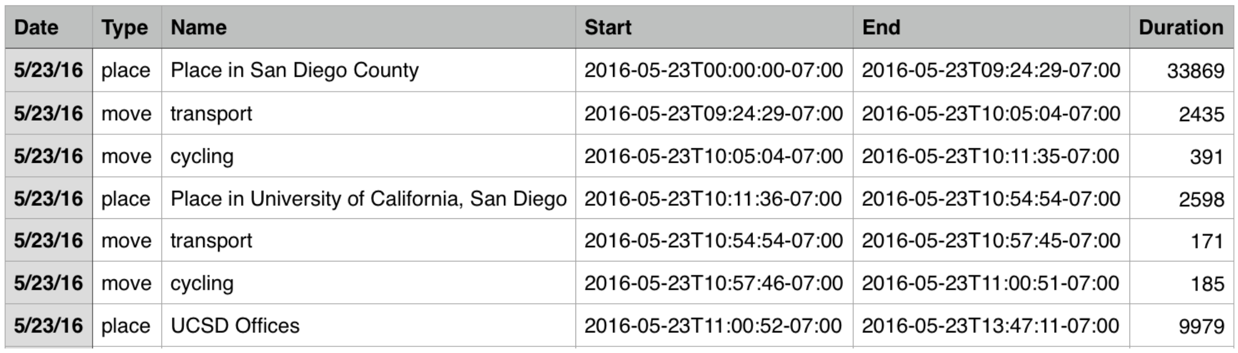
Abstract

This report details my experiments training a Recurrent Neural Network (RNN) with Long-Short Term Memory (LSTM) on my own daily activity data which I collect passively with the app *Moves*. The goal of this project was to transform the exported activity data into a machine-readable format and to train and optimize an LSTM network to predict my next move or location based off of my previous activity.

Introduction

I have been collecting data with an app on my phone called *Moves* which automatically tracks and labels a users activities and locations based on the device's built-in sensors and GPS. Since downloading this app, it has passively recorded over 590 days of my activity, which include things like my bike rides to class, drives to work, visits to family, et cetera. This data is available for export in many formats and I downloaded the CSV version and explored the data to see if it would be possible to adapt for a machine learning project. The data captured by *Moves* seemed detailed enough and spanned a long enough time range to where I decided to implement a recurrent neural net with long-short term memory to model and predict my daily activity.

Figure 1. Excerpt of exported *Moves* data before processing



Date	Type	Name	Start	End	Duration
5/23/16	place	Place in San Diego County	2016-05-23T00:00:00-07:00	2016-05-23T09:24:29-07:00	33869
5/23/16	move	transport	2016-05-23T09:24:29-07:00	2016-05-23T10:05:04-07:00	2435
5/23/16	move	cycling	2016-05-23T10:05:04-07:00	2016-05-23T10:11:35-07:00	391
5/23/16	place	Place in University of California, San Diego	2016-05-23T10:11:36-07:00	2016-05-23T10:54:54-07:00	2598
5/23/16	move	transport	2016-05-23T10:54:54-07:00	2016-05-23T10:57:45-07:00	171
5/23/16	move	cycling	2016-05-23T10:57:46-07:00	2016-05-23T11:00:51-07:00	185
5/23/16	place	UCSD Offices	2016-05-23T11:00:52-07:00	2016-05-23T13:47:11-07:00	9979

Method

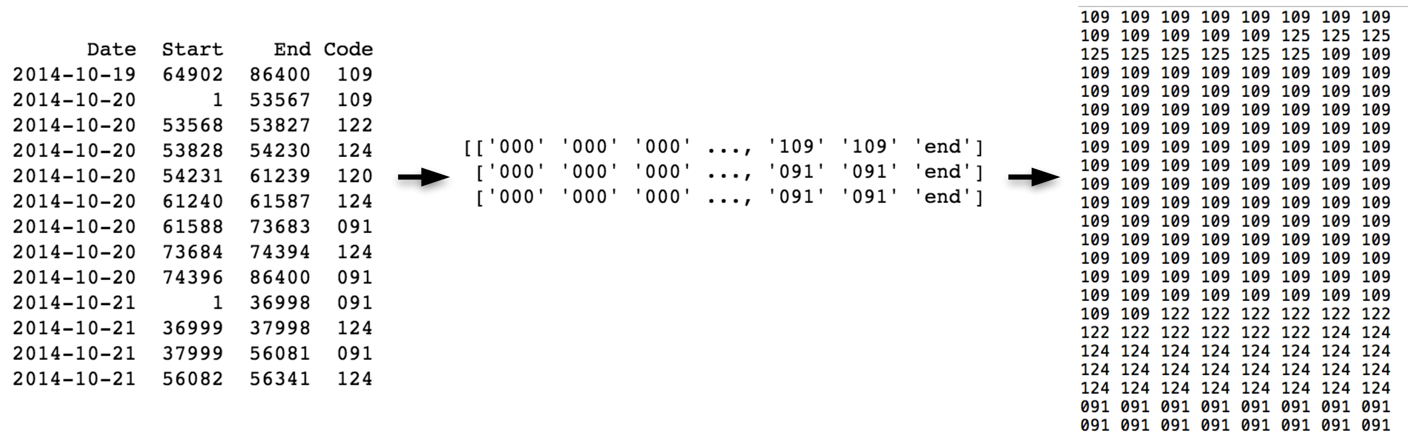
Data Processing

Each row of the file represents one activity, either a movement or place, and is labeled with the date, start and end times, and duration. In the column "Name", the activity is labeled more specifically, with an auto-generated movement name (i.e. "cycling", "transport", "walking") or place name (i.e. "Place in University of California, San Diego"). Upon looking at this data I immediately thought of different ways of transforming it into something usable for a machine learning algorithm.

I started by changing the start and end times from 24-hour clock (24:00:00) to seconds (1 to 86,400) and by using 1-of-k encoding to convert the categorical "Names" to numbers. The place names that are recorded are not very granular, so many places that I go to end up under broad categories like "Place in San Diego County", this meant that there were only about 125 unique "Names" to encode. I ended up with a data frame that looked similar to the CSV file except that some of the irrelevant columns were removed (i.e. "Duration", "Type") and the data itself had been transformed into neat, numerical values (Figure 2A).

From this transformed data, I created a matrix of dimensions 596 days by 86400 seconds with each cell containing the corresponding activity code for that point in time. Since my activity does not change that frequently, I downsampled this dataset, taking every 30th second and giving me a 596 by 2880 matrix. Finally, I appended an “end-of-sequence” token to the end of each row to signify the end of each day, this was simply done by appending a new column to the end of the matrix and adding the string ‘end’ to each cell (Figure 2B). The data was then separated by whitespaces and saved to a text file to be used as input to the LSTM model (Figure 2C).

Figure 2. Data preprocessing pipeline



```
      Date  Start    End Code
2014-10-19  64902  86400  109
2014-10-20      1  53567  109
2014-10-20  53568  53827  122
2014-10-20  53828  54230  124
2014-10-20  54231  61239  120
2014-10-20  61240  61587  124
2014-10-20  61588  73683  091
2014-10-20  73684  74394  124
2014-10-20  74396  86400  091
2014-10-21      1  36998  091
2014-10-21  36999  37998  124
2014-10-21  37999  56081  091
2014-10-21  56082  56341  124
```

```
[['000' '000' '000' ..., '109' '109' 'end']
 ['000' '000' '000' ..., '091' '091' 'end']
 ['000' '000' '000' ..., '091' '091' 'end']]
```

```
109 109 109 109 109 109 109 109
109 109 109 109 109 125 125 125
125 125 125 125 125 125 109 109
109 109 109 109 109 109 109 109
109 109 109 109 109 109 109 109
109 109 109 109 109 109 109 109
109 109 109 109 109 109 109 109
109 109 109 109 109 109 109 109
109 109 109 109 109 109 109 109
109 109 109 109 109 109 109 109
109 109 109 109 109 109 109 109
109 109 109 109 109 109 109 109
109 109 109 109 109 109 109 109
109 109 109 109 109 109 109 109
109 109 109 109 109 109 109 109
109 109 109 109 109 109 109 109
109 109 122 122 122 122 122 122
122 122 122 122 122 122 124 124
124 124 124 124 124 124 124 124
124 124 124 124 124 124 124 124
124 124 124 124 124 124 124 124
091 091 091 091 091 091 091 091
091 091 091 091 091 091 091 091
```

(A) Numerical representation of original data. (B) A 596x2280 matrix where each row represents a day, columns represent time points, and each cell value represents the corresponding activity that occurred at that time point. (C) Excerpt of final text file fed into network.

<u>LSTM</u>

People commonly choose to model sequential data with Hidden Markov Models (HMM) or Recurrent Neural Networks so I decided to experiment with RNNs because of their powerful ability to store a memory of past inputs. I used a variation of the recurrent net architecture known as Long-Short Term Memory which performs even better over longer-term dependencies. I found an existing LSTM network for word-level language modeling and repurposed it to predict my activity. The network predicts activity by modeling the probability distribution of the next event in the sequence given a sequence of previous events. I created a network with two hidden layers- each containing 256 hidden unit- and trained it to predict my activity using the deep-learning framework Torch.

Experiment

Keeping the model parameters fixed, I experimented with the optimization parameters with the goal of obtaining as low of a validation loss as possible. I first trained my network using a batch size of 1, a sequence length of 120, a learning rate of 2e-3, and the optimization algorithm Adagrad. This algorithm adapts the learning rate based on the parameters by performing large updates for infrequent parameters and small updates for frequent parameters (4). Using this algorithm and these parameters I trained my model and watched the validation loss jump from 0.7973 to 0.2391 after 30 epochs.

I was able to save my network parameters at pre-specified checkpoints during training which allowed me to experiment with the optimization parameters to achieve a lower loss. After the initial 30 epochs, I changed the sequence length to 500 so the network could take advantage of longer-dependencies and I increased the batch size to 5 so that batches of sequences were trained in parallel. After training for 5 epochs, the validation loss decreased to 0.1164. Training an additional 5 epochs with the longer sequence length only improved the loss to 0.1132. At this point I decided to change the optimization algorithm to RMSprop which divides the current gradient by the running average of squared gradients (in effect normalizing the gradients and dealing with radically diminishing learning rates). I also decreased the learning rate to 1e-3 and increased the sequence length to the length of each downsampled, daily activity sequence: 2281. With such a long sequence length, training the network on 1 epoch took much longer than with a sequence length=120. Although I tried to get my Mac's NVIDIA CUDA driver to work I was unable to and had to train on my CPU which meant a lot of the training occurred overnight while I was sleeping or during the day when I was out at work or class. This extra time spent training with RMSprop proved valuable because after 20 epochs, the validation loss decreased to 0.0350. I tried to get it lower still by changing the learning rate and training over more epochs, however the lowest validation loss I was able to achieve was 0.0346 and even with additional training the loss hovered around this value and would even slightly increase.

Sampling from these checkpoint parameters allowed me to see what the network had learned at different points in training. I generated outputs with sequence length 2281 with the thought/hope that my network would output a sequence so similar to the actual data that the final "word" would be 'end' as I intended when I appended it to the end of each sequence. The excerpts below show what the sequence outputs looked like at different points in training, with different "priming" codes, and at different temperatures.

Results

Samples of Generated Activity Sequences				
Validation Loss, Training Parameters	**Temperature=0.2 Priming Code= none**	**Temperature=0.2 Priming Code= 'end'**	**Temperature=0.9 Priming Code= none**	**Temperature=0.9 Priming Code= 'end'**
Loss = 0.7973 (after 1 epoch, Adagrad, sequence length=120)	`082 031 106 040 061 109 040 091 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 …`	`end 125 101 125 040 091 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 …`	`082 084 101 121 091 112 124 124 114 091 000 108 122 124 124 124 … 109 000 109 109 000 000 091 000 000 000 000 000 000 000 000 109 000 000 000 000 000 000 000 000 …`	`end 035 015 027 121 018 023 124 124 051 091 000 075 110 124 124 … 096 000 109 109 000 109 000 109 109 000 109 109 000 000 091 000 000 000 000 000 000 000 000 109 …`

Validation Loss, Training Parameters	Temperature=0.2 Priming Code= none	Temperature=0.2 Priming Code= 'end'	Temperature=0.9 Priming Code= none	Temperature=0.9 Priming Code= 'end'
Loss = 0.1133 (after ~35 epochs, Adagrad, sequence length=500)	`082 000 000 000 000 000 000 000 000 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 …`	`end 000 000 000 000 000 000 000 000 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 109 …`	`082 052 000 091 091 000 124 124 124 124 124 001 124 124 124 124 124 124 124 124 … 091 091 091 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 056 125 091 091 091 091 091 091 091 091 … 091 091 091 091 091 091 091 091 091 091 end 000 …`	`end 000 000 000 000 000 000 000 124 109 124 124 001 124 124 124 124 124 124 124 124 124 124 124 … 091 091 091 091 091 091 091 091 124 124 124 124 124 124 124 124 … 124 124 124 056 125 091 091 091 091 091 091 091 … 091 091 091 end 000 000 000 000 …`
Loss = 0.0346 (after ~60 epochs, RMSprop, sequence length=2281)	`082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 082 …`	`end 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 …`	082 109 109 109 109 109 109 109 109 109 … 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 109 109 109 109 109 109 109 109 109 109 109 109 … 000 000 000 000 000 000 000 000 000 000 109 109 109 109 109 109 109 109 109 …	end 000 000 000 … 000 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 124 … 125 125 125 125 125 082 082 082 082 082 082 082 082 082 082 … 082 082 082 082 082 082 082 082 082 125 125 125 125 125 125 125 125 125 125 125 125 125 125 125 109 109 109 109 109 …

Conclusions

From the generated output it is clear that the network was better at modeling the structure of the input data as the number of epochs trained increased. After the first epoch, the generated output started with what seemed like random activity codes and changed frequently, whereas in later epochs the initial codes were more likely to be repeated. For example, with a loss of 0.7973, each of the first activities in the sequence are different from one another, when in reality, my activity changes much more slowly, something the network learned after additional training. With a loss of 0.0346 and temperature of .9, the generated output looks the most similar to the input text file because the same activity occurs repeatedly (for at least ~5 or more time points) and then changes. This makes sense because each activity code that the network generates represents 30 seconds of activity (since the data was downsampled from 86400 seconds to every 30th second- 2280 time points) and my activity typically does not change every 30 seconds.

I was very happy with the results I generated, however, I was somewhat disappointed that the network did not learn the stopping token I added to each sequence. When I appended the string "end" to each sequence, I envisioned generating sequences equal in length to those that I fed in as input and that ended with the "end" token. Though this was never the case, in some generated sequences that were primed with “end”, the generated output consisted of an additional 1 or 2 “end” activities interspersed randomly amongst the other activity codes.

In the future, I would like to train with more hidden units and perhaps on a GPU so that training can occur more quickly and so that the generated outputs look more like what I expected. There is also room for me to experiment with sequence length and batch sizes, for example, perhaps my network would have learned more had I initiated training with a longer sequence length or more batches (i.e. rather than starting with batch size = 1 and sequence length = 120 starting with batch size = 5 and larger sequence length might have helped me to converge more quickly).

References

1 *Moves*,
https://www.moves-app.com/

2 Word-RNN, Lars Hiller Eidnes,
https://github.com/larspars/word-rnn

3 “The Unreasonable Effectiveness of Recurrent Neural Networks”, Andrej Kapathy,
http://karpathy.github.io/2015/05/21/rnn-effectiveness/

4 “An overview of gradient descent optimization algorithms”, Sebastian Ruder,
http://sebastianruder.com/optimizing-gradient-descent/index.html

5 Neural Networks for Machine Learning, Geoff Hinton,
https://class.coursera.org/neuralnets-2012-001/lecture