SCOOT Emulator Using Recurrent Neural Network

Introduction

This report documents the methodology followed to build a model of the Split Cycle Offset Optimization Technique (SCOOT) adaptive traffic signal control system [(1)], using Artificial Neural Networks [(2, 3)]. The derived model, later called the SCOOT emulator, is shown to replicate the adaptive behavior of SCOOT using real-world data.

The report is structured as follows: Section 2 briefly introduces the SCOOT traffic signal control system and motivates the SCOOT emulator and its input-output characteristics. Section 3 presents the formulation of the neural network and parameter estimation from historic data. Section 4 presents training and test results for the emulator.

The SCOOT System and the SCOOT Emulator

SCOOT is an adaptive traffic signal control system that controls and coordinates a network of urban intersections in real time. It responds proactively to fluctuating traffic demands. An online computer monitors traffic flow patterns in the network, using the data obtained from in-field sensors such as inductive loops or infrared sensors. The collected data is fed to SCOOT, which includes a software component that predicts queue sizes and delays in the network. These are used to optimize signal timing parameters: green splits, offsets, and signal cycle length. Each optimizer evaluates the effect of incremental changes in signal timings on the overall network performance. SCOOT divides the network into regions, each of which consists of a set of intersections, often related geographically. The signal cycle lengths, offsets, and green splits are determined independently for each region.

The SCOOT Emulator is a neural network model that approximates SCOOT's adaptive control algorithm. As with SCOOT, the emulator takes in vehicle actuations as inputs and outputs signal timings to be implemented. It is developed based on an intuitive assumption that for any adaptive traffic signal control system, the prevailing traffic conditions (as obtained from field sensors) will have a direct impact on the signal timings it optimizes. Thus, with extensive use of data – especially high-resolution data – one could reverse engineer the whole process and approximate the underlying control algorithm from the data. This is exactly what the SCOOT emulator does – it is a model learned from data produced by an actual SCOOT system.

Traffic signals at the major intersections of downtown Abu Dhabi are controlled using SCOOT, which uses data from inductive loop detectors embedded in the Abu Dhabi roads. We utilized a high-resolution dataset obtained from Abu Dhabi DOT to build the SCOOT emulator. The dataset consists of detector and signal states recorded by the SCOOT system for a set of closely spaced intersections. These detector and signal states serve, respectively, as the inputs and outputs

to the emulator. Owing to the sequential nature of the data, we used a Recurrent Neural Network [3, 4] (RNN). The RNN model takes a sequence of detector states recorded during a specific time period in the past and predicts the signal states for a future time step through a series of non-linear transformations. The parameters of the RNN model are estimated from historical data.

The availability of high-resolution data plays a key role in developing the emulator. The data we used had a resolution of 1 second, which means the detector actuations and signal states are recorded every second. One can infer various aspects of the operation of the signal from this data, such as the general traffic condition of the network, traffic arrival patterns, signal phase timings and phase sequences, and lead and lag behavior of signal phase sequences. With this data, a neural network can be employed to learn these underlying patterns, and here we are presenting just one application of it – to learn the control algorithms; this may not have been possible with data of lower resolution, say 15-minute aggregated data. In the next section, a more detailed view on the neural network used to build the emulator is presented.

RNN Model used in the SCOOT emulator

Unlike the actual SCOOT system, which calculates signal timings for a future time horizon using current measurements, the emulator only calculates the signal states for a single time step ahead, say 1 second. The rationale for choosing a 1-second prediction horizon is that we also allow the neural network to learn the gap out (and max out) behavior based on real time vehicle actuations from the data. In order to capture temporal variation in detector actuations, we chose to implement an RNN over a feed forward neural network.

The RNN model used in the emulator is structured as a classification model, owing to the nature of explanatory and predictor variables, which are both categorical. The explanatory variables are the detector states from all loop detectors, each categorized as ‘On’ or ‘Off’. The predictor variables are the signal states of all the protected phases, categorized as ‘Green’ or ‘Red’. Thus, the model takes in a sequence of Ons and Offs over a certain time period in the past and predicts an array of ‘Greens’ and ‘Reds’. The RNN architecture used is shown in Figure 1.

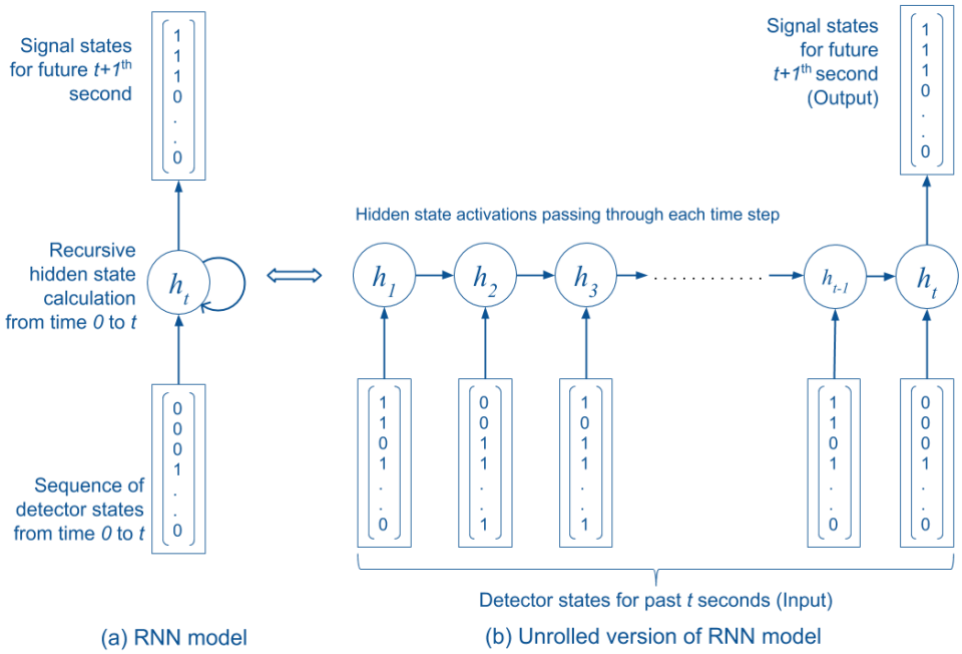


Figure 1: RNN architecture used for developing the SCOOT emulator

The RNN model consists of three layers – an input layer, a hidden layer, and an output layer. The input layer receives the sequence of detector actuations from time, $T = 0$ to time $T = t$, where t can be any reasonable chosen time period, say 60 or 120 or 180 seconds, during which traffic conditions are assumed to have an effect on the predicted signal timings. The input is, thus, a two-dimensional array with columns corresponding to the number of detectors and rows corresponding to time steps. The hidden layer consists of a Long Short Term Memory [5] (LSTM) cell, and is used instead of a vanilla RNN[1] cell due to its inherent limitations in learning long term dependencies when the number of time steps considered is large[2]. The hidden states from the hidden layer are then passed to the output layer, which consist of a sigmoid activation function, and predicts the signal states. The dimension of the output vector is the total number of signal heads. The flow of computations in the RNN is more visible in the unrolled version of the RNN model; see Figure 1b. At each time step, the hidden states are computed based on inputs from that time step and the hidden states associated with the previous time step. The output from the RNN is then computed using the hidden states from the last time step.

[1] Vanilla RNN will have the following hidden state calculation: $h_t = f(W_x x_t + W_h h_{t-1} + b_{xh})$. The hidden state at time step t depends on both the input at the current time step, x_t, and the hidden state at previous time step, h_{t-1}.

[2] In practice, vanilla RNN fails to learn long term dependencies due to a well cited problem referred to as the vanishing or exploding gradient [6]. For deeper networks, the gradients either become too short or too long, which makes the learning slower and ultimately results in under-fitting. LSTM addresses this issue by employing a memory component, which keeps track of the dependencies for a much longer time [5].

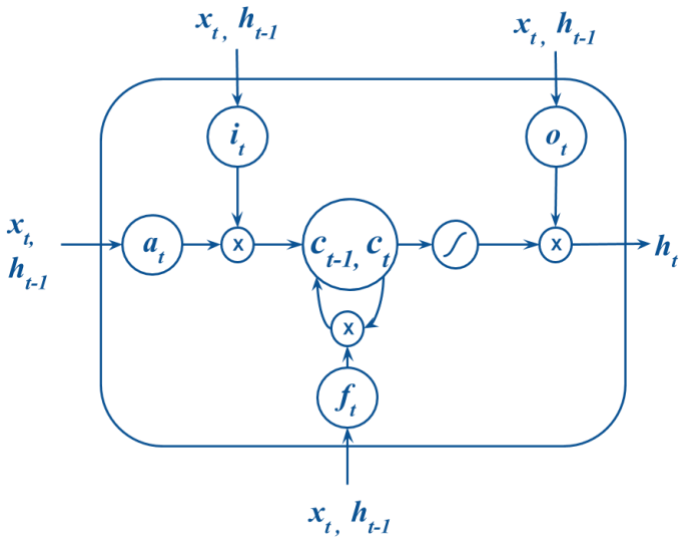


Figure 2: A single LSTM cell and the flow of computation inside it.

Figure 2 gives a closer look at an LSTM cell of the hidden layer and its input-output characteristics. Equations (1) through (6) present the hidden state calculations in the LSTM cell for a time step *t*. Unlike the hidden node in a traditional neural network, the LSTM cell has a cell memory that keeps track of its cell state for a longer period. The cell takes three inputs: input at current time step *t*, previous hidden state *t+1*, and previous cell state c_{t-1}. Equations (2), (3) and (4) represent the three control gates that controls the information to be stored in the primary cell: the input gate regulates the amount of new information flow to the cell's memory component; the forget gate regulates the amount of the old information from the cell memory; and the output gate decides what part of the updated cell state should constitute the new hidden state.

Input activation:	$a_t = \tanh(W_a.x_t + U_a.h_{t-1} + b_a)$	(1)[3]
Input gate:	$i_t = \text{sigmoid}(W_i.x_i + U_i.h_{t-1} + b_i)$	(2)[4]
Forget gate:	$f_t = \text{sigmoid}(W_f.x_f + U_f.h_{t-1} + b_f)$	(3)
Output gate:	$o_t = \text{sigmoid}(W_o.x_o + U_o.h_{t-1} + b_o)$	(4)
Internal state:	$c_t = a_t * i_t + f_t * c_{t-1}$	(5)[5]
Hidden state:	$h_t = \tanh(c_t) * o_t$	(6)
Output:	$\hat{y} = \text{sigmoid}\ (W_y.h_t + b_y)$	(7)

[3] The hyperbolic tangent function for any variable *x* is defined as, tanh (*x*) = [exp(*x*) – exp(-*x*)] / [exp(*x*) + exp(-*x*)]

[4] The sigmoid function for any variable *x* is defined as, sigmoid (*x*) = 1 / [1 + exp(-*x*)]

[5] Note that, *x* * *y* represents the element wise multiplication of arrays *x* and *y*.

The output of the RNN model is computed using Equation (7). The non-linear sigmoid function transforms the hidden states to values between 0 and 1. Note that the output equation solely depends on the hidden state at last time step, which implies that the hidden state at time *t* carries all the information from the previous time steps. Here, W_a, W_i, W_f, W_o, W_y, U_a, U_i, U_f, U_o, b_a, b_i, b_f, b_o, and b_y are the parameters of the RNN model, which are estimated from the data using back propagation.

Equations (1) through (7) represent the forward propagation model of the recurrent neural network. The predictions, denoted $\hat{y}$, are encoded in an array of dimension equal to the number of signal heads, with each element taking values between 0 and 1. Elements closer to 1 are assigned a green state, otherwise they are assigned a red state. In fact, the value of each element represents the probability that the respective signal head shows a green, conditional upon given input detector states. Now, the goal is to estimate the parameter values of the RNN model that best represents the data. In order to do so, we define a cost function, which captures the discrepancy between the predicted and actual signal states. Here, we use a cross entropy cost function, widely used for binary classification problems, and is defined below:

Cost function,
$$C = \sum_{i=1}^{m}\sum_{j=1}^{n_o} y_{i,j} * \log\left(\widehat{y_{i,j}}\right) + \left(1 - y_{i,j}\right) * \log\left(1 - \widehat{y_{i,j}}\right) \quad (8)$$

where,

$y_{i,j}$ is the actual state of signal head *j*, for i^{th} observation in the dataset.

$\widehat{y_{i,j}}$ is the predicted state of signal head *j*, for i^{th} observation in the dataset.

m is the total number of observations (examples) in the dataset.

n_o is the total number of signal heads.

We frame the above problem as an optimization problem, and the best parameters for the RNN model are estimated by minimizing the cost function over the entire dataset. Owing to the non-existence of a closed form solution to the above problem, we employ iterative optimization algorithms such as Stochastic Gradient Descent [(7)] to learn the parameters of the model from the data. In the neural network literature, the procedure is called back propagation through time [(8)], as the algorithm involves unrolling the time steps. By choosing an appropriate set of hyper-parameters (i.e., hidden layer dimensions, number of hidden layers, etc.), the weight parameters of the RNN that best fit the data can be obtained, given that proper measures [(8, 9)] are taken for the neural network model to not overfit the data. The weighted RNN model then should be tested against out-of-sample datasets in order to validate the model.

Single intersection case study

For demonstration, we present here the training and testing of a SCOOT emulator for a single intersection in Abu Dhabi, using an actual field dataset. A four-leg intersection is chosen with detector locations and other geometrical details depicted in Figure 3. There are a total of 16 detectors (inductive loops) recording vehicle actuations from all the incoming links of the intersection; each link has 3 detectors placed at the midblock section and 1 detector placed on the exclusive left turn lane. The positions of the detectors are calibrated in the field. There are 8 signal heads at the intersection, corresponding to each protected signal phase movement.

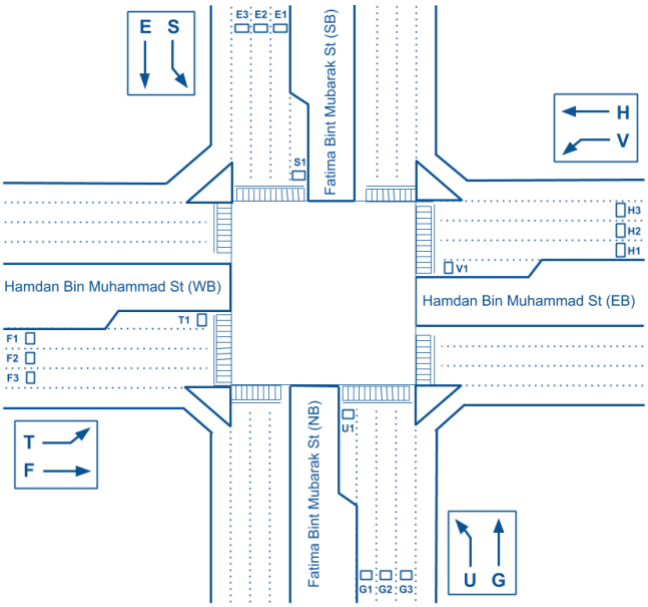


Figure 3: Intersection chosen for study

Data preparation

The raw dataset, containing the detector actuations and signal state records for the intersection over an entire day, is unstructured and needs to be preprocessed in order to make it compatible with the input format of the RNN. The dataset consists of around 86000 observations. Each observation has the detector and signal states for a single time step (1 second), which is then stored in two one-dimensional arrays of size 16 and 8, corresponding to the number of detectors and signal heads at the intersection, respectively.

Each element in the array is a binary: 0 (if detector is inactive or if signal head shows red) or 1 (if detector is active or if signal head shows green). The dataset is then rearranged in such a way that each observation contains the detector state records for the past t secs and signal states at the $t+1^{th}$ sec. Thus, the reshaped dataset has a dimension of (m, t, n_i) for detector states and a dimension of (m, n_o) for the signal states, representing the input and output to the RNN, respectively; here, m = number of observations or examples, t = number of time steps, n_i = input dimension at a single

time step (16 in this case), n_o = output dimension of signal states (8 in this case). The whole dataset is divided into training and testing datasets[6].

Forward propagation model of RNN and Graph definition using Tensorflow

In this study, we have used the Tensorflow package (11), an open source deep learning framework, to train and test the RNN. Tensorflow has the additional advantage that we only need to define the forward propagation steps involved in fitting the neural network; it automatically performs the backpropagation computations using Auto-differentiation. The forward propagation computations are defined on a *Tensorflow graph*; each edge of the graph represents a tensor (any n-dimensional data array) and each node connecting them represents a tensor operation. Abstractly, the *Tensorflow graph* captures the data flow from the input node to the output node in a neural network. The complete python (12) code for training and testing of the emulator for this case study is included with this document (*Scoot_emulator_rnn_training.py*).

We start with an arbitrary neural network architecture and hyper-parameters. *Tensorflow placeholders* (empty slots in the graph) are defined to represent the input and the output data

```
graph = tf.Graph()

with graph.as_default():
    features_ = tf.placeholder(tf.float32, [None, MaxTimeStep, Num_Input_Dims])
    labels_ = tf.placeholder(tf.float32, [None, Num_Output_Dims])
    keep_prob = tf.placeholder(tf.float32, name='keep_prob')

with graph.as_default():
    weight = tf.Variable(tf.truncated_normal([num_hidden_1, Num_Output_Dims]))
    bias = tf.Variable(tf.constant(0.01, shape=[Num_Output_Dims]))

with graph.as_default():
    cell = tf.contrib.rnn.LSTMCell(num_hidden_1, state_is_tuple=True)
    outputs, states = tf.nn.dynamic_rnn(cell, features_, dtype=tf.float32)
    _out = tf.transpose(outputs, [1, 0, 2])
    last = tf.gather(_out, MaxTimeStep - 1)

with graph.as_default():
    prediction = tf.add(tf.matmul(last, weight), bias)
    logits_ = tf.nn.sigmoid(prediction)
    cost = tf.reduce_sum(
             tf.losses.sigmoid_cross_entropy(
             multi_class_labels=labels_, logits=prediction))
    pred_sig_states = tf.cast(tf.round(logits_), tf.float32)
    correct_pred = tf.equal(pred_sig_states, labels_)
    accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
    optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Figure 4: Python code snippet for the Forward propagation model of RNN using Tensorflow

[6] Note that the dataset obtained does not include information on the amber or flashing green; however, the methodology presented in this report can be easily extended to consider the explicit amber or flashing green, using a different coding scheme like integer encoding.

feeders. The weights and biases of the neural network are defined as *Tensorflow variables* and need to be initialized with random values; here, we initialized it using a normal distribution. We used Tensorflow's built-in functions such as addition, multiplication, etc., to define tensor operations; these built-in functions will have its derivatives defined in the backend and will make it easier for the backward propagation computations. Also, we do not need to explicitly specify the trainable variables in the network, it is rather identified automatically from the graph. Figure 4 shows a Python code snippet of the forward propagation computations using Tensorflow.

In the code snippet, the first two sets of lines of code define the placeholders and variables in the *Tensorflow graph*. The next set defines the LSTM layer; we used the *lstmcell* application programming interface (API) from the Tensorflow package. It first creates an *lstm* cell and then the *lstm* layer; only the hidden states from the last *lstm* are carried over to the output layer. The final lines of code define the output layer, cost function and the optimizer. Stochastic gradient descent is used as the optimizer for training the neural network. The network architecture and hyper parameters chosen for this case study are tabulated in Table 1.

Table 1: Hyper parameters for the neural network

Network architecture	
a) Number of time steps	120
b) Input dimension at each time step	16
c) Output dimension	8
d) Hidden layer dimension	16
e) Number of hidden layers	1
Learning parameters	
f) Learning rate	0.001
g) Loss function	Cross entropy
h) Batch size	1000
i) Number of iterations	400
j) Regularization	Dropout, with $p = 0.20$

Training phase (Back propagation through time)

In the training phase, we execute the computation graph by feeding the data (both the input detector states and the output signal states), in batches, to the *Tensorflow placeholders*, previously defined in the graph. Figure 5 shows a Python code snippet of the training phase of the SCOOT emulator.

In each iteration, the weights of the trainable variables are (automatically) updated while minimizing the cost function. Throughout the process, we monitor the accuracy of predictions from the neural network, on both the training and test datasets (see Figure 6). The model is trained until we observe negligible increase in training accuracy (i.e., < 0.001) or when test accuracy starts to decline, in which case the model is said to over-fit the data, or until a pre-specified maximum

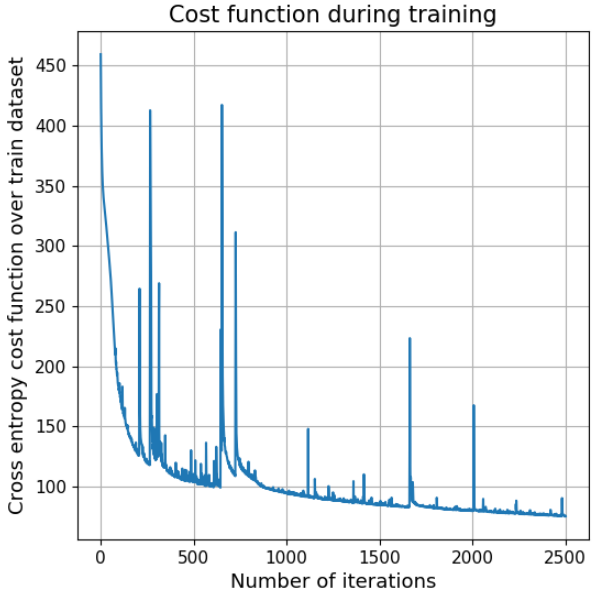
number of iterations is reached. If the trained model does not achieve reasonable accuracy, the whole procedure is repeated with a different network architecture and hyper parameters [13].

```
epochs = # number of iterations
no_of_batches = int (len (train_input) / batch_size) # number of batches
costs = []; train_accuracy = []; test_accuracy = []

with tf.Session(graph=graph) as sess:
    sess.run(tf.global_variables_initializer()) # initialize variables

    for i in range(epochs):
       ptr = 0; epoch_cost = 0;
      for j in range(no_of_batches):
           x, y = train_input[ptr:ptr+batch_size],
                        train_output[ptr:ptr+batch_size] # one batch of data
           ptr += batch_size
           _, c = sess.run([optimizer, cost],
                        {features_: x, labels_: y, keep_prob: k_p})
```

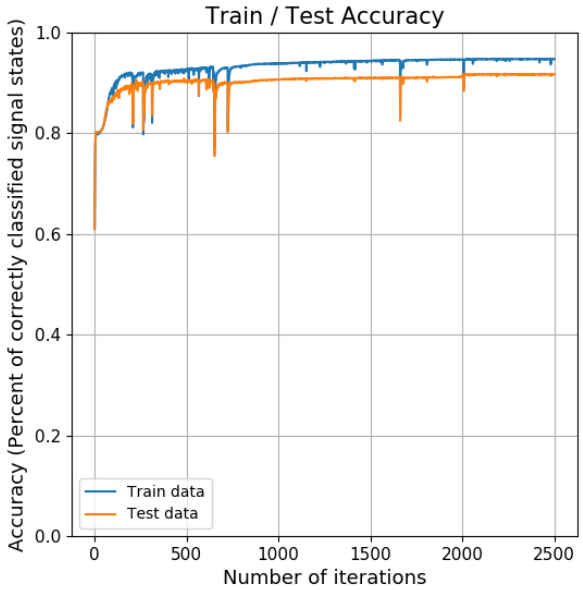
Figure 5: Python code snippet for training of RNN using Tensorflow

Validation of trained model

The SCOOT emulator for the intersection is trained over 2500 iterations and the results are presented here. Figures 6a and 6b show the cost function and the accuracy[7] of the neural network during the training phase. The accuracy of the trained model is 94.7% on the train dataset and 91.7% on the test dataset.



(a) Cost function

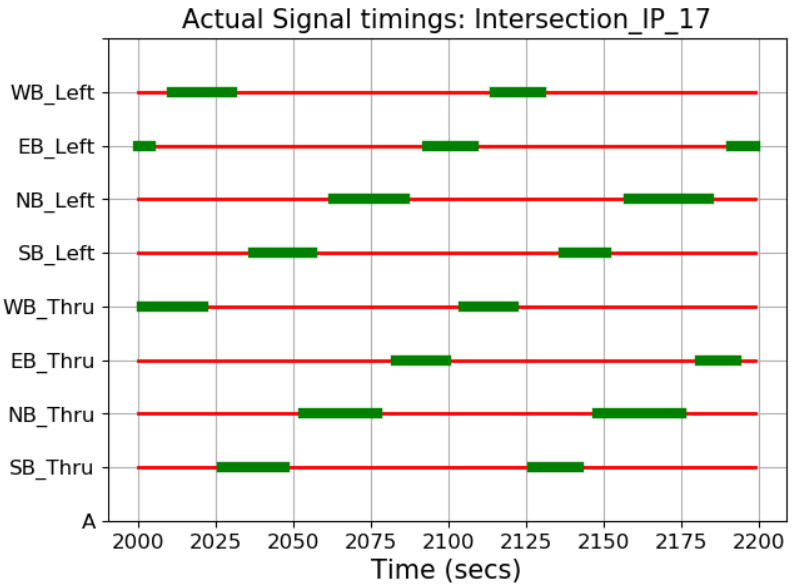


(b) Accuracy

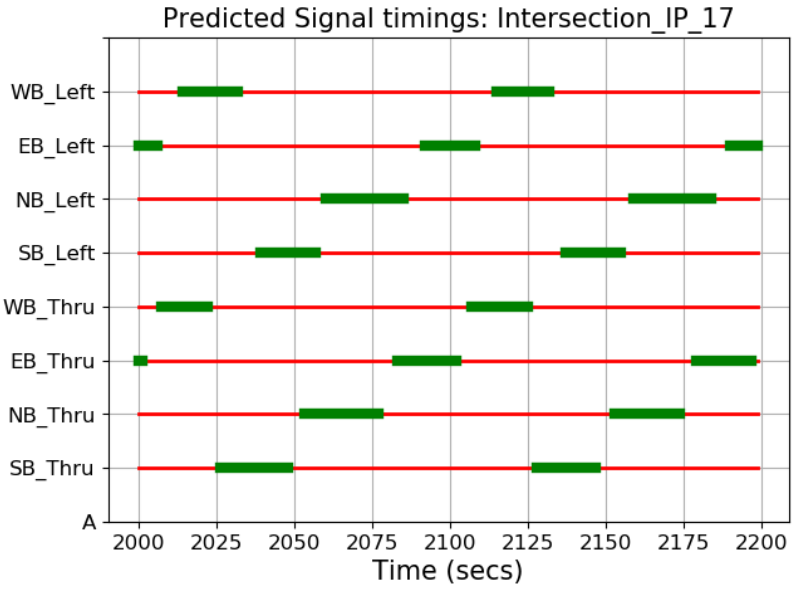
[7] Accuracy, in this case, is defined as the percentage of correctly classified signal states.

Figure 6: Performance of the neural network during training phase

Figures 7a and 7b show a comparison of the actual signal timings and the emulator predicted signal timings for a sample time period of 200 secs on the test data. We observe that the predicted signal timings for most of the signal phases match the actual signal timings found in the raw data. Also, the emulator is able to capture the phase sequences, and lead-lag behaviors from in the data.



(a) Actual signal timings as obtained from the raw dataset



(b) Predicted signal timings from the SCOOT emulator

Figure 7: Comparison of actual and predicted signal timings for the test intersection.

References

1. Hunt, P. B., Robertson, D. I., Bretherton, R. D., & Royle, M. C. "The SCOOT on-line traffic signal optimization technique", *Traffic Engineering & Control*, 23:4, (1982). pp. 190-192.
2. C.M. Bishop, "Pattern Recognition and Machine Learning", (2006), Springer, New York, NY.
3. Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y., "Deep learning", 1, (2016), MIT press, Cambridge, MA, USA.
4. Recurrent Neural Network Tutorial, http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/.
5. Hochreiter, S., & Schmidhuber, J. "Long short-term memory", *Neural computation*, 9:8, (1997), pp. 1735-1780.
6. Bengio, Y., Simard, P., & Frasconi, P. "Learning long-term dependencies with gradient descent is difficult", *IEEE transactions on neural networks*, 5:2, (1994), pp. 157-166.
7. Bottou, L. "Stochastic gradient learning in neural networks", *Proceedings of Neuro-Nımes*, 91:8, (1991), pp. 12.
8. Back propagation through time. http://ir.hit.edu.cn/~jguo/docs/notes/bptt.pdf.
9. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. "Dropout: a simple way to prevent neural networks from overfitting", *The Journal of Machine Learning Research*, 15:1, (2014), pp. 1929-1958.
10. Zaremba, W., Sutskever, I., & Vinyals, O. "Recurrent neural network regularization", (2014), arXiv preprint: https://arxiv.org/abs/1409.2329.
11. TensorFlow, https://www.tensorflow.org/.
12. Python programming language, https://www.python.org/.
13. Bergstra, J., & Bengio, Y. "Random search for hyper-parameter optimization", *Journal of Machine Learning Research*, 13: Feb, (2012), pp. 281-305.
14. Snoek, J., Larochelle, H., & Adams, R. P. "Practical bayesian optimization of machine learning algorithms", *In Advances in neural information processing systems*, (2012), pp. 2951-2959.