CS230

Time-series prediction of COVID-19 ICU admissions to assist with managing healthcare system resources

Sotirios Papasotiriou
sotpap@stanford.edu

(Healthcare)

Abstract

COVID-19 has already had a major impact on our lives after less than two years since its first appearance. The challenge presented to healthcare systems worldwide is using their resources in the best possible way to handle the waves of new cases and patients every day. In this project we utilized Recurrent Neural Networks to make time-series predictions regarding ICUs needed in a healthcare system. We experimented with various RNN layers such as LSTMs, Bidirectional LSTMs and GRUs, and using a predefined network structure, we implemented a Genetic Algorithm to perform hyperparameter tuning. We managed to achieve very low regression metrics, as well as develop time-series charts that show our network's ability to predict correctly.

1 Introduction

A little over a year has passed since the COVID-19 pandemic spread around the world and established itself as one of the most contagious and dangerous viruses of modern times. In the age of information there's a plethora of statistics and measurements available regarding patients, vaccination, test results, deaths and more for each country. To tackle this problem, we use a timeseries prediction approach utilizing a Recurrent Neural Network to predict future ICU occupancy numbers, to aid a healthcare system in preparing in advance and thus better allocating its resources in the fight against the pandemic. Input to our network is a multi-feature timeseries-modeled input, and our output will be the result of the model's regression.

2 Related work

Time-series problems were often solved using traditional prediction models, like ARIMA[1]. However, using a neural network to make predictions enables us to use multiple input features in our timeseries, among other advantages. That allows the network to better understand the importance of different features in predicting the desired value during training[2]. Due to the impact COVID-19 pandemic has had in our lives, there have already been several research papers working on the matter, treating the data as a time-series and predicting future values. Those include using autoregressive time-series models based on the two–piece scale mixture normal[3], or RNNs such as LSTMs, GRUs and Variational AutoEncoders[4][5]. What we aim to do in this project, is focus on RNNs predicting just the future ICU occupancy numbers, splitting the dataset samples depending on the location the data is from. This way we can experiment using multi-step, multi-feature inputs that make use of Neural Networks' advantage over traditional ML and time-series models, and enhance our dataset

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

size by normalizing its values in each location separately. Also the utilization of Hyperparameter Optimization Algorithms helps find the best suited network for our specific problem[6].

3 Dataset and Features

To tackle this problem we are using the datasets provided by The COVID Tracking Project[7] and Our World in Data[8] that cover COVID data on the United States state-wise, and worldwide, respectively. The datasets are 3 and 17 MBs in size, containing 20,000 and 71,000 rows of information including new cases, deaths, negative tests, vaccination information, average life expectancy, and more health related data(Figure 1). Getting the data in the desired form in order to feed it in our neural network

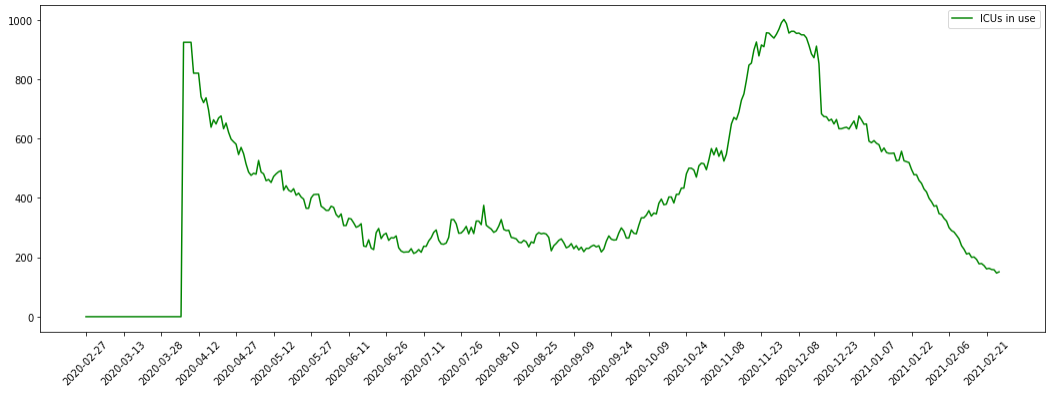

Figure 1: Data on ICU occupancy in the state of Indiana visualized as a time-series

requires us to create and apply several preprocessing functions on the input files. To begin, we load both our dataset files (OWID, TheAtlantic). We need to align the two datasets based on the features we will use, and we want our features to be as relevant to our problem as possible. The columns we chose to keep from the two datasets are: location, date, new_cases, new_deaths, icu_patients, hosp_patients, and state, date, positiveIncrease, deathIncrease, inIcuCurrently, hospitalizedCurrently, respectively. We rename them accordingly for convenience, concatenate them, and sort them by location and date. Sorting by date is very important, since we need our data to have ascending time order so we can feed it to an RNN. We also replace negative values with zeros, since negative values make no sense in our context and they can cause our model to make weird predictions. Afterwards, using the function 'get_locations' we keep a list of all available locations that have at least 200 days with non-zero ICU occupancy values. The reason is that the datasets are very scarce in what several countries do not have ICU occupancy numbers, which if not taken under consideration, would fill our network with 0 values, making it biased in its predictions. Since our data spans a little over a year, we judged that 200 days (more than 50% of the dataset's duration) is a good enough number, especially considering the delay of the spreading of the virus between countries. We then split the list we created, keeping 10 locations as dev/test cases and using the rest to create our train dataset. Date as a feature cannot offer us great information considering that we only have a single year of data, but in order to help our network to observe periodicity, we decided to enhance the dataset by replacing date by day of the week, with 0 corresponding to Monday and 6 to Sunday. For example we expect lower case numbers and possibly fewer ICU admissions on the weekends when people get tested less. We created functions to draw the data concerning a specific location from the dataset ('country_data', 'get_data'), so that we can normalize them separately and create separate input-output pairs. To achieve normalization we get each location's min and max values for every feature and compute $\frac{value-min}{max-min}$. Since min values are going to be 0 for every feature of every location, essentially the formula computed is equal to $\frac{value}{max}$ which scales our data down to (0,1) that our model can handle more easily. Finally we create a timeseries by using the last 4 weeks (28 days) data as input and the next day's ICU occupancy as output. Using 'create_dataset' we use all chosen locations to create our train or dev/test set respectively and complete our data preprocessing. After all preprocessing is done, our dataset has 14605 training examples, 1705 validation and 1705 test examples (Figure 2), and is ready to be used for training. Regarding normalization, we choose to normalize each location based on its own min and max values instead of taking the global min/max value, since the latter showed

high variance with a higher loss on the test set. The reason is that we want the changes in each country values to be equally significant regardless of the country's ICU capabilities or its population. If we choose to normalize all data the same way, its obvious that more densely populated locations with values that can change by thousands each day are more important than smaller countries whose ICU values can show only tiny changes such as 4-5 per day.

```
[[0.         0.         0.         0.0156038  0.01849618]
 [0.16666667 0.00309352 0.01008064 0.0183175  0.01970245]
 [0.33333334 0.00380419 0.00201613 0.0183175  0.02010454]
 [0.5        0.01530036 0.0141129  0.01899593 0.01970245]
 [0.6666667  0.01091092 0.01008064 0.01899593 0.01943439]
 [0.8333333  0.00865348 0.         0.02035278 0.02157888]
 [1.         0.         0.         0.02103121 0.02251709]
 [0.         0.01622006 0.01008064 0.02442334 0.02425948]
 [0.16666667 0.0068559  0.         0.02238806 0.02305321]
 [0.33333334 0.01542578 0.00604839 0.02781547 0.02559979]
 [0.5        0.00919694 0.00806452 0.02713704 0.02720815]
 [0.6666667  0.0147151  0.01008064 0.02781547 0.0292186 ]
 [0.8333333  0.02207266 0.00806452 0.02985075 0.02814636]
 [1.         0.01249948 0.         0.03188603 0.02828039]
 [0.         0.01680532 0.00201613 0.03527816 0.02975472]
 [0.16666667 0.0097822  0.02217742 0.03324288 0.03109503]
 [0.33333334 0.02813428 0.00604839 0.03188603 0.03524997]
 [0.5        0.02805067 0.00806452 0.02849389 0.03578609]
 [0.6666667  0.03114418 0.00201613 0.03256445 0.03390966]
 [0.8333333  0.02721458 0.00806452 0.03799186 0.03471385]
 [1.         0.01868651 0.         0.03934871 0.03632221]
 [0.         0.01943899 0.01008064 0.03663501 0.03886878]
 [0.16666667 0.01396263 0.00403226 0.04274084 0.03551803]
 [0.33333334 0.02132018 0.0141129  0.04138399 0.03927087]
 [0.5        0.03586807 0.00403226 0.04409769 0.03726042]
 [0.6666667  0.03210568 0.01008064 0.0468114  0.03819863]
 [0.8333333  0.02579324 0.00806452 0.04545455 0.03766251]
 [1.         0.03139501 0.00403226 0.0468114  0.03819863]]
[0.0495251]
```

Figure 2: Example of a preprocessed input and output sample

4 Methods

Once we get our data ready to be used as input to the network we begin the training process. A neural network uses variations of Stochastic Gradient Descent called Optimizers, trying to adjust its units' weights in order to minimize the loss calculated from the value it predicted and the actual value it should predict. The loss function we tried to minimize in order to optimize our model was Huber Loss. Huber Loss was essentially created to bridge the gap between Mean Absolute Error and Mean Squared Error. Mean Squared Error tends to put too much emphasis on outliers which might produce huge errors, whereas Mean Absolute Error is smoother but is not differentiable on 0. To solve this, Huber Loss uses a parameter δ as such: if the error is less than δ it uses a quadratic function, whereas if the error is greater than δ, it uses a linear function:

$$H_\delta = \begin{matrix} \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}(y_i-\hat{y}_i)^2, |y_i-\hat{y}_i| \le \delta \\ \frac{1}{n}\sum_{i=1}^{n}\delta\left(|y_i-\hat{y}_i|-\frac{1}{2}\delta\right), otherwise \end{matrix}$$

Figure 3: Huber Loss formula

The model architecture we chose is comprised of the following: -One or two Recurrent Layers utilizing dropout -A flatten layer to help feed the Recurrent layers output to dense layers -A dense layer -A dropout layer to help against overfitting -The output layer

Various python frameworks were utilized such as tensorflow[9] and keras[10] for the deep learning models and training, pandas[11] and numpy[12] for dataset handling, matplotlib[13] for data and result plotting, and scikit learn[14] for evaluation and metrics. In order to ensure the model we choose is capable of solving the problem at hand we implemented a Genetic Algorithm[15] that will help us determine our neural network's hyperparameter values. A Genetic Algorithm works as follows: We create a population of n networks with random hyperparameter values, and train them. We evaluate the results and sort them. The top networks advance to the next iteration of the algorithm, keeping their "genes" alive. All other networks have a very small chance to also advance to the next iteration.

We call the networks that advanced to the next iteration parent-networks. The rest of the available network slots, so that we once again have n networks in our population, are occupied by networks whose hyperparameters are inherited by two parent-networks. Each hyperparameter value of the new (child) network is chosen at random from the correspondent values of the parents, with a very small chance of a completely random value applied, to add mutation to the genes and help exploration. We then train the children networks and repeat the same process of sorting-choosing parents-creating children networks for each iteration, until we reach a desirable outcome.

5 Experiments/Results/Discussion

The hyperparameters chosen by our Genetic Algorithm were each layer's units, Dense activations, dropouts, the optimizer and its learning rate, as well as the batch size of training.

Hyperparameter	Possible Values
Layer Units	2, 4, 8, 16, 32, 64, 96, 128, 192, 256, 512
Dropout Rate	0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5
Activation functions	Relu, sigmoid, tanh, elu, linear
Optimizer	RMSProp, Adam, SGD, Adagrad, Adadelta, Adamax, Nadam
Learning Rate	0.001, 0.005, 0.01, 0.015, 0.02, 0.025, 0.03, 0.035, 0.04, 0.045, 0.05, 0.055, 0.06, 0.065, 0.07, 0.075, 0.08, 0.085, 0.09, 0.095, 0.1
Batch size	8, 16, 32, 64, 96, 128, 192, 256

We tested the Genetic Algorithm with various architectures regarding the Recurrent network part, namely with: a) 2 LSTM layers b) 1 LSTM layer c) 1 Bidirectional LSTM layer d) 1 GRU layer

The following table shows the hyperparameter choices and structure the Genetic Algorithm chose as best for each architectural experiment:

GA w/Double LSTM	GA w/Single LSTM	GA w/Bidirectional LSTM	GA w/GRU
128 units LSTM layer with 0 dropout	—	—	—
32 units LSTM layer with 0.2 dropout	16 units LSTM layer with 0.05 dropout	32 units Bidirectional LSTM layer with 0.05 dropout	16 units GRU layer with 0 dropout
192 units Dense layer with elu activation	128 units Dense layer with tanh activation	512 units Dense layer with relu activation	8 units Dense layer with elu activation
0.25 dropout	0.25 dropout	0.1 dropout	0 dropout
elu activation on Output layer	Linear activation on Output layer	Sigmoid activation on Output layer	elu activation on Output layer
Adamax Optimizer with 0.03 learning rate	Adagrad Optimizer with 0.08 learning rate	RMSProp Optimizer with 0.001 learning rate	Adamax Optimizer with 0.04 learning rate
8 batch size	32 batch size	16 batch size	128 batch size

We evaluate the models by using the Huber Loss test result and the RMSE (Root Mean Squared Error) and MAE (Mean Absolute Error) metrics of each model, as well as the wMAPE (weighted Mean Absolute Percentage Error) over all 10 test locations after the data has been denormalized (Figure 4).

$$RMSE=\sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i-y_i)^2}{n}} \quad MAE=\frac{1}{n}\sum_{i=1}^{n}|y_i-\hat{y}_i| \quad wMAPE=\frac{\sum_{i=1}^{n}|y_i-\hat{y}_i|}{\sum_{i=1}^{n}|y_i|}$$

Figure 4: Formulas for the evaluation methods used

The evaluation results of the best model of each architecture can be seen on the table below:

model	Huber Loss	RMSE	MAE	wMAPE
Baseline model	0.00086	0.04146	0.02563	62.49%
GA w/Double LSTM	0.00068	0.03681	0.02007	61.07%
GA w/Single LSTM	0.00140	0.05290	0.02924	60.85%
GA w/Bidirectional LSTM	0.00090	0.04250	0.02237	61.07%
GA w/GRU	0.00063	0.03553	0.01885	62.03%

We also developed a visualization function that prints out 28 days of consecutive predictions based on the input data (Figure 5). In the visualization the blue line refers to the actual values of the dataset, the green line to the predicted values, whereas the blue area around the green plot line refers to the error margin which is set as the maximum value among 10 ICUs and 5%. Plots of every architecture's top network can be found in the Appendix.

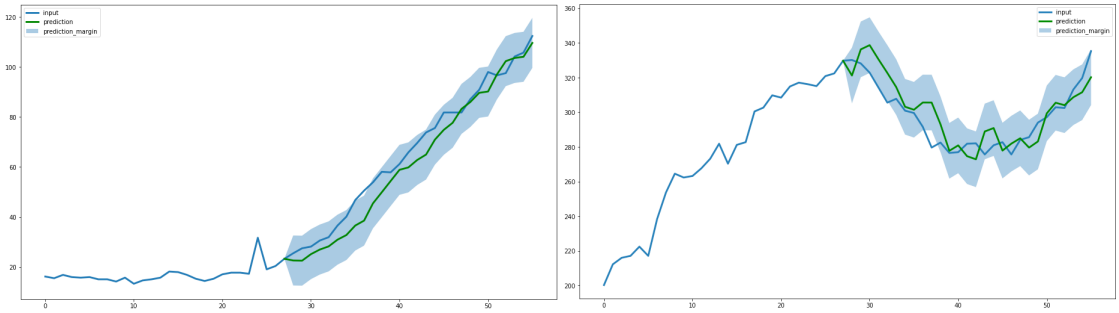

Figure 5: Visualization of a 28-day prediction of the two best evaluated models on the dataset of Portugal. The Double-LSTM model on the left, the GRU model on the right

6 Conclusion/Future Work

In conclusion, our baseline model seemed to tackle the problem very well judging from the metrics provided due to its solid architectural structure, but Hyperparameter Optimization helped push the errors to even lower values, whereas a GRU model proved to be the most successful of all our experiments. Although all the models were complex enough to be able to process the input without any trouble even with lower hyperparameter values, the Double-LSTM model proved more accurate than the Single LSTM networks. However, GRU models that have shown they can outperform LSTMs on smaller datasets[16] managed to provide slightly better metrics overall. Future work on this project could be done in data preprocessing. One idea to aid the neural network in its predictions would be to remove data with no ICU used. This happens almost exclusively during the time period before the pandemic reaches said location, and apart from not offering any information to the model whatsoever, it also makes regular MAPE unable to be computed, since MAPE involves element-wise division with those 0 true values. Other areas this project could improve upon is model architecture exploration and evaluation. Specifically, since the data is heterogeneous we could evaluate our models separately in higher and lower population areas to assess the model's ability to predict when the number predicted (after de-normalizing the predictions) is high or low separately.

References

[1] Tandon, Hiteshi, et al. "Coronavirus (COVID-19): ARIMA based time-series analysis to forecast near future." arXiv preprint arXiv:2004.07859 (2020).

[2] Lim, Bryan, and Stefan Zohren. "Time series forecasting with deep learning: A survey." arXiv preprint arXiv:2004.13408 (2020).

[3] Maleki, Mohsen, et al. "Time series modelling to forecast the confirmed and recovered cases of COVID-19." Travel medicine and infectious disease 37 (2020): 101742.

[4] Chimmula, Vinay Kumar Reddy, and Lei Zhang. "Time series forecasting of COVID-19 transmission in Canada using LSTM networks." Chaos, Solitons & Fractals 135 (2020): 109864.

[5] Zeroual, Abdelhafid, et al. "Deep learning methods for forecasting COVID-19 time-Series data: A Comparative study." Chaos, Solitons & Fractals 140 (2020): 110121.

[6] Yu, Tong, and Hong Zhu. "Hyper-parameter optimization: A review of algorithms and applications." arXiv preprint arXiv:2003.05689 (2020).

[7] https://covidtracking.com/data

[8] https://ourworldindata.org/coronavirus-source-data

[9] https://www.tensorflow.org/

[10] https://keras.io/

[11] https://pandas.pydata.org/

[12] https://numpy.org/

[13] https://matplotlib.org/

[14] https://scikit-learn.org/

[15] Xiao, Xueli, et al. "Efficient hyperparameter optimization in deep learning using a variable length genetic algorithm." arXiv preprint arXiv:2006.12703 (2020).

[16] Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." arXiv preprint arXiv:1412.3555 (2014).

Appendix

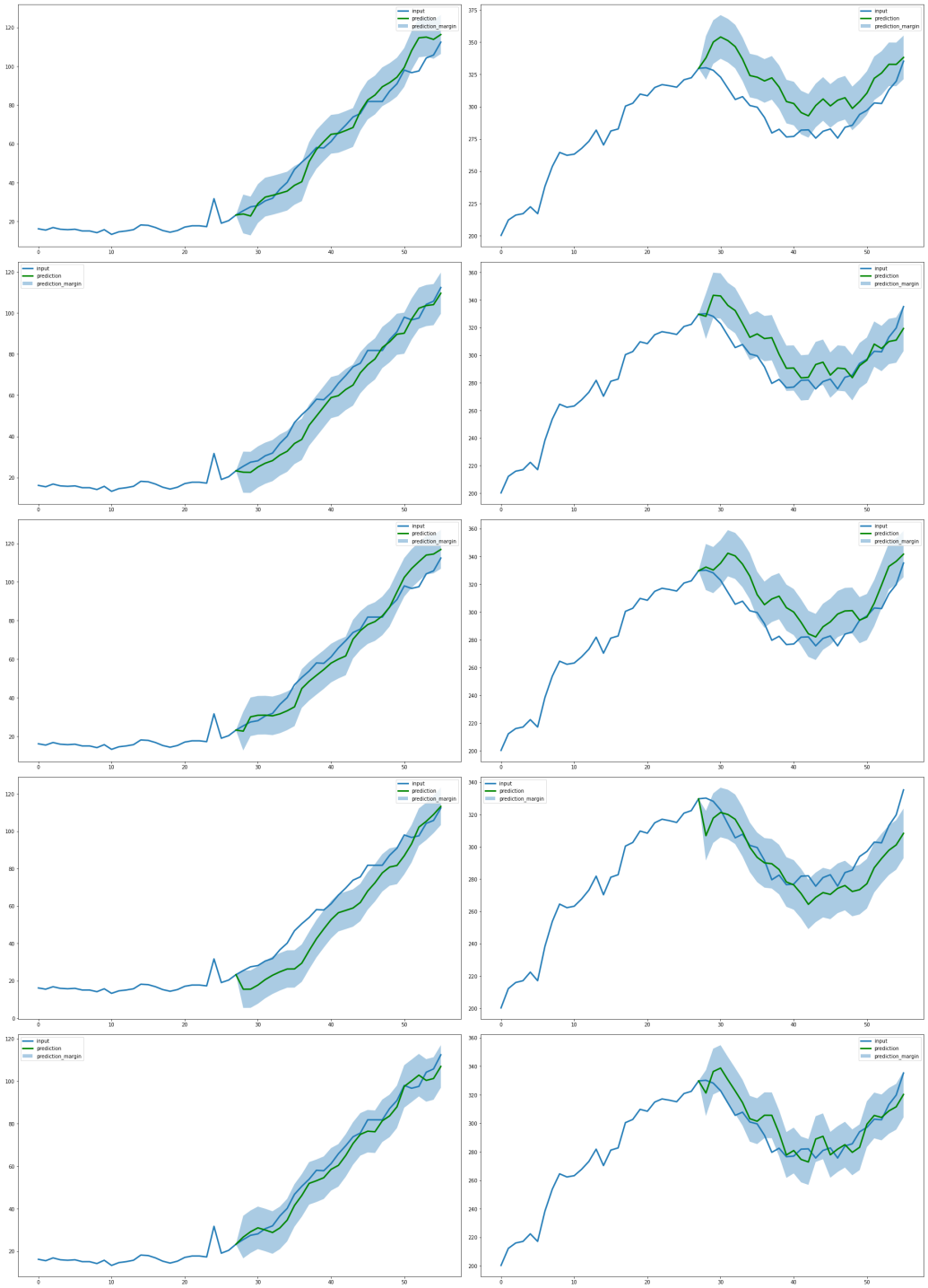

Figure 6: Visualization of two predictions on input samples from Portugal. Top to bottom: Baseline model, Double-LSTM model, Single-LSTM model, Bidirectional LSTM model, GRU model.

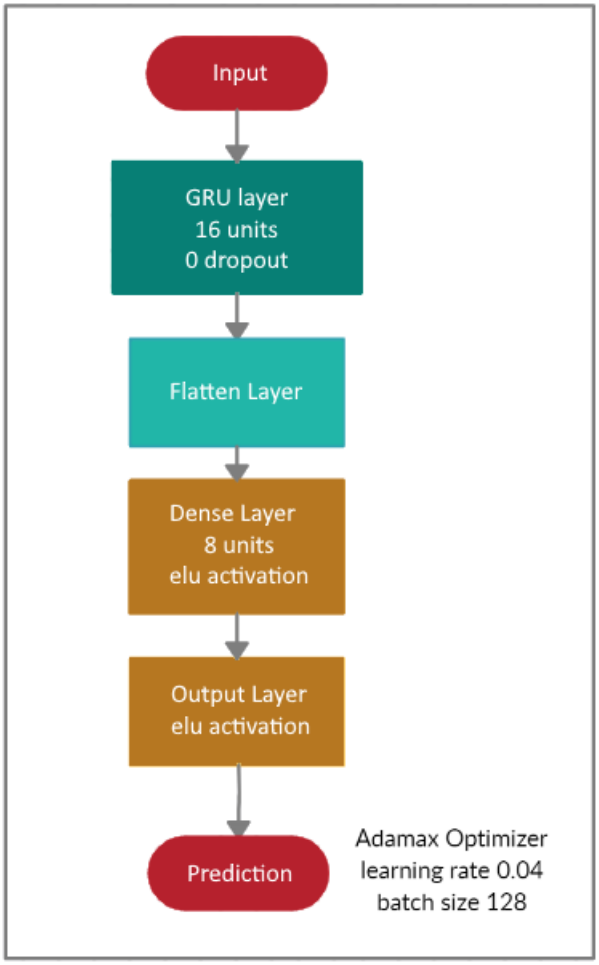

Figure 7: The GRU model that performed best during our experiments