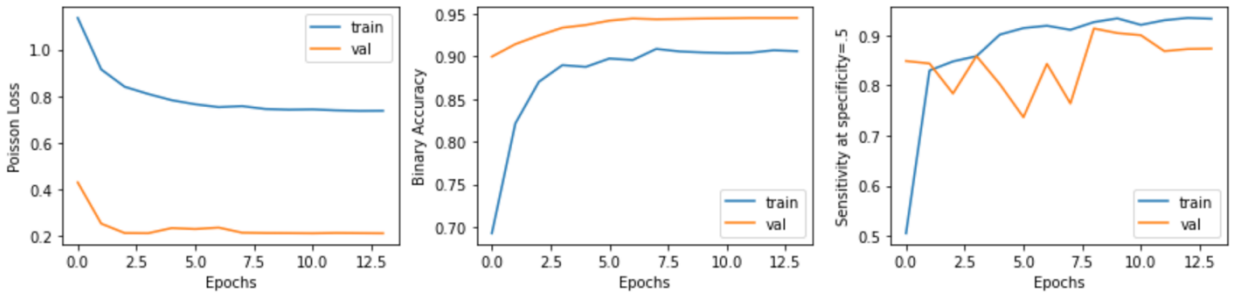



Figure 9: Training curve for subseries 2

The validation results are shown below. These results are comparable to the baseline model.

val

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.96 | 0.98 | 0.97 | 16152 |
| 1 | 0.70 | 0.54 | 0.61 | 1416 |
| accuracy | | | 0.94 | 17568 |
| macro avg | 0.83 | 0.76 | 0.79 | 17568 |
| weighted avg | 0.94 | 0.94 | 0.94 | 17568 |

val2

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.92 | 0.86 | 0.89 | 9247 |
| 1 | 0.28 | 0.42 | 0.34 | 1217 |
| accuracy | | | 0.81 | 10464 |
| macro avg | 0.60 | 0.64 | 0.61 | 10464 |
| weighted avg | 0.84 | 0.81 | 0.82 | 10464 |

Figure 10: Validation results for subseries one and two

Confusion matrices were generated in order to better understand the effect of class imbalance on prediction.

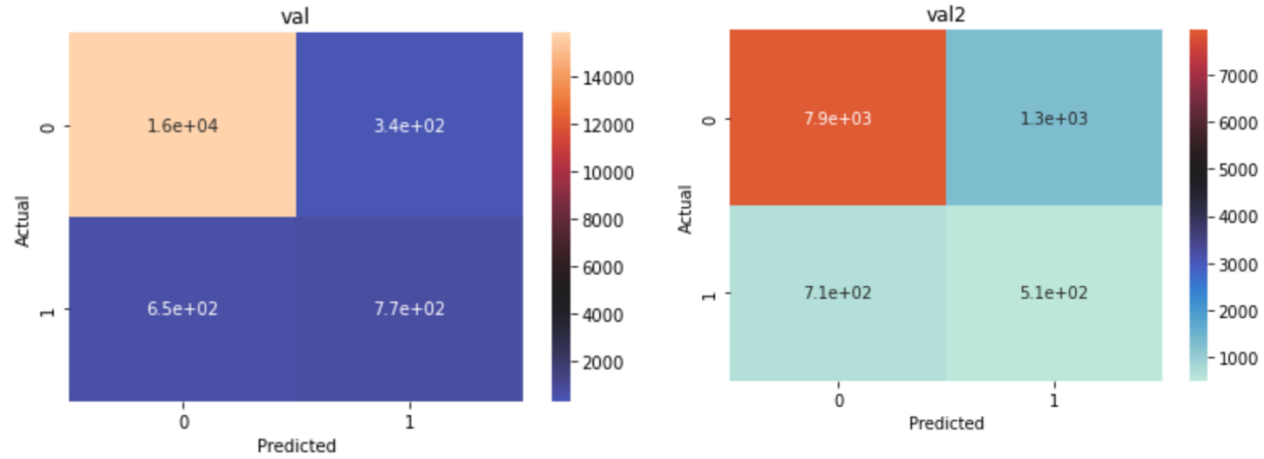



Figure 11: Confusion matrices for validation subseries one and two. Note the relatively high proportion of false negatives especially in validation 2.

After each model was trained on the training data, each model was trained on the validation data for 5 epochs. The purpose of this was to give the model the most up-to-date data on the subseries before it would make predictions.

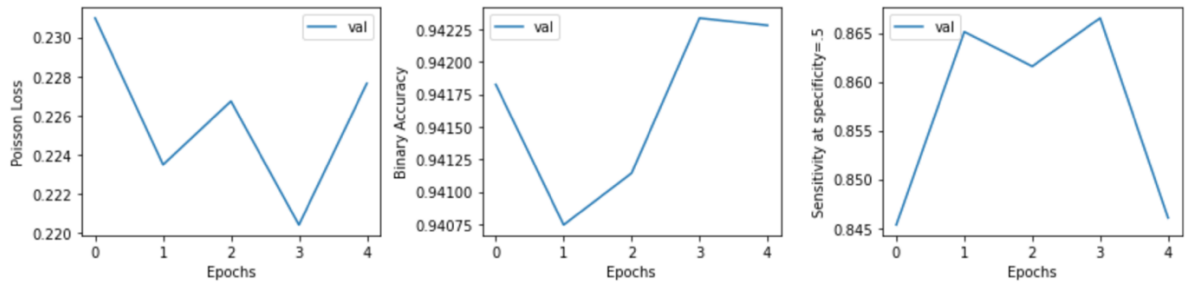



Figure 12: Validation training curve and results on subseries 1

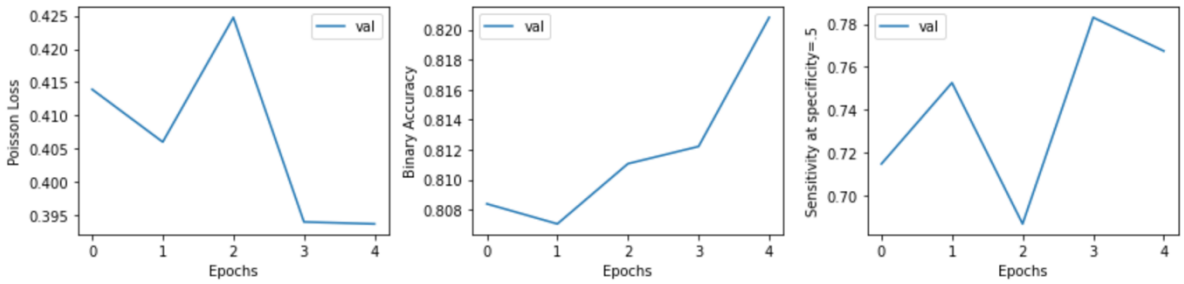



Figure 13: Validation training curve and results on subseries 2
Testing was then performed for each model.

test

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.99 | 1.00 | 0.99 | 17376 |
| 1 | 0.00 | 0.00 | 0.00 | 216 |
| accuracy | | | 0.99 | 17592 |
| macro avg | 0.49 | 0.50 | 0.50 | 17592 |
| weighted avg | 0.98 | 0.99 | 0.98 | 17592 |

test2

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.84 | 0.93 | 0.88 | 7889 |
| 1 | 0.67 | 0.45 | 0.54 | 2599 |
| accuracy | | | 0.81 | 10488 |
| macro avg | 0.76 | 0.69 | 0.71 | 10488 |
| weighted avg | 0.80 | 0.81 | 0.80 | 10488 |

Figure 14: Test results show that the model failed to detect any of the wet conditions in test subseries 1, but detected about 45% of them in test 2.

Again confusion matrices were made to illustrate performance.

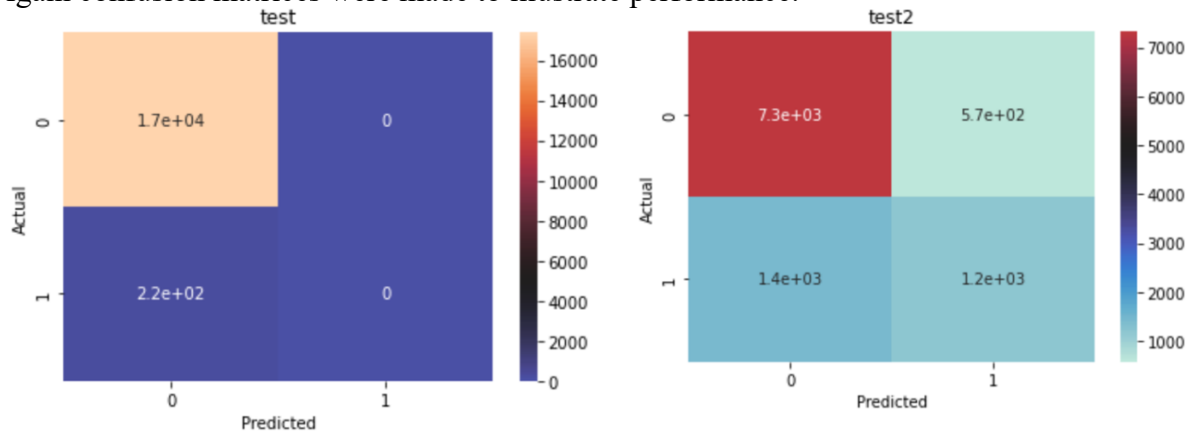



Figure 15: Confusion matrices for subseries 1 and 2

**Discussion:**

Trying to encourage prediction of the positive class was the toughest challenge of the project. The use of sample weighting, monitoring recall versus specificity (using Keras Callbacks) and the Poisson loss function helped in this process. Class weighting would have been ideal, but the complications of applying it to 24 element vector output did not seem obvious using Keras. Sample weight was instead increased if the samples had 66% or more positive labels. Deciding on what factor to use in this weighting, was a hyperparameter that I experimented with a lot. Of course this factor depends on the amount of class imbalance which was substantially different in the test sets. This change in class balances between training and testing was significant and may have contributed to poor results. Both training sets were close to 90% dry and 10% wet. Test 1 was 99% dry while test 2 was 75% dry. The Keras callback that monitored the sensitivity helped prevent predictions entirely of the negative class (but it was still not effective enough for the first test subseries). I had trouble implementing the BinaryFocalCrossentropy loss function (due to version dependencies), but I think this loss would be much more suited to class imbalances than the Poisson loss [10]. Also due to the large datasets (and some suboptimal optimization use of RAM in my coding) Google Colab's RAM was exceeded, and I was unable to use it for training.

I found that the deeper I made the model, the more it just predicted the negative class. I used high levels of dropout (at rate of 0.8) and recurrent dropout (at rate of 0.4), but the problems persisted. My final architecture of using a large SimpleRNN layer followed by a smaller LSTM layer worked alright, but given the results there is plenty of room for improvement. Due to time limitations I did not experiment with stateful=True as much as I intended to.

**Conclusions and future work:**

My results on subseries 1: 99% accuracy and undefined f1 score and subseries 2: 81% accuracy with 0.54 f1 score were not better than the baseline model's results of 98% accuracy and 0.07 f1 score on subseries 1 and 80% accuracy and 0.61 f1 score on subseries 2. However the project suggests that using recurrent neural networks for weather prediction is quite feasible. Given more time and computational resources, I would like explore bidirectional recurrent neural networks and the ConvLSTM layers, as discussed before. Prediction of average weather over an hour (rather than 6 10 minute intervals in each hour) might be more feasible. Also probability-based Bayesian models might prove to fit the domain well. The threshold I chose of what constitutes wet could be changed. Specifically, the problem could be turned into multinomial classification, such as dry, raining, severe weather, etc. In retrospect, the area I chose was very large (over 5000 $km^2$); a smaller area might further encourage the class imbalance but could make prediction easier. Lastly, in regards to the model I would like to further experiment with stateful mode enabled, and returning the hidden state so the model can be warmed up properly before testing. The resetting of the hidden state to zero vectors between batches seems to interrupt the temporal nature of the series and even with a fairly large batch size of 64, I believe the hidden state is still getting reset at the start of the validation and test data and thus critical information could being getting lost. I would like to further study RNNs to better understand how they function.

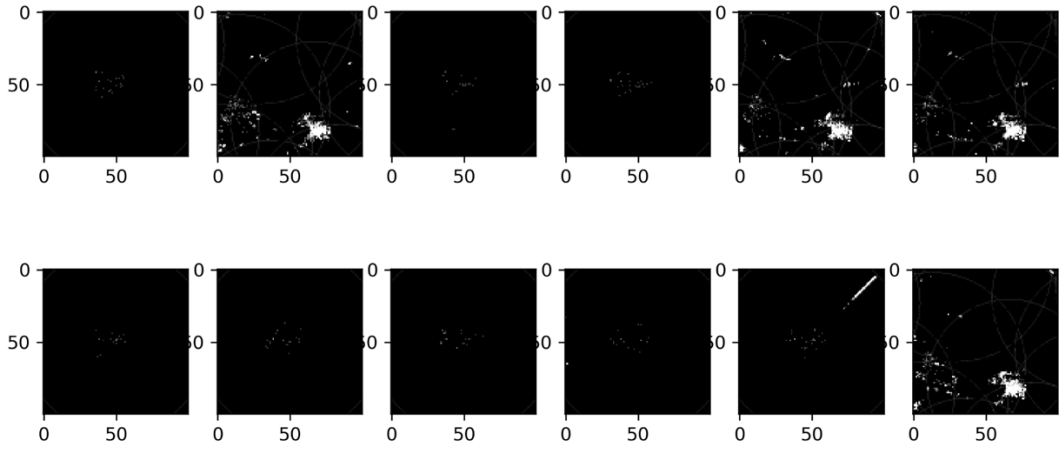
**References:**

[1] J. Cahir. "Weather Forecasting", *Britannica*.
https://www.britannica.com/science/weatherforecasting

[2] A. Chattopadhyay, P. Hassanzadeh and S. Pasha. “Predicting clustered weather patterns: A test case for applications of convolutional neural networks to spatio-temporal climate data,” *Scientific Reports, Nature Research,* 2020. https://www.nature.com/articles/s41598-020-57897-9/.
[3] M. A. Haq, A. Ahmed, I. Khan, J. Gyani1, A. Mohamed, E. Attia , P. Mangan and D. Pandi. “Analysis of environmental factors using AI and ML methods,” *Scientific Reports, Nature Portfolio*, 2022. https://www.nature.com/articles/s41598-022-16665-7.
[4] M. Fan, O. Imran, A. Singh and S. A. Ajila. “CNN-LSTM Model for Weather Forecasting,” *IEEE*, 2022.
https://www.researchgate.net/publication/367455415_Using_CNNLSTM_Model_for_Weather_Forecasting.
[5] https://www.kaggle.com/datasets/skillsmuggler/weather-radar-king-city-canada
[6] “Understanding Weather Radar”, *Weather Underground.*
https://www.wunderground.com/prepare/understanding-radar
[7] “Movement of Thunderstorms”, *Britannica.*
https://www.britannica.com/science/thunderstorm/Movement-of-thunderstorms
[8] https://www.tensorflow.org/tutorials/structured_data/time_series#multi-step_models
[9] https://www.tensorflow.org/tutorials/structured_data/imbalanced_data
[10] https://www.tensorflow.org/api_docs/python/tf/keras/losses/BinaryFocalCrossentropy
[11] https://keras.io/api/.

**Appendix A – Data samples, model input and output details:**

The 22,000 images (one is shown in figure 1) were cropped and scaled down using average pooling to 100 x 100. The precipitation pixels were scaled by dBZ value while the “dry” pixels were set to zero. Twenty four of these images were grouped together in one sample as shown below:



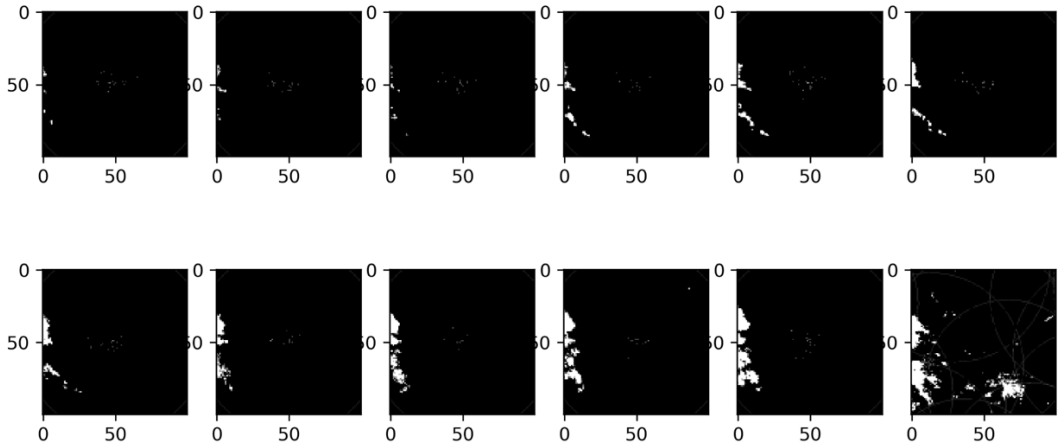


Figure 16: A single sample input was a numpy array of shape (24, 10000). “Wet pixels” are shown in white.

The next 24 images in the time series were labeled as specified before and were the targets of these images. The labels were 24 element numpy arrays with binary values such as [0, 0, 1, ..., 0] corresponding to dry (0) or wet (1) conditions in the innermost circle (40 km radius around King City).

**Appendix B – Critical code segments:**

All code in this project is original. Assistance mainly came from Keras tutorials and the reference document [11]. Preprocessing without the help of source code was slow-going and inefficient perhaps, but it was vital in the learning process. Perhaps on future projects, now that I understand the general nature of the preprocessing, source code could be used to boost efficiency (especially in regards to reducing algorithmic time complexity).

This segment of code iterates through the x and y coordinates of the center circle. The color_mapping dictionary converts the pixel value to its respective dBZ value. The function then sums them all together. A simple check comparing this value to the threshold is not shown.

```
# gets the weighted sum of the pixels in the center most circle
def get_pixel_vals(self, img):
    dbz_sum = 0
    for (x, y) in zip(self.xs, self.ys):
        for ind in range(y.shape[0]):
            pix = img.getpixel((x, int(y[ind])))
            dbz_sum += self.color_mapping[pix]
    return dbz_sum
```

Figure 14: Code segment showing calculation of the dBZ value in the 40 km radius circle.

This code shows how the X and Y vectors were made. This code was extremely computationally expensive. Each image that could be opened (if it was not corrupted) was labeled as discussed immediately above, then the image was converted to a vector. This vector

was then mapped to the dBZ values and scaled by the max dBZ value of 60. Then the 4x4 average pooling filter is applied.

```
def make_X_and_Y(files):
    Y = np.zeros(len(files)) # labels
    X = [] # inputs
    i = 0
    max_dbz = 60
    scale = np.vectorize(lambda x: 0.0 if x not in preprocessor.colors else preprocessor.color_mapping[x] / max_dbz
    for file_name in files:
        try:
            img = PIL.Image.open(file_name)
            Y[i], img = preprocessor.get_label(img)
            arr = np.asarray(img)
            rescaled = scale(arr)
            reshaped = tf.reshape(rescaled, (1, preprocessor.size, preprocessor.size, 1))

            # change reduction here avg pool - 16 to 1
            reduced = tf.nn.avg_pool(reshaped, ksize=4, strides=4, padding="SAME")

            to_append = np.array(tf.squeeze(reduced))
            X.append(to_append)

        except FileNotFoundError:
            Y[i] = np.nan
            X.append(np.nan)
            preprocessor.dbz_sums.append(np.nan)

        except PIL.UnidentifiedImageError:
            Y[i] = np.nan
            X.append(np.nan)
            preprocessor.dbz_sums.append(np.nan)

```

Figure 15: Generating X and Y. X and Y are eventually used in generating the input, output for each subseries (dataset).

The baseline model iterated through the indices of the images. The indices were [0, 1, ..., 47], [1, 2,..., 48], ... etc. The first half of the index refers to the input, and the second half the output. Thus taking the 24$^{th}$ Y value of an index and using it as the prediction for the next 24, 10 minute time steps is the baseline no-change model. This function returns the correct count and the total number of samples, in addition to the model's predictions.

```
# base model - simply guess last label
def base_model_predictions(indices, Y, x_win):
    predictions = []
    count = 0
    correct = 0
    predictions = []
    for ind in indices:
        pred = [Y[ind[x_win - 1]]] * (len(ind) - x_win)
        predictions += pred
        for i in range(x_win, len(ind)):
            count += 1
            ac = Y[ind[i]]
            if ac == pred[0]:
                correct += 1
    return correct, count, np.array(predictions)
```

Figure 16: The base model.

The model with dropout rates commented in is shown below.

```
# building the model
model = tf.keras.Sequential()
model.add(tf.keras.layers.SimpleRNN(512, input_shape=(x_win, num_feat), \
                                    return_sequences=True, recurrent_dropout=rdrop)) # .4
model.add(tf.keras.layers.Dropout(drop)) # .8
model.add(tf.keras.layers.LSTM(24, input_shape=(x_win, num_feat), \
                               return_sequences=False, recurrent_dropout=rdrop)) # .4
model.add(tf.keras.layers.Dropout(drop2)) # .5
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(48, activation="relu"))
model.add(tf.keras.layers.Dropout(drop3)) # .5
model.add(tf.keras.layers.Dense(x_win, activation="sigmoid"))
# compile
model.compile(opt, loss,  metrics=[metric, "binary_accuracy"])
```

Figure 17: The model. Note that loss was Poisson, and metric was SensitivityAtSpecificity=0.5.