Generating new song from set of songs

Shashank Kumar
shashank.kr0328@gmail.com

***Abstract*—The idea is to implement an algorithm which takes a set of vocal-less songs from a music album and generates a new song that could have been a part of this album. In *analysis phase*, the algorithm starts by breaking a song into small windows in time. Then it tries to learn transitions of, energy and phase of frequencies for the given set of songs, using supervised machine learning. Next, in the *generation phase*, user gives initial energy and phase for frequencies to the algorithm and algorithm applies the transitions to these frequencies from window to window, producing a new song.**

Keywords—Supervised learning; gradient descent; DFT;

I. INTRODUCTION

The algorithm can be divided into two phases: Analysis phase and Generation Phase. In Analysis phase input data is read and converted into data structure which makes it easy to be analysed and then uses *gradient descent* for *supervised machine learning*. In *Generation phase*, the results from *gradient descent* is used to predict the transitions as analysed by *Machine Learning algorithm*.

II. ANALYSIS PHASE

A. Loading training set

Initially all songs are made to have same sampling frequency, *fs*. Then smaller duration songs are zero padded to make length of songs same. Then the input songs are divided into windows of 40ms to make audio signal statistically stationary [1] [2]. This window is called a *time bin* (*or timeBin* in code). For each *time bin* take DFT of all songs and store it [3].

B. Data structure used to store training set

A 2D array is used to store the data read from the songs. The rows correspond to song numbers and columns correspond to *time bin.* This data structure is called *timeBinData* and each cell in this 2D array is of *struct* type, which contains the DFT of the corresponding song of the window given by *timeBin.* The Octave code is available at [3].

C. Supervised machine learning

Supervised learning is the machine learning task of inferring a function from labeled training data [4]. Each instance of learning data is a pair of values given as input and expected output. The input data is also called *features* and is often a vector. Algorithm then analyses the training data and generates a function which can then be used to predict an output value for a given set of *features*.

The algorithm uses *supervised learning* for *regression analysis*. In statistics, *regression analysis* is way for modeling the relationship between a scalar dependent variable *y* and vector of input variables, which is denoted here by *X* [5]. This relationship can be then used to predict *y* for given *X*. But, *"Correlation does not imply causation"* and caution is advisable because this may lead to false relations [6] [7] [8].

Representation of the input variables or *features* is a 2D matrix of dimension *m* x *n*, where *n* is number of features and *m* is number of training data. Before analysis this 2D vector, *X*, is padded with an additional column of ones to incorporate the bias constant. This makes *X, m* x *(n+1)* dimensional matrix.

Output or dependent variable is a column vector of length *m*, where each row correspond to output value of that training example.

Hypothesis, *h*, is a function of *features* of a given instance in training data, and constants given by n+1 dimensional column vector, *theta* (θ) and is given as:

$$h_\theta(x^{(k)}) = x^{(k.)}.\ \theta = \theta_0 + \theta_1 x_0^{\ k} + \theta_2 x_2^{\ k} + \ldots + \theta_n x_n^{\ k} \qquad (1)$$

Where, $x^{(k)}$ is row vectors which contains *features* of *k*th training data or the *k*th row of *X*. $x_i^{\ k}$ is *i*th feature for *k*th training data. θ_i is value at *i*th column of θ.

Cost function is given by:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \qquad (2)$$

Algorithm tries to minimize the cost by running *gradient descent* on *cost function*.

To find local minima of *cost function*, *J*, using *gradient descent*, algorithm iteratively takes steps proportional to negative of the gradient of the *cost function* at the current point. Algorithm stops when the gradient of current point is zero. The coefficient of proportionality is here referred to as *alpha* (α).

Gradient of *cost function*, *J*, is given by:

$$\frac{\partial}{\partial \theta_0} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \qquad (3)$$

For every iteration of *gradient descent*, θ is updated as:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_0} J(\theta), \quad j \in [0,n]$$

Substituting gradient of J from (3):

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}), \quad j \in [0,n] \qquad (4)$$

This attempts to find a minimum of a scalar function of several variables, starting at an initial estimate and is generally referred to as *unconstrained nonlinear optimization*. There are many algorithms to iteratively solve unconstrained nonlinear optimization problems. One of them is *Broyden–Fletcher–Goldfarb–Shanno* (BFGS) algorithm [9]. Octave has an inbuilt function called *fminunc* [10] which implements one of the variants of BFGS [11].

In the source code [12] *fminunc*() is invoked in following way:

[theta, cost] = fminunc(@(t)(costFunction(t, X, y)), initial_theta, options);

The first argument is passing *costFunction()* as anonymous function to *fminunc*. *initial_theta* the initial value of θ vector. *options* is a *struct* used to setup optimization function *fminunc* and is created by using *optimset()* [13] [25] function of Octave. One of the primary options was to limit *fminunc* to have maximum 400 iterations [13].

fminunc returns the value *theta, θ,* vector for which, according to *fminunc,* cost is minimum and the value of cost is returned in *cost* variable.

D. Analysis

The first *time bin* of the output signal is initialized by a random musical note or a mix of musical notes, for example C# note on piano [14]. Algorithm predicts the transition of phase and magnitude for *frequency bin* F, from *time bin* T to T+1 by doing a *supervised learning* of transition of *phase* and magnitude for *frequency bin* F, from *time bin* T to T+1 of training data.

For predicting phase for *frequency bin* F and *time bin* T+1 the *features* considered are, for each song [15] [16]:

- Phase of *frequency bin* F and *time bin* T
- Average of phases for *time bin* T, for song under consideration

This creates a 2D array in which rows correspond individual songs and columns correspond to *features*. The output of the training data is the phase for *frequency bin* F and *time bin* T+1.

For predicting magnitude for *frequency bin* F and *time bin* T+1 the *features* considered are, for each song [17] [18]:

- Magnitude of *frequency bin* F and *time bin* T
- Average of magnitudes for *time bin* T, for song under consideration

The output of the training data is the magnitude for *frequency bin* F and *time bin* T+1.

III. Generation Phase

Once the algorithm has θ for phase and magnitude of every *frequency bin,* F in *time bin,* T, has been calculated, value for phase and magnitude for output song can be predicted for *frequency bin,* F in *time bin* T + 1. The predicted value is given by following matrix operation:

$$p = [1\ in] * \theta \qquad (5)$$

Where *p* is predicted value for *frequency bin,* F in *time bin* T + 1. *in* is row vector of *features* of output signal at *time bin* T [19].

IV. Implementation Challanges

A. Execution time issue

One of the major problems faced during the implementation was the time taken for the algorithm to complete. The completion time is in order of 24 hours for 2, 4.9 minute long songs with sampling frequency of 22,000 Hz on Intel Core i5, running at 2.5GHz, 8GB RAM machine on Windows 7. On this machine CPU utilisation was never beyond 25%. And only one of the four CPU threads was used [20]. If all four CPU threads could be used the time taken can reduce by about a factor of four.

Following shows the number of predictions needed to be done:

- Fs = 22,000 Hz
- Duration = 294000 ms (after zero padding songs to make them of same length)
- *time bin* size = 40 ms
- Number of *time bins* = 7,350
- Samples in a *time bin* = 880 samples
- Number of *frequency bins* = 440
- Since each *frequency bin* is evaluated 2 times, one for phase prediction and one for magnitude prediction, total number of analysis-prediction cycles:

  2 * (Number of *frequency bins*) * (Number of *time bins*) = 6468000
- Further, each cycle has 400 iterations for *gradient descent*, which further includes multiple matrix multiplications.

B. Solution: parallel processing

To make predictions run parallel, following was done:

- Write the training data and input data for each *frequency bin* of each *time bin* to a unique file on disk.

- For each training data file generated in above step, a prediction Octave code is generated, called *predictors*. Each of these *predictors* predict only for one set of *features*. A small Java program was used to generate *predictors* [21]. These *predictors* are generated on the fly [22]. The files generated by these *predictors* is kept in [23]. Since these files are generated on the run these folders in source code are empty.
- Java code was used again to run these *predictors* as independent Octave processes. Now that OS sees multiple processes, all CPU threads are being utilised, making the prediction process fairly parallelised.

V. CONCLUSION

It was demonstrated how a new song can be generated from closely related existing set of songs. Also, the time taken to generate the new song was reduced by running different predictions in independent processes.

REFERENCES

[1] Davis, S. Mermelstein, P. (1980) *Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences*. In IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 28 No. 4, pp. 357-366

[2] X. Huang, A. Acero, and H. Hon. *Spoken Language Processing: A guide to theory, algorithm, and system development*. Prentice Hall, 2001.

[3] From source code calculateTimeBinData.m https://github.com/shanxS/song-generator

[4] *Mehryar Mohri, Afshin Rostamizadeh and Ameet Talwalkar, Foundations of Machine Learning*, The MIT Press

[5] Statistical Models: Theory and Practice, Freedman, David. Cambridge University Press.

[6] Armstrong, J. Scott (2012). "Illusions in Regression Analysis". International Journal of Forecasting (forthcoming)

[7] Tufte, Edward R. (2006). "The Cognitive Style of PowerPoint: Pitching Out Corrupts Within". Cheshire, Connecticut: Graphics Press.

[8] Aldrich, John (1995). "Correlations Genuine and Spurious in Pearson and Yule". Statistical Science

[9] BFGS algorithm reference: Avriel, Mordecai (2003), Nonlinear Programming: Analysis and Methods, Dover Publishing

[10] Help page of fminunc(), http://octave.sourceforge.net/octave/function/fminunc.html

[11] fminunc() source code, line 197, http://hg.savannah.gnu.org/hgweb/octave/file/ed1bf35dc11c/scripts/optimization/fminunc.m

[12] From source code predit.m, line 10, 11, https://github.com/shanxS/song-generator

[13] From source code predit.m, line 9, https://github.com/shanxS/song-generator

[14] Piano notes frequencies http://www.math.niu.edu/~rusin/uses-math/music/frequencies

[15] From source code saveTrainingData.m, line 34, https://github.com/shanxS/song-generator

[16] From source code getPhaseDataMatrices.m https://github.com/shanxS/song-generator

[17] From source code saveTrainingData.m, line 53, https://github.com/shanxS/song-generator

[18] From source code getMagnitudeDataMatrices.m https://github.com/shanxS/song-generator

[19] From source code predit.m , line 13, https://github.com/shanxS/song-generator

[20] Core i5 specs: http://ark.intel.com/products/67355/Intel-Core-i5-3210M-Processor-3M-Cache-up-to-3_10-GHz-rPGA

[21] Java source code to generate *predictors*: /java/src/Main.java, https://github.com/shanxS/song-generator

[22] Folder in which *predictors* are stored: /predictors, https://github.com/shanxS/song-generator

[23] Folder in which files generated by *predictors* are kept: /predictedValues https://github.com/shanxS/song-generator

[24] Help page of optimset: http://octave.sourceforge.net/octave/function/optimset.html