

---

# Music Recommendation using Recurrent Neural Networks

---

**Avishkar Bhoopchand**

Department of Computer Science  
MSc Computational Statistics and Machine Learning  
EMAIL@ucl.ac.uk

**Fahad Syed**

Department of Computer Science  
MSc Computational Statistics and Machine Learning  
fahad.syed.15@ucl.ac.uk

**Hipolito Iturraspe**

Department of Computer Science  
MSc Machine Learning  
hipolito.iturraspe.15@ucl.ac.uk

In this model we are looking at how to predict what a user will likely listen to next given their history. This could be seen recommendation list of possible songs the user may want to listen to.

Booked down problem into an NLP problem where we are looking at prediction the next most likely sequence of words not sure if relevant

## 1 Methods

## 2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a deep learning architecture ideally suited for sequential input data. By forming loops, the network is able to remember information about earlier data in the sequence in order to make better predictions. We feed the network a playlist as input and let it learn to predict the same playlist offset by 1 as a target. In this way, the network learns to predict the next song(s) a user may want to play based on the songs they've played so far in a particular playlist. [Reference: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>]

### 2.1 LSTM Cells

Vanilla RNNs use a simple non-linearity such as sigmoid or tanh in each of the cells in sequence. These vanilla RNNs are effective at using recent information for each current prediction task, but tend to quickly "forget" information from earlier in the sequence. In order to overcome this issue, more complex cells capable of modelling memory are used. These Long Short Term Memory (LSTM) cells contain a cell memory vector. They receive as input the current input data as well as the state from the previous LSTM cell in the chain. Using these inputs, the cell can decide whether to read from, write to or erase its memory using 3 soft "gates" namely the input gate, output gate and forget gate. It also decides how much of the input vs how much of its memory it wants to pass along the chain as the state, or to the output. The parameters controlling these gates [Reference: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>]

### 2.2 Architecture

The architecture of the model we built for this task is illustrated in Figure X (insert architecture image from poster). A sequence of input songs from a playlist are passed as inputs to an RNN's input cells. The embedding vector for each song is retrieved and these are passed into an LSTM cell, along with a state vector representing the internal state of the LSTM up to the particular point in the sequence. The output of the LSTM is a vector which is mapped using softmax to a probability distribution over which song the network thinks is most likely to follow after a particular point in the sequence. Note that although it is illustrated (and implemented) as if the entire sequence is passed in one go, it is best to think of the network as processing each song in the sequence one at a time, making a prediction before seeing the next song. In other words, only information from the current song in the sequence and information the network has chosen to remember about earlier songs in the sequence are used for making a prediction at any point.

Multiple layers of LSTM cells can be stacked in order to make the network deep.

Insert image of LSTM cell from poster

### 2.3 Parameters and Training

The parameters of the network include embedding vectors for each song, parameters for the 3 gates of the LSTM cells (a matrix and bias vector) separate for each layer and a matrix and bias parameter for the output softmax layer. The parameters are trained by comparing the output predictions of the network with the actual targets. The cross entropy loss is computed (formula below) and the derivative of this is fed back into the network by backpropagation in order to update the network parameters. [Reference: <http://neuralnetworksanddeeplearning.com/chap3.html>]  $C = \frac{1}{n} \sum [y \log a + (1-y) \log (1-a)]$

## 3 Implementation

### 3.1 Batching

In order to train efficiently using Stochastic Gradient Descent, batching is used whereby a batch of multiple playlists (the size of which is configurable) are passed to the network at each iteration. An issue arises due to different playlists having different lengths in the same batch. Luckily Tensorflow provides a principled way of dealing with this by allowing one to specify the lengths of each sequence in a batch. The RNNs zero their predictions when they reach the size limit of each sequence in a batch.

### 3.2 Training Method

The model is trained using a variation of Stochastic Gradient Descent called the Adam Optimiser (an implementation of which comes with Tensorflow). This is an efficient optimiser that uses first-order gradients and is able to automatically adapt its learning rate using first and second order moments [Reference: <http://arxiv.org/pdf/1412.6980v8.pdf>]. We therefore did not need to implement a learning rate decay schedule that would usually be required for stochastic gradient descent. In order to address the possibility of exploding gradients during training, gradient clipping was employed, whereby the magnitude of gradients are clipped if they exceed a threshold.

### 3.3 Configuration

The model we implemented used 2 layers, hidden state vector and embedding vector dimensions of 100, a batch size of 100, a gradient clipping threshold of 5 and an unravelled sequence length of 20 songs.

#### Conclusion / Results

Possible steps to improve Scalability, possibly hard to scale, unless there is some recursive RNN alg. I remember reading somewhere that there is one. Metric used in RNN different from actual metric used to measure performance. hard to find the gradient of day recall. Using no a large dataset

The effect of many of these properties on the user experience is unclear, and depends on the application, without doing extensive online testing it is hard to tell which features to use to measure recommender systems, that have the most benefit to the user.

Users need to trust the system. Hard problem. How long we should recommend a song if a user does not click on it.

recommendations - youtube does this like crazy, doesnt really help in all honesty. Novelty - recommend items that the user did not know about Serendipity - how successful the recommends are; possible hard to evaluate. Scalability Robustness How to calculate the cost benefit and false/true positive/negative - hard task Due to our offline learning we do not know if what we recommend is something that the user will actually wants, if they dont click on it does not mean that the user does not like or want to the content as they may have just not looked at it. And if they clicked on it does not mean that they actually want that product.

More time available, we could train a model to predict latent representations of songs using the actual audio signal and this would all use to predict new song by running them through a deep neural network and recommending them to users based on similarity

## 4 Code structure

The files we implemented for this project are listed below. Please see the comments in the files themselves for further details.

**musicmodel.py** This file provides the MusicModel class which constructs a tensorflow computation graph which represents the recurrent neural network unravelled for a configurable number of steps. Various placeholder operators from the computation graph are made publically

visible so that the relevant input data, targets and initial state can be passed and the final costs (loss) can be accessed.

**modeltrainer.py** This file provides the Trainer class which is used to train the model. The train function runs a configurable number of training epochs and after each epoch calls a list of hooks which display information about the training progress. Once training has completed, the model parameters are saved to a file so that the trained model can be later restored and evaluated. Each epoch consists of running through the batches in the training data, passing these to the model and then evaluating the model's cost function and training operator to invoke backpropagation. If the sequences in the batch are too long, they are broken up into sequence.length chunks which are fed one at a time to the network, using the final state of the last chunk as the initial state.

**yes\_reader.py** This file loads the data set which consists of a separate training and test file. The train file is split into a training and validation set. Iterators are provided which split the training data into batches of the correct sequence length.

**hooks.py** This file implements two hooks that output data during training. The first is a LossHook which merely reports the training set loss at the end of an epoch. The second runs the validation dataset through the network, calculates the loss for this dataset and reports it. The advantage of using such a hook is that the generalisation ability of the network can be assessed while it is being trained.

**deepcf.py** This file pulls the various peices together. Configuration of the model can be specified here and the training or evaluation can be invoked using command line arguments.