CS 289A Spring 2017 Final Project

Mitchell Negus NEGUS@BERKELEY.EDU
UC Berkeley, Department of Nuclear Engineering

Dayton Thorpe DAYTON@BERKELEY.EDU
UC Berkeley, Department of Physics

Abstract

We develop a Recurrent Neural Network model of Bay Area traffic. Data come from 943 sensors around the Bay Area measuring lane occupancy every 10 minutes for one year. In one version of the model, we classify traffic as low, medium, or high. In a second version, we predict the absolute level of traffic. Both models capture nearly all variation in traffic. However, much simpler models perform just as well.

1. Introduction

According to the US Department of Transportation Federal Highway Administrations Urban Congestion report for 2015, San Francisco traffic ranked fourth worst among US major metropolitan areas in terms of congestion time [1]. Bay Area roads are jammed for nearly seven hours of the day. Alleviating this traffic could provide a host of benefits: increased productivity, reduced fuel consumption, lower pollution rates, or just less frustrated commuters. This report discusses a recurrent neural network constructed to predict times of peak traffic volume so that traffic pattern information may be used to reduce congestion.

To quantify Bay Area traffic patterns, this project relied on the UC Irvine PEMS-SF Data Set. The data provides measured lane occupancy fractions for 963 lane sensors on Bay Area freeways, with each sensor collecting data once every 10 minutes for over 1 year. Traffic data collected for one sensor over the course of 1 day is shown in figure 1.

Due to the nature of the time series data format, recurrent neural networks (RNN) were chosen to serve as the predictive tool.

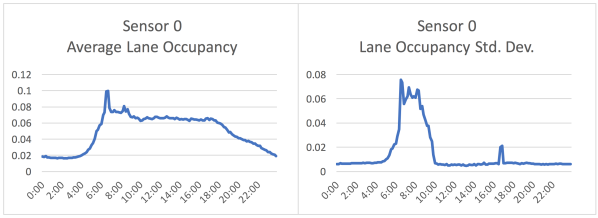

Figure 1. Lane occupancy rates for a randomly selected single sensor in the PEMS-SF Data Set.

2. Recurrent Neural Network

At each time step, t, the RNN takes as input the lane occupancy at sensor i and the hidden layer calculated at time t-1. The matrix U connects the current lane occupancy to the current hidden layer, the matrix V connects the previous hidden layer to the current hidden layer, and the matrix W connects the hidden layer to the occupancy forecast. Predictions are calculated over T time steps simultaneously. At the first time step, t_0, the previous hidden layer is set to a vector of zeros. After propagating forward through to time step $t_0 + T$, a final prediction is made for the lane occupancy at time $t_0 + T + 1$. At each step, the gradient of the loss function with respect to the matrices U, V, and W is calculated, although no change is made to those matrices.

After these T predictions are calculated, back propagation begins and makes T successive adjustments to each of the matrices U, V, and W.

3. Implementation Details

To form a complete analysis, two similar RNNs were tested. The first approach implemented a classification RNN for determining general traffic patterns, and the second implemented a regression algorithm.

Both RNNs evaluated four hours of traffic data (24 consecutive occupancy values) from a given sensor and made predictions of future occupancy in that sensor's lane based

on the four hour window. The novelty of RNNs allowed these predictions to be updated in the form of rolling four hour windows, so that the networks achieve learning in real-time.

Each of the 24 occupancy values were then fed into the 24 recurrent layers of the RNN, multiplied by the single common weight matrix, V, across layers, and passed through a rectified linear unit (ReLU) activation function. The ReLU used here is of the form

$$r(x) = \begin{cases} x, & x >= 0 \\ \alpha x & \text{else,} \end{cases}$$

with $\alpha = 0.001$. The small but non-zero slope for negative input values helps prevent hidden sites from becoming "stuck" when their input is consistently negative, then their gradients are 0, and they are always evaluated as 0, contributing nothing to the RNN.

At this stage, the two approaches differ in their implementation. The classification model predicts discrete traffic levels and updates the recurrent layers based on those discrete predictions, as opposed to the regression model which updates with respect to the continuous range of occupancies.

4. Classification: Predicting Low, Medium, High Traffic

For the classification problem, we categorized occupancy values into 3 discrete groupings: high, medium, and low traffic levels. The groupings were evenly chosen for the occupancies (i.e. for occupancy x_t at time t,$x_t < 0.33, 0.33 < x_t < 0.67$, and $0.67 < x_t < 1.0$ corresponded to low, medium, and high traffic levels respectively.

We evaluated the continuous input using the ReLU function in each recurrent layer, those hidden values were transformed into discrete outputs through a three-unit logistic function activation layer. Outputs corresponded to one-hot-encoded predictions of the traffic level at the target time, and were compared against the true traffic level at the target time.

Though we did not compute the loss at each time step explicitly, the predicted outputs and true value were back-propagated through both layers of each recurrent block using gradients found for the logistic loss function

$$L(z) = -\sum_{j=1}^{3} y_j \ln z_j + (1 - y_j) \ln(1 - z_j).$$

The update process was then executed as explained in section 3. A diagram showing the setup for a similar recurrent neural network, but with only 3 recurrent layers instead of 24, is provided in figure 4. Input occupancies at time t are given as x_t.

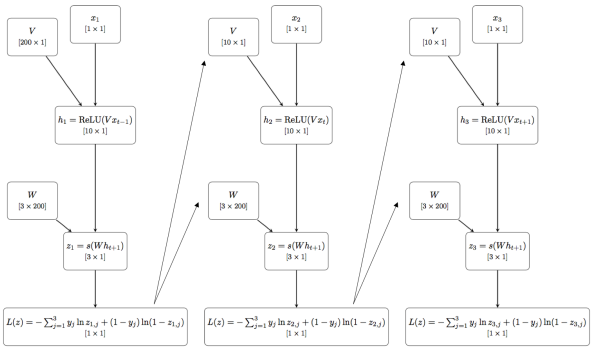

Figure 2. A RNN that is similar in design to the classification implementation that is described. Note: this RNN has only 3 recurrent layers, while our model uses 24 layers.

We trained this neural network on the first 60,000 data points (corresponding to slightly more than the first 400 days of available data with measurements at 10 minute intervals), for each of 20 randomly selected sensors. Training error over the 60,000 point interval showed consistent improvement for each sensor. Plots of the training accuracy are given in figure 4. Since the problem is a classification problem, the accuracy begins randomly at values of either 0 or 1 (whether the initial guess provided by random generation of weight matrices U and V gives a correct classification or not). It is evident from the figure that as many time windows are used for training, the training accuracy improves to about 99.5% or better.

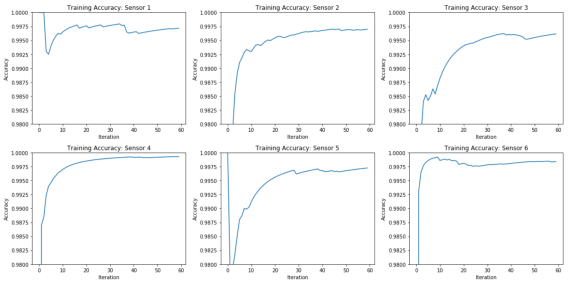

Figure 3. Lane occupancy rates for a randomly selected single sensor in the PEMS-SF Data Set.

For each of the 6 sensors shown in figure 4, we predicted traffic levels at 100 times after the conclusion of the training time window. Test points were selected at regular intervals, with the number of intervals between points not divisible by 6 to ensure that the selected times were not the same every hour or every day. Test accuracies were 99% for each sensor.

Full code for the traffic classification neural network is pro-

vided in appendix A.

5. Regression: Predicting Occupancy

For the regression model, we sought to predict the absolute level of lane occupancy. The lane occupancy is a positive real number unbounded from above, so the forecast was simply Wh, where h is the vector of hidden units. The lane occupancy cannot be negative, so a ReLU could have been employed to prevent forecasts of negative occupancy, but no such forecasts were observed, so such a transformation was unnecessary. We optimized the RNN weights by minimizing the square deviation (MSD) between the predicted and actual lane occupancy. Both the input layer and the hidden layer included a bias term set to 1. A single RNN was trained for all traffic sensors.

We trained the model on 100 randomly chosen sensors over the first 10 days of available data. The time window, T, was 4 hours, for 24 time slices separated by 10 minutes each. After propagating forward and backwards over the data in the range $[t_0, t_0 + T]$, we then advance by one time step, propagating over the data in the range $[t_0 + 1, t_0 + 1 + T]$. We tested the model over the same ten days on a different set of 100 randomly chosen sensors.

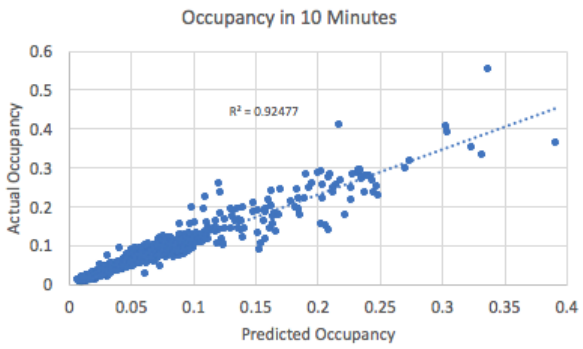

Figure 4. Actual lane occupancy versus lane occupancy predicted 10 minutes in advance. Measured by R^2, the RNN model explains 92.5% of the variation in lane occupancy.

As seen in Figure 4, the model performs very well, capturing 92.5% of the observed variation in the test set traffic.

Figure 5 shows that except for brief spikes around rush hour, the lane occupancy changes slowly over each ten minute interval. Predicting the absolute level of the traffic may not be a difficult problem because the variations are small compared to the absolute level. This observation suggests that we should compare our RNN to a simpler model for short-term traffic prediction and that we should modify our RNN to forecast the changes over each 10-minute interval rather than the absolute lane occupancy.

By zooming in on the first 10 hours of the window prediction, Figure 6, reveals approximately how the model is working. The predicted traffic is almost exactly the traffic from 10 minutes before. Figure 7 shows how the RNN compares in the test set to a much simpler model that forecasts that the traffic will stay the same. With 10 or fewer hidden units, the RNN significantly under-performs the simple model. With 20 or more hidden units, the RNN matches, but does not outperform, the simple model. With 100 hidden units, the RNN performance declines slight and begins to show signs of over-fitting as the validation error is significantly larger than the test error, a problem not apparent for the smaller hidden layers.

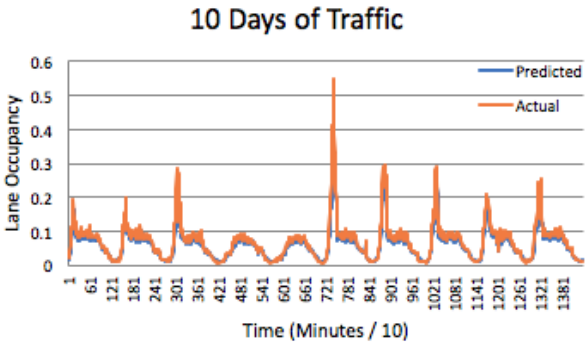

Figure 5. Time series of actual lane occupancy and predictions 10 minutes in advance over 10 days.

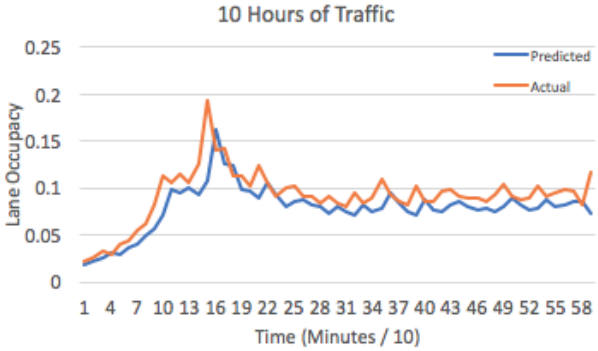

Figure 6. Time series of actual lane occupancy and predictions 10 minutes in advance over the first 10 hours of the week.

To improve the model, we then train our RNN not on the absolute level of the lane occupancy but on the change in the occupancy. To predict the absolute lane occupancy in ten minutes, we can add the RNN prediction for the change to the current lane occupancy.

Unfortunately, Figure 8 shows that our RNN fails to predict the lane occupancy change over 10 minutes. Some change in the structure of the model is required to describe changes in traffic over 10 minute intervals.

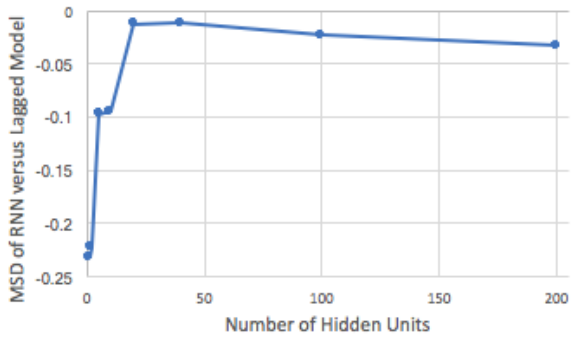

Figure 7. Performance of RNN compared to a much simpler model as a function of the number of hidden units in the RNN. Vertical axis shows the percentage improvement in the MSD when switching from the simple model to the RNN, with a negative value indicating a decline in performance. The simple model forecasts that the traffic at $t + 1$ will be the same as at time t.

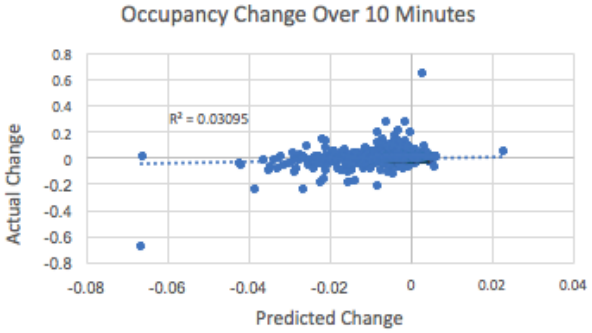

Figure 8. Actual change in lane occupancy over 10 minutes versus prediction at the beginning of the time interval. Measured by R^2, the RNN model explains a trivial 3% of the variation.

Full code for the traffic regression neural network is provided in appendix B.

6. Conclusion

Both versions of the model successfully captured most of the variation in the traffic. However, they did not significantly outperform the simpler prediction that the traffic at time t+1 is the same as the traffic at t. Attempting to train the regression model directly on the change in traffic over each interval did not improve the model performance. A modification to the model architecture is necessary to capture the change in traffic more effectively.

Appendices

A. RNN Classification Code

```
# # Recurrent Neural Network
# ## Classifier: Traffic as Low, Medium, or High

# Import necessary modules.
import numpy as np
from matplotlib import pyplot as plt


# Load data from text file.
data = np.genfromtxt('/Users/mitch/Dropbox/FinalProject/traffic2.txt')


# Define functions and derivatives of those functions to be used in the RNN.
def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))

def relu(x):
    return np.maximum(x,np.zeros_like(x))

def Leaky_ReLU_deriv(x):
    dRLU = np.empty(len(x))
    for i in range(len(x)):
        if x[i] > 0:
            dRLU[i] = 1
        else:
            dRLU[i] = 0.1
    return dRLU

# Rename function for convenience
LRD = Leaky_ReLU_deriv


# Define a recurrent neural network class, which repeats a single layer.
class OneLayerRNN:

    def __init__(self,hidden_units=10,output_units=3):
        self.V = None
        self.W = None
        self.h = None
        self.z = None
        self.hidden_units = hidden_units
        self.output_units = output_units
        self.windowlen = None
        self.futuretime = None

    def initialize(self,X):
        # Weight matrices start as normal distributions
        self.V = np.random.normal(scale=1.0/np.sqrt(self.hidden_units),size=((self.hidden_units),1))
        self.W = np.random.normal(scale=1.0/np.sqrt(self.hidden_units),size=(self.output_units,self.hidden_units))
        self.h = np.zeros((len(X),self.hidden_units))
        self.z = np.zeros((len(X),self.output_units))

        # Matrices for gradient update
        self.gV = np.zeros_like(self.V)
        self.gW = np.zeros_like(self.W)


    def forward(self,X):
        for i in range(len(X)):
            x = X[i]                                                    # traffic at time i

            self.h[i] = relu(np.dot(self.V,x)).flatten()        # calculate new layer output
            self.z[i] = sigmoid(np.dot(self.W,self.h[i]))

    def backward(self,X,y):
        """Backpropagation through time"""
        for i in range(len(X)):
            x = X[i]
            Q = self.z[i] - y
            self.gV += np.array([np.dot(self.W.T,Q)*LRD(self.h[i]*x)]).T
            self.gW += np.outer(Q,self.h[i])

    def update_VW(self,epsilon):
        """Update"""
```

```
        self.V = self.V - epsilon*self.gV
        self.W = self.W - epsilon*self.gW

    def one_hot_encode(self,occ):
        if occ < 0.33: return np.array([1,0,0])
        elif occ >= 0.33 and occ < 0.67: return np.array([0,1,0])
        elif occ >= 0.67: return np.array([0,0,1])
        else: print('Error: the given occupancy value was ',occ)


    def train(self,timeframe,epsilon,windowlen=24,futuretime=0):
        """
        Train the neural network given input data as X
        X:   - a series of occupancies
        """
        self.windowlen=windowlen
        self.futuretime = futuretime

        windowstart = 0
        firstwindow = timeframe[:windowlen]
        self.initialize(firstwindow)
        counter,total = 0,0
        trainingerrors = []
        while windowstart < len(timeframe)-windowlen-futuretime-1:
            timewindow = timeframe[windowstart:windowstart+windowlen]
            timewindowlabel = self.one_hot_encode(timeframe[windowstart+windowlen+futuretime])
            trueforecast = np.argmax(timewindowlabel)

            self.forward(timewindow)
            self.backward(timewindow,timewindowlabel)
            self.update_VW(epsilon)

            prediction = np.argmax(np.average(self.z,axis=0))

            if prediction == trueforecast:
                counter += 1
            total += 1

            TE = counter/total
            if windowstart%1000==0:
                #print(windowstart,'\t',TE)
                trainingerrors.append(TE)
            windowstart=windowstart+1
        #print(counter/total)
        return trainingerrors


    def predict(self,timeframe,predicttime,predictgap=0):
        timewindow = timeframe[predicttime-predictgap-self.windowlen:predicttime-predictgap]
        self.forward(timewindow)
        classification = np.argmax(np.average(self.z,axis=0))
        '''if classification==0:
            print('low')
        elif classification==1:
            print('med')
        elif classification==2:
            print('high')'''

        return classification
```

B. RNN Regression Code

```
import numpy as np
import copy
import sys

def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))

def ReLU(x):
    return np.maximum(x,np.zeros_like(x))

"""
def ReLU_derivative(x):
    if x >= 0: return 1
    return 0
"""

#read in data - a row is a sensor, a column is a point in time
traffic_file = open('traffic2.txt')
traffic_lines = traffic_file.readlines()
traffic_file.close()
num_sensors = len(traffic_lines)
num_sensors_considered = 1 #For now, I will look at 1 sensor at a time
num_timepoints = len(traffic_lines[0].split())

traffic = np.zeros([num_sensors,num_timepoints])
for i in range(num_sensors):
        tokens = traffic_lines[i].split()
        for j in range(num_timepoints): traffic[i,j] = float(tokens[j])

#setup neural network
rolling_window_hours = 4
rolling_window = rolling_window_hours * 6 #each time point is separated by 10 minutes
hidden_units = 200
#this will map from the input layer and the previous hidden layer to the hidden layer
v = np.random.normal(scale=1.0/np.sqrt(num_sensors_considered + hidden_units),
                        size=[hidden_units,num_sensors_considered + hidden_units + 1])
#this will map from the hidden layer to the output layer
w = np.random.normal(scale=1.0/np.sqrt(hidden_units),size=[num_sensors_considered,hidden_units + 1])

v_update = np.zeros_like(v)
w_update = np.zeros_like(w)

alpha = 0.001 #learning rate
training_set = np.random.binomial(1, 0.8, num_sensors)

intervals_ahead = 1

#train
step = 0
total_error = 0
train_max = 1440
average = 0
sensors_trained = 0
error_count = 0
LRD = 0.1 #Leaky ReLU derivative
print "Training Error"
for sensor_iteration in range(num_sensors):
    if training_set[sensor_iteration] == 0: continue
    train_sensor = sensor_iteration
    sensors_trained += 1
    if sensors_trained > 100: break
    for i in range(train_max): #range(num_timepoints-rolling_window):
        #print "sensor: ", train_sensor
        step += 1
        if step % (train_max * 10) == 0:
            print step, total_error / error_count
            total_error = 0
            error_count = 0
            alpha *= 0.75
        history = traffic[train_sensor,i:i+rolling_window]

        previous_hidden_layers = list()
        #previous_hidden_layers.append(np.zeros(hidden_units))
        previous_hidden_layers.append(np.zeros(hidden_units))
        grad_w_of_L = list()
        grad_v_of_L = list()

        #forward propagation
        for j in range(rolling_window):
```

```
            input = np.append(history[j],previous_hidden_layers[-1])
            input = np.append(input,[1.0])
            #print "input: ", input
            hidden = ReLU(np.dot(v,input))
            hidden_one = np.append(hidden,[1.0])
            previous_hidden_layers.append(copy.deepcopy(hidden))
            prediction = np.dot(w, hidden_one) #linear output
            next = traffic[train_sensor,i+j+intervals_ahead] #- traffic[train_sensor,i+j]
            error = prediction - next
            if j == rolling_window - 1:
                average += next
                total_error += np.abs(error[0])
                error_count += 1
            #print "error: ", error
            #print "hidden: ", hidden
            grad_w_of_L.append(2 * error * hidden_one)
            grad_h_of_L = 2 * error * np.transpose(w)
            grad_h_of_L = grad_h_of_L[:-1] #remove the last row of grad_h_of_L, corresponding to the response of L to changing the co
            grad_v_of_h = np.zeros(v.shape)
            for k in range(hidden_units):
                if np.dot(input,v[k]) > 0: grad_v_of_h[k] = input
                else: grad_v_of_h[k] = LRD
            grad_v_of_L.append(grad_h_of_L * grad_v_of_h)
        #print step, total_error

        #back propagation
        for j in range(rolling_window):
            v_update -= grad_v_of_L[-j-1]
            w_update -= grad_w_of_L[-j-1]

        v += alpha * v_update
        w += alpha * w_update

        v_update *= 0
        w_update *= 0

average *= 1.0 / step

#print "v: ", v
#print "w: ", w

#test
total_error = 0
MAD = 0
step = 0
sensors_tested = 0
error_count = 0
print "Test Error"
for sensor_iteration in range(num_sensors):
    if training_set[sensor_iteration] == 1: continue
    train_sensor = sensor_iteration
    sensors_tested += 1
    for i in range(train_max): #range(num_timepoints-rolling_window):
        step += 1
        #if step % (train_max * 100) == 0:
        if sensors_tested > 100:
            print step, total_error / (error_count)
            print "MAD: ", MAD / (error_count)
            sys.exit()
            total_error = 0
            error_count = 0
        history = traffic[train_sensor,i:i+rolling_window]
        previous_hidden_layers = list()
        previous_hidden_layers.append(np.zeros(hidden_units))
        for j in range(rolling_window):
            input = np.append(history[j],previous_hidden_layers[-1])
            input = np.append(input,[1.0])
            hidden = ReLU(np.dot(v,input))
            hidden_one = np.append(hidden,[1.0])
            #print hidden[0:5]
            previous_hidden_layers.append(copy.deepcopy(hidden))
            if j == rolling_window - 1:
                prediction = np.dot(w, hidden_one) #linear output
                next = traffic[train_sensor,i+j+intervals_ahead] #- traffic[train_sensor,i+j]
                #print prediction[0], next
                error = prediction - next
                total_error += np.abs(error[0])
                MAD += np.abs(next - traffic[train_sensor,i+j])
                error_count += 1
```